Skill India
कौशल भारत-कुशल भारत

MINISTRY OF SKILL DEVELOPMENT & ENTREPRENEURSHIP
GOVERNMENT OF INDIA

N·S·D·C
RE IMAGINE FUTURE

India skills

IT - ITeS SSC
nasscom

Technical Handbook

IT Software Solution for Business

SSC/Q0111

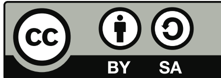
This book is sponsored by:

IT-ITeS Sector Skill Council
NASSCOM, Plot No. 7, 8, 9 & 10, 3rd Floor,
Sector 126, Noida Uttar Pradesh – 201303
Phone: +91-120-4990111
Email: sscnasscom@nasscom.in
Web: www.sscnasscom.com

First Edition

Printed in India

Copyright © 2024
Under Creative Commons License: CC-BY-SA
Attribution-ShareAlike: CC-BY-SA



Disclaimer:
The information contained herein has been obtained from sources reliable to IT-ITeS Sector Skill Council. IT-ITeS Sector Skill Council disclaims all warranties to the accuracy, completeness or adequacy of such information. IT-ITeS Sector Skill Council shall have no liability for errors, omissions, or inadequacies, in the information contained herein, or for interpretations thereof. Every effort has been made to trace the owners of the copyright material included in the book. The publishers would be grateful for any omissions brought to their notice for acknowledgements in future editions of the book. No entity in IT-ITeS Sector Skill Council shall be responsible for any loss whatsoever, sustained by any person who relies on this material.

Acknowledgements

On behalf of IT-ITeS SSC, we extend our sincere appreciation to all individuals and teams who have significantly contributed to the creation and publication of this technical handbook on the skill IT Software Solution for Business for IndiaSkills. Our sincere thanks go to Ministry of Skill Development and Entrepreneurship (MSDE) and National Skill Development Corporation (NSDC) for their contribution towards the development of this book and their constructive feedback. We owe a debt of gratitude to our leadership at IT-ITeS SSC as well as the subject matter experts for their invaluable insights that have greatly enhanced the quality of this work. We also acknowledge the unwavering support of our editorial and production teams, whose professionalism and dedication have been instrumental in bringing this project to life. Finally, we express our heartfelt appreciation to the candidates who inspire us to continuously strive for excellence. Your support and engagement are the driving forces behind our mission to empower future generations through skill building initiatives such as IndiaSkills.

About this book

IndiaSkills Competition is the country's biggest skill competition, designed to demonstrate the highest standards of skilling and offers a platform for youngsters to showcase their talent at national and international levels. This technical handbook contains information about the details related to IT Software Solution for Busines skill of IndiaSkills competition. It serves as a comprehensive guide to understanding the IndiaSkills competition and the IT Software Solution for Busines skill and its core principles- providing readers with a solid foundation in both theoretical concepts and practical applications. Designed for the candidates, subject matter experts, IndiaSkills stakeholders, and the competition enthusiasts alike, this book offers insights, understanding, and the skill-sets required to participate in the competition.

Symbols Used

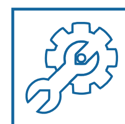
Key Learning Outcomes

Unit Objectives

Exercise

Tips

Notes



Activity

Summary

Table of Contents

S. No.	Modules and Units	Page No.
1.	**Introduction to Software Development**	**1**
	Unit 1.1: Overview of Software Development Lifecycle (SDLC)	3
	Unit 1.2: Roles and Responsibilities in Software Development	8
2.	**Problem Solving and Analysis**	**13**
	Unit 2.1: Understanding User Requirements	15
	Unit 2.2: Functional and Non-Functional Requirements Analysis	17
	Unit 2.3: Systems Analysis Techniques	19
3.	**System Design**	**23**
	Unit 3.1: Object-Oriented Design Principles	25
	Unit 3.2: Database Design	28
	Unit 3.3: UI/UX Design Principles	32
	Unit 3.4: Security and Access Control Design	35
4.	**Software Development**	**39**
	Unit 4.1: Introduction to Programming Languages and Development Environments	41
	Unit 4.2: Coding Techniques and Best Practices	43
	Unit 4.3: Building Advanced Software Applications	45
	Unit 4.4: Developing Web and Mobile User Interfaces	48
5.	**Software Testing**	**51**
	Unit 5.1: Testing Fundamentals and Techniques	53
	Unit 5.2: Designing Effective Test Cases	56
	Unit 5.3: Debugging, Error Handling and Test Reporting	58
6.	**Technical Communication**	**61**
	Unit 6.1: Technical Documentation and Tools	63
	Unit 6.2: Communicating with Stakeholders	65

Skill India
कौशल भारत-कुशल भारत

सत्यमेव जयते
MINISTRY OF SKILL DEVELOPMENT
& ENTREPRENEURSHIP
GOVERNMENT OF INDIA

N·S·D·C
RE IMAGINE FUTURE

India skills

IT - ITeS SSC
nasscom

1. Introduction to Software Development

Unit 1.1: Overview of Software Development Lifecycle (SDLC)

Unit 1.2: Roles and Responsibilities in Software Development

Key Learning Outcomes

At the end of this module, you will be able to:

1. Describe the different phases of the Software Development Lifecycle (SDLC).
2. Identify the roles and responsibilities involved in software development projects.

Unit 1.1: Overview of Software Development Lifecycle (SDLC)

Unit Objectives

By the end of this unit, the participants will be able to:

1. Define the Software Development Lifecycle (SDLC) and its various phases.
2. Explain the purpose of each SDLC phase.
3. Identify key deliverables associated with each SDLC phase.

1.1.1 SDLC and its phases

What is Software Development?

Software development is the process of creating software, which involves various key steps such as programming, maintaining source code, project conception, feasibility evaluation, business requirement analysis, software design, testing, and release.

In addition to development, software engineering encompasses project management, employee management, and other administrative functions. It can be carried out through sequential or iterative development methods to enhance flexibility, efficiency, and scheduling.

What is SDLC?

The software development lifecycle (SDLC) is the cost-effective and time-efficient process that development teams use to design and build high-quality software. The goal of SDLC is to minimize project risks through forward planning so that software meets customer expectations during production and beyond. This methodology outlines a series of steps that divide the software development process into tasks you can assign, complete, and measure.

Why is SDLC important?

Software development can be challenging to manage due to changing requirements, technology upgrades, and cross-functional collaboration. The software development lifecycle (SDLC) methodology provides a systematic management framework with specific deliverables at every stage of the software development process. As a result, all stakeholders agree on software development goals and requirements upfront and also have a plan to achieve those goals.

Here are some benefits of SDLC:

- Increased visibility of the development process for all stakeholders involved
- Efficient estimation, planning, and scheduling
- Improved risk management and cost estimation
- Systematic software delivery and better customer satisfaction

How does SDLC work?

The software development lifecycle (SDLC) outlines several tasks required to build a software application. The development process goes through several stages as developers add new features and fix bugs in the software.

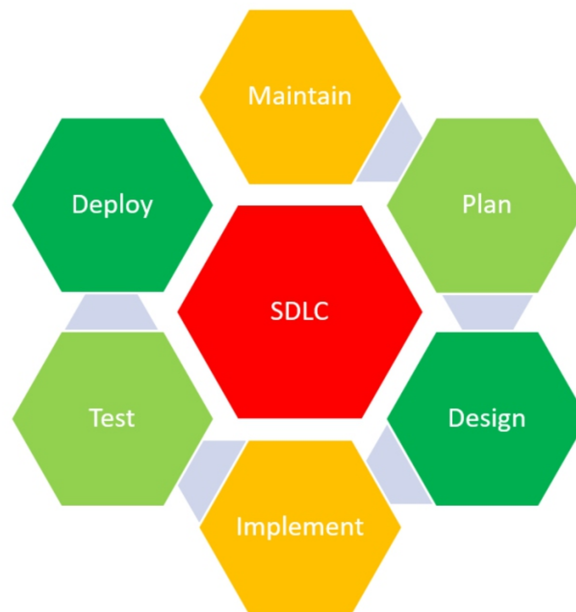


Fig. 1.1: Overview of software development lifecycle (SDLC)

The details of the SDLC process vary for different teams. However, we outline some common SDLC phases below:

Plan

The planning phase typically includes tasks like cost-benefit analysis, scheduling, resource estimation, and allocation. The development team collects requirements from several stakeholders such as customers, internal and external experts, and managers to create a software requirement specification document. The document sets expectations and defines common goals that aid in project planning. The team estimates costs, creates a schedule, and has a detailed plan to achieve their goals.

Design

In the design phase, software engineers analyze requirements and identify the best solutions to create the software. For example, they may consider integrating pre-existing modules, make technology choices, and identify development tools. They will look at how to best integrate the new software into any existing IT infrastructure the organization may have.

Implement

In the implementation phase, the development team codes the product. They analyze the requirements to identify smaller coding tasks they can do daily to achieve the final result.

Test

The development team combines automation and manual testing to check the software for bugs. Quality analysis includes testing the software for errors and checking if it meets customer requirements. Because many teams immediately test the code they write, the testing phase often runs parallel to the development phase.

Deploy

When teams develop software, they code and test on a different copy of the software than the one that the users have access to. The software that customers use is called production, while other copies are said to be in the build environment, or testing environment.

Having separate build and production environments ensures that customers can continue to use the software even while it is being changed or upgraded. The deployment phase includes several tasks to move the latest build copy to the production environment, such as packaging, environment configuration, and installation.

Maintain

In the maintenance phase, among other tasks, the team fixes bugs, resolves customer issues, and manages software changes. In addition, the team monitors overall system performance, security, and user experience to identify new ways to improve the existing software.

1.1.2 Purpose of SDLC Phase

1. Planning & Analysis

The first phase of the SDLC is the project planning stage where you are gathering business requirements from your client or stakeholders. This phase is when you evaluate the feasibility of creating the product, revenue potential, the cost of production, the needs of the end-users, etc. To properly decide what to make, what not to make, and what to make first, you can use a feature prioritization framework that takes into account the value of the software/update, the cost, the time it takes to build, and other factors.
Once it is decided that the software project is in line with business and stakeholder goals, feasible to create, and addresses user needs, then you can move on to the next phase.

2. Define Requirements

This phase is critical for converting the information gathered during the planning and analysis phase into clear requirements for the development team. This process guides the development of several important documents: a software requirement specification (SRS) or product specification, a Use Case document, and a Requirement Traceability Matrix document.

3. Design

The design phase is where you put pen to paper—so to speak. The original plan and vision are elaborated into a software design document (SDD) that includes the system design, programming language, templates, platform to use, and application security measures. This is also where you can flowchart how the software responds to user actions.

In most cases, the design phase will include the development of a prototype model. Creating a pre-production version of the product can give the team the opportunity to visualize what the product will look like and make changes without having to go through the hassle of rewriting code.

4. Development

The actual development phase is where the development team members divide the project into software modules and turn the software requirement into code that makes the product.
This SDLC phase can take quite a lot of time and specialized development tools. It's important to have a set timeline and milestones so the software developers understand the expectations and you can keep track of the progress in this stage.

In some cases, the development stage can also merge with the testing stage where certain tests are run to ensure there are no critical bugs. Keep in mind, different types of product development software will have different specialties so you'll want to pick the one that suits you best.

5. Testing

Before getting the software product out the door to the production environment, it's important to have your quality assurance team perform validation testing to make sure it is functioning properly and does what it's meant to do. The testing process can also help hash out any major user experience issues and security issues.

In some cases, software testing can be done in a simulated environment. Other simpler tests can also be automated.

The types of testing to do in this phase:

- **Performance testing:** Assesses the software's speed and scalability under different conditions
- **Functional testing:** Verifies that the software meets the requirements
- **Security testing:** Identifies potential vulnerabilities and weaknesses
- **Unit-testing:** Tests individual units or components of the software
- **Usability testing:** Evaluates the software's user interface and overall user experience
- **Acceptance testing:** Also termed end-user testing, beta testing, application testing, or field testing, this is the final testing stage to test if the software product delivers on what it promises

6. Deployment

During the deployment phase, your final product is delivered to your intended user. You can automate this process and schedule your deployment depending on the type. For example, if you are only deploying a feature update, you can do so with a small number of users (canary release). If you are creating brand-new software, you can learn more about the different stages of the software release life cycle (SRLC).

7. Maintenance

The maintenance phase is the final stage of the SDLC if you're following the waterfall structure of the software development process. However, the industry is moving towards a more agile software development approach where maintenance is only a stage for further improvement.

In the maintenance stage, users may find bugs and errors that were missed in the earlier testing phase. These bugs need to be fixed for better user experience and retention.

In some cases, these can lead to going back to the first step of the software development life cycle. The SDLC phases can also restart for any new features you may want to add in your next release/update.

It's important to note that the SDLC is not always a strictly linear process. Depending on the chosen SDLC model, there may be iterations and feedback loops between phases. However, understanding the purpose and activities within each phase provides a clear roadmap for successful software development.

1.1.3 Key deliverables associated with SDLC phase

The Software Development Life Cycle (SDLC) typically consists of several phases, each with its own set of key deliverables.

1. Planning Phase:

- **Project Charter:** Document outlining project objectives, scope, stakeholders, and constraints.
- **Project Plan:** Detailed roadmap including tasks, timelines, resources, and dependencies.
- **Feasibility Study:** Assessment of the project's viability in terms of technical, economic, and operational aspects.

2. Requirements Analysis Phase:

- **Business Requirements Document (BRD):** Detailed description of user needs, system features, and functionalities.
- **Functional Requirements Specification (FRS):** Specific requirements related to system functions, inputs, outputs, and interfaces.
- **Non-Functional Requirements Document:** Requirements concerning performance, security, usability, and other quality attributes.

3. Design Phase:

- **System Design Document (SDD):** Blueprint detailing system architecture, components, modules, and interfaces.

- **Detailed Design Document (DDD):** Elaboration of individual components/modules, algorithms, data structures, and database schemas.
- **User Interface (UI) Design:** Mockups or prototypes illustrating the graphical layout and interaction flow of the application.

4. Implementation Phase:

- **Source Code:** Actual program code written in the chosen programming language(s).
- **Unit Test Cases:** Test scripts to validate the functionality of individual units or modules.
- **Integration Test Cases:** Test scripts to verify the interaction and integration of different modules/components.

5. Testing Phase:

- **Test Plan:** Document outlining the testing strategy, scope, resources, and schedule.
- **Test Cases:** Detailed instructions for executing tests to verify system requirements and functionality.
- **Test Reports:** Documentation of test results, including defects found, their severity, and steps to reproduce them.

6. Deployment Phase:

- **Installation Guide:** Instructions for installing and configuring the software in the production environment.
- **Release Notes:** Documentation highlighting the new features, enhancements, and bug fixes included in the release.
- **User Manuals:** Guides providing instructions on how to use the software effectively.

7. Maintenance Phase:

- **Change Requests:** Formal requests for modifications, enhancements, or bug fixes to the software.
- **Bug Reports:** Documentation of defects found in the deployed software, including steps to reproduce and impact analysis.
- **Service Level Agreements (SLAs):** Agreements defining the level of support and maintenance provided to the users or customers.

Unit 1.2: Roles and Responsibilities in Software Development

Unit Objectives

By the end of this unit, the participants will be able to:

1. Identify the key roles involved in software development projects (e.g., developers, testers, analysts, project managers).
2. Describe the responsibilities and deliverables associated with each role.

1.2.1 Roles Involved in software Development Projects

Key roles involved in software development projects typically include:

1. **Developers/Programmers:** Responsible for writing code according to specifications, implementing software features, and ensuring code quality.
2. **Testers/QA Engineers:** Responsible for testing the software to identify and report defects, ensuring the quality and reliability of the final product.
3. **Business Analysts:** Responsible for gathering and analyzing requirements from stakeholders, translating business needs into technical requirements, and ensuring alignment between business goals and technical solutions.
4. **Project Managers:** Responsible for overall project planning, coordination, and execution, including managing resources, schedules, budgets, and communication with stakeholders.
5. **Software Architects:** Responsible for designing the overall architecture of the software system, defining technical standards, and ensuring the scalability, reliability, and performance of the system.
6. **User Experience (UX) Designers:** Responsible for designing intuitive and user-friendly interfaces, conducting user research, creating wireframes and prototypes, and ensuring a positive user experience.
7. **System Administrators/DevOps Engineers:** Responsible for setting up and maintaining development, testing, and production environments, automating deployment processes, monitoring system performance, and ensuring the reliability and security of the infrastructure.

1.2.2 Responsibilities and deliverables associated with each role

Certainly, let's delve into the responsibilities and deliverables associated with each role in software development:

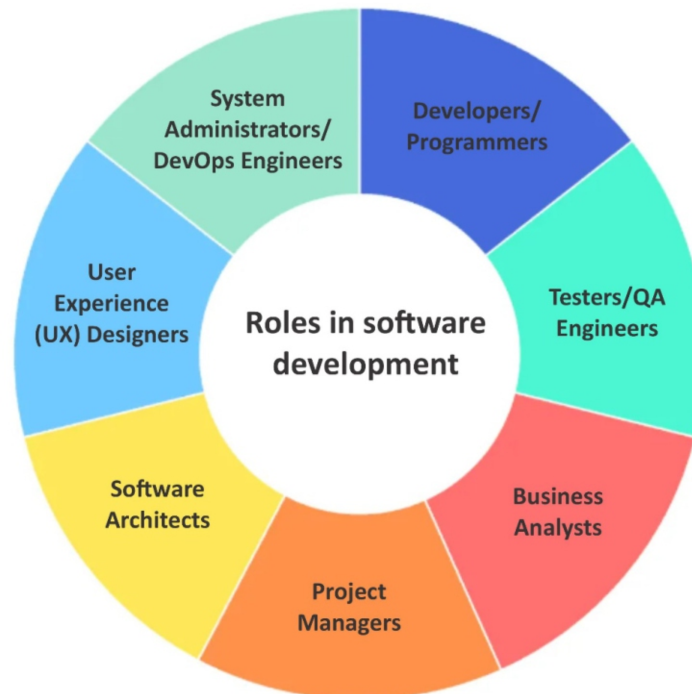


Fig. 1.2: Different roles in software development

1. Developers/Programmers:

- **Responsibilities:**
  - o Write code based on technical specifications and design documents.
  - o Implement software features and functionality.
  - o Debug and troubleshoot issues in the code.
  - o Collaborate with other team members, such as testers and architects.

- **Deliverables:**
  - o **Source code:** The actual code written to implement features.
  - o **Unit tests:** Tests written to verify the functionality of individual units or components.
  - o **Code documentation:** Documentation explaining the code's structure, functionality, and usage.

2. Testers/QA Engineers:

- **Responsibilities:**
  - o Develop test plans and test cases based on requirements and design documents.
  - o Execute various types of testing, such as functional testing, integration testing, and regression testing.
  - o Report defects found during testing and track them to resolution.
  - o Verify fixes and ensure that software meets quality standards.

- **Deliverables:**
  - o **Test plans:** Documents outlining the testing approach, scope, and resources.
  - o **Test cases:** Detailed instructions for conducting tests to verify software functionality.
  - o **Defect reports:** Documentation of identified defects, including steps to reproduce and severity.

3. Business Analysts:

- **Responsibilities:**
  - o Gather and analyze business requirements from stakeholders.
  - o Translate business needs into technical requirements and specifications.
  - o Create use cases, user stories, or functional requirements documents.
  - o Collaborate with stakeholders and the development team to ensure alignment between business goals and technical solutions.

- **Deliverables:**
  - o **Business requirements document:** Detailed description of business needs, goals, and objectives.
  - o **Functional requirements document:** Specifications outlining the software's features and functionality.
  - o **Use cases/user stories:** Scenarios describing how users interact with the software to achieve specific goals.

4. Project Managers:

- **Responsibilities:**
  - o Plan and schedule project activities, resources, and timelines.
  - o Monitor project progress and track milestones.
  - o Manage project risks and issues.
  - o Communicate with stakeholders to provide updates and address concerns.

- **Deliverables:**
  - o **Project plan:** Document outlining project objectives, scope, schedules, and resource allocation.
  - o **Status reports:** Regular updates on project progress, milestones achieved, and issues encountered.
  - o **Risk management plan:** Strategy for identifying, assessing, and mitigating project risks.

5. Software Architects:

- **Responsibilities:**
  - o Design the overall architecture of the software system.
  - o Define technical standards, guidelines, and best practices.
  - o Evaluate and select appropriate technologies and tools.
  - o Ensure the scalability, reliability, and performance of the system architecture.

- **Deliverables:**
  - o **System architecture diagrams:** Visual representations of the software system's structure and components.
  - o **Technical design documents:** Detailed descriptions of the system's architecture, components, and interfaces.
  - o **Prototypes or proof-of-concepts:** Initial implementations to validate architectural decisions and design concepts.

6. User Experience (UX) Designers:

- **Responsibilities:**
  - o Design intuitive and user-friendly interfaces.
  - o Conduct user research to understand user needs and preferences.
  - o Create wireframes, mockups, and prototypes to visualize design concepts.
  - o Ensure consistency, accessibility, and usability in the user interface design.

- **Deliverables:**
  - o **Wireframes:** Schematic diagrams illustrating the layout and structure of user interfaces.
  - o **Mockups:** Visual representations of user interfaces with design elements and content.

- o **Interactive prototypes:** Functional simulations of the software interface to demonstrate user interactions and flows.

7. System Administrators/DevOps Engineers:

- **Responsibilities:**
  - o Set up and configure development, testing, and production environments.
  - o Automate deployment processes using tools and scripts.
  - o Monitor system performance and troubleshoot issues.
  - o Ensure the security, reliability, and scalability of the infrastructure.

- **Deliverables:**
  - o **Deployment scripts:** Automated scripts for deploying software updates and configurations.
  - o **Monitoring dashboards:** Tools and interfaces for monitoring system performance and health.
  - o **Infrastructure documentation:** Documentation detailing the setup, configuration, and maintenance procedures for the infrastructure components.

Skill India
कौशल भारत-कुशल भारत
MINISTRY OF SKILL DEVELOPMENT & ENTREPRENEURSHIP
GOVERNMENT OF INDIA
N·S·D·C
RE IMAGINE FUTURE
India skills

IT - ITeS SSC
nasscom

2. Problem Solving and Analysis

Unit 2.1: Understanding User Requirements

Unit 2.2: Functional and Non-Functional Requirements Analysis

Unit 2.3: Systems Analysis Techniques

Key Learning Outcomes

At the end of this module, you will be able to:

1. Apply various techniques to gather and understand user requirements.
2. Differentiate between functional and non-functional requirements.
3. Utilize system analysis techniques like Use Case Diagrams, Class Diagrams, and Entity Relationship Diagrams to model software systems.

Unit 2.1: Understanding User Requirements

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain different techniques for gathering user requirements (interviews, questionnaires, document analysis).
2. Analyze user requirements to identify functional and non-functional needs.

2.1.1 Understanding User Requirements

Certainly! Understanding user requirements is crucial for developing software that meets the needs of its intended users. Here are some techniques for gathering user requirements and analyzing them to identify functional and non-functional needs:

1. Interviews:

- **Description:** Involves direct interaction between the project team and stakeholders or end-users to gather information about their needs, preferences, and expectations.
- **Process:** Conduct one-on-one or group interviews where open-ended questions are asked to explore user requirements in-depth.
- **Analysis:** Transcribe and categorize interview responses to identify common themes and requirements. Look for patterns and prioritize requirements based on their frequency and importance.

2. Questionnaires/Surveys:

- **Description:** Distribute structured questionnaires or surveys to a large group of stakeholders or end-users to gather feedback and requirements.
- **Process:** Design clear and concise questions covering various aspects of the software, such as features, usability, and performance. Collect responses anonymously or with respondent identification.
- **Analysis:** Aggregate survey responses and analyze the data using statistical techniques to identify trends and priorities. Look for consensus among respondents and areas of divergence.

3. Document Analysis:

- **Description:** Review existing documentation related to the project, such as business reports, user manuals, and technical specifications, to extract relevant requirements.
- **Process:** Analyze documents to identify explicit and implicit requirements, constraints, and assumptions. Look for inconsistencies or gaps in the information.
- **Analysis:** Create a requirements traceability matrix to link documented requirements to specific sources. Validate requirements with stakeholders to ensure accuracy and completeness.

4. Observation/Job Shadowing:

- **Description:** Observe users performing their tasks in their natural environment to understand their workflow, challenges, and needs.
- **Process:** Shadow users as they perform their daily activities, taking notes and documenting observations. Pay attention to pain points, workarounds, and opportunities for improvement.
- **Analysis:** Analyze observation notes to identify recurring patterns and user needs. Use insights gained to propose solutions that address observed challenges and streamline workflows.

Once user requirements are gathered, they need to be analyzed to identify both functional and non-functional needs:

- **Functional Requirements:**
  - o **Definition:** Specify what the system should do or the functions it should perform to meet user needs and achieve its objectives.
  - o **Identification:** Look for user requests or descriptions of desired system behavior in the gathered requirements.
  - o **Examples:** User authentication, data entry forms, report generation, search functionality.

- **Non-functional Requirements:**
  - o **Definition:** Specify how the system should perform or the qualities it should possess, such as performance, security, and usability.
  - o **Identification:** Identify constraints, quality attributes, or system characteristics mentioned in the requirements.
  - o **Examples:** Response time, system availability, data security, user interface responsiveness.

Unit 2.2: Functional and Non-Functional Requirements Analysis

Unit Objectives

By the end of this unit, the participants will be able to:

1. Differentiate between functional and non-functional requirements of a software system.
2. Provide examples of functional and non-functional requirements

2.2.1 Analysis of Functional and Non-Functional Requirement

1. Functional Requirements:

Functional requirements specify what the system should do. They describe the behavior and functionality of the software system in terms of specific tasks, actions, and operations it must perform to meet the user's needs and achieve its intended purpose.

Examples of functional requirements include:

- **User Management:**
    - o The system shall allow users to register, log in, and manage their profiles.
    - o Users shall be able to reset their passwords if forgotten.

- **Order Processing:**
    - o The system shall allow users to add items to a shopping cart, view the cart contents, and proceed to checkout.
    - o Upon checkout, the system shall calculate the total cost, apply discounts if applicable, and generate an order confirmation.

- **Search Functionality:**
    - o The system shall provide a search feature allowing users to search for products by name, category, or keyword.
    - o Search results shall be displayed in a paginated manner with relevant information and sorting options.

- **Reporting:**
    - o The system shall generate monthly sales reports showing total revenue, top-selling products, and sales trends.
    - o Reports shall be exportable in PDF and CSV formats.

2. Non-Functional Requirements:

Non-functional requirements specify how the system should perform. They describe the qualities, constraints, and characteristics of the software system that are not directly related to its functionality but are essential for ensuring its overall quality, performance, and user experience.

Examples of non-functional requirements include:

- **Performance:**
    - o The system shall load product pages in less than 3 seconds under normal load conditions.
    - o The system shall support a maximum of 1000 concurrent users without significant degradation in performance.

- **Security:**
  - o User passwords shall be stored using encryption algorithms and shall not be stored in plain text.
  - o Access to sensitive data shall be restricted based on user roles and permissions.
- **Usability:**
  - o The user interface shall be intuitive and easy to navigate, requiring minimal training for new users.
  - o Error messages shall be clear, concise, and provide guidance on how to resolve issues.
- **Reliability:**
  - o The system shall have a mean time between failures (MTBF) of at least 1000 hours.
  - o Backup and recovery mechanisms shall be in place to ensure data integrity and system availability in case of failures.
- **Scalability:**
  - o The system architecture shall be designed to scale horizontally to accommodate increasing user loads.
  - o Database performance shall scale linearly with the growth of data volume.

2.2.2 Difference between Functional Requirements and Non-Functional Requirements:

Functional Requirements	Non-Functional Requirements
A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
It specifies "What should the software system do?"	It places constraints on "How should the software system fulfill the functional requirements?"
Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
It is mandatory.	It is not mandatory.
It is captured in use case.	It is captured as a quality attribute.
Defined at a component level.	Applied to a system as a whole.
Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
Usually easy to define.	Usually more difficult to define.
Example: 1) Authentication of user whenever he/she logs into the system. 2) System shutdown in case of a cyber-attack. 3) A Verification email is sent to user whenever he/she registers for the first time on some software system.	Example: 1) Emails should be sent with a latency of no greater than 12 hours from such an activity. 2) The processing of each request should be done within 10 seconds 3) The site should load in 3 seconds when the number of simultaneous users are > 10000

Table 2.1: Difference between functional requirements and non-functional requirements

Unit 2.3: Systems Analysis Techniques

Unit Objectives

By the end of this unit, the participants will be able to:

1. Apply Use Case Diagrams to model system functionality from a user's perspective.
2. Utilize Class Diagrams to represent the objects and their relationships in a system.
3. Construct Entity Relationship Diagrams (ERDs) to model data entities and their relationships within a database.

2.3.1 User Needs with Use Case Diagrams

1. Use Case Diagrams:

- **Purpose:** Use case diagrams are used to model the interactions between users (actors) and the system, depicting the system's functionality from a user's perspective.
- **Components:**
  - o **Actors:** Represent external entities (e.g., users, systems) interacting with the system.
  - o **Use Cases:** Represent specific actions or functionalities that the system provides to its users.
  - o **Relationships:** Connect actors to use cases to show which actors are involved in each use case.
- **Example:** In an online shopping system, actors might include "Customer" and "Administrator," while use cases could include "Login," "Add to Cart," "Checkout," and "Manage Inventory."

2. Benefits of Use Case Diagrams:

- **Improved Communication:** Use Case Diagrams bridge the gap between technical and non-technical stakeholders. They provide a clear picture of system functionality, fostering better communication and shared understanding.
- **Requirements Gathering:** By capturing user interactions, these diagrams help identify and document system requirements. This ensures the developed system aligns with user needs.
- **Early Design Decisions:** Use Case Diagrams inform early design decisions. By understanding user workflows, developers can design a user-friendly and efficient system.

Creating Use Case Diagrams:

This sub-unit will guide you through the steps of creating Use Case Diagrams using a standardized visual notation. You will learn how to identify actors, define use cases, and illustrate the flow of events.

While Use Case Diagrams provide a high-level overview, you will also explore:

- **Types of Use Cases:** Different use cases, such as primary, secondary, and exceptional, will be covered, allowing you to model various user interactions.
- **Relationships Between Use Cases:** Understanding how use cases can be related (e.g., extend, include) provides a more comprehensive view of system functionality.

2.3.2 Object-Oriented Analysis with Class Diagrams

Object-Oriented Analysis, a powerful approach that uses Class Diagrams to model a system based on real-world objects and their interactions.

Object-Oriented Analysis:

Traditional analysis might focus on functions and processes. Object-Oriented Analysis flips the script, viewing a system through the lens of objects: things in the system world with properties and behaviors.

Class Diagrams:

- **Purpose:** Class diagrams are used to represent the static structure of a system, focusing on the objects (classes) in the system and their relationships.
- **Components:**
  - o **Classes:** Represent entities or objects in the system, along with their attributes and methods.
  - o **Associations:** Represent relationships between classes, indicating how they are connected.
  - o **Multiplicity:** Specifies the number of instances of one class that can be associated with instances of another class.
- **Example:** In a library system, classes might include "Book," "Author," and "Library," with associations indicating that each book is written by one or more authors and belongs to a library.

Benefits of Class Diagrams:

- **Clear System Understanding:** By visualizing objects and their interactions, Class Diagrams provide a clear understanding of the system's structure and functionality.
- **Improved Design:** They promote well-structured and maintainable code by encouraging developers to think in terms of reusable objects.
- **Enhanced Communication:** A visual representation facilitates communication between designers, developers, and stakeholders.

Building Class Diagram Skills:

To create Class Diagrams using the Unified Modeling Language (UML) notation, you will learn how to:

- **Identify Objects:** Look for real-world entities within the system and their relevant attributes.
- **Define Operations:** Specify the actions (methods) that objects can perform.
- **Illustrate Relationships:** Use UML notation to depict associations, inheritance, and aggregation between classes.

While Use Class Diagrams provide a high-level overview, you will also explore:

- **Types of Use Cases:** Different use cases, such as primary, secondary, and exceptional, will be covered, allowing you to model various user interactions.
- **Relationships Between Use Cases:** Understanding how use cases can be related (e.g., extend, include) provides a more comprehensive view of system functionality.

2.3.3 Data Modeling with Entity Relationship Diagrams (ERDs)

The foundation of any software system is its data. This sub-unit equips you with Entity Relationship Diagrams (ERDs), a powerful tool for data modeling. ERDs visualize the data entities (things you store information about) within a system and the relationships between them.

Entity Relationship Diagrams (ERDs):

- **Purpose:** ERDs are used to model the data entities and their relationships within a database, providing a visual representation of the database schema.

- **Components:**
  - o **Entities:** Represent the main objects or concepts in the database, such as customers, products, or orders.
  - o **Relationships:** Represent the connections between entities, indicating how they are related to each other.
  - o **Attributes:** Describe the properties or characteristics of entities.
- **Example:** In a university database, entities might include "Student," "Course," and "Professor," with relationships indicating that each student enrolls in multiple courses and each course is taught by one or more professors.

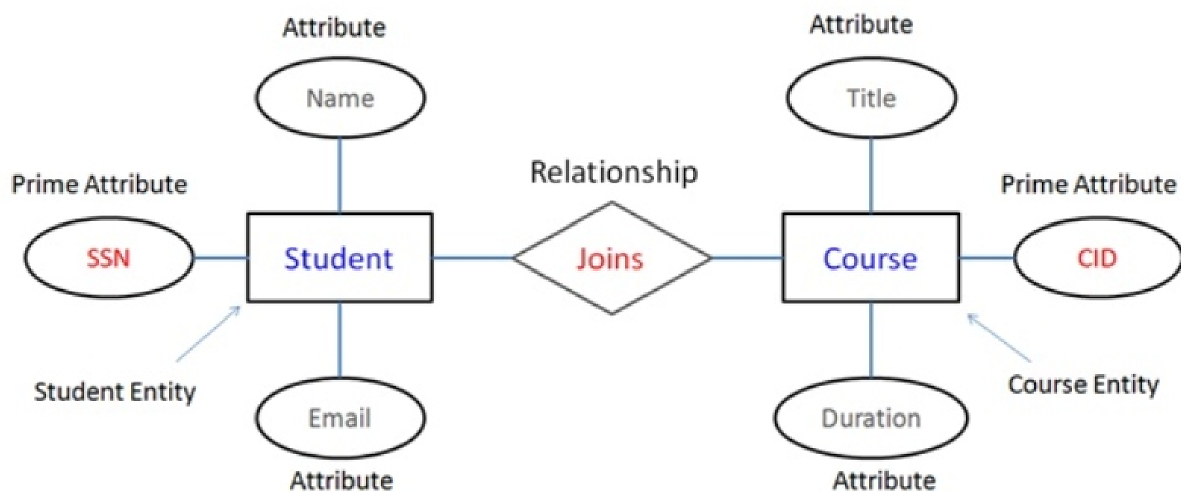


Fig. 2.1: Entity relationship diagrams (ERDs)

Why Use ERDs?

Imagine building a house without a blueprint. Data modeling with ERDs is like creating a blueprint for your database. It ensures:

- **Organized Data Structure:** ERDs define a clear structure for your data, minimizing redundancy and promoting data integrity.
- **Efficient Database Design:** By understanding relationships between entities, you can design an efficient database that optimizes storage and retrieval of information.
- **Improved Communication:** ERDs act as a common language between analysts, designers, and developers, fostering clear communication about data requirements.

Creating Effective ERDs:

This sub-unit will guide you through the steps of constructing ERDs using standardized symbols and notation. You'll learn how to:

- **Identify Entities and Attributes:** Recognize the data elements you need to store and their defining characteristics.
- **Define Relationships:** Specify the connections between entities, including cardinality (number of occurrences) and optionality (whether a relationship is mandatory).
- **Normalize Your Data:** Refine your ERD to minimize redundancy and improve data integrity – a crucial step for efficient database design.

Skill India
कौशल भारत-कुशल भारत

MINISTRY OF SKILL DEVELOPMENT
& ENTREPRENEURSHIP
GOVERNMENT OF INDIA

N·S·D·C
RE IMAGINE FUTURE

India skills

IT - ITeS SSC
nasscom

3. System Design

Unit 3.1: Object-Oriented Design Principles

Unit 3.2: Database Design

Unit 3.3: UI/UX Design Principles

Unit 3.4: Security and Access Control Design

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain the principles of object-oriented design.
2. Design relational or object-oriented databases for software applications.
3. Apply UI/UX design principles to create user-friendly interfaces.
4. Develop security and access control mechanisms for software systems.

Unit 3.1: Object-Oriented Design Principles

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain the core principles of object-oriented design (encapsulation, inheritance, polymorphism).
2. Describe the benefits of using object-oriented design for software development.

3.1.1 Object-Oriented Design Principles

What is System Design?

Systems Design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. It involves translating user requirements into a detailed blueprint that guides the implementation phase. The goal is to create a well-organized and efficient structure that meets the intended purpose while considering factors like scalability, maintainability, and performance.

Object-Oriented Design

Object-oriented design (OOD) is a structured approach to building software using reusable components. The fundamental principles to form the foundation of OOD are: encapsulation, inheritance, and polymorphism. With these principles, you can create well-organized, maintainable, and efficient software systems.

1. Encapsulation:

- **Explanation:** Encapsulation is the principle of bundling data (attributes) and methods (functions) that operate on the data into a single unit, i.e., an object. It hides the internal state of an object from the outside world and only exposes the necessary interfaces for interacting with the object.
- **Benefits:**
  - o **Modularity:** Encapsulation promotes modularity by allowing objects to be treated as black boxes, enabling independent development, testing, and maintenance.
  - o **Information Hiding:** Encapsulation hides the internal details of an object's implementation, which reduces complexity and prevents unintended access or modification of data.

2. Inheritance:

- **Explanation:** Inheritance is the principle of defining new classes based on existing classes, known as parent or superclass, to inherit their attributes and methods. It allows the creation of a hierarchy of classes where subclasses (child classes) inherit and extend the functionality of their parent classes.
- **Benefits:**
  - o **Code Reusability:** Inheritance promotes code reuse by allowing subclasses to inherit common behavior and attributes from their parent classes, reducing redundant code.
  - o **Modifiability:** Changes made to the superclass propagate to all its subclasses, providing a centralized location for modifications and updates.

3. Polymorphism:

- **Explanation:** Polymorphism is the principle that allows objects of different classes to be treated as objects of a common superclass. It enables objects to exhibit different behaviors based on their specific types or classes, even when accessed through a common interface.
- **Benefits:**
  - o **Flexibility:** Polymorphism allows for flexibility and extensibility in software design by enabling dynamic method invocation based on object types, which facilitates code reuse and adaptability.

- o **Simplicity:** Polymorphism simplifies code maintenance and enhances readability by promoting a more modular and generic design, where clients interact with objects through common interfaces without needing to know their specific implementations.

Benefits of using object-oriented design for software development:

1. **Modularity:** OOD promotes modularity by organizing code into self-contained objects, which improves code organization, maintenance, and reuse.
2. **Abstraction:** OOD allows developers to abstract complex systems into manageable components, making it easier to understand and reason about software systems.
3. **Encapsulation:** Encapsulation hides the internal details of objects, providing a clear separation between the interface and implementation, which enhances security and reduces complexity.
4. **Inheritance:** Inheritance facilitates code reuse and promotes extensibility by allowing subclasses to inherit and extend the functionality of their parent classes.
5. **Polymorphism:** Polymorphism enhances flexibility and adaptability by enabling objects to exhibit different behaviors based on their specific types, promoting code reuse and simplifying design.

3.1.2 Importance of Object-Oriented Design

Object-Oriented Design (OOD) is a practical approach that offers significant advantages for software development. The benefits of using OOD principles, highlighting how they can lead to the creation of robust, maintainable, and efficient software systems are as below:

1. Enhanced Reusability:

- **Reduced Development Time and Effort:** By leveraging inheritance and well-designed classes, developers can reuse existing code components. This eliminates the need to rewrite common functionalities, saving time and resources. Imagine building with pre-fabricated modules instead of starting from scratch!
- **Improved Code Consistency:** Reusing code components promotes consistency throughout the system. This reduces the risk of errors and inconsistencies that can arise from duplicating code in multiple places.

2. Increased Maintainability:

- **Modular Design:** Encapsulation promotes modularity, where objects become self-contained units. This makes code easier to understand, modify, and debug. Imagine working on a specific section of a house's blueprint, knowing it has minimal impact on other sections.
- **Reduced Complexity:** Breaking down the system into smaller, manageable objects reduces overall complexity. This makes it easier for developers to understand how different parts of the system interact and identify the root cause of issues.

3. Greater Flexibility and Adaptability:

- **Polymorphism in Action:** Polymorphism allows for adaptable systems that can respond to changing requirements more effectively. Imagine having a tool that can be used for various tasks depending on the situation. Similarly, objects can exhibit different behaviors based on the context.
- **Open for Extension, Closed for Modification:** The core functionalities of a class are protected through encapsulation (closed for modification). However, inheritance allows for extending functionalities through subclasses (open for extension). This promotes flexibility while maintaining stability of the core system.

4. Improved Modeling of the Real World:

- **Intuitive Design:** Object-oriented design reflects real-world entities and their interactions. This leads to a more natural and intuitive code structure that is easier to understand and maintain. Imagine a system modeling customers and orders, mirroring real-world interactions.
- **Clearer Communication:** Using objects and their relationships as building blocks fosters better communication between developers, analysts, and stakeholders. Everyone has a shared understanding of the system based on real-world concepts.

Unit 3.2: Database Design

Unit Objectives

By the end of this unit, the participants will be able to:

1. Compare and contrast relational and object-oriented database models.
2. Apply normalization techniques to design efficient and effective database structures.
3. Explain the importance of data integrity in database design.

3.2.1 Database Design

There are two primary database models: relational and object-oriented. Their structures, strengths, and weaknesses, enabling you to make informed decisions when designing your database.

Fig. 3.1: Database design

Relational and Object-Oriented Database Models:

- **Relational Database Model:**
  - o **Structure:** Relational databases organize data into tables, where each table consists of rows and columns. Tables are related through primary and foreign keys.
  - o **Data Representation:** Data is represented in a tabular format, with each row representing a record and each column representing a data attribute.
  - o **Query Language:** Relational databases use SQL (Structured Query Language) for querying and manipulating data.
  - o **Example:** MySQL, PostgreSQL, Oracle.
- **Object-Oriented Database Model:**
  - o **Structure:** Object-oriented databases store data as objects, which encapsulate both data and behavior. Objects are organized into classes, similar to object-oriented programming languages.
  - o **Data Representation:** Data is represented as objects with attributes and methods, allowing for complex data structures and relationships.
  - o **Query Language:** Object-oriented databases typically use object query languages (OQL) or extensions to SQL.
  - o **Example:** db4o, ObjectDB.

Comparison between the two models:

The optimal model depends on the specific needs of your application. Here are some key considerations:

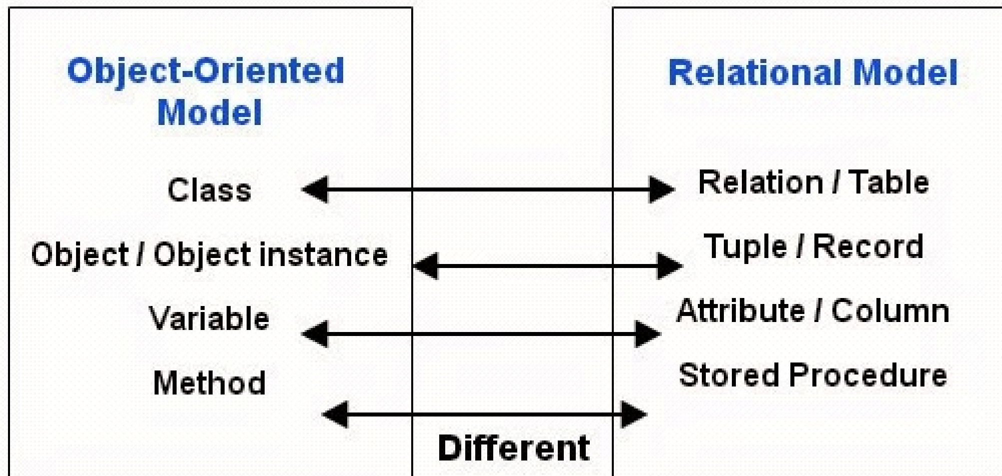


Fig. 3.2: Difference between object-oriented and relational database models

- **Data Structure:** If your data is well-defined and has simple relationships, a relational model might be sufficient. However, for complex object hierarchies and behavior, an object-oriented model might be a better choice.
- **Application Logic:** If your application heavily relies on object-oriented programming, an object-oriented database can offer a smoother development experience.
- **Performance Requirements:** For complex queries on large datasets, a well-normalized relational database might offer better performance.
- **Developer Expertise:** Consider the availability of developers skilled in working with each model.

3.2.2 Building Strong Foundations with Normalization

Normalization is the process of organizing your database tables to minimize data redundancy, improve data integrity, and enhance database efficiency. By applying normalization principles, you'll be laying a solid foundation for a robust and maintainable database.

Understanding Data Redundancy:

Data redundancy occurs when the same data element is stored in multiple locations within the database. While seemingly harmless at first, redundancy can lead to several issues:

- **Wasted Storage Space:** Duplicate data consumes unnecessary storage, impacting database size and potentially increasing costs.
- **Data Inconsistency:** If the same data needs to be updated in multiple places, there's a risk of inconsistencies arising if the updates are not performed simultaneously.
- **Maintenance Headaches:** Maintaining consistency across redundant data becomes tedious and error-prone, especially as the database grows.

Normalization Forms:

Normalization follows a series of progressive stages, each reducing redundancy to a greater degree. Here are the key normalization forms you'll encounter:

- **First Normal Form (1NF):** Eliminates repeating groups within a table. Every data element should be atomic (indivisible) and uniquely identifiable.
- **Second Normal Form (2NF):** Ensures all non-key attributes are fully dependent on the primary key, eliminating redundancy based on partial key dependencies.

- **Third Normal Form (3NF):** Removes transitive dependencies, meaning no non-key attribute depends on another non-key attribute – only the primary key.

Benefits of Normalization:

By adhering to normalization principles, you'll reap several benefits:

- **Reduced Data Redundancy:** Minimizes wasted storage space and improves overall database efficiency.
- **Enhanced Data Integrity:** Ensures data consistency by eliminating the need for updates in multiple locations.
- **Improved Query Performance:** Streamlines data retrieval by reducing complex joins and unnecessary data access.
- **Simplified Database Maintenance:** Makes database maintenance and modification easier by promoting a well-organized structure.

Applying Normalization Techniques:

This include practical steps for analyzing your database schema and applying normalization techniques to identify redundancy, decompose tables, and establish relationships using foreign keys. Through hands-on exercises, you'll gain the skills to confidently normalize your database for optimal performance and maintainability.

3.2.3 Data Integrity

Data integrity refers to the accuracy, consistency, and completeness of your data. It ensures that the information stored in your database is reliable and trustworthy, forming the foundation for sound decision-making, efficient operations, and regulatory compliance.

The Price of Compromised Integrity:

Failing to maintain data integrity can have far-reaching consequences. Here's why it matters:

- **Flawed Decisions:** Inaccurate or inconsistent data can lead to misleading insights and poor business decisions with significant financial or operational repercussions.
- **Compliance Issues:** Many regulations mandate data accuracy and security. Failing to adhere to these regulations can result in hefty fines and reputational damage.
- **Security Risks:** Inaccurate or incomplete data can weaken security measures, making your database vulnerable to attacks that exploit inconsistencies or manipulate data for malicious purposes.
- **Wasted Resources:** Time and money are wasted investigating and correcting data errors, hindering productivity and diverting resources from core activities.

Building a Fortress for Your Data:

Several strategies contribute to maintaining data integrity throughout your database's lifecycle:

- **Data Types and Constraints:** Define data types for each attribute (e.g., number, date) to restrict invalid entries. Implement constraints like primary keys, foreign keys, and check constraints to enforce data consistency and validity. This ensures data adheres to specific rules and relationships within your database schema.
- **Data Validation:** Integrate data validation rules at the point of data entry, preventing errors from entering the database in the first place. This can be achieved through user interface validation or programmatic checks within your application.
- **User Permissions:** Implement robust user access controls. Grant users only the minimum privileges necessary for their tasks, restricting unauthorized data modification, deletion, or manipulation. This minimizes the risk of accidental or intentional data corruption.

- **Data Backup and Recovery:** Regularly back up your database to a secure location. This ensures a reliable copy exists in case of data corruption or accidental deletion, allowing you to restore the database to a known good state.

Tools for Maintaining Integrity:

Modern database management systems (DBMS) offer a variety of tools to safeguard data integrity:

- **Data Dictionaries:** Store metadata about your database schema, including data types, constraints, and relationships between tables. This centralized repository provides a clear view of your data structure and the rules governing it.
- **Triggers:** Create automated routines that execute specific actions when certain events occur in the database, such as data insertion or update. These triggers can enforce integrity rules, like preventing invalid data from being saved or automatically updating related records to maintain consistency.
- **Auditing:** Enable auditing mechanisms within your DBMS to track user activity and data modifications. This detailed record allows you to identify potential integrity issues, investigate suspicious activity, and ensure accountability.

Unit 3.3: UI/UX Design Principles

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain the importance of user interface (UI) and user experience (UX) design in software development.
2. Implement UI/UX design principles to create user-friendly and intuitive interfaces.

3.3.1 Importance of UI/UX Design

User Interface (UI) and User Experience (UX) design play crucial roles in software development, as they directly impact how users interact with and perceive a software application.

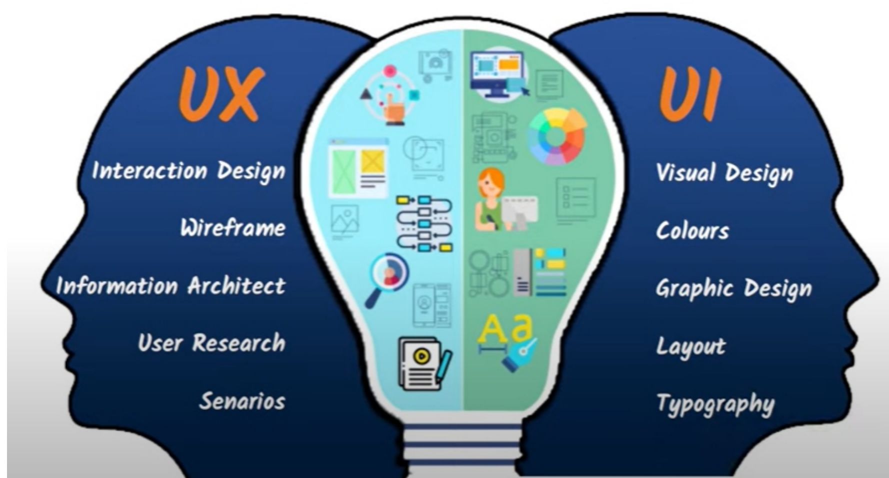


Fig. 3.2: UX vs UI

1. Enhanced User Satisfaction:

- UI/UX design focuses on creating interfaces that are intuitive, easy to use, and visually appealing. By providing a positive and enjoyable user experience, it enhances user satisfaction and encourages continued usage of the software.

2. Increased Usability:

- UI/UX design aims to streamline the user's interaction with the software by optimizing workflows, reducing cognitive load, and minimizing the learning curve. This leads to increased usability and efficiency in completing tasks, resulting in higher user productivity.

3. Improved Accessibility:

- UI/UX design considers the needs of diverse users, including those with disabilities or special requirements. By implementing accessibility features such as screen reader compatibility, keyboard navigation, and adjustable text sizes, it ensures that the software is accessible to all users.

4. Effective Communication:

- UI/UX design involves clear and concise communication of information through visual elements, text, and interactive components. By presenting information in a well-organized and understandable manner, it helps users comprehend the functionality of the software and achieve their goals effectively.

5. **Brand Image and Recognition:**
   - UI/UX design contributes to shaping the brand identity and image of the software. A visually appealing and consistent design reflects positively on the brand, instilling trust and confidence in users. Consistent branding elements also facilitate brand recognition and loyalty.

6. **Reduced Errors and Support Costs:**
   - Well-designed UI/UX helps in preventing user errors and reducing the need for user support and assistance. By guiding users through clear visual cues, informative feedback, and error prevention techniques, it minimizes user frustration and support costs.

7. **Competitive Advantage:**
   - In today's competitive market, UI/UX design can be a key differentiator for software products. A superior user experience sets apart a product from its competitors, attracting and retaining users who value usability, efficiency, and enjoyment.

3.3.2 Implementing UI/UX design principles

Implementing UI/UX design principles is essential for creating user-friendly and intuitive interfaces that provide a positive user experience.

1. **User-Centered Design:**
   - Focus on understanding the needs, goals, and behaviors of the target users throughout the design process.
   - Involve users in the design process through user research, feedback sessions, and usability testing.
   - Design interfaces that prioritize user needs and preferences, ensuring that users can easily accomplish their tasks.

2. **Consistency:**
   - Maintain consistency in visual elements, layout, navigation, and interaction patterns throughout the interface.
   - Use consistent terminology, icons, colors, and styles to create a cohesive and predictable user experience.
   - Consistency reduces cognitive load, improves learnability, and enhances usability by making the interface more intuitive and familiar to users.

3. **Simplicity:**
   - Keep the interface simple and uncluttered, avoiding unnecessary elements or features that can overwhelm users.
   - Prioritize essential content and functionality, and progressively disclose additional information or options as needed.
   - Simplify complex tasks by breaking them down into smaller, manageable steps, and provide clear guidance and feedback to users.

4. **Visual Hierarchy:**
   - Use visual hierarchy to prioritize content and guide users' attention to the most important elements and actions.
   - Use techniques such as size, color, contrast, and typography to create a clear and structured layout.
   - Emphasize key elements such as headings, buttons, and calls-to-action to help users quickly identify and understand their significance.

5. Feedback and Responsiveness:

- Provide immediate and clear feedback to users for their actions, such as button clicks, form submissions, or errors.
- Use visual cues, animations, and feedback messages to indicate the status of tasks, confirmations, or errors.
- Design interfaces that are responsive and adaptive to different devices, screen sizes, and input methods, ensuring a consistent experience across platforms.

6. Accessibility:

- Design interfaces that are accessible to users with disabilities or special needs, such as screen readers, keyboard navigation, and alternative text for images.
- Ensure sufficient color contrast, readable text sizes, and sufficient interactive elements for users with visual impairments or motor disabilities.
- Test the interface with assistive technologies and follow accessibility guidelines and standards to ensure compliance.

7. Emotional Design:

- Consider the emotional impact of the interface on users and aim to evoke positive emotions such as delight, satisfaction, and trust.
- Use visual aesthetics, animations, micro-interactions, and storytelling elements to create a memorable and enjoyable user experience.
- Design interfaces that reflect the brand personality and values, building a strong emotional connection with users.

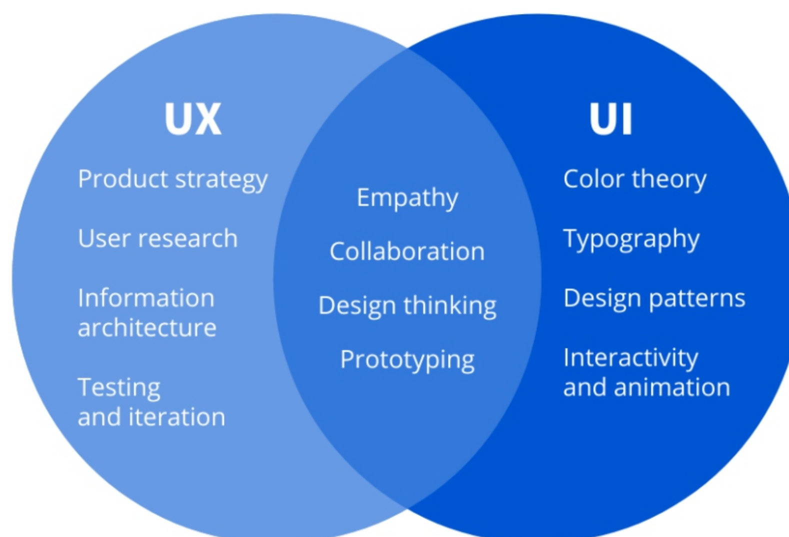


Fig. 3.3: Principles of UI/UX design

Unit 3.4: Security and Access Control Design

Unit Objectives

By the end of this unit, the participants will be able to:

1. Identify different security threats and vulnerabilities in software systems.
2. Design security mechanisms (authentication, authorization) to control access to system resources.

3.4.1 Security, threats and vulnerabilities in Software Ecosystem

Software is the backbone of countless systems and applications. This reliance on software introduces a critical aspect: security. This includes ever-present threats and vulnerabilities lurking within the software ecosystem. We'll explore the various ways malicious actors can exploit weaknesses, the common types of attacks they employ, and the potential consequences of security breaches:

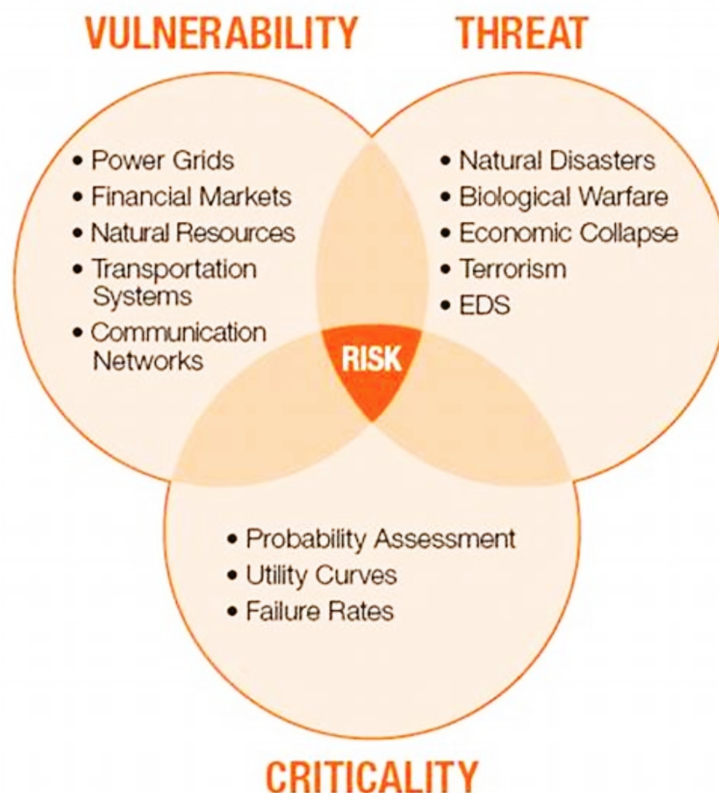


Fig. 3.4: Security, threats and vulnerabilities

i. Malware:

- **Threat:** Malicious software, including viruses, worms, Trojans, ransomware, and spyware, that infects systems and compromises their security.
- **Vulnerabilities:** Vulnerabilities in operating systems, applications, or network protocols that allow malware to exploit weaknesses and gain unauthorized access to systems.

ii. Phishing:

- **Threat:** Social engineering attacks that trick users into revealing sensitive information, such as usernames, passwords, or financial details, by impersonating legitimate entities.

- **Vulnerabilities:** Lack of user awareness and education, insecure communication channels (e.g., email), and spoofed websites that mimic trusted organizations.

iii. SQL Injection:

- **Threat:** Injection attacks that exploit vulnerabilities in web applications to execute malicious SQL queries against a database.
- **Vulnerabilities:** Insecure coding practices, such as concatenating user input directly into SQL queries without proper validation or parameterization, leading to SQL injection vulnerabilities.

iv. Cross-Site Scripting (XSS):

- **Threat:** Attacks that inject malicious scripts into web pages viewed by other users, allowing attackers to steal session cookies, hijack user sessions, or perform unauthorized actions on behalf of users.
- **Vulnerabilities:** Insecure handling of user input in web applications, such as failing to properly escape or sanitize user-generated content before rendering it in HTML pages.

v. Unauthorized Access:

- **Threat:** Attempts by unauthorized users to gain access to protected resources, systems, or data.
- **Vulnerabilities:** Weak or default passwords, insufficient authentication mechanisms, misconfigured access controls, or improper handling of session management and authentication tokens.

vi. Denial of Service (DoS) and Distributed Denial of Service (DDoS) Attacks:

- **Threat:** Attacks that overwhelm servers, networks, or applications with a high volume of malicious traffic, causing service disruption or downtime.
- **Vulnerabilities:** Lack of sufficient network bandwidth, inadequate resource management, or vulnerabilities in network protocols or application-layer services that can be exploited to launch DoS/DDoS attacks.

vii. Data Breaches:

- **Threat:** Unauthorized access to sensitive data, such as personal information, financial records, or intellectual property, resulting in data theft or exposure.
- **Vulnerabilities:** Inadequate data encryption, weak access controls, insecure storage or transmission of data, or insider threats from malicious or negligent employees.

viii. Insider Threats:

- **Threat:** Malicious actions or misconduct by employees, contractors, or trusted individuals with insider access to systems and data.
- **Vulnerabilities:** Insufficient user monitoring and auditing, lack of employee training and awareness, or inadequate segregation of duties and access controls.

3.4.2 Access Control Mechanisms for system

The software ecosystem thrives on data and functionality, but this valuable resource requires protection. This sub-unit explores the concept of access control, a cornerstone of system security. We'll delve into the mechanisms that ensure only authorized users can access specific resources, mitigating the risks posed by ever-present security threats and vulnerabilities.

The CIA Triad: A Foundation for Secure Access

Our journey begins with the CIA triad, a fundamental security principle encompassing three core objectives:

- **Confidentiality:** Ensuring only authorized users can access sensitive data. Imagine a locked safe; access control mechanisms act as the combination that safeguards confidential information.

- **Integrity:** Guaranteeing the accuracy and trustworthiness of data. Access control prevents unauthorized modifications, ensuring data remains unaltered and reliable.
- **Availability:** Maintaining system uptime and ensuring authorized users can access resources when needed. Access control mechanisms are designed to prevent denial-of-service attacks that could disrupt system availability.

Authentication: The First Line of Defense

Authentication verifies a user's claimed identity before granting access. It's like checking someone's ID at the entrance. Common authentication methods include:

- **Passwords:** The traditional approach, requiring users to enter a secret combination of characters.
- **Multi-Factor Authentication (MFA):** Adds an extra layer of security by requiring a secondary verification factor, such as a code from a mobile device, alongside a password.
- **Biometric Authentication:** Utilizes unique physical characteristics like fingerprints or facial recognition for even stronger verification.

Authorization: Granting Permissions Wisely

Once a user is authenticated, authorization determines what actions they can perform and what data they can access. Imagine granting different levels of access within a library; some users can only borrow books, while others can access the restricted archives. Here's how authorization is often implemented:

- **Access Control Lists (ACLs):** Explicitly define which users or groups have permissions to access specific resources.
- **Role-Based Access Control (RBAC):** Assigns permissions based on user roles (e.g., administrator, editor, viewer). Users inherit the permissions associated with their assigned roles.

Secure Coding Practices: Building a Strong Foundation

While access control mechanisms are crucial, security starts at the development stage. Here's how secure coding practices minimize vulnerabilities that could be exploited to bypass access controls:

- **Input Validation:** Thoroughly validate user input to prevent malicious code injection attempts.
- **Secure Data Storage:** Store sensitive data using encryption techniques to render it unreadable in case of a breach.
- **Regular Security Testing:** Proactively identify and address vulnerabilities through penetration testing and vulnerability assessments.

Skill India
कौशल भारत-कुशल भारत
MINISTRY OF SKILL DEVELOPMENT & ENTREPRENEURSHIP
GOVERNMENT OF INDIA
N·S·D·C
RE IMAGINE FUTURE
India skills

IT - ITeS SSC
nasscom

4. Software Development

Unit 4.1: Introduction to Programming Languages and Development Environments

Unit 4.2: Coding Techniques and Best Practices

Unit 4.3: Building Advanced Software Applications

Unit 4.4: Developing Web and Mobile User Interfaces

Key Learning Outcomes

At the end of this module, you will be able to:

1. Identify and utilize appropriate programming languages and development environments.
2. Implement coding techniques and best practices to write clean and maintainable code.
3. Evaluate and integrate relevant libraries and frameworks into software projects.
4. Construct multi-tier architecture for complex software applications.
5. Develop web and mobile interfaces for client-server based systems.

Unit 4.1: Introduction to Programming Languages and Development Environments

Unit Objectives

By the end of this unit, the participants will be able to:

1. Identify common programming languages used for developing software applications.
2. Select appropriate development environments based on programming language and project requirements.
3. Explain the concept of Integrated Development Environments (IDEs).

4.1.1 Introduction to Programming Languages and Development Environments

I. Common Programming Languages:

1. **Java:** Known for its platform independence, Java is widely used in enterprise-level applications, web development, and Android app development.
2. **Python:** Renowned for its simplicity and versatility, Python is utilized in web development, data science, artificial intelligence, scientific computing, and automation scripting.
3. **JavaScript:** Primarily used for front-end web development, JavaScript has gained popularity in recent years with the advent of frameworks like React and Angular for building dynamic and interactive web applications.
4. **C/C++:** These languages are powerful and widely used for system programming, game development, embedded systems, and performance-critical applications.
5. **C#:** Developed by Microsoft, C# is commonly used for building Windows desktop applications, web applications (via ASP.NET), and games using the Unity game engine.
6. **Swift:** Developed by Apple, Swift is used for building iOS, macOS, watchOS, and tvOS applications.
7. **Ruby:** Known for its simplicity and productivity, Ruby is often used in web development, particularly with the Ruby on Rails framework.

II. Development Environments:

1. **Visual Studio:** A popular IDE developed by Microsoft, Visual Studio supports various programming languages like C#, C++, Python, and JavaScript. It provides advanced debugging tools, code navigation, and integrated testing capabilities.
2. **Eclipse:** Eclipse is an open-source IDE primarily used for Java development but supports other languages through plugins. It offers features like syntax highlighting, code completion, and version control integration.
3. **PyCharm:** Specifically designed for Python development, PyCharm offers intelligent code completion, debugging, and version control integration tailored to Python developers.
4. **IntelliJ IDEA**: Similar to PyCharm, IntelliJ IDEA is a powerful IDE that supports multiple programming languages, including Java, Kotlin, and Scala. It provides features like refactoring, code analysis, and integration with build tools.
5. **Visual Studio Code (VS Code):** A lightweight, open-source IDE developed by Microsoft, VS Code supports a wide range of programming languages through extensions. It offers features like debugging, Git integration, and customizable user interface.

Integrated Development Environments (IDEs):

Fig. 4.1: Integrated development environments (IDEs)

An Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. It typically includes a source code editor, build automation tools, and a debugger. IDEs aim to streamline the development process by providing features such as syntax highlighting, code completion, and project management tools within a unified interface. IDEs enhance developer productivity and help maintain code quality through features like code refactoring and version control integration. Examples of popular IDEs include Visual Studio, Eclipse, and IntelliJ IDEA.

Unit 4.2: Coding Techniques and Best Practices

Unit Objectives

By the end of this unit, the participants will be able to:

1. Implement proper coding techniques (e.g., indentation, naming conventions, commenting) to write clean and maintainable code.
2. Explain the importance of code reusability and modularity.
3. Discuss the concept of code refactoring and its benefits.
4. Introduce basic version control concepts (e.g., Git) for managing code changes.

4.2.1 Coding Techniques and Best Practices

Implement proper coding techniques:

1. **Indentation:** Consistent indentation improves code readability, making it easier to understand the structure of the code. Indentation also helps identify code blocks and nesting levels.
2. **Naming conventions:** Descriptive and meaningful names for variables, functions, and classes enhance code readability and maintainability. Following naming conventions, such as camelCase or snake_case, helps maintain consistency across the codebase.
3. **Commenting:** Adding comments to explain complex logic, algorithmic approaches, or important decisions in the code enhances understandability for other developers. Comments should be clear, concise, and updated alongside code changes.

Explain the importance of code reusability and modularity:

1. **Code reusability:** Writing reusable code components allows developers to efficiently leverage existing solutions rather than reinventing the wheel. This reduces development time, minimizes code duplication, and improves maintainability.
2. **Modularity:** Breaking down a software system into modular components with well-defined interfaces enables easier development, testing, and maintenance. Modularity promotes code organization, facilitates collaboration among team members, and enhances scalability.

Discuss the concept of code refactoring and its benefits:

1. **Code refactoring:** Refactoring involves restructuring existing code to improve its design, readability, and maintainability without altering its external behavior. This process may include renaming variables, extracting reusable functions, optimizing performance, or simplifying complex logic.
2. **Benefits of code refactoring:**
   - Enhances code readability and understandability.
   - Improves code maintainability by reducing technical debt.
   - Increases software quality and reduces the likelihood of bugs.
   - Facilitates easier code reviews and collaboration among team members.
   - Supports future enhancements and scalability of the codebase.

Introduce basic version control concepts (e.g., Git) for managing code changes:

1. **Version control:** Version control systems like Git track changes to source code files over time, allowing developers to collaborate effectively, maintain a history of changes, and revert to previous versions if necessary.

2. **Key concepts of Git:**
   - **Repositories:** Stores the entire history of a project, including all file versions and branches.

- **Commits:** Snapshot of changes made to the codebase at a specific point in time, accompanied by a commit message describing the changes.
- **Branches:** Independent lines of development that allow for parallel work on different features or bug fixes.
- **Merge:** Combines changes from one branch into another, integrating new features or bug fixes into the main codebase.
- **Pull requests:** Proposed changes submitted by developers for review and integration into the main codebase.
- **Conflict resolution:** Handling conflicts that arise when two or more developers make changes to the same part of a file concurrently.

Unit 4.3: Building Advanced Software Applications

Unit Objectives

By the end of this unit, the participants will be able to:

1. Evaluate the benefits of using libraries and frameworks in software development.
2. Select appropriate libraries and frameworks based on project requirements.
3. Integrate relevant libraries and frameworks to enhance application functionality and efficiency.
4. Explain the concept of multi-tier architecture and its advantages.
5. Design and develop software applications using a multi-tier approach.

4.3.1 Evaluating and selecting appropriate libraries and frameworks

These are the benefits of using libraries and frameworks in software development:

i. **Time-saving:** Libraries and frameworks provide pre-written code components and functionalities, allowing developers to leverage existing solutions rather than building everything from scratch. This significantly reduces development time and effort.

ii. **Code reuse:** Libraries and frameworks promote code reuse by encapsulating common functionalities and best practices. Developers can easily integrate these components into their projects, minimizing code duplication and improving maintainability.

iii. **Consistency:** Using established libraries and frameworks helps maintain consistency across the codebase. They often follow standardized coding conventions and design patterns, ensuring a uniform structure and style throughout the application.

iv. **Scalability:** Libraries and frameworks are designed to handle various scalability challenges, such as managing large datasets, handling concurrent requests, and optimizing performance. They provide scalable solutions and architectural patterns that support the growth of the application over time.

v. **Community support:** Popular libraries and frameworks typically have vibrant developer communities. This community support includes documentation, forums, tutorials, and third-party plugins/extensions, which help developers troubleshoot issues, share knowledge, and stay updated with best practices.

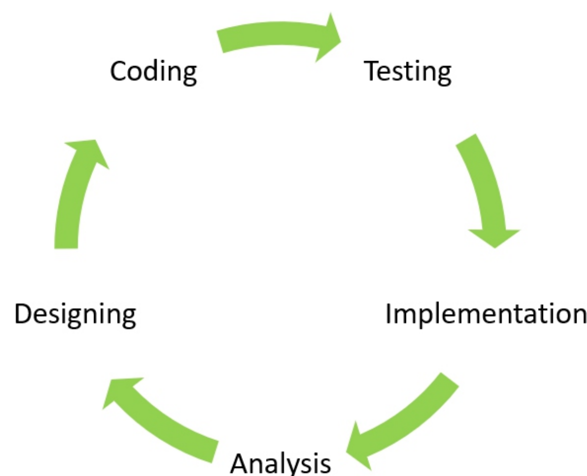


Fig. 4.2: Frameworks in software development

Select appropriate libraries and frameworks based on project requirements:

i. **Compatibility:** Choose libraries and frameworks that are compatible with the programming language and technology stack used in the project. Ensure that they integrate seamlessly with existing components and tools.

ii. **Feature set:** Evaluate the features and functionalities offered by libraries and frameworks to ensure they align with the project requirements. Consider factors such as data processing capabilities, security features, performance optimizations, and support for specific use cases.

iii. **Community adoption:** Assess the popularity and adoption rate of libraries and frameworks within the developer community. Widely-used and well-maintained libraries are more likely to have robust documentation, regular updates, and community support.

iv. **Performance:** Measure the performance impact of integrating a library or framework into the project. Evaluate factors such as resource utilization, execution speed, and scalability to ensure that the chosen solution meets performance requirements.

v. **Flexibility:** Choose libraries and frameworks that offer flexibility and customization options. Look for extensibility features, plugin architectures, and configurable settings that allow developers to tailor the solution to fit the specific needs of the project.

4.3.2 Multi-tier and develop software app using a multi-tier approach

Multi-tier Architecture:

Multi-tier architecture, also known as n-tier architecture, is a software architecture pattern that divides an application into multiple logical layers or tiers, each responsible for specific functionality. These tiers are typically categorized as presentation tier, application tier (also known as business logic or middle tier), and data tier (also known as data access or persistence tier).

1. **Presentation Tier (Client Tier):** This tier interacts directly with users and handles the presentation logic of the application. It includes components such as user interfaces, web browsers, mobile apps, and other client-side technologies. The presentation tier is responsible for rendering the user interface, processing user input, and displaying information to users.
2. **Application Tier (Middle Tier):** The application tier contains the business logic and processing logic of the application. It handles tasks such as data validation, business rules enforcement, workflow management, and application logic execution. This tier is often implemented using server-side technologies such as web servers, application servers, and microservices.
3. **Data Tier (Data Persistence Tier):** The data tier is responsible for managing data storage and retrieval. It includes components such as databases, file systems, data warehouses, and external data sources. The data tier provides mechanisms for storing, querying, and manipulating data, ensuring data integrity, and managing transactions.

Advantages of Multi-tier Architecture:

1. **Scalability:** Multi-tier architecture allows for horizontal scalability by distributing the application across multiple servers or nodes. Each tier can be scaled independently based on demand, improving performance and accommodating increasing user loads.
2. **Modularity:** The separation of concerns into distinct tiers promotes modularity and encapsulation. Each tier can be developed, deployed, and maintained independently, facilitating easier updates, enhancements, and troubleshooting.

3. **Flexibility:** Multi-tier architecture enables flexibility in technology selection and deployment. Different tiers can use different technologies and platforms based on their specific requirements, allowing for the use of best-of-breed solutions and integration with legacy systems.
4. **Security:** By enforcing clear boundaries between tiers and implementing security measures at each level, multi-tier architecture enhances security and mitigates risks. Access controls, encryption, authentication, and authorization mechanisms can be implemented at appropriate tiers to protect sensitive data and resources.
5. **Maintainability:** The modular structure of multi-tier architecture improves maintainability by isolating changes and reducing the impact of modifications. Developers can update or replace individual tiers without affecting the entire application, making it easier to evolve and adapt the software over time.

Now, let's discuss how to design and develop software applications using a multi-tier approach:

Design and Development of Software Applications using a Multi-tier Approach:

1. **Identify Functional Requirements:** Analyze the functional requirements of the application to determine the tasks and responsibilities of each tier. Define the interactions and interfaces between tiers to ensure clear communication and data flow.
2. **Choose Technologies and Platforms:** Select appropriate technologies and platforms for each tier based on the project requirements, scalability goals, performance considerations, and team expertise. Consider factors such as programming languages, frameworks, databases, and deployment options.
3. **Define Communication Protocols:** Establish communication protocols and APIs (Application Programming Interfaces) between tiers to facilitate interaction and data exchange. Use standardized protocols such as HTTP, RESTful APIs, SOAP, or messaging queues to ensure interoperability and compatibility.
4. **Implement Business Logic:** Develop the business logic and application logic in the middle tier using appropriate programming languages, frameworks, and design patterns. Implement validation rules, business rules, workflows, and other processing logic to fulfill the functional requirements of the application.
5. **Manage Data Access:** Design and implement data access and persistence mechanisms in the data tier using databases, ORMs (Object-Relational Mapping), or other data storage solutions. Define data models, schema designs, indexing strategies, and data access patterns to optimize performance and ensure data integrity.
6. **Develop User Interfaces:** Create user interfaces and presentation layers in the presentation tier using web technologies, UI frameworks, or native UI libraries. Design intuitive interfaces, responsive layouts, and interactive components to provide a seamless user experience across different devices and platforms.
7. **Testing and Quality Assurance:** Conduct thorough testing and quality assurance activities to validate the functionality, performance, security, and usability of the multi-tier application. Perform unit testing, integration testing, system testing, and acceptance testing to identify and fix defects early in the development lifecycle.
8. **Deployment and Maintenance:** Deploy the multi-tier application to production environments using deployment automation tools, containerization platforms, or cloud services. Monitor application performance, availability, and security post-deployment, and apply patches, updates, and optimizations as needed to ensure ongoing reliability and scalability.

Unit 4.4: Developing Web and Mobile User Interfaces

Unit Objectives

By the end of this unit, the participants will be able to:

1. Distinguish between web-based and mobile application development.
2. Identify the key considerations for designing user interfaces for web and mobile applications.
3. Utilize appropriate technologies and tools to build user interfaces for web and mobile applications.
4. Implement user interaction elements and functionalities in web and mobile applications.

4.4.1 Difference between Developing Web and Mobile Interfaces

Basis	Mobile App Development	Web Development
Definition	Mobile Apps are software programs that run on smartphones and tablets.	Web applications, on the other hand, are accessed through a web browser and are very adaptable to any device.
Build process	A developer is often hired by a company to produce a native or hybrid mobile app.	HTML5, CSS, and JavaScript may all be used together.
Functionality	When compared to full-fledged website visitors, mobile app users frequently receive restricted functionality, and many of these applications are focused on a specific aim.	Web apps provide a far broader range of functions than mobile apps.
Platform dependency	These fall in the category of Native applications that are created to work with something like a mobile device's operating system (OS).	Web apps provide a far broader range of functions than mobile apps. Adobe Photoshop, for example, provides customers with both a mobile app and a web version.
Education	A bachelor's degree in software engineering, mobile computing, computer science, mobile application development, or any programming-focused field of study.	Bachelor's degree or associate degrees in any programming-focused field of study.
Connectivity and Updates	The primary distinction between these two applications is that Mobile apps may frequently function when disconnected.	Online apps require an active Internet connection to function.
Salary	The average annual income in the US for a Mobile Application Developer is $91245 per year.	The national average compensation in the US for a Web Developer is $66593 per year.
Skills	May use the Swift programming language, Objective-C, and the XCode IDE, whereas Android developers can use Java or Kotlin with the Eclipse IDE. HTML, JavaScript, and CSS are the most often used languages for hybrid apps.	Web development languages and frameworks available to developers. HTML, JavaScript, Python, PHP, and Ruby are some of the most popular languages. Laravel, and Rails

Table 4.1: Difference between developing web and mobile interfaces

4.4.2 Considerations for designing user interfaces

1. **Platform and Device Compatibility:** Understand the differences between web and mobile platforms and their respective devices. Design interfaces that are responsive and adaptable to different screen sizes, resolutions, orientations, and input methods (touch, mouse, keyboard).
2. **User Experience (UX):** Focus on providing a seamless and intuitive user experience that guides users through the application's functionalities. Pay attention to navigation patterns, information architecture, and interaction design to minimize cognitive load and maximize usability.
3. **Visual Design:** Create visually appealing interfaces that reflect the brand identity and resonate with the target audience. Use consistent branding elements, typography, color schemes, and imagery to establish a cohesive visual language and enhance brand recognition.
4. **Accessibility:** Ensure that the interface is accessible to users with disabilities or impairments. Follow accessibility standards and guidelines (e.g., WCAG) to make content perceivable, operable, and understandable for all users. Provide alternative text for images, keyboard navigation, and support for screen readers.
5. **Performance:** Optimize performance to deliver fast and responsive user experiences. Minimize loading times, reduce server requests, and optimize assets (images, scripts, stylesheets) to enhance page load speed and responsiveness. Implement lazy loading, caching mechanisms, and asynchronous data fetching to improve performance.
6. **Content Presentation:** Present content in a clear, concise, and structured manner to facilitate readability and comprehension. Use hierarchical organization, visual hierarchy, and whitespace to prioritize content and guide users' attention to important elements. Break content into digestible chunks and use descriptive headings, bullet points, and multimedia (videos, images) to enhance engagement.
7. **Touch and Gestures:** Design for touch interactions on mobile devices by optimizing touch targets, spacing, and gesture-based interactions. Ensure that interactive elements are large enough and spaced adequately to prevent accidental taps or gestures. Provide visual feedback (e.g., animations, color changes) to confirm user actions and improve touch responsiveness.
8. **Cross-Browser and Cross-Platform Compatibility:** Test the interface across different web browsers (Chrome, Firefox, Safari, Edge, etc.) and platforms (iOS, Android, Windows, macOS) to ensure consistent behavior and appearance. Address browser-specific quirks and implement progressive enhancement and graceful degradation techniques to provide a consistent experience across platforms.
9. **Feedback and Error Handling:** Provide feedback to users for their actions and guide them through the interface with informative messages, tooltips, and notifications. Clearly communicate error messages and recovery options to help users recover from errors gracefully. Use validation indicators and inline validation to prevent user errors proactively.
10. **User Feedback and Iteration:** Gather user feedback through usability testing, surveys, and analytics to identify pain points, usability issues, and areas for improvement. Iterate on the design based on user feedback and usage data to refine the interface and enhance the overall user experience continuously.

4.4.3 Utilize appropriate technologies and implementing for web and mobile

Utilize appropriate technologies and tools to build user interfaces for web and mobile applications:

1. **Web Application Development:**
   - **HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and JavaScript:** Fundamental technologies for building web interfaces. HTML provides the structure, CSS controls the presentation and layout, and JavaScript adds interactivity and behavior to the interface.
   - **Frontend Frameworks:** Utilize frontend frameworks like React.js, Angular, or Vue.js to streamline development, manage state, and create reusable UI components.
   - **CSS Preprocessors:** Use tools like Sass or Less to enhance CSS with variables, mixins, and functions, making styling more efficient and maintainable.
   - **Responsive Design:** Employ techniques like media queries and flexbox/grid layouts to create responsive web interfaces that adapt to different screen sizes and devices.
   - **Browser Developer Tools:** Take advantage of browser developer tools (e.g., Chrome DevTools) for debugging, inspecting elements, and optimizing performance.

2. **Mobile Application Development:**
   - **Native Development:** Build mobile interfaces using platform-specific languages and SDKs, such as Java or Kotlin for Android (using Android Studio) and Swift or Objective-C for iOS (using Xcode).
   - **Cross-Platform Development:** Utilize frameworks like React Native, Flutter, or Xamarin to develop mobile applications that run on multiple platforms using a single codebase.
   - **UI Design Tools:** Use design tools like Adobe XD, Sketch, or Figma to create wireframes, mockups, and prototypes of mobile interfaces before implementation.
   - **Testing Tools:** Employ mobile testing frameworks like Appium or XCTest for iOS and Espresso or UI Automator for Android to automate UI testing and ensure app reliability across devices.

Implement user interaction elements and functionalities in web and mobile applications:

1. **Web Application Interaction Elements**:
   - o **Forms and Inputs:** Implement forms with various input types (text, email, password, etc.) and validation to collect user data.
   - o **Navigation:** Design intuitive navigation menus, breadcrumbs, and links to help users navigate through the application.
   - o **Buttons and Controls:** Use buttons, dropdowns, checkboxes, radio buttons, sliders, and other UI controls for user interaction.
   - o **Modals and Dialogs:** Incorporate modal windows and dialogs for displaying alerts, messages, and interactive content.
   - o **Animations and Transitions:** Enhance user experience with smooth animations, transitions, and visual effects using CSS animations or JavaScript libraries like GSAP or Anime.js.

2. **Mobile Application Interaction Elements:**
   - o **Gestures:** Implement touch gestures such as tapping, swiping, pinching, and dragging to enable intuitive interactions on mobile devices.
   - o **Navigation Patterns:** Utilize navigation patterns like tabs, bottom navigation bars, navigation drawers, and swipeable views for seamless navigation between screens.
   - o **Touch Feedback:** Provide visual feedback (e.g., ripple effect, animation) to indicate user interactions and actions.
   - o **Mobile-specific Inputs:** Use mobile-specific input elements like date pickers, time pickers, and native dropdowns for better usability on touch devices.
   - o **Push Notifications:** Integrate push notifications to engage users and provide timely updates and alerts within the mobile application.

5. Software Testing

Unit 5.1: Testing Fundamentals and Techniques

Unit 5.2: Designing Effective Test Cases

Unit 5.3: Debugging, Error Handling and Test Reporting

Key Learning Outcomes

At the end of this module, you will be able to:

1. Explain the various types of software testing (unit, integration, system, acceptance).
2. Design test cases with expected results to validate software functionality.
3. Implement debugging techniques and handle errors effectively.
4. Apply black box and white box testing methods to identify software defects.
5. Document and report test processes for effective communication.

Unit 5.1: Testing Fundamentals and Techniques

Unit Objectives

By the end of this unit, the participants will be able to:

1. Define different types of software testing (unit, integration, system, acceptance testing) and their purposes.
2. Discuss how testing helps ensure software quality, reliability, and user satisfaction.
3. Identify how testing activities contribute to different SDLC phases (planning, development, deployment, etc.).
4. Distinguish between black box and white box testing approaches.

5.1.1 Different Types of Software testing and its contribution to SDLC

What is Software Testing?

Software testing is an important process in the software development lifecycle. It involves verifying and validating that a software application is free of bugs, meets the technical requirements set by its design and development, and satisfies user requirements efficiently and effectively.

This process ensures that the application can handle all exceptional and boundary cases, providing a robust and reliable user experience. By systematically identifying and fixing issues, software testing helps deliver high-quality software that performs as expected in various scenarios.

Fig. 5.1: Software testing principles

Different types of software testing (unit, integration, system, acceptance testing) and their purposes:

1. **Unit Testing:** Unit testing involves testing individual units or components of the software in isolation. The purpose is to validate that each unit functions correctly as per its specifications. It helps identify bugs early in the development process and facilitates code refactoring and maintenance.
2. **Integration Testing:** Integration testing verifies the interaction between different units or modules of the software. It ensures that integrated components work together as expected and handle data exchange and communication properly. Integration testing helps uncover interface defects and integration errors.

3. **System Testing:** System testing evaluates the behavior of the entire software system as a whole. It validates that the software meets specified requirements and functions correctly in its intended environment. System testing covers functional and non-functional aspects such as performance, reliability, security, and compatibility.
4. **Acceptance Testing:** Acceptance testing involves validating the software against business requirements and user expectations. It is typically performed by end-users or stakeholders to determine if the software satisfies acceptance criteria and is ready for deployment. Acceptance testing ensures that the software meets user needs and delivers value to stakeholders.

How testing helps ensure software quality, reliability, and user satisfaction

1. **Bug Detection:** Testing helps identify defects, errors, and inconsistencies in the software, allowing developers to fix issues before deployment. This contributes to software quality by improving reliability and reducing the likelihood of post-release failures.
2. **Validation and Verification:** Testing validates that the software meets specified requirements and verifies that it functions correctly in various scenarios. This ensures that the software behaves as expected and delivers the intended functionality, enhancing reliability and user satisfaction.
3. **Risk Mitigation:** Testing helps mitigate risks associated with software defects, performance issues, security vulnerabilities, and usability problems. By identifying and addressing risks early in the development lifecycle, testing minimizes the likelihood of negative impacts on users and stakeholders.
4. **Continuous Improvement:** Testing provides feedback on the software's performance, usability, and quality, enabling continuous improvement and refinement. By analyzing test results and incorporating feedback from testing activities, developers can enhance software reliability, usability, and user satisfaction over time.

Testing activities contribute to different SDLC phases (planning, development, deployment, etc.):

1. **Planning Phase:** Testing activities in the planning phase involve defining test objectives, strategies, and scope, as well as estimating resources and timelines for testing activities.
2. **Development Phase:** Testing activities in the development phase include unit testing, where developers test individual units of code, and integration testing, where they verify the interaction between integrated components.
3. **Testing Phase:** The testing phase encompasses system testing and acceptance testing, where testers evaluate the entire software system against specified requirements and user expectations.
4. **Deployment Phase:** Testing activities in the deployment phase involve final validation and verification of the software to ensure readiness for deployment, including compatibility testing, performance testing, and acceptance testing by end-users or stakeholders.

5.1.2 Difference between black box and white box testing

Parameter	Black Box Testing	White Box Testing
Definition	Tests software without knowledge of the internal structure.	Tests software with knowledge of the internal structure.
Alias	Also known as data-driven, box testing, and functional testing.	Also known as structural, clear box, code-based, or glass box testing.
Base of Testing	Based on external expectations; internal behavior is unknown.	Internal working is known; tests are designed accordingly.
Usage	Ideal for higher levels like system and acceptance testing.	Best suited for lower levels like unit and integration testing.
Programming Knowledge	Not needed.	Required.
Implementation Knowledge	Not required.	Complete understanding is necessary.
Automation	Challenging to automate due to dependency on external behavior.	Easier to automate.
Objective	To check the functionality of the system under test.	To check the quality of the code.
Basis for Test Cases	Can start after preparing the requirement specification document.	Can start after preparing the detailed design document.
Tested By	End users, developers, and testers.	Primarily testers and developers.
Granularity	Low.	High.
Implementation Knowledge	Based on trial and error.	Focuses on data domain and internal boundaries.
Automation	Less exhaustive and time-consuming.	Exhaustive and time-consuming.
Objective	Not the best method for algorithm testing.	Best suited for algorithm testing.
Basis for Test Cases	Not required.	Required. Code security is a concern if testing is outsourced.
Tested By	Well-suited for large code segments.	Helps in removing extra lines of code, revealing hidden defects.
Granularity	Testers with lower skill levels can test the application without knowledge of the implementation or programming.	Requires expert testers with vast experience.

Table 5.1: Difference between black box and white box testing

Unit 5.2: Designing Effective Test Cases

Unit Objectives

By the end of this unit, the participants will be able to:

1. Develop test cases that target various functionalities of a software application.
2. Apply test case design techniques (equivalence partitioning, boundary value analysis, error guessing) to create comprehensive test coverage.
3. Differentiate between positive and negative test scenarios and their importance.
4. Predict and document clear and concise expected results for each test case.

5.2.1 Develop and apply Test cases

Develop test cases that target various functionalities of a software application:

1. **Identify Test Scenarios:** Understand the functional requirements of the software application and identify different scenarios that need to be tested. These scenarios could cover user interactions, system behaviors, edge cases, and error conditions.
2. **Create Test Cases:** Based on the identified scenarios, develop specific test cases that outline the steps to be executed, including input data, actions performed, and expected outcomes. Each test case should target a unique aspect of the application's functionality.

Test case design techniques to create comprehensive test coverage:

1. **Equivalence Partitioning:** Divide input data into equivalence classes that share similar characteristics and are likely to produce similar results. Test cases are then created to represent each equivalence class, ensuring comprehensive coverage while minimizing redundancy.
2. **Boundary Value Analysis:** Focuses on testing the boundaries and extreme values of input data. Test cases are designed to evaluate the behavior of the software at the boundaries between equivalence classes, as these areas are more likely to contain errors.
3. **Error Guessing:** Based on domain knowledge, past experiences, and intuition, testers identify potential error scenarios that may not be covered by formal techniques. Test cases are then created to simulate these error conditions and validate the application's response.

5.2.2 Differentiate between positive and negative test scenarios

The importance of designing both positive and negative test cases for comprehensive mobile app testing:

Positive Test Scenarios	Negative Test Scenarios
The Password text box should allow 8 characters input.	The password text box should throw an error or should not accept when less than 8 characters are entered.
The Password text box should allow 15 characters of input.	The password text box should throw an error or should not accept when more than 15 characters are entered.
Any values between 8 and 15 characters long should be accepted by the Password text box.	The password text box should not accept special characters as input
It should accept any combination of letters and numbers in Password text box.	The password text box should not accept a combination of numbers only or a combination of letters only.

Table 5.2: Differentiate between positive and negative test scenarios

5.2.3 Predict and document clear and concise for test cases

Predicting and documenting clear and concise expected results for each test case is crucial for effective testing.

1. **Understand the Test Objective:** Before predicting expected results, it's essential to understand the objective of the test case. Clearly define what specific functionality or behavior you are testing.
2. **Review Requirements:** Refer to the software requirements specification or user stories to understand the expected behavior of the feature being tested. This helps in aligning your expected results with the intended functionality.
3. **Consider Inputs and Preconditions:** Take into account the inputs, initial conditions, or prerequisites required for executing the test case. The expected results should be based on these inputs and **conditions.**
4. **Define Expected Outputs or Behavior:** Clearly specify what output or behavior the software should exhibit when the test case is executed successfully. This could include actions performed by the system, data displayed on the user interface, or changes in the system state.
5. **Document Expected Results:** Document the expected results in a clear and concise manner alongside the test case description. Use simple language and avoid ambiguity to ensure that anyone executing the test can easily understand the expected outcome.
6. **Include Positive and Negative Scenarios:** For positive test cases, document the expected behavior when valid inputs or correct conditions are provided. For negative test cases, specify the expected behavior when invalid inputs or unexpected conditions are encountered.
7. **Account for Variability:** Consider any factors that may introduce variability in the expected results, such as different configurations, environments, or user inputs. Document how these factors may affect the expected outcome.
8. **Validate Expected Results:** Review and validate the expected results to ensure they accurately reflect the intended behavior of the software. Verify that the expected results align with the test objective and are achievable based on the test inputs and conditions.
9. **Update Expected Results as Needed:** As the software evolves and changes are made, review and update the expected results accordingly to reflect any modifications to the functionality or behavior being tested.

Unit 5.3: Debugging, Error Handling and Test Reporting

Unit Objectives

By the end of this unit, the participants will be able to:

1. Implement debugging techniques to identify and fix errors in software code.
2. Design robust error handling mechanisms to gracefully manage errors during application execution.
3. Utilize appropriate methods to document test processes and results effectively.
4. Effectively communicate testing findings and recommendations to stakeholders through comprehensive reports.

5.3.1 Debugging, Error Handling and Test Reporting

Debugging techniques to identify and fix errors in software code:

1. **Logging:** Incorporate logging statements strategically throughout the code to track the program's execution flow, variable values, and potential errors. Logging helps in identifying the sequence of events leading to an issue and provides valuable insights for debugging.
2. **Debugging Tools:** Utilize debugging tools provided by integrated development environments (IDEs) or standalone debuggers to step through the code, inspect variables, and identify the root cause of issues. Set breakpoints at critical points in the code to pause execution and examine the program state.
3. **Code Review:** Conduct peer code reviews to leverage the collective expertise of team members in identifying logic errors, syntax issues, and potential bugs. Code reviews provide an opportunity for collaboration and knowledge sharing, leading to better code quality.
4. **Unit Testing:** Write unit tests to validate the behavior of individual code components in isolation. Unit tests help in identifying bugs early in the development process and provide a safety net for refactoring and code changes.

Advanced Debugging Techniques:

Advanced debugging techniques encompass a range of sophisticated methods and strategies used to identify, isolate, and fix issues within software applications:

- Gain hands-on experience applying techniques like:
  - o **Code Stepping:** Execute code line by line to examine variable values and program flow.
  - o **Setting Breakpoints:** Pause code execution at specific points to inspect variables and the call stack.
  - o **Logging Statements:** Strategically place log messages to track variable values, function calls, and program behavior.
- Learn how to interpret logs and analyze stack traces to pinpoint the root cause of errors.

5.3.2 Error Handling in Mobile Apps

Error Handling in Mobile Apps is a critical aspect of mobile application development, ensuring that users have a smooth and reliable experience while using the app. This covers the strategies and best practices for effectively identifying, handling, and reporting errors within mobile applications:

1. **Exception Handling:** Implement try-catch blocks or exception handling mechanisms to gracefully handle exceptions and prevent application crashes. Catch specific types of exceptions and handle them appropriately, either by logging the error, displaying user-friendly messages, or taking corrective actions.

2. **Error Logging:** Log errors and exceptions systematically, including relevant information such as stack traces, error codes, and contextual data. Centralize error logging to a designated location or service for easier monitoring, analysis, and troubleshooting.
3. **Fallback Mechanisms:** Provide fallback mechanisms or alternative pathways for critical operations to mitigate the impact of errors. For example, if a network request fails, fallback to a cached data source or display a default response to the user.
4. **Graceful Degradation:** Design applications to gracefully degrade functionality in the event of errors or unsupported conditions. Prioritize essential features and ensure that the application remains functional, albeit with reduced capabilities, in adverse conditions.

5.3.3 Test Process and Result Reporting

Test Process and Result Reporting is a critical component of the software development lifecycle, ensuring that the quality and functionality of the software are rigorously assessed and reported.

Utilize appropriate methods to document test processes and results effectively:

1. **Test Plan Documentation:** Begin by creating a detailed test plan that outlines the scope, objectives, resources, and schedule of the testing process. Include information about the testing approach (e.g., manual, automated), testing tools, and the roles and responsibilities of team members involved in testing.
2. **Test Case Documentation:** Document each test case thoroughly, including a clear description of the test scenario, input data or actions, expected results, preconditions, and postconditions. Organize test cases logically and categorize them based on functionality, feature, or priority for easier reference and management.
3. **Test Execution Logs:** Maintain comprehensive logs of test execution, recording details such as the date and time of testing, test environment configurations, test case status (pass/fail), any deviations from expected results, and relevant screenshots or logs. These execution logs provide a detailed record of testing activities and aid in traceability and debugging.
4. **Defect Tracking:** Utilize defect tracking tools or issue management systems to log and track defects identified during testing. Document each defect with essential information such as defect description, severity, priority, steps to reproduce, environment details, and the status of resolution. Regularly update the defect log with any changes or progress in defect resolution.

5.3.4 Communicating Test Reports to Stakeholders

Communicating Test Reports to Stakeholders is a crucial aspect of the software development process, ensuring that the outcomes of testing activities are effectively conveyed to relevant stakeholders.

Effectively communicate testing findings and recommendations to stakeholders through comprehensive reports:

1. **Test Summary Reports:** Prepare comprehensive test summary reports that provide an overview of the testing process, including testing objectives, scope, approach, resources utilized, and key findings. Summarize test results, including test coverage metrics, defect statistics, and any notable observations or insights.

2. **Visualizations:** Use visual aids such as charts, graphs, and diagrams to present testing data and trends effectively. Visualizations can help stakeholders quickly grasp complex information and identify patterns or trends in test results. Include visual representations of test coverage, defect trends over time, and pass/fail ratios for different test categories.
3. **Detailed Test Reports:** Provide detailed test reports for each testing phase or iteration, including detailed test results, defect summaries, and recommendations for improvement. Include a breakdown of test case execution status, defect statistics, test coverage analysis, and any additional insights or observations from the testing process.

Skill India
कौशल भारत-कुशल भारत
MINISTRY OF SKILL DEVELOPMENT & ENTREPRENEURSHIP GOVERNMENT OF INDIA
N·S·D·C RE IMAGINE FUTURE
India skills

6. Technical Communication

Unit 6.1: Technical Documentation and Tools

Unit 6.2: Communicating with Stakeholders

Key Learning Outcomes

At the end of this module, you will be able to:

1. Create technical documentation using industry standards like flowcharts, diagrams, and code comments.
2. Effectively communicate technical information to clients and stakeholders.

Unit 6.1: Technical Documentation and Tools

Unit Objectives

By the end of this unit, the participants will be able to:

1. Explain the importance of comprehensive technical documentation in software development.
2. Utilize industry-standard tools and techniques for effective documentation.

6.1.1 Technical Documentation and Tools

Comprehensive technical documentation is crucial in software development as it serves as a blueprint for understanding the software's structure, functionality, and usage. It provides a centralized and reliable source of technical data that enables project teams to make informed decisions quickly and ensures that all stakeholders are on the same page.

Technical documentation offers several benefits, including:

1. **Ease of Onboarding:** New developers can quickly understand a project's structure and requirements, making their onboarding process seamless.
2. **Maintainability**: Documentation ensures that developers can maintain, debug, and refactor the software more efficiently as it evolves.
3. **Knowledge Sharing:** Documentation promotes the sharing of knowledge and best practices among developers, reducing the risk of knowledge silos.
4. **Quality Assurance:** Documentation helps QA teams understand the software's functionality and requirements, resulting in more effective testing.
5. **Customer Support:** Well-documented software enables support teams to resolve customer issues more efficiently.
6. **Reduced Dependence:** Good documentation reduces the reliance on specific individuals, ensuring that projects can continue without significant disruption if key developers leave.
7. **Reusability:** Documentation encourages code reusability by making it easier for developers to identify and understand existing code snippets for future projects.

6.1.2 Industry-Standard Tools and Techniques for Effective Documentation

Technical Documentation Tools and Techniques are essential for creating comprehensive, accessible, and well-structured documentation that effectively communicates technical concepts, processes, and instructions to various audiences.

To create effective technical documentation, follow these best practices:

1. **Understand the Document's Goals and Target Audience:** Define the primary purpose and objectives of the document, and tailor it to the intended audience.
2. **Gather Relevant Information:** Ensure you have a clear grasp of the product's functionality and purpose by gathering information from developers, stakeholders, and other sources.
3. **Plan the Document Structure:** Organize the document into logical sections with headings and subheadings to facilitate easy navigation.
4. **Write Clear and Concise Content:** Use simple language and avoid jargon to make the documentation accessible to a broader audience.

5. **Use Visual Aids:** Include diagrams, screenshots, and other visuals to make the documentation more engaging and easier to consume.
6. **Keep It Up-to-Date:** Maintain the documentation as the project evolves, updating it to reflect changes, enhancements, or new features.
7. **Collaborate:** Involve multiple stakeholders, such as product managers and designers, in the documentation process to ensure a holistic view of the product.

Examples of Software Documentation

Software documentation encompasses a wide range of materials that provide information about software products, their features, functionalities, and usage. Here are some examples of software documentation:

1. **Data Model Documentation:** Information about the data structures and relationships used by the software, including entities, attributes, and relationships defined in the data model, along with examples of how the data model is utilized by the software
2. **API Documentation:** Clear and comprehensive documentation for APIs, including details about API endpoints, request/response structures, and authentication mechanisms.
3. **User Manuals:** Comprehensive documents showcasing all the software product's features and functions, aiming to teach users how to use the product effectively
4. **Quick Start Guides:** Concise guides designed to help users learn about software and start using it without delving into intricate details, providing a quick introduction to the software
5. **Tutorials:** Learning aids designed to share knowledge and skills related to a particular software topic, such as using specific modules of enterprise software
6. **User-Focused Documentation:** Guides and articles that help users understand the software, including details about downloading, setting up, and troubleshooting the software
7. **Internal Documentation:** Documents such as PRDs, user stories, and roadmaps, organized and accessible within a collaborative wiki for internal use
8. **Technical Documentation for Programming Languages and Frameworks:** Information providing key details for developers, including programming languages, frameworks, and APIs.

These examples illustrate the diverse nature of software documentation, catering to different audiences and serving various purposes within the software development and user experience realms.

Unit 6.2: Communicating with Stakeholders

Unit Objectives

By the end of this unit, the participants will be able to:

1. Identify different stakeholders involved in software development projects (e.g., developers, testers, clients, end-users).
2. Tailor technical communication to different audiences.
3. Develop effective communication skills (e.g., clear delivery, handling questions) to engage the audience.

6.2.1 Communication with stakeholders

Software development is a collaborative effort involving individuals with varying interests and priorities. Understanding these stakeholders and their perspectives is crucial for effective communication. We'll explore common stakeholder groups and their roles:

Internal Stakeholders:

1. **Developers:** The individuals responsible for writing, testing, and maintaining the software code. They are primarily concerned with technical details, feasibility, and code quality.
2. **Testers:** The team responsible for identifying and resolving software bugs. They focus on ensuring functionality, performance, and user experience.
3. **Project Managers:** Individuals who oversee the planning, execution, and delivery of the software project. They manage resources, timelines, and budgets.
4. **Product Owners:** The representatives who define the product vision, features, and priorities. They bridge the gap between business needs and technical development.

External Stakeholders:

1. **Clients/Sponsors:** The entities funding the software development project. They are interested in the project's return on investment, alignment with business goals, and adherence to timelines and budgets.
2. **Investors:** Individuals or organizations providing financial backing for the project. Their primary concern is the project's potential for success and financial return.
3. **End-Users:** The individuals who will ultimately use the software application. Their focus is on usability, functionality, and the software's ability to meet their needs.

Considering the audience and customizing the message accordingly is key to effective communication. This approach nurtures collaboration, keeps stakeholders well-informed, and ultimately plays a vital role in the success of a software development project.

6.2.2 Tailoring Technical Communication

It involves the deliberate adaptation of communication strategies and content to effectively engage and inform diverse stakeholders within the software development ecosystem.

1. **Adapting Communication to Diverse Audiences in Software Development:** In the realm of software development projects, stakeholders encompass a spectrum of technical expertise. Effective communication demands tailoring the approach to suit each audience:
   - For Technical Audience (Developers, Testers):
     - o Utilize technical jargon and delve into specific technicalities.
     - o Incorporate code snippets, architecture diagrams, and in-depth technical discussions.

   - For Non-Technical Audience (Clients, Investors, End-Users):
     - o Simplify technical concepts.
     - o Omit jargon and concentrate on project impacts and benefits.
     - o Employ clear, concise language complemented by visual aids (charts, graphs, mockups)

2. **Clarity and Brevity:** Irrespective of the audience, aim for clarity and conciseness:
   - Simplified Language: Employ easily understandable language to elucidate complex concepts.
   - Logical Structure: Methodically organize your message (introduction, body, conclusion).
   - Supporting Information: Substantiate points with data, visuals, and examples (charts, graphs, screenshots).

Customized Examples:

Below are instances of adapting communication for varied audiences:

- **Bug Report:**
  - o **For Technical Audience:** Present detailed steps to replicate the bug, error codes, and potential code fixes.
  - o **For Non-Technical Audience:** Describe the bug's impact on user experience and the anticipated resolution timeline.
- **Project Update:**
  - o **For Technical Audience:** Address technical challenges, code implementation specifics, and upcoming milestones.
  - o **For Non-Technical Audience:** Showcase progress on features, address client concerns, and exhibit user interface mockups.

By tailoring your communication style and content, you bridge the chasm between technical expertise and stakeholder needs. This cultivates collaboration, ensures comprehensive information dissemination, and contributes to the triumph of a software development project.

6.3.3 Effective Communication Skills

The success of your communication hinges not just on the content itself, but also on your ability to deliver it in a way that captures attention, fosters understanding, and facilitates a productive exchange.

1. Presentation Skills:

Transforming technical information into an engaging presentation requires honing your delivery skills. Here's how:

i. Practice and Confidence: Regularly practice your presentation to ensure fluency and project confidence. This reduces delivery anxiety and allows you to focus on connecting with your audience.

ii. **Strategic Visual Aids:** Don't overload your slides with text. Use visuals strategically to enhance understanding. Charts, graphs, screenshots, and even well-placed images can significantly improve audience engagement.

iii. **Body Language and Eye Contact:** Maintain eye contact with your audience and use appropriate body language to convey enthusiasm and professionalism. Avoid nervous mannerisms that can distract from your message.

2. Handling Questions:

A successful presentation is a two-way conversation. Be prepared to handle questions effectively:

i. **Active Listening:** Actively listen to questions and clarify if necessary. Don't assume you understand the question before responding.

ii. **Tailored Responses:** Provide clear and concise answers tailored to the audience's level of understanding. Avoid overly technical jargon if your audience is not technical.

iii. **Acknowledging Limitations:** It's okay not to have all the answers. Acknowledge limitations and offer to follow up with additional information if needed.

By mastering these presentation and audience interaction skills, you can transform technical information into an engaging experience. This fosters a more collaborative environment, ensures stakeholders are actively involved, and ultimately contributes to a successful project outcome.

Skill India
कौशल भारत - कुशल भारत

MINISTRY OF SKILL DEVELOPMENT
& ENTREPRENEURSHIP
GOVERNMENT OF INDIA

N·S·D·C
RE IMAGINE FUTURE

India skills