External Exam of SE(SSCS3010)

Chapter -1 Introduction to SE

1. Define Software

A software is : a computer program, data structures and descriptive information

2. Define SE

A SE is an establishment and use of sound engineering principles in order to obtain the economically software that is reliable and works efficiently in real machines

3. State characteristics of software.

Software is developed and engineered
Software doesn't wear out

4. Differentiate between system software and application software.

Feature	System Software	Application Software
Function	Manages hardware, provides a platform for other software	Helps users perform specific tasks
Dependency	Runs independently, critical for computer operation	Dependent on system software, cannot run without an operating system
Operation Timing	Starts when the computer is turned on, and operates continuously	Runs when initiated by a user and closes when the user stops it
Examples	Operating systems (Windows, Linux), device drivers, system utilities	Microsoft Office, web browsers, photo editors

User Interaction	Minimal direct user interaction mainly runs in the background	Significant user interaction, user-friendly interfaces

5. .Explain objectives of SoftwareEngineering.

Objectives of Software Engineering

1. **Maintainability:** It should be feasible for the software to evolve to meet changing requirements.
2. **Efficiency:** The software should not make wasteful use of computing devices such as memory, processor cycles, etc.
3. **Correctness:** A software product is correct if the different requirements specified in the SRS Document have been correctly implemented.
4. **Reusability:** A software product has good reusability if the different modules of the product can easily be reused to develop new products.
5. **Testability:** Here software facilitates both the establishment of test criteria and the evaluation of the software concerning those criteria.
6. **Reliability:** It is an attribute of software quality. The extent to which a program can be expected to perform its desired function, over an arbitrary time period.
7. **Portability:** In this case, the software can be transferred from one computer system or environment to another.
8. **Adaptability:** In this case, the software allows differing system constraints and the user needs to be satisfied by making changes to the software.
9. **Interoperability:** Capability of 2 or more functional units to process data cooperatively.

6. Explain the layered technology of Software Engineering.

Quality: Main principle of Software Engineering is Quality Focus.
An **engineering approach** must have a **focus on quality**.
Total Quality Management **(TQM)**, **Six Sigma**, **ISO** 9001, ISO 9000-3, CAPABILITY MATURITY MODEL **(CMM)**, **CMMI** & similar approaches encourages a continuous process improvement culture.

Process Layer: It is a foundation of Software Engineering, It is the glue the holds the technology layers together and enables logical and timely development of computer software.

- It **defines** a **framework** with activities for effective delivery of software engineering technology
- It establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Method: It provides **technical how-to's** for building software

- It **encompasses many tasks** including communication, requirement analysis, design modeling, program construction, testing and support

Tools: Software engineering tools provide automated or semiautomated support for the process and the methods

- Computer-aided software engineering (**CASE**) is the scientific application of a **set of tools** and **methods** to a software system which is meant to **result in high-quality, defect-free, and maintainable software products**.
- CASE tools automate many of the activities involved in various life cycle phases.

7. What are the generic activities of software process?

he **generic activities** in any software process model typically include:

1. **Communication** – Understanding the requirements through meetings with stakeholders.
2. **Planning** – Estimating costs, resources, schedules, and tasks.
3. **Modeling** – Designing data models, architecture, and interfaces.
4. **Construction** – Coding and testing the software.
5. **Deployment** – Delivering the product to users, followed by maintenance and support.

8. the importance of software engineering in real-world applications.

Software engineering ensures that software is:

- **Reliable**: Minimizes bugs and failures (e.g., medical devices or aircraft systems).
- **Scalable**: Supports growing users and data (e.g., e-commerce platforms like Amazon).
- **Efficient**: Optimizes performance (e.g., real-time stock trading apps).
- **Maintainable**: Easier to update or modify over time.
- **Cost-effective**: Reduces long-term development and maintenance costs.

Real-world example: In **hospital management systems**, software engineering helps manage patient records, appointments, billing, and diagnostics accurately and securely.

9.Evaluate the importance of layered technology in Software Engineering with a real-time example from banking systems.

Layered technology in software engineering is a structured approach where the system is divided into layers such as:

1. **Tools Layer** – IDEs, compilers, test tools.
2. **Methods Layer** – Design methods, coding practices, testing strategies.
3. **Process Layer** – Frameworks guiding the software lifecycle.
4. **Quality Focus Layer** – Ensures software meets required standards.

Importance in Banking Systems:

- **Example**: In **online banking**, the process layer ensures secure transaction protocols, the methods layer ensures proper authentication flows, and the tools layer supports secure coding.
- Ensures **security**, **data consistency**, and **scalability** while separating concerns for better maintenance.

10. Explain types of software with examples.

Software can be broadly classified into the following types:

1. **System Software** – Supports hardware functionality.
   Example: Operating Systems (Windows, Linux), device drivers.
2. **Application Software** – Performs specific user tasks.

Example: MS Word, Google Chrome, Tally.

3. **Embedded Software** – Built into hardware devices.
   Example: Software in washing machines, car engines.

4. **Web Applications** – Runs on browsers.
   Example: Gmail, Facebook, online banking portals.

5. **Mobile Applications** – Designed for smartphones/tablets.
   Example: WhatsApp, Uber, Paytm.

6. **AI/ML Software** – Performs intelligent tasks like recognition or prediction.
   Example: Chatbots, recommendation systems like Netflix.

Chapter-2

1. Define software process model and explain its use.

Software Process Model is a standardized format for planning, organizing, and running a software development project. It defines the steps, activities, and sequence followed from requirement gathering to deployment and maintenance.

Use:

- Provides a structured approach to development.
- Helps in time and cost estimation.
- Ensures quality and efficient workflow.
- Supports risk management and team coordination.

---

2. List Different Types of Software Process Models

- Waterfall Model
- Prototype Model
- Incremental Model
- Spiral Model
- Agile Model
- V-Model

- RAD (Rapid Application Development) Model

---

3. Explain the Waterfall Model with Advantages and Disadvantages

Waterfall Model:

A linear sequential model where each phase must be completed before moving to the next.

Phases:

1. Requirements
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

Advantages:

- Simple and easy to understand.
- Works well for smaller projects with well-defined requirements.
- Each phase has specific deliverables.

Disadvantages:

- Rigid — no going back to a previous phase.
- Not suitable for complex, evolving projects.
- Late discovery of errors (testing occurs after implementation).

---

4. Compare Waterfall Model and Spiral Model

Feature	Waterfall Model	Spiral Model
Process Type	Linear & sequential	Iterative with risk analysis
Risk Handling	Poor	Excellent (built-in risk assessment)
Flexibility	Low (no feedback loops)	High (repeated refinement)
Cost	Low for small projects	High due to repeated cycles

Ideal For	Simple, fixed-requirement projects	Large, complex, high-risk systems

---

5. Explain Prototype Model with a Neat Diagram

Prototype Model:

A working model or mock-up of the software is developed to understand requirements clearly.

Steps:

1. Collect requirements
2. Develop initial prototype
3. User evaluates prototype
4. Refine as per feedback
5. Final system is developed

📌 *Diagram:*

```
User Requirements → Quick Design → Build Prototype → User Evaluation → Refine → Final Software
```

Advantages:

- Improved user involvement and feedback
- Better requirement understanding

Disadvantages:

- Time-consuming iterations
- Users may mistake prototype as the final product

---

6. Describe Incremental Model and Its Usage in Software Development

Incremental Model:

The product is designed, implemented, and tested incrementally (piece by piece) until complete.

Usage:

- Projects needing early delivery of parts
- When requirements are known partially

Advantages:

- Delivers working product early
- Easier testing and debugging
- Flexible to changes

Disadvantages:

- Requires good planning
- Integration complexity increases

---

7. Explain Spiral Model in Detail with Its Advantages and Limitations

Spiral Model:

A risk-driven iterative model that combines elements of Waterfall and Prototyping.
Each loop (spiral) represents a phase:

1. Planning
2. Risk Analysis
3. Development
4. Evaluation

✅ Advantages:

- Best for large, high-risk projects
- Continuous risk management
- Early partial releases possible

❌ Limitations:

- Expensive and complex
- Not suitable for small projects
- Requires experienced teams

---

8. Evaluate Waterfall Model for Use in Banking Software System

- Banking systems need security, reliability, and stability.
- Waterfall may work only if requirements are fixed and known in advance.
- However, banking systems evolve (e.g., new regulations), so lack of flexibility is a drawback.
  ✅ Suitable for core, stable modules
  ❌ Not ideal for frequent change-driven features like user interfaces or mobile apps

---

9. Compare All Four Models w.r.t Risk Handling and Flexibility

Model	Risk Handling	Flexibility
Waterfall	Poor	Low
Prototype	Moderate	High
Incremental	Moderate to Good	Good
Spiral	Excellent	Very High

---

10. When Would You Choose Prototype Model Over Waterfall Model?

Choose Prototype when:

- Requirements are unclear or evolving
- User feedback is critical
- Rapid visual validation is needed

Justification:
Unlike Waterfall (which freezes requirements early), Prototyping allows:

- Frequent user interaction
- Better requirement refinement
- Lower risk of developing the wrong product

Chapter-3

1Define software requirement and explain its types with examples.

Define Software Requirement and Explain Its Types with Examples

Software Requirement is a condition or capability that a software system must possess to satisfy user needs, business goals, or regulatory requirements.

Types of Requirements:

Functional Requirements:

a. Define **what** the system should do.
b. Example:*The system shall allow users to log in with a username and password.*

Non-Functional Requirements:

c. Define **how** the system should behave.
d. Example: *The system should respond to user actions within 2 seconds*

User Requirements:

e. Written in natural language for end-users.
f. Example: *As a user, I should be able to search for books in the library.*

System Requirements:

g. Detailed, technical, and used by developers.
h. Example: *The system shall support up to 10,000 concurrent users.*

What is requirement engineering? Explain its process.

Requirement Engineering (RE) is the process of identifying, documenting, and maintaining software requirements.

Process of Requirement Engineering:

1. **Feasibility Study:**
   - Evaluate if the requirements can be implemented.
   - Example: Budget, timeline, technology constraints.
2. **Requirement Gathering:**
   - Collect information from stakeholders.
   - Example: Interviews, surveys, observations.
3. **Requirement Analysis:**

- Identify conflicts, dependencies, and priorities.
- Example: Validating user vs business needs.

4. **Requirement Specification:**
   - Document the requirements in a formal SRS.
   - Example: Using IEEE SRS format.
5. **Requirement Validation:**
   - Ensure requirements are complete, correct, and feasible.
   - Example: Reviews, walkthroughs, prototyping.
6. **Requirement Management:**
   - Handle changes and maintain version control.
   - Example: Change requests from clients.

3. Describe Different Requirement Gathering Techniques

Technique	Description	Example
Interviews	One-on-one sessions with stakeholders	Interview a librarian to understand daily tasks
Questionnaires	Distribute forms with structured questions	Gather user expectations for an app
Observation	Watch users interact with the current system	See how receptionists manage appointments
Workshops/Meetings	Collaborative sessions for brainstorming	Sprint planning meeting with developers and users

Prototyping	Build sample models to understand needs	Wireframe of a hospital registration screen
Document Analysis	Review existing documentation	Study current system reports and logs

4. Characteristics of a Good Software Requirement Specification (SRS)

A good SRS should be:

- **Correct** – Accurately represents the user's needs.
- **Complete** – Includes all functionalities.
- **Unambiguous** – Each requirement has one meaning.
- **Consistent** – No conflicting requirements.
- **Verifiable** – Can be tested.
- **Modifiable** – Easy to update.
- **Traceable** – Can trace each requirement to its origin.

5. Importance and Contents of an SRS Document

Importance:

- Acts as a contract between stakeholders and developers.
- Reduces development cost and rework.
- Serves as a basis for design, development, and testing.

Contents of SRS (IEEE Standard):

1. Introduction
    - Purpose
    - Scope
    - Definitions
2. Overall Description
    - System Environment
    - Constraints
3. Specific Requirements
    - Functional
    - Non-functional

4. Appendices
5. Index

6. Prepare SRS for a Library Management System

SRS: Library Management System

1. **Introduction:**
    - Purpose: Automate book issuance, returns, and member management.
    - Scope: For university libraries.
2. **Functional Requirements:**
    - Add/edit/delete books
    - Register members
    - Issue and return books
    - Generate reports
3. **Non-Functional Requirements:**
    - Response time < 3 seconds
    - Secure login for admins
4. **System Requirements:**
    - Runs on Windows/Linux
    - MySQL as backend database

7. Explain Requirement Engineering Tasks with Examples

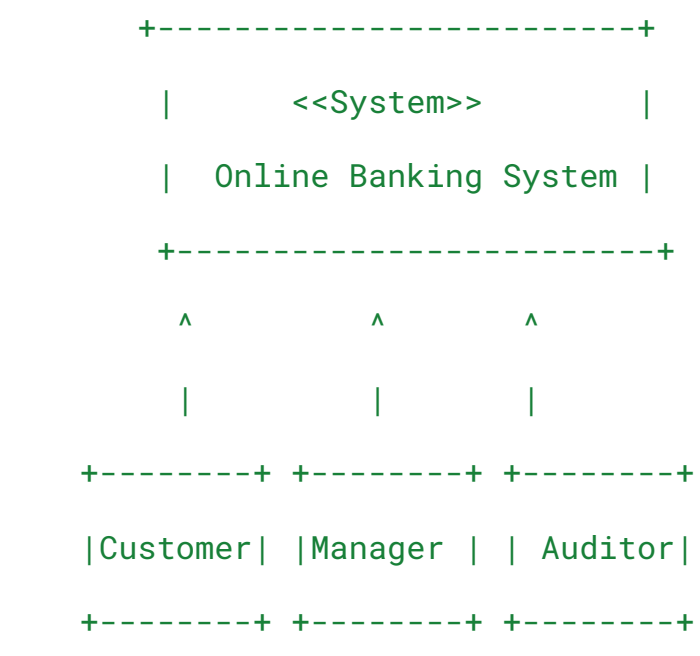
Task	Description	Example
Inception	Understand business context	Meet with hospital staff
Elicitation	Gather raw requirements	Use interviews, surveys
Elaboration	Refine and expand	Model workflows

Negotiation	Resolve conflicts	Decide on system priorities
Specification	Document requirements	Create SRS
Validation	Confirm accuracy	Review with stakeholders
Management	Handle changes	Update SRS with change log

8. Evaluate Challenges in Gathering Requirements for a Hospital Management System

- **Diverse Stakeholders:** Doctors, nurses, admins have different needs.
- **Regulatory Compliance:** Must meet health data standards (HIPAA, etc.).
- **Unclear Requirements:** Non-technical staff may struggle to express needs.
- **Integration:** With existing systems (lab, pharmacy).
- **Change Requests:** Frequent due to evolving medical practices.
- **Criticality:** Errors can lead to patient harm.

9. Use Case Diagram for an Online Banking System



Use Cases:

- Login
- View Account
- Transfer Money
- Pay Bills
- View Transaction History
- Manage Users (Manager)
- Generate Audit Report (Auditor)

10. Functional and Non-Functional Requirements of an Online Food Delivery System

Functional Requirements:

- User registration and login
- Search restaurants and food items
- Place orders
- Make payments
- Track orders
- Rate and review restaurants

Non-Functional Requirements:

- Performance: Response time < 2 sec
- Scalability: Handle 10,000+ users
- Security: End-to-end encryption for payments
- Availability: 99.9% uptime
- Usability: User-friendly mobile interface
- Maintainability: Modular codebase

Chapter-4

Define system design. What is the need of design in software engineering?

System design (in software engineering) is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. It translates requirements (what the system must do) into a blueprint (how it will be done), covering both high-level structural decisions and lower-level component behaviour.

Design is needed because:

Bridges requirements → implementation: turns abstract requirements into

implementable components.

Manages complexity: decomposes a large system into manageable parts.

Enables reuse & maintainability: good designs make future changes easier.

Improves quality: helps address non-functional requirements (performance, security, scalability).

Facilitates communication: provides clear diagrams and specs for stakeholders (developers, testers, clients).

Risk reduction: reveals architectural risks early and guides prototyping.

List and explain various design principles.
Abstraction
Architecture
Pattern
Separation of Concern
Modularity
Information Hiding
Functional Independence
Refinement
Aspects

What is a design model? Explain with components.

A **design model** is a collection of inter-related artifacts that describe the system's structure and behaviour at design time. It is more detailed than the requirements model and less detailed than the final code.

Data design — logical and physical data structures, database schema, data flow.
Architectural view — high-level building blocks (subsystems, major components) and their relationships.
Interface design — definitions of APIs, data formats, protocols between components.
Procedure design — It **transforms structural elements** of software into **procedural description** of software components

Describe different design concepts used in software engineering.
The beginning of **wisdom** for a **software engineer** is to **recognize** the **difference** between **getting program to work** and **getting it right**

Top-down and Bottom-up design: top-down decomposes system from high-level to low-level; bottom-up composes small components into larger systems.
Design patterns: reusable solutions to common problems (Singleton, Factory, Observer, Strategy, MVC).
Layering: organize system into layers (presentation, business, data).
Client-Server & Microservices: distribution approaches—monolithic vs many small services.

Event-driven & Message-oriented architectures: async processing via events/queues.
Concurrent & Parallel design: threads, processes, synchronization strategies.
Resilience & Fault-tolerance design: failover, redundancy, circuit breakers.
Scalability design: horizontal scaling, sharding, statelessness.
Security by design: threat modeling, encryption, secure auth flows.
Transactional & consistency design: ACID transactions, eventual consistency.
Interface and protocol design: REST, gRPC, WebSocket decisions.
Resource management: pooling, caching strategies, back-pressure.

Explain architectural design and its importance in system Development.
Large systems are decomposed into subsystems
Sub-systems provide related services
Initial design process includes
Identifying sub-systems
Establishing a framework for sub-system control and communication
Stakeholder Communication: High-level presentation of system
System Analysis: Big effect on performance, reliability, maintainability and other –ilities (Usability, Maintainability,Scalability, Reliability, Extensibility, Security, Portability)
Large-scale Reuse: Similar requirements similar architecture

Discuss layered architecture model with real-time software example (e.g., ATM or E-commerce).

Layered architecture organizes a system into layers where each layer has a specific role and interacts only with adjacent layers.

Common layers (example):

1. **Presentation (UI) layer** — user interfaces.
2. **Application / Service (Business) layer** — business logic, use-cases.
3. **Domain / Model layer** — core domain objects and rules.
4. **Data Access layer** — database access, repositories.
5. **Integration / Infrastructure layer** — external services, messaging, caching.

ATM (real-time example):

- Presentation layer: ATM UI screens (enter PIN, choose transaction).
- Service layer: Transaction processing (withdraw, deposit, balance inquiry).
- Domain layer: Account, Card, Transaction entities and business rules (withdraw limits).

- Data Access: Bank's account database or core banking API.
- Integration: Card validation, network switch, logging, secure PIN verification.
  Flow: User → ATM UI → Transaction Service → Account DB → Response.

Compare between design model and architectural model with examples.

Aspect	Architectural model	Design model
Level	High-level (system structure)	Mid/low-level (component internals)
Focus	Major components, their interactions and deployment	Class/module design, algorithms, data structures, interfaces
Audience	Architects, project managers, stakeholders	Developers, testers, implementers
Artifacts	Component diagrams, deployment diagrams, high-level sequence	Class diagrams, sequence diagrams, ERD, API specs
Decisions	Distribution, tech stack, scalability, security approach	Class responsibilities, method signatures, DB schema, control flow
Example	"System will be microservices: Auth, Catalog, Orders, Payments; use Kafka for messaging; deployed on Kubernetes."	"Order service: OrderManager class, OrderRepository (Postgres), REST endpoints, sequence for placing order, DB schema for order table."

Apply the concept of modularity and abstraction in designing an **Online Ticket Booking System**.

High-level modules:

1. **User Interface Module**
   - Functions: browse events, seat map, search, login/signup, booking UI.
   - Abstraction: UI consumes service APIs; does not know internal business rules.
2. **Authentication & Authorization Module**

- Handles login, OAuth, roles (user/admin), session management.

3. **Event & Inventory Module**
   - Manages events, showtimes, seating layouts, ticket inventory.
   - High cohesion: all seat-related operations live here.
4. **Search & Catalog Module**
   - Search events by date, venue, artist; filters.
5. **Booking / Reservation Module**
   - Handles seat reservation (hold), booking confirmation, cancellation.
   - Important: atomicity of seat allocation; implements seat-locking.
6. **Payment Module**
   - Integrates payment gateways, handles refunds, payment verification.
   - Abstraction: exposes PaymentService interface; multiple gateway implementations behind it.
7. **Notification Module**
   - Sends emails/SMS/push (booking confirmation, reminders).
8. **Reporting & Analytics Module**
   - Sales reports, occupancy stats, revenue dashboards.
9. **Admin Module**
   - Create/edit events, pricing, seating maps, promos.
10. **Integration module**
    - External systems: payment gateways, third-party ticket sellers.

Applying modularity:

- Each module is a bounded context with its own data (or schema), exposing REST/gRPC APIs.

- Use service contracts (API specs) so modules can be developed/tested independently.
- Example: BookingService exposes `reserveSeats(userId, eventId, seatList)` and `confirmBooking(reservationId, paymentInfo)`.

Evaluate the impact of poor design principles on software quality.

Poor design undermines nearly every quality attribute:

- **Maintainability suffers** — hard to change, causing long bug fixes and high cost.
- **Reliability decreases** — more bugs and runtime failures due to unclear responsibilities and poor error handling.
- **Scalability limited** — single monolithic components cause bottlenecks.
- **Performance problems** — inefficient data flows, blocking calls, lack of cache.
- **Security vulnerabilities** — poor encapsulation and sloppy access control lead to data leaks.
- **Testability lowered** — tightly coupled code is hard to unit test; leads to low test coverage.
- **Poor user experience** — latency, inconsistent behaviour, frequent downtime.
- **Higher cost & schedule slips** — refactoring and firefighting consume time and budget.
- **Technical debt accumulation** — leads to eventual rewrite costs.

Create a system design for a **Smart Home Automation System** highlighting major components and architecture.

Requirements (brief)

- Control devices (lights, HVAC, locks, cameras).
- Local and remote control (mobile app + voice).
- Schedules and automation rules (if motion -> turn on light).
- Scenes and user preferences.
- Security (auth, encryption).

- Offline capability (local control if internet down).
- Scalability to many devices and integration with 3rd-party services (Alexa, Google).

Major components

1. **Edge Devices (Sensors & Actuators)**
    - Smart bulbs, thermostats, locks, motion sensors, cameras (Zigbee, Z-Wave, Wi-Fi, BLE).
2. **Local Hub / Gateway (Edge Controller)**
    - Protocol translator (Zigbee ↔ IP), local rule engine, device registry, MQTT broker locally.
    - Functions: device discovery, local automation, OTA updates, local security.
3. **Device Manager Service (Cloud)**
    - Keeps canonical device state, versions, health monitoring.
4. **Automation / Rule Engine (Cloud + Edge)**
    - Executes user-defined rules and ML-driven automations.
    - Edge engine handles latency-critical rules; cloud engine handles complex/learning rules.
5. **User Interface**
    - Mobile app / Web UI: dashboards, rules editor, schedules, device control.
6. **Auth & Identity Service**
    - User accounts, multi-factor, OAuth/SSO, RBAC for guest access.
7. **Messaging / Event Bus**
    - MQTT (device → hub/cloud for telemetry), Kafka or cloud pub/sub for events.
8. **Data Storage**
    - Time-series DB for sensor telemetry, relational DB for configuration,

document DB for logs.

9. **Voice & Third-party Integration**
    - Connectors to Alexa, Google Home, IFTTT.
10. **Security Layer**
    - Device authentication (certs), TLS for transport, encrypted at rest, secure boot for devices.
11. **OTA Update Service**
    - Manage firmware rollouts with canary releases.
12. **Analytics & ML**
    - Power usage patterns, anomaly detection, predictive automation.
13. **Monitoring & Logging**
    - Health dashboards, error alerts.

Architecture (layered view)

- **Device Layer** — sensors/actuators (Zigbee/Wi-Fi).
- **Edge Layer (Gateway/Hub)** — local rule engine, device/firmware management, local decision making, local storage/cache.
- **Communication Layer** — secure channels (MQTT/TLS), message brokers.
- **Cloud Services Layer** — device manager, automation engine, API gateway, data stores, analytics.
- **Application Layer** — Mobile/Web UIs, third-party integrations (voice), admin consoles.
- **Management & Security Layer** — auth, update service, monitoring, logs.

Protocols & technologies (suggested)

- Device comms: MQTT over TLS, CoAP for constrained devices, Zigbee/Z-Wave for mesh.

- API layer: REST + WebSocket for real-time pushes.
- Event streaming: Kafka or cloud pub/sub.
- Databases: Time-series (InfluxDB/Timescale) for telemetry; Postgres for user/config; Redis for caching and locks.
- Deployment: Microservices on Kubernetes, edge services in small ARM Linux on hub.

Example Use-case Flow: Motion-triggered light

1. Motion sensor detects movement → publishes MQTT event to local hub.
2. Local hub rule engine checks rule `if motion at night then turn on corridor light for 2 minutes`.
3. Hub sends command to bulb via Zigbee → bulb turns on instantly (local, low-latency).
4. Hub sends telemetry to cloud for logging and analytics.

Fault-tolerance & offline considerations

- Local hub must handle core automations if cloud unavailable.
- Exponential backoff & queueing for intermittent connectivity.
- Device firmware has safe fallback behaviour.

Security considerations

- Unique device certificates for mutual TLS.
- Encrypted local storage for sensitive keys.
- Least privilege for APIs and admin consoles.
- Secure OTA with signed images.

Deployment & scaling

- Edge hubs scale per site (1 per home). Cloud services auto-scale; use CDN for static assets and push notifications.

Chapter - 5

1. Define User Interface Design

User Interface (UI) Design is the process of designing the visual layout, interactive components, and overall look-and-feel of a software system that allows users to interact with it effectively.
It focuses on ease of use, aesthetics, and functionality through elements like menus, buttons, icons, forms, navigation bars, and feedback messages.

2. Explain the golden rules of User Interface Design.

Place the User in Control

- **During a requirements-gathering session for a major new information system, a key user was asked about.**
- **Following are the design principles that allow the user to maintain control:**

Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
Provide for flexible interaction
Allow user interaction to be interruptible and undoable.
Streamline interaction as skill levels advance and allow the interaction to be customized.
Hide technical internals from the casual user.
Design for direct interaction with objects that appear on the screen.

Reduce the User's Memory Load

- **The more a user has to remember, the more error-prone the interaction with the system will be.**
- **Following are the design principles that enable an interface to reduce the user's memory load:**

Reduce demand on short-term memory.
Establish meaningful defaults
Define shortcuts that are intuitive
The visual layout of the interface should be based on a real-world metaphor
Disclose information in a progressive fashion

Make the Interface Consistent

- **The interface should present and acquire information in a consistent fashion.**
- **Following are the design principles that help make the interface consistent**

Maintain consistency across a family of applications
If past interactive models have created user expectations, do not make changes unless there is a compelling (convincing) reason to do so

3. **What are the important elements of a good UI?**
   Clarity – Clear labels, icons, navigation.
   Simplicity – Minimal design, avoid clutter.
   Consistency – Fonts, colors, and interactions are uniform.
   Feedback – System responds to user actions.
   Accessibility – Supports users with disabilities.
   Efficiency – Quick navigation, shortcuts for frequent users.
   Aesthetics – Visually appealing and balanced design.

4. **Discuss the significance of feedback and consistency in UI design.**

Feedback:

a. **Tells users the system has received input.**
b. **Reduces confusion and builds confidence.**
c. **Example: Loading spinner while submitting a form.**

Consistency:

d. **Users don't need to re-learn controls.**
e. **Increases usability and trust.**
f. **Example: Facebook uses the same "Like" button style across web and mobile apps.**

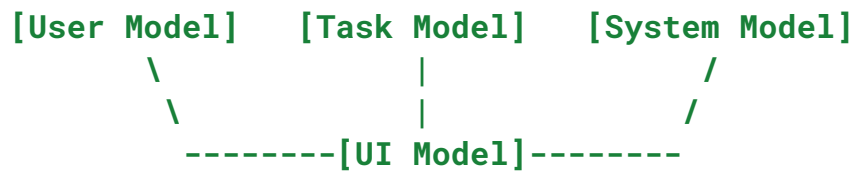
5. **Explain the User Interface Design Model with a neat diagram.**

The UI Design Model integrates user, task, and system considerations.

Components:

- **User Model – Characteristics of target users (age, experience, disabilities).**
- **Task Model – Activities the user performs (searching, uploading, buying).**
- **System Model – System's architecture, data flow, and capabilities.**
- **UI Model – Actual screens, menus, dialogs.**

Diagram (simplified):



6. **Design a basic UI layout for an **Online Shopping Cart** using UX principles.**

   | Logo | Search Bar | Cart (2) |

   | Categories | Product List with Images |

```
----------------------------------------
| Cart Summary (Right Side):          |
| - Item name + qty + price           |
| - Total Price                       |
| - Checkout Button                   |
----------------------------------------
```

7. **Evaluate the usability of a social media platform from a UX perspective.**

Example: Instagram

- **Strengths:**
  - **Simple scrolling feed.**
  - **Visual-first design (photos, videos).**
  - **Consistent icons (heart, comment, share).**
- **Weaknesses:**
  - **Complex privacy settings (confusing for new users).**
  - **Too many features (Reels, Stories, Posts, Shopping) → may overwhelm.**

Conclusion: While Instagram offers high engagement, its overloaded UI affects simplicity and user focus.

8. **Differentiate between UI and UX design with suitable examples.**

Aspect	UI Design	UX Design
Focus	Visuals & interactions	User experience & satisfaction
Elements	Buttons, menus, colors, typography	Research, wireframes, usability testing
Goal	Attractive, usable screens	Efficient, meaningful experience
Example	The "Buy Now" button style on Amazon	The entire buying journey from search → cart → delivery

9. **Explain how UI design impacts software accessibility with real world examples.**

   Font size & contrast – Large, high-contrast text helps visually impaired users.
   Example: **Wikipedia offers adjustable text size.**

   Keyboard navigation – Users with motor disabilities rely on keyboard.
   Example: **Google Docs supports full keyboard shortcuts.**

Screen reader support – Proper labels/alt-text for blind users.
Example: **Twitter allows adding alt-text to images.**
Color-blind friendly palettes – Avoid red/green-only distinctions.
Example: **Trello uses shapes + colors in labels.**

Impact: Inclusive design broadens reach, ensures legal compliance (e.g., WCAG, ADA).

10. **Apply the golden rules of UI design in creating a **Mobile Banking App** interface.**

Mobile Banking App UI Layout

- Home Screen: Balance overview, Quick Actions (Transfer, Pay Bills, Recharge).
- Menu: Transactions, Beneficiaries, Settings, Help.
- Notifications: Payment alerts, security warnings.
- Profile: User info, logout.

Application of Golden Rules

1. Place the User in Control
    - Allow users to cancel a transaction anytime before final confirmation.
    - Provide options to reorder dashboard features (customize what they see first).
    - Enable undo for accidental actions (like wrong amount entered).
2. Reduce the User's Memory Load
    - Show recent beneficiaries and favorite billers automatically.
    - Auto-fill account numbers after first use.
    - Provide hints/tooltips (e.g., explain transaction limits).
3. Make the Interface Consistent
    - Use the same button style and color scheme across all screens (e.g., "Pay Now" button always green).
    - Keep icons (home, transfer, settings) uniform across app and web version.
    - Maintain consistent terminology (e.g., always "Beneficiary," not "Payee" in some places).

Chapter - 6

1. Define UML and its importance in software engineering.

- **Definition: UML (Unified Modeling Language) is a standardized modeling language used to visualize, specify, construct, and document software systems.**

- **Importance:**

- Provides a common language for developers, designers, and stakeholders.
- Helps in visualizing system structure and behavior before coding.
- Supports object-oriented design concepts.
- Reduces complexity by breaking the system into diagrams.

---

2. What is a Use Case Diagram? Explain with suitable symbols.

- **Definition:** A use case diagram represents functional requirements of a system, showing interactions between actors (users/external systems) and use cases (system functions).
- **Symbols:**
  - Actor: Stick figure 👤
  - Use Case: Oval ○ (e.g., "Login")
  - System Boundary: Rectangle enclosing use cases
  - Associations: Line connecting actor to use case
  - Include/Extend: Dotted arrows showing dependency

---

3. Draw and explain a use case diagram for Online Food Ordering System.

Actors: Customer, Restaurant, Delivery Boy, Admin
Use Cases: Register/Login, Browse Menu, Place Order, Make Payment, Track Order, Update Menu, Manage Orders

📌 **Diagram (Text Form)**

```
Customer ----> (Register/Login)
Customer ----> (Browse Menu) ----> (Place Order) ----> (Make Payment)
Customer ----> (Track Order)
```

```
Restaurant ----> (Update Menu), (Manage Orders)
Delivery Boy ----> (Deliver Order)
Admin ----> (Manage Users), (Generate Reports)
```

---

4. Explain the Class Diagram and its components with example.

- **Definition: A class diagram represents the static structure of a system by showing classes, their attributes, operations, and relationships.**
- **Components:**
  - **Class: Rectangle with three parts (Name, Attributes, Methods).**
  - **Relationships: Association, Inheritance, Aggregation, Composition.**
- **Example: For a Bank System:**
  - **Class `Customer` (name, accountNo, balance; deposit(), withdraw()).**
  - **Class `Account` associated with Customer.**
  - **Inheritance: `SavingsAccount` and `CurrentAccount` inherit from `Account`.**

---

5. Draw a Class Diagram for Library Management System.

Classes:

- `Book (title, author, ISBN; issueBook(), returnBook())`
- `Student (name, rollNo; searchBook())`
- `Librarian (name, empId; addBook(), deleteBook())`
- `Transaction (transId, issueDate, returnDate)`

📌 **Relationships:**

- **Student *borrows* Book via Transaction.**

- **Librarian *manages* Book.**

---

6. What is a Data Flow Diagram? Explain levels of DFD.

- **Definition: A DFD shows how data moves through a system, highlighting processes, data stores, inputs, and outputs.**
- **Levels:**
  - **Level 0 (Context Diagram): High-level view (single process, main actors).**
  - **Level 1: Breakdown of main process into sub-processes.**
  - **Level 2+: Further detailed decomposition.**

---

7. Draw DFD (Level 0 and 1) for Student Admission System.

- **Level 0:**
  - **External Entities: Student, Admin**
  - **Process: "Admission System"**
  - **Data Stores: Student DB**
  - **Flows: Student submits form → System → Admin verifies → Confirmation to Student**
- **Level 1 (Expanded):**
  - **Processes: Fill Application, Verify Documents, Fee Payment, Generate ID Card**
  - **Data Stores: Student DB, Payment DB**

---

8. Explain the concept and structure of a Data Dictionary.

- **Definition: A Data Dictionary is a repository that stores details about data items (names, types, formats, sources, usage).**
- **Structure:**
  - **Data Name**
  - **Description**
  - **Type/Format**
  - **Source/Destination**
  - **Relationships/Constraints**

---

9. Create a Data Dictionary for Hospital Management System.

Data Item	Description	Type	Source	Destination
Patient_ID	Unique identifier for patient	Integer	Registration	Patient DB
Name	Patient's full name	String	Registration	Patient DB
Doctor_ID	Unique doctor identifier	Integer	Admin Input	Doctor DB
Appointment_Date	Date of appointment	Date	Patient Input	Appointment DB
Bill_Amount	Total amount of treatment	Decimal	Accounts	Billing DB

---

10. Compare Use Case Diagram and DFD in terms of purpose and usability.

Aspect	Use Case Diagram	Data Flow Diagram (DFD)
Focus	Functional interactions (who does what)	Data movement (what data flows where)

Representation	**Actors, use cases, relationships**	**Processes, data stores, flows, entities**
Abstraction Level	**High-level functional requirements**	**Detailed system workflows**
Usability	**Used in requirement gathering and stakeholder communication**	**Used in system analysis and process refinement**
Example	**Online Food Ordering → "Place Order" use case**	**Order data flows → Restaurant DB → Delivery**

Chapter - 7

Q1. Define Error, Fault, and Failure with suitable examples.

- **Error (Mistake):** A human mistake made by the developer during coding or design.
  Example: Developer writes `a = b + c` instead of `a = b – c`.
- **Fault (Defect/Bug): A** flaw in the software caused by an error in the code.
  Example: The incorrect statement (`a = b + c`) is part of the source code.
- **Failure:** When the software behaves incorrectly during execution because of a fault.
  Example: The program gives the wrong calculation result when executed.

Q2. Differentiate between Verification and Validation.

Aspect	Verification	Validation
Meaning	Ensures product is built correctly (process-oriented).	Ensures product meets user needs (product-oriented).
Focus	Checks design, documentation, code correctness.	Checks actual software behavior against requirements.
Performed by	QA team, developers.	QA team, end-users.
Methods	Reviews, walkthroughs, inspections.	Testing, acceptance tests.
Example	Checking the login page design document is as per requirement.	Checking the login page actually allows valid users to log in.

Q3. What is Black Box Testing? Explain its types.

Definition: Testing that focuses on the functionality of software without looking at internal code/logic.

Types:

1. Functional Testing – tests features as per requirements.
2. Boundary Value Testing – checks behavior at input limits.
3. Equivalence Partitioning – divides inputs into valid/invalid groups.
4. Decision Table Testing – tests combinations of inputs.
5. State Transition Testing – checks system response to state changes.
6. User Acceptance Testing (UAT) – validates with end-users.

Q4. Explain White Box Testing and its techniques.

Definition: Testing the internal structure, logic, and code of the program.

Techniques:

1. Statement Coverage – ensure every line executes at least once.
2. Branch Coverage – test all possible decision branches (if/else).
3. Path Coverage – test all independent paths through the code.
4. Loop Testing – test loops with 0, 1, and many iterations.
5. Condition Coverage – evaluate each condition true/false at least once.

Q5. Compare Black Box and White Box Testing.

Aspect	Black Box	White Box
Knowledge	Tester doesn't know internal code.	Tester knows internal logic/code.
Focus	Functional behavior.	Internal structure and logic.
Performed by	QA testers.	Developers/advanced testers.
Techniques	Boundary values, equivalence partitioning.	Path, statement, branch coverage.
Example	Testing login page functionality.	Testing algorithm correctness inside login function.

Q6. What is Unit Testing? How is it different from Integration Testing?

- Unit Testing: Testing individual components (functions, classes, modules) in isolation.
  Example: Testing the `addItem()` method in a shopping cart module.
- Integration Testing: Testing combined modules to check data flow and interaction.
  Example: Checking if `addItem()` correctly updates the database and UI together.

Difference: Unit testing checks *one component at a time*, integration testing checks *interaction among components*.

Q7. Describe System Testing and Performance Testing with examples.

- System Testing: Testing the entire integrated system against requirements.
  Example: Testing a banking application end-to-end (login, transfer, logout).

- Performance Testing: Evaluating system speed, scalability, and stability.
  - Load Testing: Check system under expected user load.
  - Stress Testing: Push beyond limits.
  - Volume Testing: Large data input.
    Example: Testing an e-commerce site with 10,000 users simultaneously.

Q8. Explain the process and advantages of using test cases.

Process:

1. Identify requirements.
2. Design test scenarios.
3. Write detailed test cases (input, steps, expected output).
4. Execute test cases.
5. Record results and defects.

Advantages:

- Ensures coverage of requirements.
- Helps detect defects early.
- Provides documentation for regression testing.
- Improves software quality.

Q9. Evaluate a scenario for Online Exam System using Verification & Validation techniques.

- Verification:
  - Review requirement documents (e.g., exam should auto-submit after 2

hours).
    - Check design of the timer logic.
    - Inspect database schema for student results.
- Validation:
    - Test if exam actually ends at 2 hours.
    - Check login and question navigation work correctly.
    - Validate that results are stored and displayed properly.

Q10. Design a testing strategy for a Flight Reservation System covering all levels of testing.

1. Unit Testing:

- Test individual modules like login, flight search, booking, payment.

2. Integration Testing:

- Test interactions between modules (e.g., booking module ↔ payment gateway).

3. System Testing:

- End-to-end test of booking a flight: login → search → book → payment → confirmation email.

4. Performance Testing:

- Load test with thousands of users searching/bookings simultaneously.

5. Security Testing:

- Verify user authentication, data encryption, prevent SQL injection.

6. User Acceptance Testing (UAT):

- Check system usability with actual users before release.

Chapter-8

Q1. What is Software Effort Estimation and why is it needed?

- **Definition:**
  Effort estimation is the process of predicting the amount of time, cost, and resources required to complete a software project.

- **Why Needed?**

- Helps in **project planning & budgeting**.
- Prevents **delays & overruns**.
- Assists in **resource allocation**.
- Improves **project monitoring & control**.
- Builds **client trust** through realistic timelines.

Q2. Describe various techniques for effort estimation.

1. **Expert Judgment (Delphi Technique):** Based on experience of experts.
2. **Analogy-based Estimation:** Compare with past similar projects.
3. **Parametric Models (COCOMO):** Use formulas and project size (LOC/FP).
    - Example: Effort = a × (Size)^b × EAF
4. **Function Point Analysis (FPA):** Estimate effort based on inputs, outputs, files, user interactions.
5. **Top-down Estimation:** Overall estimate → broken into sub-tasks.
6. **Bottom-up Estimation:** Each task estimated → combined for total effort.
7. **Use-case Point Method:** Uses complexity of use cases in project.

Q3. Explain the purpose and structure of a Software Project Plan.

- **Purpose:** To define **scope, schedule, resources, risks, quality standards**, and **deliverables** of a project.
- **Structure:**
    1. Project Objectives
    2. Scope of Work
    3. Project Deliverables
    4. Effort & Resource Estimation
    5. Project Schedule (milestones, Gantt chart)

6. Risk Management Plan
7. Quality Assurance Plan
8. Communication Plan

Q4. Describe the steps involved in Software Project Scheduling.

1. Define project tasks.
2. Estimate effort & resources.
3. Break down tasks into WBS.
4. Identify task dependencies.
5. Allocate resources to tasks.
6. Develop timeline (using Gantt chart, PERT/CPM).
7. Monitor and update schedule.

Q5. What is WBS? Explain its use in software project management.

- **WBS (Work Breakdown Structure):** A hierarchical decomposition of the project into smaller, manageable tasks.
- **Uses:**
  - Helps in **planning and scheduling**.
  - Clarifies **task ownership**.
  - Supports **progress tracking**.
  - Improves **effort estimation accuracy**.

Q6. Create a sample WBS for an E-learning Portal.

E-Learning Portal WBS:

1. **Requirements Analysis**
    - Collect user needs
    - Define system features
2. **Design**
    - UI/UX design
    - Database design
    - System architecture
3. **Development**
    - Frontend (course pages, dashboards)
    - Backend (login, course management, payments)
    - Database (user profiles, course content)
4. **Testing**
    - Unit Testing
    - Integration Testing
    - System Testing
5. **Deployment**
    - Server setup
    - Go-live
6. **Maintenance**
    - Bug fixes
    - Feature updates

Q7. What is a Gantt chart? Illustrate its usage in a software project.

- **Definition:** A **bar chart** that represents project tasks against a timeline.
- **Usage:**

- Visualizes **start and end dates**.
- Shows **task dependencies**.
- Helps track **progress**.

Example:

Task	Duration	Timeline
Requirement	5 days	
Design	7 days	
Development	15 days	
Testing	8 days	
Deployment	3 days	

Q8. Evaluate the effectiveness of scheduling in Agile versus Waterfall model.

Aspect	Agile	Waterfall
Scheduling	Iterative (sprints of 2–4 weeks).	Linear (fixed upfront schedule).
Flexibility	Highly adaptable to changes.	Rigid – hard to change once planned.
Deliverables	Frequent working prototypes.	Delivered at end.
Risk Handling	Early detection of issues.	Risks found late in cycle.
Example	Online shopping app – deliver login in Sprint 1, payment in Sprint 2.	Entire system delivered after all phases.

Q9. Design a project schedule using Gantt chart for Inventory Management System.

Tasks & Timeline:

1. Requirement Analysis – 5 days
2. Database Design – 7 days
3. Development – 15 days
    - Module 1: Inventory Entry
    - Module 2: Stock Update
    - Module 3: Reports
4. Testing – 10 days
5. Deployment – 3 days

📊 **Gantt Chart (Text View):**

```
Requirement Analysis:  ▓▓▓▓▓
Database Design:       ▓▓▓▓▓▓▓
Development:           ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓
Testing:               ▓▓▓▓▓▓▓▓▓▓
Deployment:            ▓▓▓
```

Q10. Compare effort estimation and scheduling in small vs large projects.

Factor	Small Projects	Large Projects
Estimation	Easier, fewer variables, based on expert judgment.	Complex, requires formal models (COCOMO, FPA).
Scheduling	Simple timelines, fewer dependencies.	Detailed scheduling with WBS, Gantt, PERT/CPM.
Risk	Lower risk, quick corrections possible.	Higher risk, delays have big impact.
Resources	Small teams, informal management.	Large teams, formal project management.

External Exam of SE(SSCS3010)