Module - 1

1. Defining Software and Its Characteristics

- **Software** is defined as a combination of computer programs (instructions), data structures, and descriptive information. This encompasses the code itself, the way data is organized, and the documentation that explains its operation and use.
- Software is developed/engineered, not manufactured in the classical sense.
- Software doesn't wear out physically but can deteriorate due to changes over time.
- Most software is custom-built, though the industry is moving toward component-based construction.

2. The Importance of Software Engineering

- Software engineering is crucial due to the complexity of modern systems. A disciplined approach is needed for designing, developing and managing complex software.
- Without proper engineering practices, projects can suffer from issues such as delays, budget overruns, unreliability, and maintenance difficulties. These issues can lead to severe consequences, such as the Ariane 5 rocket explosion due to a software error.
- The London Ambulance Service's software failure in 1992, which resulted in 46 deaths, further highlights the critical need for well-managed software development.
- Software engineering applies a systematic, disciplined, and quantifiable approach to software development, operation, and maintenance.

3. Types of Software

- The sources identify different types of software, including:
  - System software
  - Application software
  - Artificial intelligence software
  - Engineering and scientific software
  - Embedded software
  - Product-line software
  - Web applications (WebApps)
- Software can also be categorized as **generic products**, which are stand-alone systems marketed to any customer, and **customized or bespoke products**, which are developed for specific customer needs.

4. Software Engineering vs. Computer Science and Systems Engineering

- **Computer science** focuses on theory and fundamentals, while **software engineering** deals with the practical aspects of software design, development, and delivery.
- **Systems engineering** is an interdisciplinary field that focuses on the design and management of complex projects, encompassing all aspects of computer-based systems, including hardware, software, and processes. Software engineering is a part of systems engineering.

5. Software Process

- A **software process** (also known as a software methodology) is a set of related activities that lead to software production. These activities can involve new development or modifications to existing systems.
- The software process includes tasks (small, specific objectives), actions (sets of tasks producing a major work product), and activities (groups of related tasks and actions).
- A process is a collection of activities, actions, and tasks performed to create a work product.
- A generic framework for software engineering includes five activities: Communication, Planning, Modeling, Construction, and Deployment.
- Software processes emphasize repeatable actions, consistency, efficiency, standardization, and quality at every stage.
- The software process acts as a "recipe" that defines the roadmap for creating software.

6. Software Product

- A **software product** is the tangible outcome of the development process – the functional and usable software delivered to users.
- It focuses on meeting user needs and solving specific problems.
- The software product has its own life cycle that includes maintenance, updates, and iterations based on feedback.
- The software product is the "cake baked from the recipe".
- A product is the final output of a project and focuses on the end result. It is created based on customer needs and expectations.

7. Software Project

- A **software project** is a temporary endeavor aimed at developing a particular software product within a defined timeframe and budget.
- It focuses on achieving specific objectives within its scope and is allocated specific resources, tools, and teams.
- The project ends once the desired product is delivered.
- The software project is the instance of using the process to create a product within set constraints.

8. Relationship Between Process, Product, and Project

- The process guides the project to achieve a desired product.
- The product is the result of a project manufactured by a group of people.
- The process improves the quality of the project and serves as a template to direct the project.
- The process is a set of sequenced steps, whereas the product is the final production.

9. Software Process Activities

- Key software process activities include:
  - **Specification:** Defining customer needs and constraints.
  - **Development:** Design and programming.
  - **Validation:** Checking if the software meets requirements.
  - **Evolution:** Modifying software based on customer or market changes.

10. Software Process Models

The sources discuss several software development process models:

- **Waterfall Model:**
  - A linear sequential model where each phase must be completed before the next begins.
  - It is easy to understand and use, suitable when requirements are well-known and stable.
  - It is more suitable for small projects.
  - It produces documentation at every stage.
  - However, it's inflexible to changes and has high risk, taking a long time to complete.
- **Incremental Model:**
  - An iterative process where a small set of software requirements is implemented and enhanced in versions until the complete system is developed.
  - It is used when requirements are unclear, or may change.
  - It carries less risk than the Waterfall model.
  - Errors are easily identified, and it is more flexible than the Waterfall model.
  - It is suitable for projects with lengthy development schedules.
- **RAD (Rapid Application Development) Model:**
  - Focuses on faster and higher-quality development using workshops, user testing, and software component reuse.
  - It is best for developing prototypes quickly for testing software functionalities.
  - It is flexible, reduces development time, and increases reusability.
  - It is suitable when requirements are well-known and technical risks are limited.
- **Prototyping Model:**
  - A prototype is developed, tested, and refined based on customer feedback until an acceptable prototype is achieved.
  - It is best when customers don't know the exact requirements beforehand or when there is a lot of interaction with end users.
  - It reduces the risk of project failure, but can be costly and may not accurately represent the final product.
- **Spiral Model:**
  - Supports coping with risks by creating a prototype at every phase of development.
  - It analyzes risks at each phase and is more flexible than other SDLC models.
  - It is called a meta-model, subsuming the other classical SDLC models.
  - It is suitable when frequent releases are necessary, and requirements are complex and ambiguous.
- **Concurrent Process Model:**
  - Activities of different phases are done at the same time, allowing project managers to know the state of each phase.
  - It is applicable to all types of software development and gives immediate feedback from testing.

11. Agile Software Development

- Agile is a methodology that focuses on flexibility, collaboration, and efficiency for delivering quality software products.
- Agility refers to the ability to react to any kind of change.
- Agile is an approach focused on continuous delivery of working software in rapid iterations.
- It emphasizes individuals and interactions over processes and tools, working software over documentation, customer collaboration over contract negotiation, and responding to change over following a plan.
- Key benefits of Agile are adaptability, collaborative teamwork, and customer focus.
- **The Agile Manifesto** emphasizes delivering value and collaborating with customers through a committed group focused on achieving results quickly.
- **12 Principles of Agile** include customer satisfaction, welcoming change, frequent delivery, collaboration, motivated teams, face-to-face communication, working software, constant pace, good design, simplicity, self-organization, and reflection and adjustment.
- **MVP (Minimum Viable Product)**: An early version of a product with just enough core features to attract early-adopter customers and validate a product idea early in the product development cycle.

12. Agile Tools

- Several tools are used in Agile software development: Jira (ticketing system), Docker (container management), Kubernetes (automated deployment), Jenkins/Drone (CI/CD), and Slack (communication).

13. Agile Frameworks

- **Scrum:** An agile project management framework used primarily for software development, which delivers new software every 2-4 weeks.
  - Key components of Scrum are sprints (basic unit of development), product owner, Scrum Master, product backlog, sprint planning meetings, daily scrums, sprint review meetings, and sprint retrospectives.
  - A **product backlog** consists of features that you want to implement but have not yet prioritized for release.
  - A **sprint backlog** contains user stories that need to be completed during a specific period of time.
  - A **user story** is an informal explanation of a software feature from the end user's perspective.
  - Scrum is effective due to its focus on customers, and is a proven method for optimal collaboration.
  - It is ideal for complicated projects, and companies that value results and customer satisfaction.
  - Scrum has a short fixed release schedule with adjustable scope known as sprints.
- **Kanban:** A visual workflow management framework used for prioritizing and resolving bug fixes. It uses a visual board to represent workflow stages.
  - Team members work on tasks by pulling cards, prioritizing by severity, and having visibility of the bug-fixing progress.
- **Extreme Programming (XP):** An agile method that takes an 'extreme' approach to iterative development, with frequent releases and continuous testing.
  - XP aims to increase software efficiency and quality through testing and code refactoring.
  - XP is based on 12 principles including listening to the customer, simple things first, working in tandem, unit testing, refactoring, design on the fly, continuous integration, continuous delivery, code focused teams, quality control, responsible teams, and being in touch with reality.
  - Core values of XP include communication, simplicity, feedback, respect, and courage.
  - XP supports incremental development, customer involvement, people not processes, and change supported through regular releases.
  - Key practices include: incremental planning, small releases, simple design, test-first development, refactoring, pair programming, collective ownership, continuous integration, sustainable pace, and on-site customer.

14. Key Differences between Scrum and XP

- Scrum focuses on project management and productivity, while XP concentrates on software quality and engineering techniques.

- Sprints in Scrum usually last longer (2-4 weeks), whereas XP emphasizes shorter sprints (1-2 weeks).
- In Scrum, the product owner prioritizes tasks, while in XP, developers implement tasks in a strict priority order.
- Scrum relies on openness, focus, and commitment. Whereas, XP depends on communication, simplicity, and feedback.

15. System Engineering

- System engineering helps improve traceability which is the ability to trace all processes. It uses best practices such as baselining current performance, building business cases for live traceability, and creating quick wins.
- System engineering methods include stakeholder analysis, interface specification, design tradeoffs, configuration management, systematic verification and validation, and requirements engineering.
- **Design tradeoffs** involve balancing functionality, performance, security, maintainability, and usability.
- **Configuration Management** involves controlling changes by identifying items, setting up connections between things, and controlling changes.

16. Verification and Validation

- **Verification** ensures system-level requirements are met, while **validation** ensures the system performs as designed and intended.
  - Verification includes methods like reviews, walkthroughs, inspections, and desk-checking.
  - Validation includes methods like models, box testing, and non-functional testing.
- Verification checks if a component conforms to a specification, while validation checks if it meets the requirements and expectations.
- The validation process answers the question "Is it the right solution to the problem?", and evaluates requirements, functional and physical architectures, and implementation.
- Examples of verification include design and code review, unit testing, and integration testing.
- Examples of validation include usability testing, A/B testing, user acceptance testing (UAT), and performance testing.

This comprehensive overview should provide you with a solid understanding of the topics discussed in the sources. If you have specific questions about any of these concepts, feel free to ask!

Module - 2

Okay, here is a detailed overview of the topics covered in the provided sources, which include software project management, planning, agile methodologies, team dynamics, and cost estimation:

1. Project Planning and Scope Management

- **Project Planning Overview**: Project planning is a crucial initial step before software production begins. It involves several key components:
  - **Defining goals and objectives**
  - Establishing **success metrics**
  - Identifying **stakeholders** and their roles
  - Defining the **project scope and budget**
  - Setting **milestones, deliverables, and project dependencies**
  - Creating a **timeline and schedule**, along with a **communication plan**
- **Steps in Creating a Project Plan:** The sources outline steps to ensure proper project execution:
  - **Define stakeholders:** Identify all parties with an interest in the project
  - **Define Roles**: Establish responsibilities of the stakeholders
  - **Introduce Stakeholders**: Hold meetings to discuss scope, budget, goals, and roles before the project begins
  - **Set goals**: Establish clear targets for performance management
  - **Prioritize Tasks**: Rank tasks by importance and break down larger tasks into smaller objectives
  - **Create a Schedule**: Set up a system to track deadlines and implement corrective actions
  - **Assess Risks**: Identify potential issues and develop mitigation strategies during planning
  - **Communicate**: Establish communication channels and expectations
  - **Re-assess**: Re-evaluate the project at significant milestones
  - **Final Evaluation:** Reflect on the project's successes and areas for improvement
- **Scope Management**: This process creates boundaries for the project by defining what is included and excluded. Key actions during scope management include:
  - Defining the scope
  - Deciding on verification and control methods
  - Dividing the project into smaller, manageable parts
  - Verifying the scope
  - Controlling the scope and incorporating necessary changes

2. Work Breakdown Structure (WBS)

- **Definition**: A WBS is a hierarchical decomposition of the total scope of work to be carried out by the project team to accomplish the project objectives. It assists the project manager in developing a clear vision of the end product.
- **Steps for Preparing a WBS**:
  - Identify final project products
  - Identify major deliverables
  - Incorporate levels of detail until management requirements are met
  - Review and refine the WBS until stakeholders agree
- **Key Points in Developing a WBS**:
  - Include all approved scope
  - Reflect how the project manager intends to manage the project
  - Consider reporting requirements
  - Consider the size and complexity of the project
  - Consider resources executing the work (e.g., contractors vs. in-house)
- **WBS Structure**: The WBS is typically structured in a hierarchical manner with different levels:
  - Top level can represent the organization

    - Lower levels break down into tasks
    - An example is provided of a WBS for the construction of a single family home
- **Creating a WBS Step by Step**:
    - List every major deliverable
    - Break down each deliverable into smaller components
    - Split each work component into work packages
    - Identify the dependencies
    - Prioritize and assign tasks
- **Do's and Don'ts in WBS**:
    - Use nouns, not verbs
    - Ask for feedback
    - Follow the 100% rule
    - Keep elements mutually exclusive
- **Types of WBS:** WBS can be based on process/phase, deliverables/product, or roles
- **WBS vs. Project Schedule**:
    - WBS clarifies the scope and "what" needs to be done
    - Project schedule provides detailed activities, duration, milestones and critical paths
    - WBS forms the basis of the project schedule

3. Milestones and Deliverables

- **Milestones**: These are recognizable endpoints of project activities. They are distinct, logical stages of a project. Milestones are used as checkpoints for:
    - Project start and end dates
    - Need for external review or input
    - Checking budget
    - Submission of deliverables
    - Represent a clear sequence of events in project development
- **Deliverables**: These are outcomes, results, or products (like software, documents, assets) that are submitted to clients or end-users. Deliverables have:
    - A due date
    - Measurable attributes
    - They often satisfy a milestone or due date
    - Deliverables are generally milestones, but not every milestone is a deliverable

4. Cost and Estimates

- **Project Estimation:** This is used to help draft project plans and budgets early in the software development life cycle.
- **Software Size Estimation**: This can be done using KLOC (Kilo Lines of Code) or Function Points
- **Effort Estimation**: In Agile, this is done using user stories vs. sprints
- **Time Estimation**: Software tasks are broken down using a WBS, and time is scheduled on a daily or monthly basis
- **Cost Estimation**: Factors include:
    - Size of software
    - Software quality
    - Hardware
    - Additional software or tools, licenses
    - Skilled personnel
    - Travel and communication costs
    - Training and support
- **Empirical Estimation Techniques**:
    - **Putnam Model**: Maps time and efforts with software size
    - **COCOMO (Constructive Cost Model)**: Divides software into organic, semi-detached, and embedded categories
        - **Basic COCOMO:** Estimates effort based on lines of code
        - **Intermediate COCOMO**: Refines basic estimates using 15 cost drivers such as product, hardware, personnel and project attributes
        - **COCOMO Parameters:** Includes software size, team size, developer experience, environment, innovation, and deadlines
        - **COCOMO outputs**: Effort and schedule
        - **COCOMO equations**: Effort, Development Time, Average Staff Size, and Productivity are defined mathematically
    - **Function Points (FP):** A metric used to measure the functionality delivered by a system.
        - **FP calculation**: Uses direct measures of software's information domain.
        - **FP parameters**: External Inputs, External Outputs, External Inquiries, Internal Logical Files (ILF), External Interface Files (EIF)
        - **Complexity adjustment**: Assesses parameters for complexity and applies a complexity adjustment factor (CAF)
        - **FP formulas**: Count total * CAF
        - **FP usage**: Estimates cost, effort, errors, components and source lines
        - **FP Metrics**: Errors/FP, $/FP, Defects/FP, Pages of documentation/FP, Errors/PM, Productivity (FP/PM), $/Page of Documentation
        - **FP Example**: The source provides a worked example with values

5. Agile Project Management

- **Agile Overview**: An iterative approach to software development with continuous releases and customer feedback. It increases development speed, collaboration, and adaptability to market trends
- **Kanban**: A popular framework for implementing agile software development. Kanban emphasizes real-time communication, transparency, and visual representation of work.
    - **Kanban Board**: Visual tool with cards, columns, to improve efficiency. Includes elements such as Visual signals, columns, work in progress limits, a commitment point and a delivery point
    - **Kanban elements**
        - **Visual Signals:** Projects and work items are written on cards
        - **Columns:** Represent workflow stages (e.g., To Do, In Progress, Complete)
        - **WIP Limits:** Maximum number of cards in one column at a time

        - **Commitment Point:** When a project is picked up by the team
        - **Delivery Point:** End point when a product is handed to the customer
    - **Kanban Components**: List of work, Columns or lanes, Work In Progress limits, and Continuous Releases

6. Managing Team Dynamics and Communication

- **Team Dynamics**: This is the complex web of connections and behavior among team members. It includes skills in teamwork, communication, decision-making, and conflict resolution. It impacts a team's effectiveness, innovation and culture.
- **Factors Influencing Team Dynamics**:
    - Diversity of knowledge and experience
    - Psychological safety and trust
    - Communication abilities
    - Leadership style
    - Team size, makeup, location (co-located or remote)
- **Relationship with Project Outcomes**: Good team dynamics leads to higher quality products, collaboration and problem solving
- **Building Strong Team Dynamics**:
    - Establish trust and psychological safety
    - Encourage open and effective communication
    - Promote shared goals and accountability
    - Foster continuous learning and improvement
- **Tools for Tracking and Sharing Information:**
    - **Agile project management tools**: Task management, collaboration, communication, documentation and reporting features are all supported. Examples include Jira, Trello, and Asana.
    - **Collaboration tools**: Slack, Microsoft Teams, and Google Workspace facilitate communication and information sharing within Agile teams.
    - **Agile project management software**: Products such as Scrumwise or Rally provide solutions for sprint planning, backlog management, team coordination and progress tracking
    - **QR codes**: These can be used to access project-related information like user stories or task details

7. Agile vs. Non-Agile Web App Development

- **Agile Approach**:
    - Uses sprints (1-2 weeks) to achieve specific goals
    - Includes milestones and deliverables for each sprint
    - Emphasizes iterative development, prototypes and testing
    - Examples of sprints are provided
- **Non-Agile Approach**:
    - Uses phases like discovery, design, development, testing, and deployment
    - Includes defined milestones for each phase
    - Has a defined set of deliverables

This detailed overview should provide you with a comprehensive understanding of the various concepts discussed in the sources. Let me know if you have any more specific questions about any of these topics.

Module - 3

Here is a comprehensive structured overview of the topics from the sources, organized according to the syllabus you provided:

1. Software requirements and its types

Software requirements are the descriptions of the services that a software system should provide and the constraints under which it must operate. They define what the software should do to meet the needs of its users and stakeholders. This includes the functionalities and features expected from the system, such as processing user inputs, storing and retrieving data, generating reports, or controlling external devices. Constraints on operation are also part of software requirements, encompassing limitations like performance requirements (e.g., response time), security requirements (e.g., data encryption), usability requirements (e.g., ease of use), and compatibility requirements (e.g., operating system compatibility).

The sources identify several types of software requirements: Business requirements, Stakeholder requirements, Solution requirements, User requirements, and System requirements. Solution requirements are further categorized into Functional and Non-functional requirements. Domain requirements are also discussed as a specific category.

- **Functional Requirements:**
    - Functional requirements define a function that a system or system element must be qualified to perform. They describe the behavior of the system as it correlates to the system's functionality. These requirements focus on tasks, activities, and user goals for easier project management.
    - Characteristics include helping to understand the system's functions, making the system operational, being mandatory, easy to define, describing what the product does, and concentrating on the user's requirement.
    - Formats for preparing functional requirements include use cases, models, prototypes, user stories, and diagrams.
    - Examples for an online web app include the system sending a confirmation email upon new user account creation, sending an approval request after user enters personal information, a search feature, reviewing/changing items in a cart, creating accounts/logging in, sending notifications, and allowing user feedback or ratings.
    - Types of functional requirements mentioned are Authentication, Authorization levels, Data processing, User interface and user experience (UI/UX), Reporting, System integration, Transaction handling, Error handling and logging, and Backup and recovery.
- **Non-Functional Requirements:**
    - Non-functional requirements are not related to the software's functional aspect. They specify criteria used to decide the operation instead of specific behaviors. They denote the general characteristics, behavior of the system, and features that affect the user experience.
    - Basic non-functional requirements include usability, reliability, security, storage, cost, flexibility, configuration, performance, and legal or regulatory requirements.
    - Non-functional requirements are divided into two main categories: Execution qualities (like security and usability), observable at run time, and Evolution qualities (like testability, maintainability, extensibility, and scalability), embodied in the static structure of the software system.
    - Characteristics include helping to understand the system's performance, being not mandatory (though may be desirable), and helping to verify the software's performance.

    - Examples for an online web app include website pages loading time, system handling a large number of users without performance deterioration, payment processing gateway compliance (like PCI DSS), and compatibility (e.g., running on different operating systems).
    - Types of non-functional requirements mentioned are Usability, Security, Reliability, Performance, Availability, and Scalability.

- **Domain Requirements:**
    - Domain requirements are related to a particular category like software purpose, industry, or other domain of projects. They can be functional or non-functional.
    - These are essential functions that a system of a specific domain must necessarily exhibit.
    - They often meet established standards or widely accepted feature sets for that software project category.
    - Domain requirements typically arise in industries like military, medical, and financial sectors. They are identified from the specific domain and are not user-specific.
    - Examples include software in medical equipment, which must be developed per IEC 60601 regarding basic safety and performance, or academic software needing the functionality to access lists of faculty and students by grade to maintain records efficiently.
    - A software might be functional and usable but not acceptable for production if it fails to meet domain requirements.

Other types briefly mentioned are:

- **Business requirements:** The high-level goals and objectives a system must achieve to support the organization's strategic needs, focusing on the "why".
- **Stakeholder requirements:** Capture the needs and expectations of stakeholders.
- **User requirements:** Describe the needs and expectations of the end-users.
- **System requirements:** Describe the overall requirements of the entire system, including software and hardware components.

2. Requirements Engineering process

Requirements engineering is the process of identifying, eliciting, analyzing, specifying, validating, and managing the needs and expectations of stakeholders for a software system. It is a systematic and strict approach to defining, creating, and verifying requirements. The process involves finding, analyzing, documenting, and checking requirements.

The sources outline several steps or activities in the requirements engineering process:

- **Requirements Elicitation:** This is the process of gathering information about the needs and expectations of stakeholders. It involves technical staff working with customers to find out about the application domain, required services, and operational constraints. Stakeholders can include end-users, managers, maintenance engineers, domain experts, and trade unions.
- **Requirements Analysis:** This step involves analyzing the information gathered during elicitation to identify high-level goals and objectives, and any constraints or limitations. It includes activities like domain understanding, requirements collection, classification, conflict resolution, prioritization, and checking. Techniques such as use case analysis, data flow diagrams, requirement decomposition, and risk analysis can be used. Conflicts among stakeholder requirements are identified and resolved, and requirements are prioritized.
- **Requirements Specification:** This involves documenting the identified requirements in a clear, consistent, and unambiguous manner. Requirements are prioritized and grouped into manageable chunks. Deliverables include the Software Requirements Specification (SRS) document, detailing functional, non-functional, user, and system requirements. Specifications must capture functionality, external interface, performance, quality attributes, constraints, safety, reliability, and maintainability. Qualities of good specifications include being valid, unambiguous, complete, understandable, consistent, ranked, verifiable, modifiable, and traceable.
- **Requirements Validation:** This step checks that the requirements are complete, consistent, accurate, and testable, and that they meet stakeholder needs and expectations. It is concerned with demonstrating that the requirements define the system the customer truly wants. Fixing requirements errors later is significantly more expensive. Checks include validity, consistency, completeness, realism, and verifiability.
- **Requirements Management:** This involves managing requirements throughout the software development life cycle. It includes tracking and controlling changes and ensuring requirements remain valid and relevant. Requirements are inevitably incomplete and inconsistent, changing as business needs and system understanding evolve. Different viewpoints may have contradictory requirements, and priorities can change. Activities include change control, version control, and traceability.

Other process activities mentioned include:

- **Feasibility Studies:** A short, focused study to decide if a proposed system is worthwhile, checking if it contributes to organizational objectives, can be engineered with current technology/budget, and can integrate with other systems. This involves information assessment, collection, and report writing.

3. Requirement Elicitation

Requirements elicitation is the process of gathering and defining the requirements for a software system. Its goal is to ensure the software development process is based on a clear and comprehensive understanding of the customer’s needs and requirements. It involves interacting with stakeholders through various techniques.

Eliciting requirements typically happens in stages, where a business analyst collects information from the client, conducts elicitation sessions with stakeholders, and gets approval before handing them to developers.

Key benefits of requirements elicitation include establishing precise scope and budget, avoiding confusion during development, adding business value, revealing hidden and assumed requirements, and allowing development of only relevant functionality. Its importance lies in ensuring compliance with business objectives and regulations, increasing user satisfaction, saving time and money by preventing rework, and providing a basis for traceability and documentation.

Advantages of requirements elicitation include clarifying and refining requirements, improving communication and collaboration, increasing the chances of developing quality software that meets needs, avoiding misunderstandings, managing expectations, supporting early identification of risks, facilitating accurate project planning, and increasing user/stakeholder confidence.

Disadvantages can include being time-consuming and expensive, requiring specialized skills, being impacted by changing business needs, political and organizational factors, a lack of stakeholder commitment, conflicting priorities, and potentially resulting in incomplete or inaccurate requirements if not properly managed.

Challenges in requirements analysis (often intertwined with elicitation) can include stakeholders not knowing what they really want, expressing requirements in their own terms, having conflicting requirements, organizational/political factors influencing requirements, and requirements changing during the process as new stakeholders emerge or the business environment changes. A specific challenge in interviewing is stakeholder uncertainty or unrealistic demands, communication gaps (jargon, implicit knowledge), and political factors.

4. System Modeling – Requirements Specification and Requirement Validation

System modeling refers to developing simplified, abstract representations of a system's components, their interactions, and behaviors. These models serve as tools for visualizing, specifying, and validating system requirements and design.

System modeling plays a role in Requirements Specification and Validation:

- **Specification:** Models capture essential requirements and functionalities, guiding the development process. The sources note that models like use cases, prototypes, user stories, and diagrams are formats for preparing functional requirements. Data Flow Diagrams (DFDs) help in visualizing data flow, essential for accurately capturing requirements.
- **Validation:** Models enable testing of the system's design against requirements to ensure it meets stakeholder expectations. Prototyping, which involves using an executable model of the system, is a technique for requirements validation. DFDs also help clarify system boundaries during elicitation, ensuring all necessary functionalities are identified and documented, which aids validation later.

System modeling involves different models and techniques:

- **System Models:** Context, Interaction, Structural, Behavioral, Data.
- **Modelling Techniques:** Data Flow Models (DFD), Entity-Relationship Model (ERD), State Diagram Models, Unified Modelling Language (UML).
    - DFDs visualize data flow, processes, data stores, and external entities, helping to understand complex processes and identify bottlenecks. DFDs can be created at different levels of detail (Context Diagram, Level 1, Level 2, etc.).
    - ERDs represent the logical design of a database, abstracting real-world objects as entities and their associations as relationships. They are crucial for database design, identifying key elements (entities, attributes, relationships), providing a visual representation, facilitating table creation, supporting constraint definition, and enabling normalization.
    - UML is a standardized language for visualizing, specifying, constructing, and documenting software systems artifacts. It includes various diagrams representing structural aspects (Class, Object, Component, Deployment) and behavioral aspects (Use Case, Activity, State Transition, Sequence, Communication, Timing).
        - **UML Use Case Diagrams:** Provide a shared understanding of system functionality, capturing use cases and actors (users/external systems). They show a graphical overview of functionality and are suitable for explaining a system to a non-technical audience.
        - **UML Activity Diagrams:** Behavioral diagrams representing the flow of control, like flowcharts but can show concurrent activities. They are suitable for confirming logic with business users, especially for complex business logic.
        - **UML State Chart Diagrams:** Visual representation of a system's states and the events causing transitions, modeling behavior with well-defined states. Typically used for complex systems.
        - **UML Sequence Diagrams:** Show object interactions and message passing over time, depicting the chronologically structured event flow through a use case.
        - **UML Class Diagrams:** Graphical representation of a system's static structure, showing classes, attributes, operations, and relationships (Association, Aggregation, Composition, Generalization/Specialization). They are the foundation for component and deployment diagrams.
        - **UML Component Diagrams:** Model the physical parts of the system (files, executables, libraries), breaking down large systems into manageable components and visualizing their relationships and interfaces. They verify that required functionalities are implemented by components and aid communication.
        - **UML Deployment Diagrams:** Visualize the physical hardware and software deployment, showing how software components (artifacts) are deployed on hardware (nodes) and their communication paths. They help visualize hardware topology and plan system architecture.

Requirements Specification is documented in an SRS (Software Requirements Specification). Requirements Validation techniques mentioned include Requirements reviews/inspections (systematic manual analysis), Prototyping (using an executable model), Test-case generation (developing tests for testability), and Automated consistency analysis (checking consistency of structured requirements). Reviews involve both client and contractor staff and can be formal or informal. Automated checks can verify verifiability, comprehensibility, traceability, and adaptability.

5. Requirements Elicitation techniques

Various techniques are used to gather requirements from stakeholders:

- **Interviews:** Objective is to understand customer expectations. Representatives are selected based on expertise. Interviews can be Open-ended (no pre-set agenda) or Structured (prepared agenda, sometimes with a questionnaire). Interviews may not capture implicit requirements or reveal organizational constraints. Effective techniques include open-mindedness, active listening, and using prompts.
- **Surveys:** Questionnaires to collect data from a large number of stakeholders.
- **Observation:** Observing users interacting with existing systems or performing tasks in their work environment.
- **Focus groups:** Group discussions to explore ideas and gather feedback.
- **Workshops:** Collaborative sessions to brainstorm and refine requirements. Requirement workshops are also listed as tools.
- **Prototyping:** Creating early versions of the system to gather user feedback.
- **Brainstorming Sessions:** A group technique to generate new ideas and share views. Requires a facilitator to handle group bias and conflicts. Ideas are documented and potentially prioritized.
- **Facilitated Application Specification Technique (FAST):** A team-oriented approach aimed at bridging the expectation gap between developers and customers. Attendees list objects part of the environment, produced by, or used by the system; lists are combined, redundancies eliminated, teams develop mini-specifications, and a draft specification is written.
- **Quality Function Deployment (QFD):** Emphasizes requirements valuable to the customer, focusing on customer satisfaction. Identifies three types of requirements: Normal (explicitly discussed), Expected (obvious, not explicitly stated), and Exciting (beyond expectations).
- **Use Case Approach:** Uses Actors, Use cases, System boundary, and Dependencies to understand requirements. UML Use Case diagrams are a technique within this approach.
- **Ethnography:** An observational technique involving immersing oneself in the stakeholder's work environment to understand actual practices, social interactions, and organizational context. Value lies in discovering implicit requirements and revealing how people *actually* work. Benefits include uncovering hidden requirements (workarounds, informal practices) and improving system design (informing prototypes, reducing refinement cycles). Limitations include a focus on the end-user perspective, being less suitable for groundbreaking innovation, and being time-consuming.

Other tools involved in requirement engineering that can be considered elicitation tools include observation reports, questionnaires, use cases, user stories, mind mapping, and role playing.

6. Requirements management in Agile

Requirements management is the process of managing changing requirements during the requirements engineering process and system development life cycle. This involves tracking and controlling changes, managing conflicts, ensuring requirements remain valid, and aligning with evolving stakeholder needs. Requirements are inherently incomplete and inconsistent, with new requirements emerging and existing ones changing in priority due to shifting business needs, conflicting viewpoints, and changes in the environment.

In Agile environments, requirements are often managed using **Stories and Scenarios**.

- **Stories:** These are high-level narratives describing how the system might be used. They are effective for setting the "big picture" and facilitating broad discussions, potentially using shared tools like wikis.
- **Scenarios:** These are more detailed descriptions of specific user interactions. Scenarios are typically structured with information on inputs, outputs, normal flow, error handling, and system state. They can be used to derive specific system requirements.

The relationship between stories and scenarios is that stories provide the overall context, while scenarios delve into specific use cases. Stories can be used to generate more detailed scenarios. An example is a story about a teacher using a system to support student projects, leading to a scenario about a student uploading photos that the teacher moderates.

The benefits of using stories and scenarios in Agile include improved communication, helping stakeholders understand the system more concretely, a real-world focus that helps ensure the system meets actual user needs, and early identification and resolution of potential problems.

Module - 4

Based on the provided sources, here is a comprehensive structured overview of the topics according to your syllabus:

Software design is a core part of software engineering, applied regardless of the software process model. It begins after software requirements are analyzed and modeled and is the final step in modeling before construction (code generation & testing). Design uses elements from the requirements model (scenario-based, class-based, flow-oriented, and behavioral) to produce essential design models. These design models include Data/Class Design, Architectural Design, Interface Design, and Component Level Design. Software design directly impacts software construction and maintainability, ensures quality, provides representations that can be assessed for quality, and serves as the foundation for subsequent software engineering activities. An iterative process, it translates requirements into a blueprint, starting high-level and becoming more detailed with each refinement tracing back to requirements.

Design Concepts and Principles - Abstraction - Refinement - Modularity Cohesion Coupling

Several design concepts and principles guide software designers in creating better designs and applying advanced methods. These concepts help answer questions like how to divide software into components, separate details, and define high-quality design standards.

- **Design Principles**
  - **Avoid ‘Tunnel Vision’**: Do not focus excessively on one design aspect while ignoring others; maintain a broad perspective considering all requirements.
  - **Traceable to the Analysis Model**: The design should connect directly to requirements and analysis models, with every design feature having a clear reason based on earlier analysis.
  - **Don’t Reinvent the Wheel**: Use existing solutions, libraries, and design patterns to save time and ensure proven implementations.
  - **Minimize Intellectual Distance**: The software design should be as close as possible to the real-world problem to make the system easier to understand, modify, and maintain.
  - **Uniformity & Integration**: Maintain a consistent structure, style, and logic across all components and avoid disconnected work by different teams.
  - **Accommodate Change**: Design software to easily adapt to new features or requirements using modular and scalable approaches.
  - **Graceful Degradation**: The system should not crash completely if something goes wrong; implement error handling and fallback mechanisms.
  - **Design is NOT Coding; Coding is NOT Design**: Design focuses on planning and structure, while coding is implementation; good design guides coding but is separate.
  - **Assess Quality During Design**: Evaluate design for efficiency, scalability, and correctness while it is being created, rather than waiting until the end.
  - **Review to Minimize Conceptual Errors**: Use regular peer reviews and feedback to catch logical flaws and prevent misunderstandings early.
- **Abstraction**
  - Abstraction helps break down a problem into different levels. It is the process of hiding implementation details and showing only essential features, reducing complexity by allowing focus on *what* a system does rather than *how* it is implemented.
  - **Types of Abstraction**:
    - **Procedural Abstraction**: Breaking a process into well-defined procedures (functions/methods) that perform specific tasks without exposing internal implementation. Example: A `calculateInterest()` function hides the mathematical formula.
    - **Data Abstraction**: Defining data structures that hide implementation details and expose only necessary functionalities. Example: A `Car` class with methods like `startEngine()` without exposing internal mechanics. In a Calculator example, a private `result` variable is accessed only through a public `getResult()` method. The `add()` and `subtract()` methods encapsulate calculation logic.
    - **Control Abstraction**: Hiding the details of control flow and execution logic, allowing focus on high-level tasks. Example: Using `for-each` loops instead of managing loop counters manually. A `runCalculator()` function abstracts the program's control flow with a menu-driven interface.
- **Refinement**
  - Refinement is a **stepwise approach** that breaks down a complex system or problem into smaller, more manageable components. It is a **top-down strategy**, also known as Stepwise Refinement.
  - The goal is to refine procedural details until they can be implemented in a programming language. It starts from an abstract level and gradually becomes more detailed, moving from high-level concepts to detailed procedural steps.
  - Details are added progressively at each level, adding specificity to a function or process, ensuring a well-structured design.
  - Example: Designing a login system moves from "Authenticate user" to "Prompt user for username and password," then validating input, checking credentials, and finally granting/denying access, with detailed implementation steps like using encryption or OAuth. Similarly, processing a file moves from a high-level task to opening, reading, processing, and saving, with further breakdown of steps like reading line by line or applying data transformations.
  - Refinement helps break down a large process into smaller, self-contained functions, forming a hierarchy from abstract steps to detailed implementation. It is analogous to partitioning high-level requirements into functional modules during requirements analysis.
- **Modularity**
  - Modularity is the practice of breaking down a system into **smaller, manageable, and independent components (modules)**. These modules can work independently but combine to form a complete system. Software architecture and design patterns follow modularity principles.
  - **Key benefits of Modularity**:
    - **Easier to understand**: Breaking a system into smaller parts makes it more manageable.
    - **Improves reusability**: Modules can be reused in different projects.
    - **Simplifies debugging & maintenance**: Issues can be fixed within individual modules.
    - **Enhances scalability**: Adding new features is easier.
    - **Encourages parallel development**: Different teams can work on modules simultaneously.
  - **Criteria for Evaluating Modularity**:
    - **Modular Decomposability**: The system should be systematically divided into subproblems, helping problem-solving.
    - **Modular Composability**: Existing modules should be reusable in new systems, reducing development time.
    - **Modular Understandability**: Each module should be understandable as a standalone unit.
    - **Modular Continuity**: Small changes in requirements should affect only individual modules.
    - **Modular Protection**: An error in one module should not impact others.
  - **Balancing Act**: There's an optimal number of modules (M). **Undermodularity** (too few modules) makes software complex. **Overmodularity** (too many small modules) increases integration complexity and cost.
- **Cohesion**
  - Cohesion refers to the **degree to which the elements inside a module belong together**. A highly cohesive module performs a single, well-defined task with minimal dependency on others.
  - **Higher cohesion is desirable** as it makes software easier to maintain, test, and extend. Lower cohesion indicates randomly grouped functionalities.
  - **Types of Cohesion (from lowest to highest, desirable)**:
    - **Coincidental Cohesion**: Unrelated functions grouped randomly due to poor design. Example: A module printing a report, clearing the screen, and playing

music. **Problem**: Difficult to maintain/modify due to lack of logical grouping.
    - **Logical Cohesion**: Functions performing logically similar operations grouped, but the exact function depends on external control. Example: A module handling all output types (screen, printer, file) based on an input flag. **Problem**: Functionality depends on external conditions, less optimal.
    - **Temporal Cohesion**: Functions grouped because they execute at the same time (e.g., startup), not functional relation. Example: Module initializing variables, opening files, and clearing the screen at startup. **Problem**: If one task changes, unrelated tasks may be affected.
    - **Communication Cohesion**: Functions operate on the same data and are grouped. Example: A module retrieving, logging, and displaying temperature data. **Merits**: Improves modularity compared to lower types. **Problem**: Still ties functions together due to shared data, slightly harder to modify individual parts.
    - **Functional Cohesion**: The module performs a single, well-defined task. Example: A module calculating the average of a list of numbers. **Benefit**: **Highest, most desirable**. Easy to maintain, test, and extend as it focuses on one responsibility.

- **Coupling**
    - Coupling refers to the **degree of dependency between different software modules or components**. It measures how closely connected modules are.
    - **Lower coupling is desirable** because it makes software easier to modify, test, and maintain. High coupling should be avoided as it increases complexity and reduces flexibility.
    - **Types of Coupling (from strongest to weakest, desirable)**:
        - **Content Coupling**: One module modifies the internal data or logic of another module. Example: Module A directly changing a variable inside Module B. **Problem**: **Strongest, most undesirable**. Violates information hiding, makes debugging/maintenance difficult. If Module B changes internally, Module A breaks.
        - **Common Coupling**: Multiple modules share the same global data. Example: Many modules accessing/modifying a global configuration variable. **Problem**: Changes by one module cause unexpected errors in others, creating hidden dependencies.
        - **External Coupling**: Modules depend on external systems like databases, files, or network services. Example: A web application depending on a third-party API. **Problem**: System vulnerable to external changes or failures, reduces flexibility/reliability.
        - **Control Coupling**: One module controls another's behavior by passing control flags or parameters. Example: Function A passing a flag to tell Function B what to do. **Problem**: If Function B or the flag meaning changes, Function A may need modification, creating tight coupling.
        - **Stamp Coupling**: Modules pass complex data structures (objects, records) instead of specific required data. Example: Passing an entire student object when only the ID is needed. **Problem**: Creates unnecessary dependencies, making changes more complex.
        - **Data Coupling**: Modules share only the necessary data in function calls. Example: Passing only `name` and `age` instead of a full user object. **Benefit**: **Weakest, most desirable**. Modules are independent, share minimal data, easier to maintain, test, extend.

Architectural Design

Architectural design defines the **structure of software components and their interactions**, serving as a framework for detailed design. Architectural patterns help solve common design problems.

- **Key Properties of Architectural Design**:
    - **Structural properties**: Defines system components (modules, objects) and their interactions.
    - **Extra-functional properties**: Addresses performance, security, reliability, adaptability, etc..
    - **Families of related systems**: Uses repeatable design patterns for similar systems.

- **Architecture Model Types**:
    - **Structured Model**: Represents architecture as a collection of components (modules, objects, filters) and their interactions. Example: Layered Architecture (Presentation → Business Logic → Data Access). Library Management System example: modules for User Management, Book Inventory, Transaction Management.
    - **Framework Model**: Increases abstraction by identifying repeatable architectural design frameworks. Example: MVC framework. Library Management System example: Built using a Web Application Framework like Django or Spring Boot providing components for authentication, database access, routing, etc..
    - **Dynamic Model**: Focuses on behavioral aspects, showing how structure/configuration changes due to events. Example: Real-time traffic system adjusting signals. Library Management System example: Sequence diagram showing borrowing a book (User
      System
      Book Inventory -> Transaction Management). Data Flow Diagram showing data flow for borrowing.
    - **Process Model**: Represents the business or technical processes the system accommodates. Example: Order Processing System steps (Placement → Payment → Inventory Check → Shipping).
    - **Functional Model**: Represents the functional hierarchy, showing how functions interact/depend. Example: Banking System breaking "Manage Accounts" into sub-functions like "Create Account," "Deposit," "Withdraw". Library Management System example: Functional requirements list (Register User, Login User, Search Books, Borrow Book, Return Book).

- **Architecture Concepts**
    - **Control Hierarchy (Program Structure)**: Defines how components are organized and how control flows from one module to another. Measures include Depth (levels), Width (span), Fan-out (modules controlled by one), Fan-in (modules controlling one). Superordinate modules control subordinates. Library Management System example: Main Application controls User Management, Book Inventory, and Transaction Management modules.
    - **Structural Partitioning**:
        - **Horizontal Partitioning**: Splits software into separate functional branches (Input, Process, Output). Divides into layers based on functionality (presentation, business logic, data storage). **Merits**: Easier to test/debug, maintain/modify, reduces side effects, makes software extensible. **Demerits**: More data must be passed, increases complexity in control flow. Library Management System example: Layers for Presentation, Business Logic, and Data.
        - **Vertical Partitioning (Factoring)**: Top-down structure with control/decision-making at top levels (high-level modules) and execution at low levels (low-level modules handle input, computation, output). Divides system into independent functional units or subsystems. **Effects**: Change in a control module affects subordinates; change in a worker module has minimal impact. Allows for better scalability and modularity. Library Management System example: Subsystems for User Management, Book Inventory, Transaction Management communicating via defined interfaces.
    - **Logical Data Representation**: Defines how data elements relate to each other. Refers to how data is structured and organized, focusing on logical relationships between entities. Specifies data organization, methods of access, associations, and processing alternatives. Library Management System example: Defines relationships like a `Transaction` linking a `User` to a `Book`, independent of physical storage.

- **Architectural Mapping using Data Flow**: Helps transition from data flow diagrams (DFDs) to software architecture using a six-step process: Identify flow type, Define flow boundaries, Map DFD to program structure, Define control hierarchy, Refine design using heuristics (functional independence, cohesion, coupling), and Elaborate architectural description. Two major types of information flow for detailed design are Transform Mapping and Transaction Mapping.

Detailed Design Transaction Transformation

Architectural mapping includes Transform Mapping and Transaction Mapping, transitioning from DFDs to structured architecture.

- **Transform Mapping**
    - Applies when the system has a **transform flow**, where data moves through processing steps in a well-defined sequence.

    - **Characteristics**: Linear data flow, input converted to internal representation, processed at a transform center (core logic), output converted back to external representation.
    - **Steps**: Identify flow type (transform), Define flow boundaries (external to internal data transition), Identify Transform Center (core processing), Perform First-Level Factoring (top-down structure: top-level control, middle-level logic, low-level execution), Second-Level Factoring (map DFD transformations to modules), Refine Architecture (improve cohesion, reduce coupling).
    - Example: Payroll System - Input Domain (Collect Work Hours, Validate Data) → Transform Centre (Calculate Pay) → Output Domain (Generate Pay slips, Update Records).

- **Transaction Mapping**
    - Applies when the system has a **transaction flow**, with multiple possible execution paths based on actions.
    - **Characteristics**: System behavior triggered by an external transaction (user input, event), branches into different paths based on transaction type, a transaction center controls decision making and routes requests.
    - **Steps**: Identify the Transaction Center (module controlling branching), Define Input Transactions (external triggers), Establish Processing Paths (unique flow for each transaction type), Map DFDs to Program Structure (assign modules to handle transactions), Refine Software Architecture (optimize independence, modularity, scalability).
    - Example: ATM System - Transaction Center (Menu Selection Process/Transaction Dispatcher) routes requests to Transaction Modules (Withdraw, Deposit, Balance Enquiry) based on user selection.

Refactoring of Designs

Refactoring is a **reorganization technique** that improves software design or code **without changing its functionality or behavior**. Martin Fowler defines it as modifying a system to enhance its internal structure while maintaining external behavior.

- **Purpose**: Eliminates redundancy, optimizes algorithms/data structures to improve efficiency, and enhances readability, maintainability, and scalability.
- **Identifying the Need**: Detects poorly constructed or inappropriate design elements, identifies low-cohesion components, and recognizes unnecessary complexities.
- **Benefits**: Leads to **higher cohesion**, results in software easier to integrate, test, and maintain, and reduces technical debt.
- **Example**: A component performing three loosely related functions is refactored into three separate, high-cohesion components, improving modularity, testing, and maintainability.

Object-Oriented Design

While not a separate syllabus point in the source structure, concepts related to Object-Oriented (OO) Design are present:

- **Data/Class Design** converts class models into design class realizations, defines data structures for implementation, and uses CRC diagrams and class attributes. This integrates with software architecture design. This process aligns with OO principles of defining classes and their attributes.
- **Abstraction**, a key design concept, is fundamental to OO programming. **Data Abstraction** and **Procedural Abstraction** types directly correspond to OO concepts like encapsulation (hiding data) and methods (encapsulating procedures). The Calculator example demonstrating private data members and public methods is an OO approach to data and procedural abstraction.
- **Modularity principles** like Separation of Concerns (each class/object has a single responsibility), Reusability (classes/objects can be reused), Maintainability (changes are localized), Scalability (new features can be added with new classes/objects), and Testability (objects/classes can be tested independently) are core benefits realized through OO design.
- Architectural models like the **Structured Model** representing components as modules or objects and the **Framework Model** using structures like MVC are often implemented using OO paradigms.

User-Interface Design

Interface Design focuses on **user interfaces (UI), APIs, and external system interactions**. It is based on usage scenarios and behavioral models.

- **Golden Rules**:
    - **Place the User in Control**: Design reactive interfaces allowing user control. Avoid forcing actions, allow flexible modes. Support multiple interaction methods. Enable interruptible and undoable actions. Streamline interaction for advanced users (e.g., macros). Hide system internals from casual users. Enable direct manipulation of on-screen objects.
    - **Reduce the User’s Memory Load**: Use visual cues (recognition over recall). Provide meaningful defaults with customization options. Design intuitive shortcuts (mnemonics). Use real-world metaphors. Use progressive disclosure of information (high-level first, then details).
    - **Make the Interface Consistent**: Apply uniform design rules visually. Standardize input mechanisms and navigation. Provide meaningful context (titles, icons, colors). Maintain consistency across applications in a family. Respect established user expectations (e.g., common shortcuts).

- **Four Models in Interface Design**:
    - **User Model**: Establishes end-user profiles (demographics, abilities, goals), categorizing users (Novices, Knowledgeable Intermittent, Knowledgeable Frequent).
    - **Design Model**: Created by software engineers to align with the user model and guide design.
    - **Mental Model (System Perception)**: The user’s internal understanding of how the system works, influenced by experience and familiarity.
    - **Implementation Model**: Combines the system’s appearance (look and feel) with documentation (manuals, help files). **Goal**: Align this model with the user’s mental model. The key principle is "Know the user, know the tasks".

- **User Interface Design Process**: It is an iterative process, represented by a spiral model with four framework activities.
    - **Interface Analysis and Modeling**: Understanding user profiles, skill levels, environment, defining categories, eliciting requirements, conducting task analysis, analyzing the physical work environment.
    - **Interface Design**: Defining interface objects, actions, and screen representations to meet usability goals and ensure efficient task performance.
    - **Interface Construction**: Starting with a prototype to evaluate scenarios, refining using UI toolkits. **Prototyping is essential** for evaluation and refinement.
    - **Interface Validation**: Validating that the interface supports all tasks and variations, assessing ease of use, learnability, and user acceptance.
    - The process is **Iterative**, involving multiple passes to elaborate details incrementally.

- **User Interface Design Issues**:
    - **System Response Time**: Measured from user action to system response. Key characteristics are length (long delays frustrate) and variability (consistent times preferable). Goal is to minimize variability.
    - **Help Facilities**: Essential for guiding users. Considerations include availability (for all/subset), access (menus, keys, commands), representation (windows, suggestions, hypertext), structure (flat, layered, hypertext), and return (easy return to interaction).
    - **Error Handling**: Error messages should be clear, constructive, and nonjudgmental. Effective messages use understandable language, provide recovery advice, highlight consequences, include cues, and avoid blaming the user. Poor messages increase frustration/confusion.
    - **Menu and Command Labelling**: Ensure consistency and clarity. Considerations: correspondence (menu options match commands), form (control sequences,

keys, typed words), learnability (easy to learn/remember, reminders), customization (allow changes), clarity (labels/submenus self-explanatory).
    - **Application Accessibility**: Design to accommodate users with physical challenges (visual, hearing, mobility). Follow accessibility guidelines (W3C) for inclusivity. Driven by ethical, legal, and business imperatives.
    - **Internationalization**: Creating "globalized" software with generic core adaptable to different locales/languages. Localization features customize for markets. Challenges include varying screen layouts, alphabets, and symbols. Tools like Unicode address multilingual support.

- **User Interface Characteristics**: Visually Apparent and Forgiving (clear options, easy recovery). Hide System Internals (autosave, undo without worrying about system mechanics). Minimize User Input (maximize functionality with minimal required user info).
- **User Interface Design Principles**: Include Anticipation (predict user needs), Communication (clearly show status/location), Consistency (uniform navigation, icons, aesthetics), Controlled Autonomy (free navigation within security/convention limits), Efficiency (prioritize user productivity), Flexibility (support task-oriented and exploratory users, undo), Focus (keep interface on primary tasks), Fitt’s Law (place related options close for quick selection), Human Interface Objects (use reusable objects from libraries), Latency Reduction (use multitasking, feedback, entertain during delays), Learnability (design for quick learning/minimal relearning), Metaphors (use appropriate real-world metaphors), Maintain Work Product Integrity (autosave, recovery), Readability (clear fonts, size, contrast), Track State (store interaction state), Visible Navigation (bring content to user, minimal required navigation).

Module - 5

Here is a comprehensive overview of the topics from the provided sources, structured according to the syllabus you provided:

1. Strategic Approach to Software Testing
Software testing is defined as the **process of evaluating a software application** to ensure it meets specified requirements and is free of defects. It involves executing a program or system with the intent of **finding errors**, verifying functionality, and ensuring expected performance.
The **objectives of software testing** include:

- Detecting Bugs
- Validating Requirements
- Improving Quality
- Ensuring User Satisfaction
- Ensuring Compliance to Industry Standards and Regulations

Several **software testing strategies** are mentioned, often aligning with these objectives or testing levels:

- **Requirement-based Testing**: Focuses on verifying whether the software meets the specified requirements. Test cases are designed based on documented requirements.
- **Functional Testing**: Aims to validate the functional aspects by testing individual functions or features. Involves designing test cases to exercise functionalities and verify intended performance.
- **Integration Testing**: Tests the interaction and integration between various software components or modules, ensuring they work together seamlessly and exchange data accurately.
- **System Testing**: Validates the entire software system, including all integrated components and subsystems, to ensure they function correctly and meet overall system requirements.
- **Performance Testing**: Evaluates the software's performance under specific conditions like load, stress, or scalability, aiming to identify bottlenecks, measure response times, and ensure the software handles expected workloads.
- **Regression Testing**: Focuses on retesting previously tested functionalities after changes to ensure new defects haven't been introduced or unintended side effects caused.
- **User Acceptance Testing (UAT)**: Performed by end-users or representatives to verify if the software meets their expectations and requirements, ensuring it's ready for deployment.
- **Security Testing**: Aims to identify vulnerabilities and ensure resistance against potential threats, including testing authentication, authorization, data encryption, and protection against common attacks.
- **Usability Testing**: Assesses user-friendliness, ease of use, and overall user experience, evaluating interface design, navigation, and user interaction.
- **Maintenance Testing**: Performed after software updates or modifications to ensure changes haven't introduced new defects or adversely affected existing functionalities.

2. Testing Fundamentals
Software testing is the process of evaluating a software application to ensure it meets requirements and is free of defects. It involves executing a program or system to find errors, verify functionality, and ensure expected performance. The core objectives are detecting bugs, validating requirements, improving quality, ensuring user satisfaction, and compliance.

The sources identify various **types of software testing**:

- **Manual Testing**: Test cases are executed manually without automation.
- **Automated Testing**: Scripts and tools (e.g., Selenium, JUnit) automate test execution.
- **Functional Testing**: Verifies specific functions of the application (e.g., unit testing, integration testing).
- **Non-Functional Testing**: Tests performance, security, usability, etc..
- **White Box Testing**: The internal code structure is tested.
- **Black Box Testing**: The tester evaluates functionality without looking at the internal code.

White Box Testing examines the internal structure, logic, and code. It's also known as clear box, open box, or glass box testing.

- **Focus**: Tests are designed based on knowledge of the internal implementation, with testers having access to source code, algorithms, and architecture.
- **Objective**: To ensure all internal operations function as expected and to identify hidden errors, security vulnerabilities, and inefficiencies in the code.
- **Types of Defects Identified**: Logical errors, broken/poorly structured paths, syntax errors, unreachable/redundant code.
- **Techniques**:
    - **Basis Path Testing**: A structured method identifying basis paths (independent paths) in the control flow graph (CFG). Each path corresponds to a unique sequence of statements/decision outcomes. Objective is to ensure every independent path is executed at least once, aiming for 100% path coverage.
        - **Control Flow Graph (CFG)**: Graphical representation of program logic, where nodes are statements/blocks, edges are control flow, and decision points create branches.
        - **Cyclomatic Complexity (M)**: A metric determining the number of independent paths in the CFG. It can be calculated as $M = E - N + 2P$ (where E=edges, N=nodes, P=connected components, usually P=1) or $M = D + 1$ (where D=number of decision points). Cyclomatic Complexity equals the number of test cases needed for full path coverage. Steps involve drawing the CFG, calculating complexity, identifying independent paths, designing test cases, and executing them.

- **Statement Coverage**: Aims to cover as much source code as possible. Code coverage measures how much code has unit tests validating its function.
- **Branch Coverage**: Checks if each branch (e.g., True/False in an IF statement) is processed at least once. Requires test conditions for both true and false branches.
- **Path Coverage**: Examines all paths in the program. It's a thorough strategy ensuring all paths are explored at least once. It's more effective than branch coverage and useful for complex applications. Basis path testing relates to path coverage.
- **Graph Matrices**: A data structure that can assist in automating path tracing for basis path testing, especially for large graphs where manual tracing is difficult. It's a square matrix representing nodes and connections. A connection matrix uses link/edge weights (e.g., 1 for a connection, 0 otherwise). A connection matrix can also be used to find cyclomatic complexity.

Black Box Testing focuses on checking the functionality of the system without knowing the internal structure/design, comparing input values with output values. It is also known as **behavioral testing**. A tester without internal knowledge can perform it.

- **Errors Found**: Errors in independent modules, functional validity, interface errors, system behaviors/performance, maximum load/stress.
- **Techniques**:
  - **Boundary Value Analysis**: Tests the boundaries of input domains, focusing on values at the edges of equivalence classes because errors often occur there. For a range [min, max], it tests just inside, exactly on, and just outside these boundaries.
  - **Equivalence Testing (Equivalence Class Partitioning)**: Reduces test cases by dividing input data into equivalence classes where members are expected to behave similarly. If one value in a class works, others are assumed to work too.
    - **Equivalence Class**: A set of inputs expected to behave the same way.
    - **Valid Equivalence Classes**: Inputs valid according to requirements, representing normal/acceptable inputs.
    - **Invalid Equivalence Classes**: Inputs invalid according to requirements, representing edge cases or error conditions.
  - **Decision Table Based Testing**: Tests systems with complex business logic or multiple input conditions by creating a table of all possible input combinations (conditions) and corresponding outputs (actions). Ensures all possible scenarios are covered. Includes **Conditions**, **Actions**, and **Rules** (unique combination of conditions specifying actions).
  - **State Transition Testing**: Tests systems that exhibit different behaviors based on their current state and inputs, particularly useful where output/behavior depends on the sequence of previous inputs. Goal is to verify correct transitions between states based on rules/inputs. Includes **State** (system condition), **Transition** (movement between states), **Event/Input** (trigger causing transition), and **Output/Action** (result/behavior during transition).

3. Test Plan
A **test plan** is a detailed document outlining the strategy, objectives, scope, resources, schedule, and deliverables for testing a software product. It acts as a **blueprint** for testing activities, ensuring all stakeholders understand the process.
Key components of a test plan include:

- **Scope**: Defines what will be tested and what is excluded.
- **Objectives**: Specifies the goals of the testing effort.
- **Approach and Methodology**: Describes the testing strategies and techniques to be used.
- **Resources and Timelines**: Lists personnel, tools, and time required.
- **Test Environment**: Details the setup and characteristics of the environment where testing occurs.
- **Tools**: Specifies software and hardware tools needed.
- **Defect Management**: Outlines the process for reporting and managing defects.
- **Risk Management**: Identifies potential risks and mitigation strategies.
- **Exit Parameters**: Defines when testing can be considered complete.
  Importance of a Test Plan:
- **Improved Quality**: Helps identify and mitigate defects early.
- **Resource Utilization**: Ensures efficient allocation.
- **Faster Time to Market**: Streamlines development via effective strategies.
- **Risk Mitigation**: Identifies risks early.
- **Cost Reduction**: Can eliminate significant testing costs through better planning.

4. Test Design
Test design is a **crucial phase** in the Software Testing Life Cycle (STLC) where structured test scenarios and cases are created to ensure software quality. It's the process of defining **how testing should be done** by identifying test conditions, developing test cases, and specifying test data and expected results.
Key aspects of test design:

- **Test Conditions**: Statements about the test object (functions, transactions, quality attributes) that need to be verified.
- **Test Cases**: Detailed instructions defining steps and expected results for testing specific functionality.
- **Test Data**: The data used to execute test cases, which should be realistic and cover various scenarios.
- **Test Design Techniques**: Various methods used for comprehensive testing, such as equivalence partitioning, boundary value analysis, and state transition testing.

5. Test Execution
Test execution is the process of **running test cases** on a software application to verify that it meets predefined requirements and specifications. This phase involves executing the test scripts written during test design.

- **Purpose**: To identify bugs, errors, or other issues by comparing actual results with expected outcomes.
- **Process**: Involves selecting a subset of test cases, assigning them, executing tests, reporting defects, and capturing test status.
- **Tools**: Often utilizes tools to automate or facilitate the running of tests.
- **Importance**: Crucial for ensuring software quality, reliability, and compliance.
- **Features**: Validates software against requirements, identifies and reports defects, often used for regression testing, and requires setting up a test environment.

6. Reviews, Inspection and Auditing
These are methods used to examine software products and processes to identify issues and ensure quality and compliance.

- **Reviews**: Structured meetings where stakeholders examine software products to **identify defects**, gather feedback, and ensure the product meets specifications. They can be informal or formal. Goal is to evaluate suitability and identify discrepancies.
  - **Types**: Management Reviews, Technical Reviews, Walk-throughs.
- **Inspections**: Formal reviews led by trained moderators to **detect and identify software product anomalies**. More structured than reviews, involving a detailed examination of specific sections. Conducted by a team.
  - **Key Features**: Led by an impartial facilitator, involves a checklist, focuses on detecting defects early.
- **Auditing**: Involves an **independent examination** of software products and processes to assess **compliance** with applicable regulations, standards, guidelines, plans, and specifications. Used to evaluate overall quality and adherence to internal policies or external regulations.
  - **Types**: Internal Audits, External Audits, Compliance Audits, Process Improvement Audits.
    A table summarizes their **Purpose, Scope, and Frequency**: Reviews evaluate suitability (Broad scope, Regular frequency); Inspections detect anomalies (Specific sections, Regular during development); Audits assess compliance (Entire process/product, Less frequent/scheduled).

7. Regression Testing
Regression Testing is performed to ensure that **recent changes** (bug fixes, new features, code modifications) **do not adversely affect existing functionality**. It involves **re-running previously executed test cases** to verify the system still behaves as expected after changes.

- **Key Characteristics**: Purpose is to **detect unintended side effects** caused by code changes. Can be applied at unit, integration, or system levels. It is **often automated** because tests are frequently repeated.
- **When Performed**: After bug fixes, adding new features, refactoring code, or before releasing a new version.
- **Steps**: Identify affected areas, select relevant test cases, execute tests (manually or automated), analyze results and report failures.
- **Advantages**: Ensures application stability after changes, helps maintain software quality, reduces the risk of introducing new defects.
- **Challenges**: Can be time-consuming if manual, requires maintaining an up-to-date test suite, automation tools may need frequent updates.

8. Mutation Testing
Mutation Testing is a **white-box testing technique** used to **evaluate the quality of test cases**. It involves making small, deliberate changes (mutations) to the source code and then running the test suite to check if the tests can detect these changes.

- If the tests fail on the mutated code, the mutant is "killed," indicating the tests are effective.
- If the tests pass, the mutant "survives," indicating a potential weakness or incompleteness in the test suite.
- **How it Works**:
  - **Mutant Creation**: Small changes are introduced to the code (e.g., changing operators, modifying conditions, removing statements).
  - **Test Execution**: The existing test suite is run on the mutated code.
  - **Result Analysis**: Determine if mutants are "killed" or "survive".
- **Key Metrics**: **Mutation Score** is the percentage of mutants killed by the test suite; a higher score indicates better test quality.
- **Advantages**: Helps identify weak/incomplete test cases, encourages thorough testing sensitive to code changes, provides a quantitative measure of test effectiveness.
- **Challenges**: Computationally expensive due to the large number of mutants, requires significant effort to analyze results and improve tests, may produce false positives if mutations are unrealistic.

9. Object Oriented testing
Object-Oriented Testing is a specialized testing approach tailored for **object-oriented software systems (OOP)**. It is needed because OOP concepts like encapsulation, inheritance, and polymorphism may make traditional testing techniques insufficient. It focuses on verifying the **correctness of objects, classes, and their interactions**.

- **Types of Testing**:
  - **Class Testing**: Tests individual classes in isolation, focusing on methods, attributes, and their behavior.
  - **Integration Testing**: Tests interactions between classes and objects, verifying correct collaboration.
  - **Inheritance Testing**: Tests the behavior of derived classes, ensuring correct inheritance and method overriding.
  - **Polymorphism Testing**: Tests dynamic binding, ensuring overridden methods behave as expected.
  - **Scenario-Based Testing**: Tests real-world use cases involving multiple objects and their interactions.
- **Challenges**:
  - **Encapsulation**: Limited visibility into private members makes testing harder.
  - **Inheritance**: Changes in base classes can affect derived ones, requiring extensive retesting.
  - **Polymorphism**: Dynamic method binding complicates predicting runtime execution.
  - **State Dependencies**: Objects often maintain internal state, making isolation difficult.
- **Techniques**: Random Testing, Fault-Based Testing, Scenario-Based Testing, Use Case Testing.
- **Advantages**: Aligns with OOP principles (modularity, reuse), facilitates testing complex systems, encourages early detection of design flaws.
- **Challenges**: Requires deep understanding of OOP concepts, test case design can be complex due to inheritance/polymorphism, difficult to isolate objects for unit testing due to dependencies.

10. Testing Web based System
Testing web-based systems is crucial because they run on browsers, interact over the internet, and must function correctly across various devices, browsers, and network conditions while maintaining performance, security, and usability. It involves multiple types of testing.
Types of testing for web-based systems mentioned include:

- Functional Testing
- Cross Browser Testing
- Cross Platform testing
- Performance Testing
- Security Testing
- Usability Testing
- Compatibility Testing
- Database Testing
- API Testing
- Accessibility Testing
- Localization and Internalization Testing
- Regression Testing

11. Mobile App testing
Mobile app testing is the process of **evaluating mobile applications** for functionality, usability, performance, and security across various devices and operating systems (Android, iOS). This ensures the app meets user expectations and works seamlessly on different platforms.
Types of mobile app testing:

- **Functional Testing**: Verifies that all app features work as intended (login, data entry, search, etc.).
- **Performance Testing**: Assesses how the app performs under different load conditions, such as varying network bandwidth and user traffic.
- **Security Testing**: Identifies potential security vulnerabilities to protect against data breaches and unauthorized access.
- **Usability Testing**: Ensures the app is intuitive, easy to use, and provides a good user experience.
- **Compatibility Testing**: Checks app compatibility across different devices, operating systems, and screen resolutions.
  Steps in Mobile App Testing:
- Planning and Preparation
- Test Case Development
- Test Execution
- Defect Reporting and Management

- Regression Testing
- Reporting and Review

12. Mobile test Automation and tools
Mobile test automation involves using specialized tools and scripts to **automatically test mobile applications** across various devices and platforms. It simulates user interactions like tapping, scrolling, and swiping to evaluate functionality, performance, and usability. The primary goal is to ensure quality and reliability by detecting defects and preventing regressions early.
Benefits of Mobile Test Automation:

- **Efficiency and Speed**: Automates repetitive tasks, reducing testing time and enabling faster releases.
- **Cost-Effectiveness**: Leads to long-term savings by reducing manual effort despite initial setup costs.
- **Accuracy**: Minimizes human errors.
- **Scalability**: Facilitates testing across multiple devices/platforms simultaneously.
  How Mobile Test Automation Works:
- Script Creation: Developing test scripts that simulate user interactions.
- Test Execution: Running these scripts on various devices and platforms.
- Result Analysis: Reviewing test results to identify defects.
  Tools for Mobile Test Automation:
- **Appium**: Open-source, supports multiple platforms (Android/iOS).
- **Espresso**: Designed specifically for Android.
- **XCUITest**: Used for automating iOS app testing.
- **BrowserStack**: Cloud-based access to real devices for compatibility testing.
- **App Automate**: Provides on-demand access to real devices for automated testing.

13. DevOps Testing
DevOps testing is the **integration of testing processes into the DevOps lifecycle**, which combines development (Dev) and operations (Ops) to improve collaboration, efficiency, and software quality. It involves **continuous and automated testing** throughout the entire Software Development Lifecycle (SDLC), ensuring software is delivered quickly and reliably without compromising quality.
Key Features of DevOps Testing:

- **Continuous Testing**: Testing is performed at every stage of the SDLC, providing immediate feedback on code quality.
- **Automation**: Utilizes automated testing tools to accelerate the execution of various test types.
- **Shift-Left Approach**: Introduces testing early in the development process to catch defects sooner, reducing the time and cost of fixing bugs.
- **Collaboration and Communication**: Fosters teamwork among development, testing, and operations teams for shared quality responsibility.
- **Real-Time Monitoring and Feedback**: Implements continuous monitoring for quick identification of production issues.
  Types of testing integrated in DevOps:
- Unit Testing
- Component Testing
- Integration Testing
- API Testing

14. Cloud and Big Data Testing

- **Cloud Testing**: Refers to testing software applications **hosted on cloud computing platforms**. It evaluates the **scalability, reliability, and performance** of cloud-based applications across various cloud environments (public, private, hybrid). It leverages cloud infrastructure to simulate real-world conditions.
  - **Key Aspects**: Scalability Testing, Load Testing, Security Testing (focusing on data protection in cloud environments), Compatibility Testing (across different cloud platforms).
  - **Approaches**:
    - **Testing of the Whole Cloud**: Treats the cloud as a single entity, testing its overall features and infrastructure.
    - **Testing Within a Cloud**: Conducted internally within the cloud environment, focusing on the application's performance and functionality hosted there.
    - **Testing Across the Clouds**: Testing applications across different cloud environments (public, private, hybrid) to ensure consistency and identify compatibility issues.
    - **SaaS Testing in the Cloud**: Involves functional and non-functional testing based on system requirements for SaaS applications.
- **Big Data Testing**: Involves **validating the quality and functionality of big data applications**. It ensures big data systems handle large volumes of structured and unstructured data efficiently.
  - **Focus**: Data quality, schema integrity, pipeline performance, and algorithm accuracy .
  - **Aspects**: Data Quality Testing , Schema Testing , Pipeline Testing , Algorithm Testing .

Module - 6

Here is a comprehensive structured overview of the topics discussed in the provided sources, organized according to the syllabus points you provided:

Software Maintenance

- Software maintenance is the **general process of changing a system after it has been delivered**.
- It is the process of **modifying and updating a software system after it has been delivered** to the customer.
- Maintenance is **necessary to ensure that the software continues to meet the needs of the users** over time.
- The goals of software maintenance are **to keep the software system working correctly, efficiently, and securely** and **to ensure that it continues to meet the needs of the users**. This includes fixing bugs, adding new features, improving performance, and updating the software to work with new hardware or software systems.

Types of Maintenance

The sources discuss software maintenance types in two ways: first listing three types based on purpose (Fault repairs, Environmental adaptation, Functionality addition) and then listing four types often referred to in practice (Corrective, Adaptive, Perfective, Preventative). Both are covered below.

Three Types based on Purpose:

- **Fault repairs:** This involves **fixing bugs and vulnerabilities**. Coding errors are relatively cheap to correct, design errors are more expensive, and requirements errors are the most expensive to repair because they may require extensive system redesign. An example is investigating and fixing a crash in a mobile app caused by a null

pointer exception and releasing a patch. This aligns with fixing errors, bugs, or defects in the software.

- **Environmental adaptation:** This involves **adapting the software to new platforms and environments**. This type of maintenance is required when aspects of the system's environment, such as hardware, the platform operating system, or other support software, change. Application systems may need modification to cope with these changes. An example is updating a mobile app to ensure compatibility with the latest operating system version, such as modifying notification APIs or adjusting UI elements. This aligns with modifying the software to work in a new environment or with new technologies.
- **Functionality addition:** This involves **adding new features and supporting new requirements**. This is necessary when system requirements change in response to organizational or business change. The scale of required changes can often be much greater than for other types of maintenance. An example is adding a feature to categorize tasks or a dark mode to a mobile app based on user requests. This aligns with improving the software's performance, usability, or functionality based on user feedback.

Four Types (commonly referred to):

- **Corrective maintenance:** This is universally used to refer to **maintenance for fault repair**. It involves **fixing errors and bugs** in the software system. It is necessary when something goes wrong, including faults and errors. These issues can have a widespread impact and must be addressed quickly.
- **Adaptive maintenance:** This **sometimes means adapting to a new environment** and **sometimes means adapting the software to new requirements**. It involves **modifying the software system to adapt it to changes in the environment**, such as changes in hardware, software, government policies, and business rules. It is done to keep up with customer needs and market changes and demands.
- **Perfective maintenance:** This **sometimes means perfecting the software by implementing new requirements**; in other cases, it means **maintaining the functionality but improving its structure and performance**. It involves **improving functionality, performance, and reliability, and restructuring the software system to improve changeability**. It is done to boost performance and improve the software overall.
- **Preventative Maintenance:** This involves **making necessary changes, upgrades, adaptations and more to prevent future problems**. Preventative maintenance may address small issues that currently lack significance but could become larger problems in the future. These are called **latent faults** that need detection and correction to prevent them from becoming effective faults. An example is refactoring code, updating dependencies, and writing better documentation to ensure an app remains easy to modify and extend. This aligns with making changes to prevent future problems and improve maintainability.

Software Configuration Management (SCM) – Overview

- SCM is essentially a **set of practices used to track and control changes** made to software during its development lifecycle.
- It's like keeping a detailed log of everything that goes into the software: **who made the changes, why they were made, and how they affect the overall program**.
- It is a **systematic process used in software engineering to manage, organize, and control changes** in documents, codes, and other entities during the Software Development Life Cycle.
- The primary goal of SCM is to **increase productivity with minimal mistakes**.

SCM Tools / Activities

The configuration management of a software system product involves four closely related activities:

- **Version control:** This involves **keeping track of the multiple versions** of system components and ensuring that **changes made by different developers do not interfere with each other**. It tracks changes to source code and other artifacts over time. It enables collaboration by allowing multiple developers to work on the same project simultaneously without conflicts.
- **System building:** This is the process of **assembling program components, data, and libraries, then compiling and linking** these to create an executable system. This is also referred to as **Build Automation** and involves automating the process of compiling code, running tests, and packaging software for deployment.
- **Change management:** This involves **keeping track of requests for changes** to delivered software from customers and developers. It includes **working out the costs and impact** of making changes and **deciding if they should be implemented**. It involves managing requests for changes, tracking their status, and ensuring proper approval workflows. It also maintains a history of all changes made to the software.
- **Release management:** This involves **preparing software for external release** and **keeping track of the system versions that have been released** for customer use. It involves managing the release of software versions to different environments (e.g., development, testing, production). It ensures that the correct version of the software is deployed.

Additional SCM Activities:

- **Configuration Auditing:** This verifies that the software meets its requirements and adheres to defined standards. It provides traceability of changes and configurations.
- **Baseline Management:** This involves establishing and managing baselines (snapshots of the system at specific points in time) for reference and rollback purposes.
- **Collaboration and Workflow:** This facilitates communication and coordination among team members. It supports branching and merging strategies for parallel development.

Software Reengineering

- Software Reengineering is the process of **analyzing, redesigning, and modifying existing software systems** to improve their quality, performance, maintainability, or functionality.
- It's essentially **giving old software a makeover** to make it more efficient, effective, and up-to-date.
- The goals are **to improve the quality and maintainability** of the software system while **minimizing the risks and costs** associated with redevelopment from scratch.
- An example case is upgrading an old COBOL payroll system to Java with improved structure and updated data handling.

Software Reengineering Process:

The process involves several steps, often applied in a case scenario like the COBOL to Java payroll system upgrade:

- **Source code translation:** Using a translation tool to convert the program from an old language to a modern version of the same language or a different language. Example: Automatically convert COBOL code to Java code.
- **Reverse Engineering:** Analyzing the program to extract high-level details (e.g., data flow, business rules). This helps document its organization and functionality and is usually completely automated. Example: Create system documentation detailing payroll calculation and storage.
- **Program structure improvement:** Analyzing and modifying the control structure to make it easier to read and understand. This can be partially automated but may require manual intervention. Example: Refactor Java code by improving control flow and removing redundant code, simplifying complex nested if statements.
- **Program modularization:** Grouping related parts of the program together and removing redundancy. This may involve architectural refactoring. This is a manual process. Example: Split the system into independent modules like tax calculation and employee management, creating separate classes.
- **Data reengineering:** Changing the data processed by the program to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure. It usually involves data cleanup (finding/correcting mistakes, removing duplicate records), which can be very expensive and prolonged. Example: Update from flat-file storage to a relational database (MySQL), migrating employee records.
- **Reengineered Program and Data:** Delivering the final system integrated with modern databases. Example: The new Java-based payroll system integrated with MySQL.

Reverse Engineering

- Reverse Engineering is the process of **deconstructing a system or object to understand its design, functionality, and inner workings**.
- It involves **dismantling a product to unveil its internal mechanisms** so engineers and other professionals can understand how it works.

Reverse Engineering Process:

The process typically involves the following steps:

- **Gather Information:** Collecting critical information about the product before dismantling. This involves measuring dimensions, identifying source designs, and deducing original coding.
- **Produce a Model:** Based on the gathered information, creating a sketch or model, often a 3D CAD model if hardware embedded. Examining the model can help understand the product's design purpose.
- **Dissect the Object:** Conducting a systematic disassembly, often layer by layer, following a predetermined process. Parts can be organized for easier reassembly. Sequential dismantling assists in inspecting, measuring, and assessing individual parts to evaluate functionalities.
- **Evaluate the Product:** After separating components, evaluating each part to diagnose specific issues and resolve errors. Information generated helps achieve design goals. Observations can be recorded and documented to recommend improvements. Scanning the product sketch to a computer can help when rebuilding.
- **Reconstruct the Item:** During rebuilding, changes can be made and tested to improve the product. This may generate an improved version. If the goal is to reconstruct the object, components are combined to recreate the product.

Software Reuse

- Software reuse refers to the practice of **using existing software components, artifacts, or knowledge** in the development of new software systems.
- Instead of building every part from scratch, developers **leverage previously created and tested elements to save time, reduce costs, and improve quality**.
- This concept is **central to modern software engineering practices** as it promotes efficiency, consistency, and scalability.

Types of Software Reuse:

The sources list several types of software reuse:

- **Code Reuse:** The most common form, involving reusing source code, libraries, or modules. Examples: Using open-source libraries (e.g., React) or reusing utility functions.
- **Design Reuse:** Reusing architectural patterns, design templates, or frameworks. Examples: Applying the Model-View-Controller (MVC) pattern or using microservices architecture templates.
- **Requirements Reuse:** Reusing documented requirements or specifications from previous projects. This ensures consistency and reduces effort.
- **Test Reuse:** Reusing test cases, test scripts, or testing frameworks. Example: Leveraging automated test suites for regression testing.
- **Documentation Reuse:** Reusing user manuals, technical documentation, or guidelines from prior projects. This can significantly reduce the effort for new documentation.
- **Component Reuse:** Listed as a type, implying the reuse of larger, self-contained software components. (Details beyond the listing are not provided in the sources).

Module - 7

Here is a comprehensive overview of the topics from the provided sources, structured according to the syllabus you provided: Product and Process Metrics, Quality Standards Models (ISO, TQM, Six-Sigma), Process Improvement Models (CMM & CMMI), Quality Control and Quality Assurance, Quality Management, Quality Factors, and Methods of Quality Management.

These topics are part of Software Quality Assurance (SQA). SQA is a systematic process aimed at ensuring software products meet specified requirements and are free from defects. It encompasses the entire software development lifecycle, focusing on improving the process to prevent errors and deliver high-quality software. SQA involves monitoring and auditing processes, methodologies, and standards to ensure compliance with established quality benchmarks. The goals of Quality Assurance include ensuring the software meets functional and non-functional requirements, minimizing risks, enhancing customer satisfaction, and maintaining consistency and repeatability in the development process.

1. Product and Process Metrics

Metrics are categories of measurements used to evaluate and improve the quality of software development and the final software. They are divided into Process Metrics and Product Metrics.

- **Process Metrics** focus on the processes involved in software development. They measure the efficiency, effectiveness, and quality of the activities and workflows used to develop software. The primary goal is to improve the development process itself. Process metrics are typically measured throughout the development lifecycle. Their impact is long-term, focusing on process improvement.
  - **Purpose of Process Metrics:** To identify bottlenecks or inefficiencies in the development process, track improvements over time as process changes are implemented, ensure best practices are being followed consistently, and provide feedback for continuous process improvement.
  - **Examples of Process Metrics:**
    - **Defect Density:** Number of defects found per unit of code (e.g., per 1,000 lines of code). This helps assess the quality of the development process.
    - **Code Review Efficiency:** Percentage of defects identified during code reviews versus those found later. This indicates the effectiveness of the review process.
    - **Cycle Time:** The time to complete a phase (e.g., requirements to deployment). Shorter cycle times may indicate a more efficient process.
    - **Test Coverage:** The percentage of code covered by automated tests. Higher coverage generally indicates a more thorough testing process.
    - **Defect Removal Efficiency (DRE):** Percentage of defects found and fixed before release compared to the total. A high DRE suggests an effective defect detection and removal process.
    - **Change Failure Rate:** Percentage of changes resulting in failures or incidents. Lower rates indicate a more stable process.
- **Product Metrics** focus on the software product itself. They evaluate its quality, functionality, and performance. These metrics help ensure the final product meets user expectations and adheres to quality standards. Product metrics are measured primarily after the product is built or released. Their impact is immediate on product quality and user experience.
  - **Purpose of Product Metrics:** To assess the overall quality and reliability of the software product, ensure the product meets functional and non-functional requirements, identify areas needing improvement or additional testing, and provide feedback to developers and testers on the final product's quality.
  - **Examples of Product Metrics:**
    - **Defect Count:** Total number of defects found after release. Fewer defects indicate higher quality.
    - **Mean Time Between Failures (MTBF):** Average time between system failures. A higher MTBF indicates a more reliable product.
    - **Mean Time to Repair (MTTR):** Average time to fix an issue once discovered. Lower MTTRs suggest faster resolution.
    - **Performance Metrics:** Measures like response time, throughput, and resource utilization, assessing performance under various conditions.
    - **Usability Metrics:** Like task success rate, error rate, and user satisfaction, evaluating ease of use.

        - **Code Complexity:** Measures like cyclomatic complexity, assessing codebase complexity. Simpler code is generally easier to maintain and less error-prone.

2. Quality Standards Models (ISO, TQM, Six-Sigma)

A Quality Standard Model is a structured framework or set of guidelines that defines criteria, processes, and practices for ensuring quality. These models provide a benchmark for organizations to measure, evaluate, and improve their processes and outputs. In software engineering, they outline principles and practices for developing, testing, and maintaining high-quality software, serving as a reference point for establishing consistent, repeatable processes. Examples include ISO/IEC models, CMMI, Six Sigma, ISO 9001, TQM, and IEEE Standards.

- **ISO Quality Model**
    - The ISO Quality Model is a framework that defines and measures software quality by breaking it down into key characteristics. The most widely used version is **ISO/IEC 25010**, which replaced the older ISO/IEC 9126.
    - The core idea is that software quality is a combination of measurable attributes or characteristics describing how well the software performs in different areas.
    - **ISO/IEC 25010 - 8 Key Quality Characteristics:**
        1. **Functionality:** Does the software do what it's supposed to do?. Example: A calculator app correctly performs operations. Includes Correctness, Security, and Interoperability. Note Security is also listed as a separate characteristic.
        2. **Performance Efficiency:** How fast and resource-efficient is the software?. Example: A website loads quickly. Includes Response Time, Throughput, Resource Usage.
        3. **Compatibility:** Does the software work well with other systems, devices, or platforms?. Example: An app works on Windows and macOS.
        4. **Usability:** Is the software easy to use and understand?. Example: A mobile app has an intuitive interface. Includes Learnability, Operability, User Error Protection.
        5. **Reliability:** Does the software work consistently without crashing or failing?. Example: A banking app stays operational 24/7. Includes Fault Tolerance, Availability, Recoverability.
        6. **Security:** How well does the software protect data and prevent unauthorized access?. Example: An e-commerce site encrypts user passwords. Includes Confidentiality, Integrity, Authentication.
        7. **Maintainability:** Is the software easy to update, fix, or modify over time?. Example: Developers can quickly add features. Includes Modularity, Reusability, Analyzability.
        8. **Portability:** Can the software be easily moved or adapted to different environments?. Example: A game runs on Android and iOS. Includes Adaptability, Installability, Replaceability.
    - **How it Works:** Measure Quality (using metrics for sub-characteristics), Improve Software (identifying weaknesses), and Ensure Consistency (providing a common language for quality).
    - **Comparison (ISO/IEC 9126, ISO/IEC 25010, ISO 9001):**
        - ISO/IEC 9126 focused on software product quality and post-development evaluation with 6 characteristics, now replaced by ISO/IEC 25010.
        - ISO/IEC 25010 focuses on software product and system quality across the entire lifecycle with 8 characteristics, and is the current standard.
        - ISO 9001 focuses on organizational processes and quality management in all industries, emphasizing customer focus, continuous improvement, and compliance.

- **Six Sigma**
    - Six Sigma is a **data-driven methodology** aimed at improving quality by identifying and eliminating defects or errors in processes. It focuses on reducing variability to ensure consistency, leading to near-perfect quality. The term refers to a statistical concept where a process produces **no more than 3.4 defects per million opportunities (DPMO)**.
    - **Six Sigma seeks to:** Minimize defects, Improve process efficiency, and Enhance customer satisfaction.
    - **Principles:** Customer Focus, Data-Driven Decisions (using statistical analysis), Process Improvement (optimizing processes), and Continuous Improvement.
    - **DMAIC Framework:** A methodology for improving existing processes.
        - **Define:** Identify the problem and set goals based on customer needs.
        - **Measure:** Collect data on current process performance.
        - **Analyze:** Use statistical tools to identify root causes of defects/inefficiencies.
        - **Improve:** Implement solutions to address root causes and optimize the process.
        - **Control:** Monitor and sustain improvements.
    - **Levels (Belts):** Green Belt (team members), Black Belt (project leaders), Master Black Belt (mentors), Yellow Belt (support).
    - **Example:** A team aims to reduce post-release bugs by 50%. They Analyze past defect rates, identify poor code reviews as a cause, Improve by implementing stricter guidelines, and Control by monitoring future defect rates.

- **Total Quality Management (TQM)**
    - TQM is a **holistic approach** focused on continuous improvement, customer satisfaction, and the involvement of all stakeholders. In software engineering, it ensures processes and products meet or exceed customer expectations by fostering a culture of quality throughout the organization.
    - **TQM in software engineering emphasizes:** Customer-Centric Approach, Continuous Improvement, Employee Involvement, and Process Orientation.
    - **Principles:**
        - **Customer Focus:** Deliver software satisfying customer needs (e.g., using user surveys).
        - **Continuous Improvement:** Constant refinement of processes (e.g., using Agile methodologies).
        - **Employee Empowerment:** Active participation and ownership from all team members (e.g., brainstorming sessions).
        - **Process-Centric Approach:** Defining, documenting, and standardizing processes for consistency (e.g., coding standards, automated testing).
        - **Data-Driven Decision Making:** Decisions based on measurable data (e.g., tracking defect rates, user feedback).
        - **Leadership Commitment:** Management support, resources, and leading by example.
        - **Systematic Problem Solving:** Using structured methods like PDCA or Root Cause Analysis (RCA).
        - **Long-Term Thinking:** Focusing on sustainable improvements and robust processes.
    - **Implementation:** Includes detailed steps across the lifecycle: Requirement Gathering, Design and Development, Testing and Quality Assurance, Deployment and Maintenance, and a Feedback Loop.
    - **Challenges:** Cultural Resistance, Resource Constraints, Measurement Complexity, and Time-Consuming nature.
    - **Example:** A company uses TQM by conducting user interviews, implementing Agile, holding team meetings on bug reduction, introducing coding standards/reviews, and tracking defect/satisfaction rates, leading to reduced defects and improved retention.

3. Process Improvement Models (CMM & CMMI)

These are frameworks to assess and improve the maturity of processes within an organization, particularly in software development.

- **Capability Maturity Model (CMM)**
    - CMM assesses and improves process maturity. Process Maturity measures how structured, standardized, and capable a process is of delivering consistent and efficient results, involving documentation, controls, and continuous improvement mechanisms.
    - **Five Levels (Staged):**

1. **Initial:** Processes are ad hoc, chaotic, unpredictable.
2. **Repeatable:** Basic project management processes are established, enabling repeatable results.
3. **Defined:** Processes are well-documented, standardized, and integrated across the organization.
4. **Managed:** Processes are measured and controlled using quantitative data.
5. **Optimizing:** Continuous process improvement is emphasized through innovation and feedback.

- **Application:** Process Evaluation, Enhanced Project Management (consistent execution, risk reduction), Quality Improvement (standardizing processes), Optimized Resource Utilization, and Continuous Improvement.

- **Capability Maturity Model Integration (CMMI)**
  - CMMI is the current industry standard, largely replacing the older CMM.
  - **Comparison with CMM:**
    - **Scope:** CMM focuses only on software process improvement; CMMI covers software, systems, hardware, and services.
    - **Model Type:** CMM is prescriptive (rigid levels); CMMI is flexible (allows customization).
    - **Structure:** CMM follows staged levels; CMMI supports both staged (maturity levels) and continuous (capability levels) representations.
    - **Improvement Approach:** CMM focused on individual processes; CMMI is an integrated approach addressing multiple disciplines.
    - **Adoption:** CMM is older and largely replaced; CMMI is current and widely adopted.
  - **Capability Levels (0-3):** Describe the capability of individual processes.
    - **0 - Incomplete:** Process is not performed or only partially performed.
    - **1 - Performed:** Process is executed but may be unpredictable.
    - **2 - Managed:** Process is planned, monitored, controlled, and follows policies.
    - **3 - Defined:** Process is documented and tailored from organizational standards; best practices are followed.
  - **Maturity Levels (1-5):** Describe the maturity of an organization's overall processes (staged representation).
    - **1 - Initial:** Processes are ad hoc, chaotic, or unpredictable.
    - **2 - Managed:** Basic project management is established; processes are repeatable but may vary.
    - **3 - Defined:** Organization-wide standard processes are established and used consistently.
    - **4 - Quantitatively Managed:** Processes are measured and controlled using quantitative techniques.
    - **5 - Optimizing:** Continuous process improvement is implemented using innovation and feedback.

4. Quality Control and Quality Assurance

QA and QC are distinct but related concepts in quality management.

- **Quality Assurance (QA):**
  - QA is a **proactive, process-oriented** approach focused on preventing defects in the process used to make a product.
  - **Key Characteristics:** Ensures process quality, Prevents issues *before* they occur, Involves standards, procedures, and guidelines, Focuses on continuous improvement, Typically includes audits, training, process checklists, and reviews.
  - **Includes:** Writing coding standards, conducting code reviews, and defining testing protocols. QA is about "Doing things the right way".

- **Quality Control (QC):**
  - QC is a **reactive, product-oriented** approach focused on identifying defects in the final product.
  - **Key Characteristics:** Ensures product quality, Detects issues *after* they occur, Involves testing, inspection, and validation, Focuses on meeting quality specifications, Usually involves manual/automated testing.
  - **Includes:** Executing test cases, finding bugs, and reporting issues before a release. QC is about "Checking the Final Product".

- **QC vs QA Comparison:**
  - **Focus:** QA is on Process; QC is on Product.
  - **Objective:** QA is to Prevent Defects; QC is to Detect Defects.
  - **Type:** QA is Proactive; QC is Reactive.
  - **Performed by:** QA is by Process/QA team; QC is by Testers/Inspectors.
  - **Techniques:** QA uses Process audits, documentation; QC uses Testing, inspection.
  - **Outcome:** QA results in Improved process; QC results in Verified product.

5. Quality Management

Quality Management is the process of ensuring that products or services meet consistent standards and satisfy customer expectations. It is about making things right, keeping them right, and improving them over time.

- **Formula:** Quality Management = Plan it right (Planning) + Do it right (QA) + Check it right (QC) + Keep getting better (Improvement).
- **Four Parts:**
  1. **Quality Planning:** Deciding what "quality" means, setting goals, standards, and procedures *before* work begins (e.g., defining performance benchmarks).
  2. **Quality Assurance:** Creating and following processes to avoid mistakes, focusing on *how* the work is done (e.g., code reviews, CI/CD pipelines).
  3. **Quality Control:** Checking the final product to find and fix defects, focusing on *what* is produced (e.g., testing a software app).
  4. **Continuous Improvement:** Learning from mistakes and improving processes (e.g., PDCA, Six Sigma, Kaizen), collecting feedback for future releases.

6. Quality Factors

Quality Factors are the key characteristics that define how good a software product is. They are used to evaluate, measure, and ensure software quality from both a technical and user perspective.

- **Common Quality Factors:** Functionality, Performance (Efficiency), Reliability, Usability, Maintainability, Portability, Security, and Testability.
- **Detailed Aspects:**
  - **Functionality:** Correctness, Security, Interoperability.
  - **Performance Efficiency:** Response Time, Throughput, Resource Usage.
  - **Reliability:** Fault Tolerance, Availability, Recoverability.
  - **Usability:** Learnability, Operability, User Error Protection.
  - **Maintainability:** Modularity, Reusability, Analyzability.
  - **Portability:** Adaptability, Installability, Replaceability.
  - **Security:** Confidentiality, Integrity, Authentication.
  - **Testability:** How easy it is to test for bugs.

- **Quality Factors by McCall's:** Groups factors into three categories:
  - **Product Operations:** Correctness, Reliability, Efficiency, Integrity (Security), Usability.
  - **Product Revision:** Maintainability, Flexibility, Testability.
  - **Product Transition:** Portability, Reusability, Interoperability.

7. Methods of Quality Management

Various methods and models are used within Quality Management to achieve quality goals. These include:

- **PDCA Cycle** (Plan–Do–Check–Act).
- **Six Sigma**.
- **Statistical Process Control (SPC)**.
- **Total Quality Management (TQM)**.
- **ISO 9001** (Quality Standards).
- **Failure Mode and Effects Analysis (FMEA)**.

These concepts collectively form the foundation for ensuring high-quality software products and processes in software engineering.