Software Engineering

Topic 1: Introduction to Software Engineering

- **Definition**: Software Engineering involves designing, building, and maintaining large software systems using systematic, engineering approaches.
- **Key Concepts**:
    - **Software**: Instructions that make a computer work, can be categorized as:
        - **System Software**: Operates the computer hardware.
        - **Application Software**: Performs specific tasks for users.
    - **Software Engineering**: Theories, methods, and tools for developing, managing, and evolving software products.
    - **Important Definitions**:
        - **I Sommerville**: Concerned with theories, methods, and tools for developing software products.
        - **B.W. Boehm**: Practical application of scientific knowledge in software design and construction.
        - **IEEE Standard 610.12**: Systematic, disciplined, quantifiable approach to software development and maintenance.

Importance of Software Engineering

- **Economic Impact**: Developed nations' economies depend heavily on software.
- **Software-Controlled Systems**: Increasingly prevalent in various sectors.
- **Expenditure**: Significant fraction of GNP in developed countries.

Characteristics of a Good Software Engineer

- **Systematic Methods**: Familiarity with software engineering principles.
- **Domain Knowledge**: Good technical knowledge of the project range.
- **Programming Skills**: Proficiency in coding.
- **Communication Skills**: Oral, written, and interpersonal skills.

Types of Software Products

- **Generic Products**: Stand-alone systems sold on the open market (e.g., databases, word processors).
- **Customized Products**: Systems commissioned by a specific customer, tailored to their needs (e.g., control systems for electronic devices).

Project-based vs. Product-based Software Engineering

- **Project-based**: Develops software based on external client requirements, with potential changes during the development process.

- **Product-based**: Focuses on a business opportunity identified by a company, rapidly developed to capture market share.

Software Product Line

- **Definition**: A set of software products sharing a common core, with customer-specific adaptations.
- **Platform Example**: Facebook, which provides a set of functionalities and supports the creation of additional applications.

Software Execution Models

- **Stand-alone**: Executes entirely on the customer's computers.
- **Hybrid**: Partially on customer's computers, partially on developer's servers.
- **Software Service**: Entirely on developer's servers, accessed via browser or app.

Developing Product Vision

- **Definition**: Simple statements defining the essence of the product.
- **Moore's Vision Template**:
  - **Target Customer**: Who will use the product.
  - **Need or Opportunity**: What problem the product addresses.
  - **Product Category**: What type of product it is.
  - **Key Benefit**: Why customers should buy it.
  - **Primary Differentiation**: What sets it apart from competitors.

Information Sources for Product Vision

- **Domain Experience**: Developers' work experience.
- **Product Experience**: Users' insights on better functionality.
- **Customer Experience**: Discussions with prospective customers.
- **Prototyping**: Developing prototypes to explore ideas.

Product Management Concerns

- **Customer Understanding**: Regular contact with customers to understand their needs.
- **Business Goals**: Ensuring the software meets the company's business goals.
- **Technology Issues**: Awareness of important technology issues for customers.

Programming vs. Software Engineering

- **Time Management**: Engineers focus on time and the need for change.
- **Scale and Efficiency**: Concerned with large-scale software and organizational efficiency.
- **Trade-offs**: Making complex decisions with high-stakes outcomes.

Software Systems

- **Abstract and Intangible**: Not governed by physical laws, can become complex and expensive to change.
- **Common Issues**:
  - **Increasing Demands**: Rapid development and delivery of complex systems.
  - **Low Expectations**: Drifting into software development without SE methods.

Software Engineering as a Discipline

- **Not Just Technical**: Includes project management, tool development, and organizational constraints.
- **Systematic Approach**: Produces high-quality software within schedule and budget.

Essential Attributes of Good Software

- **Maintainability**: Ability to evolve with changing needs.
- **Dependability and Security**: Reliability, security, and safety.
- **Efficiency**: Optimal use of system resources.
- **Acceptability**: User-friendly and compatible with other systems.

Importance of Software Engineering Methods

- **Economic Efficiency**: Cheaper in the long run due to reduced change costs.
- **Software Process**: Systematic sequence of activities (specification, development, validation, evolution).

Software Engineering vs. Computer Science

- **Computer Science**: Focuses on theories and fundamentals.
- **Software Engineering**: Practicalities of developing and delivering useful software.
- **Systems Engineering**: Concerned with the development of complex systems, including hardware and process design.

General Issues in Software Development

- **Heterogeneity**: Building flexible, dependable software.
- **Business and Social Change**: Rapid development and adaptation of software.
- **Security and Trust**: Ensuring software security and maintaining information security.

Lecture 2

Recap: What is Software Engineering?

- **Software**: Instructions given to a computer (computer programs), encompassing both system software and application software.
- **Software Engineering**: Theories, methods, and tools for developing, managing, and evolving software products.

Software Costs

- Software often costs more than hardware.
- Maintenance costs usually exceed development costs, especially for long-lived systems.
- Focuses on cost-effective software development.

Software Process Activities

1. **Software Specification**: Defining the software to be produced and the constraints on its operation.
2. **Software Development**: Designing and programming the software.
3. **Software Validation**: Ensuring the software meets customer requirements.
4. **Software Evolution**: Modifying software to reflect changing customer and market needs.

General Issues Affecting Software Engineering

- **Heterogeneity**: Need for systems to operate across diverse networks and devices.
- **Business and Social Change**: Rapid adaptation to new technologies and market demands.
- **Security and Trust**: Ensuring reliable and secure software.
- **Scale**: Developing software for a wide range of scales, from small embedded systems to large cloud-based systems.

Key Challenges in Software Engineering

- **Increasing Diversity**: Coping with varied system requirements.
- **Reduced Delivery Times**: Meeting tight deadlines.
- **Developing Trustworthy Software**: Ensuring reliability and security.

Costs of Software Engineering

- Development costs: Approximately 60% of total costs.
- Testing costs: Approximately 40% of total costs.
- Custom software often has higher evolution costs than development costs.

Best Software Engineering Techniques and Methods

- Different techniques are suitable for different systems (e.g., prototypes for games, complete specifications for safety-critical systems).
- No single method is universally the best.

Impact of the Web on Software Engineering

- Availability of software services and highly distributed service-based systems.
- Advances in programming languages and software reuse.

Application Types

1. **Stand-alone Applications**: Run on local computers without network connection.

2. **Interactive Transaction-based Applications**: Accessed remotely, e.g., web applications.
3. **Embedded Control Systems**: Control hardware devices.
4. **Batch Processing Systems**: Process data in large batches.
5. **Entertainment Systems**: Designed for personal use and entertainment.
6. **Modeling and Simulation Systems**: Used for scientific and engineering purposes.
7. **Data Collection Systems**: Collect data from the environment using sensors.
8. **Systems of Systems**: Composed of multiple integrated software systems.

Software Engineering Fundamentals

1. **Managed Development Process**: Use an understood process tailored to the software type.
2. **Dependability and Performance**: Essential for all systems.
3. **Specification and Requirements Management**: Clear understanding and management of what the software should do.
4. **Software Reuse**: Reuse existing software when possible.

Software Engineering Ethics

- Wider responsibilities beyond technical skills.
- **Honesty and Ethical Responsibility**: Essential for professional respect.
- **Ethical Behavior**: More than following the law, involves morally correct principles.

Professional Responsibility Issues

1. **Confidentiality**: Respecting the confidentiality of employers or clients.
2. **Competence**: Accurately representing one's level of competence.
3. **Intellectual Property Rights**: Awareness and protection of intellectual property laws.
4. **Computer Misuse**: Avoiding misuse of technical skills and others' computers.

Lecture 3: Software Process

The Software Process

- **Definition**: A structured set of activities required to develop a software system.
- **Software Process Model**: An abstract representation of a process from a particular perspective.

Key Process Activities

1. **Specification**: Defining what the system should do.
2. **Design and Implementation**: Defining the system's organization and implementing it.
3. **Validation**: Ensuring the system meets customer requirements.
4. **Evolution**: Modifying the system in response to changing needs.

Process Descriptions

- **Products**: Outcomes of a process activity.
- **Roles**: Responsibilities of the people involved.
- **Pre- and Post-conditions**: Statements true before and after a process activity.

Plan-driven and Agile Processes

- **Plan-driven Processes**: Activities planned in advance, progress measured against the plan.
- **Agile Processes**: Incremental planning, easier to accommodate changes.
- **Practical Approach**: Most processes blend plan-driven and agile elements.

Software Process Models

1. **Waterfall Model**
   - **Phases**: Requirements definition, system and software design, implementation and unit testing, integration and system testing, operation and maintenance.
   - **Drawbacks**: Difficulty accommodating changes once the process is underway.
   - **Usage**: Suitable for well-understood requirements, large systems engineering projects.
2. **Incremental Development**
   - **Concurrent Activities**: Specification, development, validation interleaved.
   - **Benefits**: Reduced cost of change, easier customer feedback, rapid delivery.
   - **Problems**: Less visible process, structure degrades with changes.
3. **Integration and Configuration**
   - **Approach**: Systems integrated from existing components.
   - **Types of Reusable Software**: Stand-alone systems, object collections, web services.
   - **Stages**: Requirements specification, software discovery and evaluation, requirements refinement, application system configuration, component adaptation and integration.
   - **Advantages**: Reduced costs, faster delivery.
   - **Disadvantages**: Requirements compromises, loss of control over evolution.

Software Process Activities

1. **Specification**: Establishing required services and constraints.
   - **Requirements Engineering Process**:
     - Elicitation and analysis
     - Specification
     - Validation
2. **Design and Implementation**: Converting specification into an executable system.
   - **Design Activities**: Architectural design, database design, interface design, component selection and design.

3. **Validation**: Verification and validation to ensure conformity with specifications.
   - **Testing Stages**: Component testing, system testing, customer testing.
4. **Evolution**: Modifying software to meet new requirements.
   - **Process**: Assess existing systems, define requirements, propose changes, modify systems.

Coping with Change

- **Inevitability of Change**: Large projects face business changes, new technologies, and platform changes.
- **Cost of Change**: Includes rework and implementing new functionality.

Approaches to Reduce Rework Costs

1. **Change Anticipation**: Activities to anticipate changes (e.g., prototyping).
2. **Change Tolerance**: Process designed to accommodate changes at low cost (e.g., incremental development).

Coping with Changing Requirements

1. **System Prototyping**: Quickly developing a system version to check requirements and design feasibility.
2. **Incremental Delivery**: Delivering system increments for customer feedback and experimentation.

Benefits and Process of Prototyping

- **Benefits**: Improved usability, closer match to user needs, improved design quality, reduced development effort, improved maintainability.
- **Process**: Establish objectives, define functionality, develop prototype, evaluate prototype.

Throw-away Prototypes

- **Discard After Development**: Not suitable as a production system basis due to lack of non-functional requirements, documentation, and structure degradation.

Incremental Delivery

- **Approach**: Development and delivery broken into increments, prioritized by user requirements.
- **Advantages**: More realistic evaluation, supports change.
- **Challenges**: Difficult for replacement systems.

Process Improvement

- **Objective**: Enhance software quality, reduce costs, accelerate development.
- **Approaches**:
  - **Process Maturity Approach**: Focus on improving process and project management.
  - **Agile Approach**: Focus on iterative development and reducing process overheads.

Process Improvement Cycle

1. **Analyze**: Assess current process, identify weaknesses.
2. **Measure**: Measure attributes of the process or product.
3. **Change**: Propose and implement process changes.

Lecture 4: Rapid Software Development

- **Agile Development**: Emerged in the late 1990s to reduce delivery time for working software systems.
- **Characteristics**:
  - Interleaved program specification, design, and implementation.
  - Developed in versions or increments with stakeholder involvement.
  - Extensive use of tools, such as automated testing.

Plan-driven vs. Agile Development

- **Plan-driven Development**:
  - Based around separate development stages with predefined outputs.
  - Allows for iteration within activities.
- **Agile Development**:
  - Interleaves specification, design, implementation, and testing.
  - Outputs are decided through negotiation during development.

Agile Methods

- **Focus on Code**: Rather than design.
- **Iterative Approach**: Quick delivery and evolution of working software.
- **Reduced Overheads**: Limits documentation, quick response to changes.

Principles of Agile Methods

1. **Customer Involvement**: Continuous involvement and feedback.
2. **Incremental Delivery**: Software developed in increments with customer-specified requirements.
3. **People Over Process**: Emphasize the skills and creativity of the development team.
4. **Embrace Change**: Design systems to accommodate changes.
5. **Maintain Simplicity**: Focus on simplicity in software and development processes.

Agile Project Management

- **Responsibility**: Ensures software is delivered on time and within budget.
- **Approach**: Adapted to incremental development and agile practices.

Scrum

- **Phases**:
  - Outline planning and architecture design.
  - Sprint cycles to develop increments.
  - Project closure and documentation.
- **Terminology**:
  - **Development Team**: Self-organizing group responsible for development.
  - **Product Backlog**: List of tasks or features to be developed.
  - **ScrumMaster**: Ensures Scrum process is followed and removes impediments.
  - **Sprint**: Iteration of development, usually 2-4 weeks.
  - **Velocity**: Measure of how much work a team can handle in a sprint.

Scaling Agile Methods

- **Challenges**: Improved communication and coordination in larger projects.
- **IBM's Agility at Scale Model**: Core agile development, disciplined agile delivery, and scaling factors such as team size and geographic distribution.

Software Project Management

- **Activities**: Planning, estimating, scheduling, and resource allocation.
- **Success Criteria**: Timely delivery, budget adherence, meeting customer expectations, and maintaining a cohesive team.
- **Distinctions**: Intangible products, novel projects, variable processes.

Risk Management

- **Importance**: Due to uncertainties in software development.
- **Process**:
  - **Risk Identification**: Identifying potential risks.
  - **Risk Analysis**: Assessing likelihood and impact.
  - **Risk Planning**: Developing plans to avoid or minimize risks.
  - **Risk Monitoring**: Continuous monitoring of risks.

Managing People

- **Factors**:
  - **Consistency**: Equal treatment without favoritism.
  - **Respect**: Acknowledge different skills and contributions.
  - **Inclusion**: Involve all team members.
  - **Honesty**: Transparency about project status.

Motivating People

- **Motivation Factors**: Basic needs, personal needs, and social needs.
- **Role of Manager**: Organize work and environment to encourage effective work.

Teamwork

- **Group Activity**: Most software projects require teamwork.
- **Cohesive Groups**: Motivated by group success and individual goals.
- **Effective Team**: Balance of technical skills and personalities.

Project Scheduling

- **Process**: Organizing work into tasks, estimating time and resources, assigning people.
- **Challenges**: Estimating difficulty, managing productivity, and allowing for contingencies.
- **Tools**: Graphical notations like bar charts to illustrate schedules.

Lecture 5: Requirements Engineering

Requirements Engineering

- **Definition**: The process of establishing services required from a system and constraints on its operation and development.
- **System Requirements**: Descriptions of system services and constraints generated during the requirements engineering process.
- **Requirements Engineering (RE)**: The process of finding out, analyzing, documenting, and checking services and constraints.

Requirements Abstraction

- **Purpose**: Define needs abstractly for competitive bidding and detailed system definition by the contractor.

Types of Requirements

1. **User Requirements**: Natural language statements plus diagrams, written for customers.
2. **System Requirements**: Detailed descriptions of system functions, services, and constraints, part of the contract between client and contractor.

User and System Requirements

- **User Requirements**: Broad statements, high-level, understandable by customers.
- **System Requirements**: Detailed, specific, technical descriptions for developers.

Feasibility Studies

- Conducted early in the RE process to answer:
    1. Does the system align with organizational objectives?
    2. Can it be implemented within schedule and budget?

3. Can it integrate with other systems?

System Stakeholders

- **Types**: End users, system managers, system owners, external stakeholders.

Functional and Non-Functional Requirements

1. **Functional Requirements**: Services the system should provide, reactions to inputs, behavior in situations, and what the system should not do.
2. **Non-Functional Requirements**: Constraints on system services/functions, such as timing, standards, and constraints on the development process.

Functional Requirements

- **Characteristics**:
  - Describe system services.
  - Depend on software type, users, and system environment.
  - High-level for user requirements, detailed for system requirements.
- **Issues**: Imprecision and ambiguity leading to varied interpretations.

Requirements Completeness and Consistency

- **Complete**: Descriptions of all required facilities.
- **Consistent**: No conflicts in descriptions.
- **Challenge**: Achieving both due to complexity.

Non-Functional Requirements

- Define properties and constraints like reliability, response time, storage.
- **Types**:
  - Product requirements: behavior of delivered product.
  - Organizational requirements: consequences of policies and procedures.
  - External requirements: factors external to the system and development process.

Metrics for Non-Functional Requirements

- **Examples**:
  - Speed: Transactions/second, response time.
  - Size: Mbytes, ROM chips.
  - Ease of use: Training time, help frames.
  - Reliability: Mean time to failure, availability.
  - Robustness: Time to restart after failure.
  - Portability: Target-dependent statements, target systems.

Requirements Engineering Processes

- Vary based on application domain, people, and organization.
- **Common Activities**:
  - Requirements elicitation and analysis.
  - Requirements specification.
  - Requirements validation.
- **Iterative Activity**: Processes are interleaved.

Spiral View of Requirements Engineering

- **Activities**:
  - Requirements elicitation.
  - Requirements specification.
  - Requirements validation.
  - Prototyping and reviews.

Requirements Elicitation and Analysis

- **Process**: Deriving system requirements through observation, discussions, task analysis.
- **Participants**: End-users, managers, maintenance engineers, domain experts, trade unions.
- **Stages**:
  1. Requirements discovery.
  2. Requirements classification and organization.
  3. Requirements prioritization and negotiation.
  4. Requirements specification.

Requirements Elicitation Techniques

- **Interviewing**: Formal/informal, open/closed.
- **Observation or Ethnography**: Observing social and organizational factors, deriving requirements from actual work practices.

Requirements Specification

- **Activity**: Translating analyzed information into a documented set of requirements.
- **Types**:
  - User requirements: Understandable by end-users and customers.
  - System requirements: Detailed technical information.
- **Purpose**: Part of a contract for system development.

Requirements and Design

- **Principle**: Requirements state what the system should do, design describes how it does it.

- **Practice**: Inseparable due to interactions with system architecture, interoperability, and domain requirements.

Use Cases

- Describe interactions between users and a system using graphical models and structured text.
- **Components**:
  - Actors: Human or system participants.
  - Interactions: Named ellipses representing classes of interactions.
- **Purpose**: Document all possible interactions with the system.

Software Requirements Document

- **Purpose**: Official statement of system requirements, including user and system requirements.
- **Users**: System customers, engineers, managers, maintenance engineers.

Requirements Validation

- **Process**: Checking that requirements define the desired system.
- **Checks**:
  1. Validity: Support customer needs.
  2. Consistency: No conflicts.
  3. Completeness: All required functions included.
  4. Realism: Implementable within constraints.
  5. Verifiability: Can be checked.

Requirements Validation Techniques

- **Techniques**:
  - Requirements reviews.
  - Prototyping.
  - Test-case generation.

Requirements Change

- **Reasons**:
  1. Changes in business and technical environment.
  2. Differing customer and user requirements.
  3. Diverse user community with conflicting priorities.
- **Outcome**: Compromises in final system requirements.

Lecture 6: System Modelling

System Modelling

- **Definition**: The process of developing abstract models of a system, presenting different views or perspectives.
- **Purpose**: Helps analysts understand system functionality and communicate with customers.
- **Notations**: Predominantly uses Unified Modeling Language (UML).

Existing and Planned System Models

- **Existing System Models**: Used during requirements engineering to clarify current functionality and discuss strengths/weaknesses.
- **New System Models**: Used to explain proposed requirements and design proposals during requirements engineering.

Different System Perspectives

1. **External Perspective**: Models the system's context or environment.
2. **Interaction Perspective**: Models interactions between the system and its environment or between system components.
3. **Structural Perspective**: Models the organization or data structure of the system.
4. **Behavioral Perspective**: Models the dynamic behavior and system response to events.

Types of UML Diagrams

1. **Activity Diagrams**: Show activities in a process or data processing.
2. **Use Case Diagrams**: Show interactions between a system and its environment.
3. **Sequence Diagrams**: Show interactions between actors and the system or between system components.
4. **Class Diagrams**: Show object classes and their associations.
5. **State Diagrams**: Show system reactions to internal and external events.

Context Models

- **Purpose**: Illustrate the operational context of a system, showing what lies outside system boundaries.
- **System Boundaries**: Define what is inside and outside the system, affecting system requirements and stakeholder influence.

Process Perspective

- **Purpose**: Reveal how the system is used in broader business processes.
- **Tool**: UML activity diagrams for defining business process models.

Interaction Models

- **Purpose**: Identify user requirements, highlight communication problems, and understand system performance and dependability.
- **Tools**: Use case diagrams and sequence diagrams.

Structural Models

- **Purpose**: Display system organization in terms of components and their relationships.
- **Types**: Static models (system design structure) and dynamic models (organization during execution).
- **Tool**: Class diagrams.

Behavioral Models

- **Purpose**: Model dynamic behavior and system response to stimuli (data or events).
- **Tools**: Data-driven models and event-driven models.
- **Examples**: Activity diagrams for data processing, state diagrams for event responses.

Model-Driven Engineering (MDE)

- **Definition**: An approach where models are the primary outputs of the development process.
- **Benefits**: Raises the level of abstraction, reduces concerns with programming details or execution specifics.
- **Model Driven Architecture (MDA)**: A model-focused approach using UML to describe a system, enabling automated generation of executable programs from high-level models.

Types of Models in MDA

1. **Computation Independent Model (CIM)**: Models important domain abstractions.
2. **Platform Independent Model (PIM)**: Models system operation without reference to implementation specifics.
3. **Platform Specific Models (PSM)**: Transformations of the PIM for specific platforms, potentially in multiple layers.

Adoption Challenges of MDE/MDA

- **Tool Support**: Specialized tools are required for model conversion.
- **Complex Systems**: Implementation is less of an issue than requirements engineering, security, dependability, integration, and testing.
- **Agile Methods Conflict**: MDA's extensive up-front modeling contradicts agile principles.
- **Limited Adoption**: Due to factors like specialized tools, long-lifetime system concerns, and preference for agile methods.

Lecture 7: System Design

Architectural Design

- **Definition**: Architectural design involves understanding how a software system should be organized and designing the overall structure.
- **Importance**: Serves as the critical link between design and requirements engineering, identifying main structural components and their relationships.

- **Output**: An architectural model describing the system's organization as a set of communicating components.

Agility and Architecture

- Early agile processes design an overall system architecture.
- Refactoring system architecture is expensive due to the impact on many components.

Levels of Architectural Abstraction

1. **Architecture in the Small**: Concerns the architecture of individual programs, focusing on how they are decomposed into components.
2. **Architecture in the Large**: Concerns the architecture of complex enterprise systems, including other systems, programs, and components distributed across different computers.

Advantages of Explicit Architecture

1. **Stakeholder Communication**: Facilitates discussions among stakeholders.
2. **System Analysis**: Enables analysis of whether the system meets non-functional requirements.
3. **Large-scale Reuse**: Allows architecture reuse across systems and development of product-line architectures.

Use of Architectural Models

1. **Facilitate Discussion**: High-level views help communicate with stakeholders without getting bogged down in details.
2. **Document Architecture**: Produces a complete system model showing components, interfaces, and connections.

Architectural Design Decisions

- Creative process addressing functional and non-functional requirements.
- Key decisions include:
  - Distribution across hardware
  - Use of architectural patterns/styles
  - Structural decomposition into sub-components
  - Strategy for component operation control
  - Documentation of the architecture

Architectural Views

- **4+1 View Model (Krutchen 1995)**:
  1. **Logical View**: Key abstractions as objects or object classes.
  2. **Physical View**: System hardware and software distribution.
  3. **Process View**: Interacting runtime processes.
  4. **Development View**: Software decomposition for development.

Architectural Patterns

- **Definition**: Stylized descriptions of good design practice, reusable in different environments.
- **Example**: Model-View-Controller (MVC) Pattern, which separates presentation, interaction, and data management.

Model-View-Controller (MVC) Pattern

- **Components**:
    - **Model**: Manages system data.
    - **View**: Manages data presentation.
    - **Controller**: Manages user interaction.
- **Advantages**: Data and representation independence, multiple data presentations.
- **Disadvantages**: Added code complexity.

Layered Architecture

- **Definition**: Models interfacing of sub-systems, organizing related functionalities into layers.
- **Advantages**: Incremental sub-system development, limited impact of interface changes.
- **Example**: User interface, core business logic, system utilities, system support.

Repository Architecture

- **Definition**: Central repository for managing system data accessible to all components.
- **Use Case**: Large data sharing systems.
- **Example**: Integrated Development Environment (IDE) architecture.

Client-Server Architecture

- **Definition**: Distributed model showing data and processing distribution.
- **Components**:
    - **Servers**: Provide specific services.
    - **Clients**: Request services.
    - **Network**: Facilitates client-server interaction.
- **Advantages**: Distributed services, general functionality available to all clients.
- **Disadvantages**: Single point of failure, unpredictable performance, management issues.

Pipe and Filter Architecture

- **Definition**: Functional transformations process inputs to produce outputs.
- **Advantages**: Easy to understand, supports reuse, matches workflow structure.
- **Disadvantages**: Increased system overhead, data format agreement needed.

Application Architectures

- **Purpose**: Designed to meet organizational needs, often sharing common architecture across applications.
- **Examples**:
    1. **Transaction Processing Applications**: Process user requests and update system databases.
    2. **Information Systems**: Depend on interpreting events from the system's environment.
    3. **Language Processing Systems**: Process and interpret formal languages.

Object-Oriented Design

- **Process**: Involves designing object classes and relationships, defining system objects and interactions.
- **Stages**:
    1. Understand context and external interactions.
    2. Design system architecture.
    3. Identify principal system objects.
    4. Develop design models.
    5. Specify object interfaces.

Design Models

- **Structural Models**: Describe static structure of object classes and relationships.
- **Dynamic Models**: Describe interactions between objects.
- **Examples**: Subsystem models, sequence models, state machine models.

Interface Specification

- **Purpose**: Ensures parallel design of objects and components.
- **Tool**: UML class diagrams.

Design Patterns

- **Definition**: Reusing abstract knowledge about problems and solutions.
- **Elements**: Name, problem description, solution description, statement of consequences.

Implementation Issues

- **Reuse**: Focus on reusing existing components or systems.
- **Configuration Management**: Manages changing software systems, supporting integration and version control.
- **Host-Target Development**: Development on a host system, execution on a target system.

Open Source Development

- **Definition**: Source code is published, and volunteers participate in development.
- **License Models**:

1. GNU General Public License (GPL)
2. GNU Lesser General Public License (LGPL)
3. Berkeley Standard Distribution (BSD) License

Lecture 8: Component-Based Development

What is a Component?

- **Definition**: A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.
  - **Object-oriented view**: A component contains a set of collaborating classes.
  - **Traditional view**: A component includes processing logic, internal data structures, and an interface for invocation and data passing.
  - **Process-related view**: Emphasizes using existing software components or design patterns to build systems.
- **Example**: A shopping cart in an e-commerce WebApp, which can be reused across various applications with slight modifications.

Component-Based Software Engineering (CBSE)

- **Emergence**: Late 1990s, due to limitations in object-oriented development for effective reuse.
- **Concept**: Reusing abstract components as standalone service providers.
- **Process**: Involves defining, implementing, integrating, or composing loosely coupled, independent components into systems.

Essentials of CBSE

1. **Independent Components**: Specified by their interfaces, separating interface from implementation.
2. **Component Standards**: Facilitate component integration, embodied in a component model.
3. **Middleware**: Supports component integration.
4. **Development Process**: Adapts to evolving requirements and available components.

Benefits and Principles of CBSE

- **Benefits**:
  - Independent components do not interfere with each other.
  - Hidden component implementations.
  - Communication through well-defined interfaces.
  - Replaceability if the interface is maintained.
  - Component infrastructures offer standard services.

Standards and Component as a Service

- **Competing Standards**: Sun's Enterprise Java Beans, Microsoft's COM and .NET, CORBA's CCM.
- **Interoperability Solution**: Executable services based on standards to avoid communication issues.

Components and Component Models

- **Characteristics**:
  - **Independent**: Can be deployed without other specific components.
  - **Composable**: Interactions through public interfaces.
  - **Deployable**: Self-contained and operable as standalone entities.
  - **Documented**: Fully documented for potential users.
  - **Standardized**: Conforms to a standard component model.
- **Interfaces**:
  - **Provides Interface**: Services provided to other components (API).
  - **Requires Interface**: Services needed from other components for operation.

Component Models

- **Definition**: Standards for component implementation, documentation, and deployment.
- **Examples**: EJB model, Microsoft's .NET model, CORBA Component Model.
- **Elements**:
  - **Interfaces**: Operation names, parameters, exceptions.
  - **Usage**: Unique names or handles for remote access.
  - **Deployment**: Packaging as independent, executable routines.

CBSE Processes

- **Types**:
  1. **CBSE for Reuse**: Developing components/services for reuse, often generalizing existing components.
  2. **CBSE with Reuse**: Developing new applications using existing components/services.
- **Supporting Processes**:
  1. **Component Acquisition**: Finding and acquiring components for reuse or development.
  2. **Component Management**: Cataloging, storing, and maintaining reusable components.
  3. **Component Certification**: Checking and certifying components meet specifications.

Component Composition

- **Types**:
  1. **Sequential Composition**: Calling existing components in sequence.
  2. **Hierarchical Composition**: One component calls services provided by another.

3. **Additive Composition**: Combining multiple components to create a new one.

- **Challenges**: Incompatible interfaces, conflicts between functional and non-functional requirements, and trade-offs between adaptability and performance.

Adaptor Components

- **Purpose**: Address incompatibility by reconciling interfaces of composed components.
- **Considerations**:
    - Effective component composition for functional requirements.
    - Future change adaptability.
    - Emergent properties of the composed system.

Lecture 10: Software testing

Introduction to Testing

Purpose of Testing:

- Demonstrate that a program functions as intended.
- Discover program defects before deployment.
- Executing a program using artificial data to check for errors, anomalies, and non-functional attributes.

Key Points:

- Testing reveals the presence of errors, not their absence.
- Part of a broader verification and validation (V&V) process, including static validation techniques.

Goals of Software Testing:

1. Demonstrate that the software meets its requirements.
2. Identify incorrect, undesirable, or non-conformant behavior.

Validation Testing

- Ensures the system performs correctly with a given set of test cases reflecting expected use.
- A successful validation test shows the system operates as intended.

Defect Testing

- Aims to discover faults or defects by designing test cases that expose defects.
- A successful defect test makes the system perform incorrectly, revealing a defect.

Input-Output Model of Software Testing:

- The system accepts inputs and generates outputs.

- Validation testing uses correct inputs to generate expected outputs.
- Defect testing focuses on inputs that reveal problems with the system.

Verification vs. Validation

- **Verification**: "Are we building the product right?" Ensures software conforms to specifications.
- **Validation**: "Are we building the right product?" Ensures software meets user requirements.

V&V Goal: Establish confidence that the software is fit for purpose.

Software Inspection and Testing

- Inspections and reviews analyze system requirements, design models, and code without executing the software (static V&V).
- Testing involves executing the software and observing behavior (dynamic V&V).

Software Inspection:

- Focuses on the source code and other readable representations of the software.
- Advantages include static process benefits, handling incomplete systems, and broader quality attribute considerations.

Model of Software Testing Process:

- Test cases: Specify inputs and expected results.
- Test data: Inputs devised to test the system.

Stages of Software Testing:

1. **Development Testing**: Tests during development to find bugs.
2. **Release Testing**: Separate team tests a complete system version before release.
3. **User Testing**: Users test the system in their environment.

Development Testing

Stages:

1. **Unit Testing**: Tests individual methods or object classes.
2. **Component Testing**: Integrates individual units to create composite components.
3. **System Testing**: Integrates components to test the system as a whole.

Unit Testing:

- Involves testing operations, attributes, and possible states of an object.
- Choosing effective test cases is crucial.

Component Testing:

- Focuses on the component interface to ensure it behaves as specified.
- Types of interfaces include parameter, shared memory, procedural, and message passing interfaces.

System Testing:

- Involves testing reusable components, off-the-shelf systems, and newly developed components.
- Use case-based testing is effective, involving system interactions.

Test-Driven Development (TDD)

- Incorporates testing into code development.
- Develop code incrementally along with a set of tests.
- Benefits include code coverage, regression testing, simplified debugging, and system documentation.

TDD Process:

1. Identify functionality.
2. Write a test.
3. Run tests.
4. Implement functionality.
5. Repeat until all tests pass.

Release Testing

- Tests a system release intended for customers and users.
- Goals include verifying specified functionality, performance, dependability, and ensuring no failures during normal use.

Requirements-Based Testing:

- Derives tests from system requirements to check if they are satisfied.
- Example: Testing drug allergy warnings in a medical system.

Scenario Testing:

- Uses realistic scenarios to develop test cases.
- Example: Testing a healthcare system's functionality during a home visit.

Performance Testing:

- Tests emergent properties like performance and reliability under load.
- Includes stress testing to test failure behavior.

User Testing

Types:

1. **Alpha Testing**: Users test the software at the developer's site.
2. **Beta Testing**: Users test a release of the software and report issues.
3. **Acceptance Testing**: Customers test to decide if the software is ready for deployment.

Acceptance Testing Process:

1. Define acceptance criteria.
2. Plan acceptance testing.
3. Derive acceptance tests.
4. Run acceptance tests.
5. Negotiate test results.
6. Accept or reject the system.

Agile Methods and Acceptance Testing:

- Integrated with development, tests defined by the user/customer and run automatically.
- No separate acceptance testing process.

Lecture 11: Software Quality

- **Quality Definition**: Traditionally based on conformance with a detailed product specification, but often interpreted differently by developers and customers.
- **Fitness for Purpose**: Focuses on the system meeting its intended purpose.

Key Questions for Quality Assessment:

1. Has the software been properly tested?
2. Is the software dependable?
3. Is the software's performance acceptable?
4. Is the software usable?
5. Is the software well-structured and understandable?
6. Have standards been followed during development?

Software Quality Attributes:

- Safety
- Security
- Reliability
- Resilience
- Robustness
- Understandability
- Testability
- Adaptability
- Modularity
- Complexity
- Learnability
- Portability

- Usability
- Efficiency
- Reusability

Quality Conflicts:

- Not possible to optimize all attributes simultaneously (e.g., security vs. performance).
- Quality plan prioritizes the most important attributes for development focus.

Software Standards

- **Importance**: Encapsulate best practices to avoid past mistakes and provide a framework for quality.
- **Types of Standards**:
    - **Product Standards**: Apply to the software product being developed (e.g., programming style).
    - **Process Standards**: Define processes to be followed during development (e.g., version release process).

Standards Development:

- Involve engineers in selecting standards.
- Regularly review and modify standards to reflect new technologies.
- Provide tool support to reduce clerical work.

ISO 9001 Standards Framework:

- International standards for quality management systems, applicable to organizations designing, developing, and maintaining products, including software.
- Sets general quality principles, processes, and organizational standards documented in a quality manual.

Reviews and Inspections

Purpose: Quality assurance activities that check project deliverables' quality through document and code reviews.

Software Review Process:

1. **Pre-review Activities**: Planning and preparation for the review.
2. **Review Meeting**: The author presents the document or program for review.
3. **Post-review Activities**: Addressing issues and problems raised during the review.

Program Inspection:

- Peer reviews to find bugs without executing the program.
- Involves a careful, line-by-line review of the source code using a checklist for common programming errors.

Quality Management and Agile Development:

- Agile methods focus on code quality through practices like refactoring and test-driven development.
- Quality management in agile development relies on shared good practices (e.g., code reviews before check-in, testing code changes).

Software Measurement and Metrics

Software Measurement:

- Concerned with quantifying attributes like complexity or reliability.
- Helps assess software quality and the effectiveness of processes, tools, and methods.

Software Metrics:

- **Control Metrics**: Associated with software processes (e.g., effort required to repair defects).
- **Predictor Metrics (Product Metrics)**: Associated with the software itself (e.g., lines of code, number of reported faults).

Uses of Measurements:

1. Assign values to system quality attributes.
2. Identify substandard system components.

Problems with Measurement in Industry:

- Difficulty quantifying ROI for metrics programs.
- Lack of standard metrics and processes.
- Need for specialized tools.
- Non-standardized software processes in many companies.
- Focus on code-based metrics and plan-driven processes, while more software is developed through configuration.

Product Metrics:

- **Dynamic Metrics**: Collected during system execution, assessing efficiency and reliability.
- **Static Metrics**: Collected from representations like design or code, assessing complexity, understandability, and maintainability.

Static Software Product Metrics:

- Examples include measures of code complexity, cohesion, and coupling.

CK (Chidamber and Kemerer) Object-Oriented Metrics Suite:

- A set of metrics for assessing object-oriented designs.

Software Component Analysis:

- Measures components separately using various metrics.
- Compares metrics to historical data to identify anomalies indicating quality issues.

Lecture 12: Configuration Management

Configuration Management (CM)

- **Definition**: CM is concerned with the policies, processes, and tools for managing changing software systems.
- **Purpose**: Manages evolving systems to avoid losing track of changes and component versions incorporated into each system version.

Key Activities in Configuration Management:

1. **Version Control**: Keeping track of multiple versions of system components and ensuring changes by different developers do not interfere with each other.
2. **System Building**: Assembling program components, data, and libraries, compiling, and linking to create an executable system.
3. **Change Management**: Tracking requests for changes to delivered software, evaluating costs and impacts, and deciding on implementation.
4. **Release Management**: Preparing software for external release and tracking system versions released for customer use.

Version Management

Definition: Keeping track of different versions of software components and systems, ensuring non-interference of changes made by different developers.

Key Concepts:

- **Codelines**: A sequence of versions of source code derived from earlier versions, usually for system components.
- **Baselines**: Definition of a specific system version, specifying component versions, libraries, configuration files, and other system information.

Features of Version Management Systems:

- **Version and Release Identification**: Assigning unique identifiers to managed versions of a component.
- **Change History Recording**: Keeping records of changes made to create new component versions.
- **Independent Development**: Tracking checked-out components to ensure changes by different developers do not interfere.
- **Project Support**: Supporting the development of multiple projects sharing components.
- **Storage Management**: Storing differences between files rather than maintaining multiple copies.

Codeline Branching and Merging: Necessary to merge codeline branches to create a new version of a component, combining changes made in different parts of the code.

System Building

Definition: The process of creating a complete, executable system by compiling and linking system components, external libraries, configuration files, and other information.

Tools for System Integration and Building:

1. **Build Script Generation**
2. **Version Control System Integration**
3. **Minimal Recompilation**
4. **Executable System Creation**
5. **Test Automation**
6. **Reporting**
7. **Documentation Generation**

Build Platforms:

1. **Development System**: Includes development tools; developers check out code from the version control system into a private workspace.
2. **Build Server**: Used to build definitive, executable versions of the system; maintains the definitive versions of the system.
3. **Target Environment**: The platform on which the system executes.

File Identification:

- **Modification Timestamps**: Signature is the time and date when the file was modified.
- **Source Code Checksums**: Signature is a checksum calculated from data in the file, ensuring unique identification of source code versions.

Change Management

Definition: Ensures changes are applied to the system in a controlled way, prioritizing urgent and cost-effective changes.

Change Management Process:

1. **Request Analysis**: Assess costs and benefits.
2. **Approval**: Approve cost-effective changes.
3. **Tracking**: Monitor components changed in the system.

Factors Influencing Change Implementation:

1. Consequences of not making the change.
2. Benefits of the change.
3. Number of users affected.
4. Costs of making the change.
5. Product release cycle.

Agile Methods and Change Management: Customers are involved in deciding change priorities, working with the team to assess impact. Refactoring is seen as a necessary part of development, not an overhead.

Derivation History: Maintaining records of changes made to each component.

Release Management

Definition: Managing system releases distributed to customers.

Types of Releases:

- **Major Releases**: Deliver significant new functionality.
- **Minor Releases**: Repair bugs and fix customer-reported problems.

Release Components:

1. **Configuration Files**
2. **Data Files**
3. **Installation Program**
4. **Documentation**
5. **Packaging and Publicity**

Release Planning:

- Consider technical and organizational factors, including release timing, marketing strategies, and customer readiness.
- Document system releases to ensure future re-creation for long-lifetime embedded systems.

Lecture 13: Software Evolution

Software Change

- **Need for Change**: Operational software systems need to evolve to remain useful due to:
    - Business changes and evolving user expectations.
    - Correction of operational errors.
    - Adaptation to changes in hardware and software platforms.
    - Improvement of performance or other non-functional characteristics.
    - Competition necessitating new features.

Investment in Change: Businesses invest in maintaining and evolving software to preserve its value. This often results in higher maintenance costs compared to new development.

Evolution and Servicing

- **Evolution Phase**: Significant changes to software architecture and functionality occur in response to stakeholder requirements.
- **Servicing Phase**: Involves essential but minor changes to keep the system operational, often while planning for system replacement. Users must work around discovered problems.

Evolution Processes

Factors Influencing Evolution:

1. Type of software being maintained.
2. Development processes used by the organization.
3. Skills of the involved personnel.

Change Implementation: Iterative process involving design, implementation, and testing of system revisions. It often begins with program understanding, especially when new developers are involved.

Urgent Change Requests: Arise due to:

1. Serious system faults requiring immediate repair.
2. Unexpected environmental changes disrupting operations.
3. Unanticipated business changes (e.g., new competitors, legislation).

General Model of Software Evolution Process: Cyclical process of change identification and system evolution continuing throughout the system's lifetime.

Legacy Systems

Definition: Older systems using outdated languages and technologies, often maintained over long periods with a degraded structure.

Strategic Options for Legacy Systems:

1. Scrap the system completely.
2. Maintain the system as is.
3. Reengineer the system for better maintainability.
4. Replace all or part of the system.

Legacy System Assessment:

- **Low quality, low business value**: Scrap the system.
- **Low quality, high business value**: Reengineer the system.
- **High quality, low business value**: Continue normal maintenance if affordable.
- **High quality, high business value**: Maintain and continue normal maintenance.

Business Value Assessment: Involves evaluating system use, business process support, system dependability, and reliance on system outputs.

Technical Assessment: Considers both the application system and its operational environment.

Software Maintenance

Definition: The process of changing a system after delivery.

Types of Maintenance:

1. **Fault Repairs**: Fixing bugs and vulnerabilities.
2. **Environmental Adaptation**: Adjusting the software to new platforms and environments.

3. **Functionality Addition**: Adding new features to meet changing requirements.

Maintenance Effort Distribution: Maintenance costs often exceed initial development costs due to the need for new team understanding, lack of incentives for maintainability, unpopularity of maintenance work, and structural degradation over time.

Maintenance Prediction: Involves assessing parts of the system likely to cause problems and predicting maintenance costs for budgeting purposes.

Process Metrics for Maintenance:

1. Number of corrective maintenance requests.
2. Average time for impact analysis.
3. Average time to implement change requests.
4. Number of outstanding change requests.

Software Reengineering

Purpose: To make legacy systems easier to maintain by improving their structure and understandability.

Reengineering Activities:

- Redocumenting the system.
- Refactoring system architecture.
- Translating programs to modern languages.
- Updating data structures and values.

Advantages:

- Reduced risk compared to new development.
- Lower costs compared to developing new software.

Reengineering Process: Systematic process to improve legacy systems.

Refactoring: Continuous improvement to slow down structural degradation, focusing on improving program structure, reducing complexity, and enhancing understandability without adding functionality.

Comparison:

- **Reengineering**: Conducted after significant maintenance to create a more maintainable system.
- **Refactoring**: Continuous process during development and evolution to prevent degradation