DI04000011: Software Engineering

GTU Faculty

Defining Software in Software Engineering

What is Software?

Software is not merely program code. In software engineering, it is defined as a composite entity comprising:

- **Instructions (Computer Programs):** Executable code that provides desired features, functions, and operational performance when executed.
- **Data Structures:** Schema and data manipulation components that enable programs to adequately store, process, and manipulate information.
- **Documentation:** Operational manuals, design specifications, user guides, and technical references describing program operation and maintenance.

Dual Role of Software

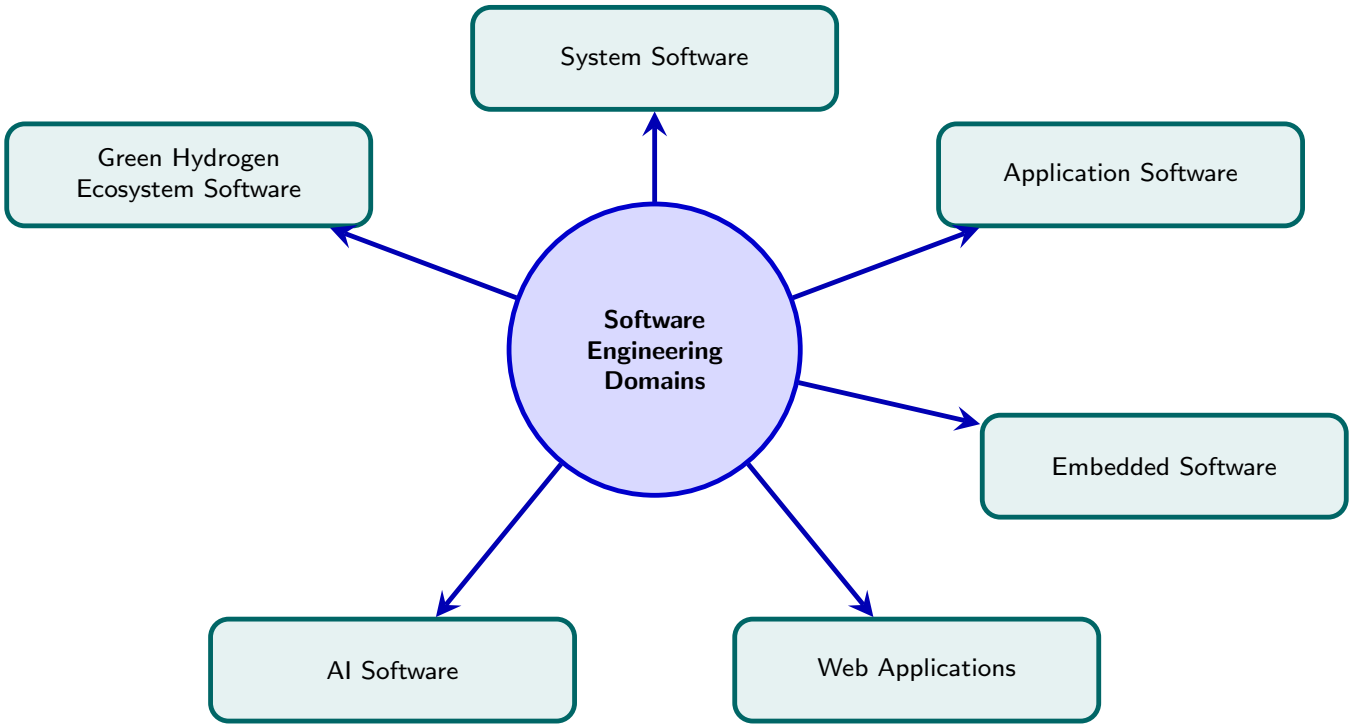
- **As a Product:** Delivers computing potential, processes business data, generates intelligence, and controls physical devices.
- **As a Vehicle for Product Delivery:** Acts as the underlying infrastructure framework for control (Operating Systems), network transmission, and software development suites.

Engineering Attributes of Software

Software differs fundamentally from physical hardware across key engineering dimensions:

1. **Software is Engineered, Not Manufactured:** Physical manufacturing relies on assembly lines and material quality. Software quality is determined by design engineering, architectural rigor, and analytical modeling.
2. **Software Does Not Wear Out:** Hardware exhibits a physical "bathtub curve" failure rate due to environmental wear, friction, and stress. Software suffers from operational degradation due to continuous modification, side effects, and changing environments.
3. **Customization vs. Component Assembly:** Although component-based software engineering is growing, significant software remains custom-built rather than assembled from off-the-shelf standardized parts.

Taxonomy of Software Application Domains



Core Domains: System & Application Software

System Software

- Infrastructure software designed to service other programs directly.
- Characterized by heavy interaction with physical hardware, concurrent multi-user execution, complex data structures, and deterministic timing constraints.
- **Examples:** Operating system kernels, compilers, device drivers, telecommunication switching software, device firmware.

Application Software

- Standalone programs designed to solve specific business needs or automate real-world administrative and engineering procedures.
- Focuses on domain-specific logic, transaction management, user interaction, and report generation.
- **Examples:** Point-of-Sale (POS) systems, Enterprise Resource Planning (ERP), inventory management systems, accounting platforms.

Core Domains: Embedded Software & Web Applications

Embedded Software

- Resides in read-only memory (ROM) inside consumer, industrial, or biomedical equipment.
- Executes dedicated control functions under stringent real-time response constraints and constrained memory footprints.
- **Examples:** Automotive Electronic Control Units (ECUs), industrial controllers, microwave firmware, medical pacemakers.

Web Applications (WebApps)

- Network-centric software systems executing across distributed client-server environments accessible via browser interfaces.
- Combines backend database management, service-oriented architecture (SOA), API integration, and high concurrency handling.
- **Examples:** E-commerce portals, cloud-based Software-as-a-Service (SaaS), banking web applications.

Modern Domain: Artificial Intelligence Software

Characteristics of AI Software

- Employs non-algorithmic, heuristic, dynamic, and probabilistic computational models to solve non-deterministic problems.
- Leverages deep neural networks, machine learning algorithms, and probabilistic trees instead of static decision structures.
- Adaptive software capability: learns, updates parameters, and optimizes performance based on continuous operational data streams.

Key Software Engineering Applications

- **Pattern & Speech Recognition:** Natural language processing (NLP), machine vision, automated translation systems.
- **Automated Decision Engines:** Predictive maintenance models, autonomous navigation systems, automated fault diagnosis platforms.

Specialized Domain: Green Hydrogen Ecosystem Software

Introduction to Green Hydrogen Software Engineering

Green Hydrogen production (water electrolysis powered by renewable energy) requires software solutions to handle variable power inputs, electrolyzer dynamics, and safety protocols.

Core Software Roles in Green Hydrogen

- **SCADA & Industrial Automation:** Real-time Supervisory Control and Data Acquisition monitoring current, voltage, pressure, and thermal state across electrolyzer stacks.
- **Energy Management Systems (EMS):** Predictive algorithmic software allocating variable solar/wind power generation to electrolyzer units to maximize efficiency.
- **Digital Twin Simulation:** Cyber-physical simulation software tracking stack degradation, membrane health, and system output in real time.

Software in Hydrogen Production & Distribution

Software Role in Production

- **Electrolyzer Stack Control Logic:** Software algorithms controlling water feed rates, cooling loops, and power electronics tuning during dynamic grid variations.
- **Safety Interlock Systems:** High-integrity software executing automated emergency shutdown (ESD) upon detecting hydrogen leakage or oxygen contamination risks.

Software Role in Storage & Distribution

- **Pressure & Cryogenic Management:** Control software maintaining high-pressure gas storage (350–700 bar) or liquid hydrogen storage (-253°C).
- **Distribution Logistics & Tracking:** Supply chain optimization software coordinating fuel cell vehicle refueling stations and tube-trailer dispatch.

Comparative Analysis of Software Application Domains

Table: Comparison of Key Software Application Domains in Software Engineering

Domain	Primary Focus	Key Constraints / Challenges	Representative Systems
System Software	Hardware management & platform services	Deterministic timing, high performance, hardware dependency	OS Kernels, Compilers, Device Drivers
Application Software	Business processes & workflow automation	Data integrity, transaction throughput, UI usability	ERP Platforms, Banking Systems, CRM
Embedded Software	Dedicated device control	Real-time deadlines, low memory footprint, reliability	Automotive ECUs, IoT Nodes, Medical Sensors
Web Applications	Networked service delivery	Security, high user concurrency, multi-browser compatibility	SaaS Platforms, Portals, E-Commerce
AI Software	Heuristic reasoning & pattern learning	Model explainability, compute intensity, dataset quality	Neural Networks, NLP Engines, Vision Tools
Green Hydrogen	Renewable power integration & safety	High-safety compliance, real-time control, sensor integration	SCADA, Electrolyzer EMS, Digital Twins

Lecture 1 Summary & Transition to Process Models

Key Lecture Takeaways

- Software is an integrated system comprising executable code, structured data models, and descriptive documentation.
- Software application domains range from low-level system programs to modern green hydrogen ecosystem control systems.
- The criticality of software in green hydrogen highlights the importance of engineering methodologies in cyber-physical safety systems.

Transition to Software Process Models

- Building complex software across diverse domains requires structured process frameworks.
- Next Topic: **Software Process Models** (Waterfall, Incremental, Evolutionary, and Agile Methodologies).

GTU Course: Software Engineering (DI04000011)

Unit: Software Process Models
Focus: Comprehensive definition of software, attributes, and categorical application domains.

- **Core Objectives:**
  - Formulate a precise engineering definition of software beyond raw executable code.
  - Differentiate between software deterioration models and hardware wear-and-tear.
  - Analyze core software application domains: System, Application, Embedded, Web, and AI.
  - Explore specialized engineering software: Green Hydrogen Ecosystems (Production & Distribution).
- **Engineering Significance:**
  - Application domain characteristics directly govern software process model selection and quality assurance strategies.

Defining Software: Concept, Nature & Dual Role

What is Software?

Software is not merely program code. In Software Engineering, software is defined as a tripartite entity:

1. **Instructions (Computer Programs):** Executable code that when executed provides desired features, function, and performance.
2. **Data Structures:** Frameworks that enable programs to adequately manipulate information.
3. **Documentation:** Descriptive text in both hard copy and electronic formats describing the operation and use of the programs.

The Dual Role of Software

- **Software as a Product:** Delivers computing potential embodied by hardware; acts as an information transformer (computes, manages, modifies, acquires, or transmits data).
- **Software as a Vehicle/Deliverer:** Serves as the control engine for product delivery (e.g., operating systems, networking software, middleware, and software development platforms).

Software Characteristics: Hardware vs. Software Failure

- **Software is Engineered, Not Manufactured:**
  - High initial cost lies in engineering design, not physical production/manufacturing.
  - Software quality is built-in during development, not inspected into manufactured units.
- **Software Does Not Wear Out (Failure Curve Characteristics):**
  - Hardware suffers from physical wear-and-tear due to environmental stress, vibration, and thermal fatigue (Bath-tub curve).
  - Software is immune to physical environmental degradation.
  - However, software **deteriorates** due to side-effects of uncoordinated maintenance changes (cumulative complexity & software entropy).
- **Custom-Built vs. Component-Based Assembly:**
  - Modern SE emphasizes reusable software components, though custom logic remains prevalent.

Taxonomy of Software Application Domains

Table: Comparative Matrix of Software Application Domains

Domain	Core Characteristics	Key SE Challenges	Typical Examples
System Software	Tight hardware coupling, heavy resource sharing	Micro-second latency, concurrency safety	Compilers, OS Kernels, Device Drivers
Application	Standalone processing, user-centric interfaces	Usability, complex business rules	ERP systems, CAD, Spreadsheet tools
Embedded Software	Read-Only Memory, real-time reactive constraints	Hard real-time deadlines, low memory	Automotive ECU, IoT Firmware
Web Applications	Distributed, multi-tiered, client-server architecture	High scalability, dynamic security threats	E-Commerce, Web Portals, Cloud Apps
AI Software	Non-deterministic execution, data-driven logic	Algorithm validation, model drift management	Deep learning, NLP, Autonomous Robotics
Green Hydrogen	Hybrid IoT-SCADA, electrochemical modeling	Real-time optimization, safety-critical telemetry	Electrolyzer Control, Fuel Cell Dispatch

Domain Exploration: System & Application Software

1. System Software

- Programs written to service other programs directly.
- Characterized by intensive interaction with computer hardware, heavy multi-user sharing, complex data structures, and low-level processing interfaces.
- *Examples:* Operating System kernels (Linux, Windows), device drivers, memory managers, command compilers, and network protocol stacks.

2. Application Software

- Standalone programs that solve specific business or technical operational needs.
- Processes business transactions or controls real-time operational data in human-centric domains.
- *Examples:* Enterprise Resource Planning (ERP), Point-of-Sale (POS) systems, graphic design tools, and CAD packages.

Domain Exploration: Embedded & Web Applications

3. Embedded Software

- Resides within a product or system and is used to implement and control features and functions for the end-user and for the system itself.
- Runs on microcontrollers with constrained memory, processing power, and strict real-time operational constraints.
- *Examples:* Anti-lock Braking Systems (ABS), pacemakers, smart thermostats, and industrial sensor microcontrollers.

4. Web Applications

- Hypertext-centric and network-based applications providing client-server interactive services over Internet/Intranet protocols.
- Focuses on data integration, multi-tier architectures, responsive user interface design, and continuous deployment workflows.
- *Examples:* Cloud dashboards, streaming services, and online banking platforms.

5. Artificial Intelligence (AI) Software

- Makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward algorithmic analysis.
- Incorporates Machine Learning (ML), Deep Neural Networks (DNN), expert systems, and pattern recognition engines.

Software Engineering for AI vs. Traditional SE

- **Non-deterministic Behavior:** Logic is learned from empirical datasets rather than explicitly hardcoded by engineers.
- **Data Quality Dependency:** System reliability relies heavily on training dataset diversity, data preprocessing pipelines, and bias mitigation.
- **Model Lifecycle Management:** Requires MLOps pipelines to handle continuous re-training, model monitoring, and drift detection.

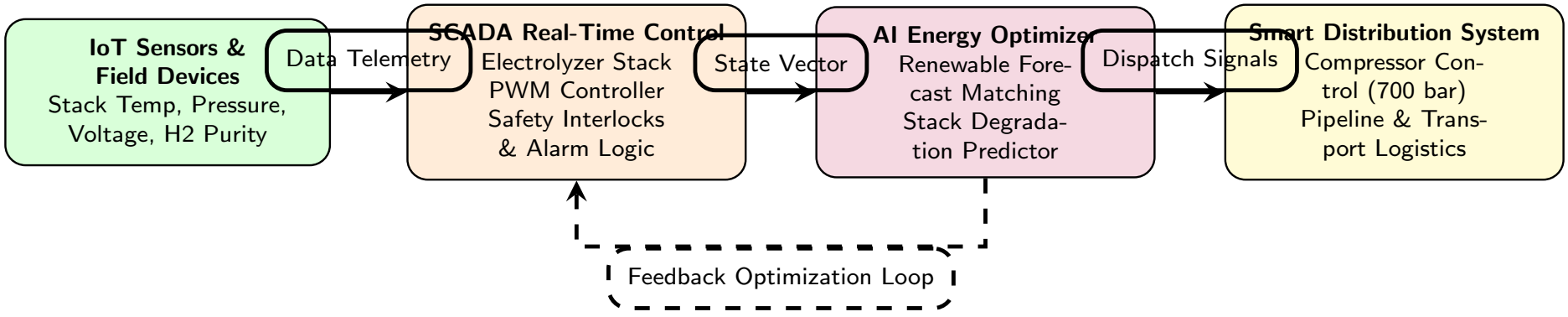
Specialized Domain: Green Hydrogen Ecosystem Software

Green Hydrogen & Software Engineering Context

Green hydrogen production involves water electrolysis using renewable power (solar/wind). Software acts as the core orchestrator to handle intermittent energy inputs and volatile demand patterns.

- **Role in Hydrogen Production:**
  - *Electrolyzer Process Control:* Real-time software monitors cell voltage, stack temperature, pressure, and gas purity to maintain peak Faraday efficiency.
  - *Renewable Power Matching:* Algorithmic scheduling aligns electrolyzer operation with dynamic solar/wind generation profiles.
- **Role in Hydrogen Storage & Distribution:**
  - *Compressor & Pipeline Control:* Software controls high-pressure compression (350–700 bar) and storage vessel gas balance.
  - *Smart Grid Dispatching:* AI-driven dispatch software predicts hydrogen demand and optimizes transport logistics via pipeline or trailer networks.

Software Architecture in Green Hydrogen Ecosystems



Software Engineering System Responsibilities

- **Hard Real-Time Control:** Sub-millisecond safety shutoff via SCADA and Programmable Logic Controllers (PLCs).
- **Predictive Analytics:** Machine Learning models optimizing stack lifetime, thermal efficiency, and operational costs.

Summary & Impact on Software Process Selection

Summary of Key Takeaways

- Software is composed of code, data structures, and documentation; it deteriorates rather than wears out physically.
- Software spans diverse domains ranging from low-level System Software to web-scale dynamic applications and non-deterministic AI systems.
- Specialized industrial software, such as Green Hydrogen Ecosystem Software, fuses embedded SCADA control with high-level AI dispatch algorithms.

Domain Drive on Software Process Models

- **Safety-Critical (Embedded / Hydrogen SCADA):** Demands heavy formal process models (Waterfall / V-Model) with strict validation.
- **Dynamic / Evolving (Web Apps / AI):** Favors Agile, Evolutionary, or DevOps lifecycles for rapid iteration and model retraining.

Defining Software: Concept and Dual Role

What is Software?

Software is defined as a combination of:

- **Instructions:** Computer programs that when executed provide desired features, function, and performance.
- **Data Structures:** Data structures that enable the programs to adequately manipulate information.
- **Documentation:** Descriptive information in both hard copy and electronic formats that describes the operation and use of the programs.

The Dual Role of Software

- **As a Product:** Operates as an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information.
- **As a Vehicle for Delivering Product:** Functions as the control infrastructure for operating systems, networks, data transmission, and hardware control systems.

Characteristics of Software: Software vs. Hardware

Key Software Characteristics

1. **Software is developed or engineered, not manufactured:** Although quality is achieved through good design, manufacturing phase for software does not exist in the traditional physical sense.
2. **Software does not "wear out":** Hardware exhibits an S-shaped bathtub curve (infant mortality leading to wear-out due to environmental stress). Software suffers from deterioration caused by maintenance changes (side effects).
3. **Custom-built vs. Component-based Assembly:** Most software continues to be custom built, though modern software engineering heavily emphasizes reusable software components.

The Software Failure Curve

Undiscovered errors cause high initial failure rates. As corrections are applied, failure rates drop, but each maintenance side-effect creates spikes in the failure rate over time.

Taxonomy of Software Application Domains

Table: Software Application Domains Matrix

Domain	Primary Focus	Key Constraints	Typical Examples
System Software	Hardware infrastructure control	High execution speed, resource constraints	OS, Compilers, Drivers
Application Software	Standalone business logic	User interaction, transaction processing	ERP, CRM, DBMS
Embedded Software	Microcontroller control	Memory limits, real-time response	Automotive ECU, Smart Meter
Web Applications	Distributed networked computing	Concurrency, security, scalability	E-Commerce, SaaS Portals
AI Software	Non-algorithmic data modeling	Computational power, training data quality	Deep Learning, NLP Models
Green Hydrogen Software	Clean energy production/distribution	Safety-critical (SIL), SCADA telemetry	Electrolyzer Control, Pipeline SCADA

System Software and Application Software

1. System Software

System software is a collection of programs written to service other programs.

- Characterized by heavy interaction with computer hardware.
- High degree of resource sharing and complex data structures.
- **Examples:** Operating System kernels, device drivers, compilers, text editors, utility software, and hardware interface protocols.

2. Application Software

Application software consists of standalone programs that solve specific business needs.

- Processes business or technical operations in real-time or batch mode.
- Focuses on data processing, user interface management, and database control.
- **Examples:** Enterprise Resource Planning (ERP), Point of Sale (POS), Accounting Software, and Inventory Management Systems.

3. Embedded Software

Resides within a product or system and is used to implement and control features and functions for the end-user and for the system itself.

- Executes within Read-Only Memory (ROM) or Microcontrollers.
- Performs limited and esoteric functions under strict real-time constraints.
- **Examples:** Braking systems in automobiles, microwave controls, medical pacemaker chips, IoT devices.

4. Web Applications (WebApps)

Centric to client-server architectures operating over hyperlinked networks.

- Evolves from simple static HTML pages to complex, dynamic cloud applications.
- High emphasis on security, concurrency, dynamic content presentation, and database integration.
- **Examples:** Cloud computing portals, web-based banking, online streaming systems.

5. Artificial Intelligence (AI) Software

AI software makes use of non-algorithmic algorithms to solve complex problems that are not amenable to computation or straightforward analysis.

- Focuses on pattern matching, heuristic learning, probabilistic reasoning, and knowledge representation.
- Shifts software engineering paradigms from traditional deterministic code logic to probabilistic model training pipelines.

Key Sub-domains of AI Software

- **Machine Learning & Deep Learning:** Neural network architectures for classification and prediction.
- **Expert Systems & Knowledge Graphs:** Rule-based engines for automated decision-making.
- **Robotics & Computer Vision:** Perception-action software loops for autonomous vehicles.

6. Green Hydrogen Ecosystem Software

An emerging cyber-physical application domain integrating embedded real-time control, industrial IoT, and supervisory control systems to manage clean energy transformation.

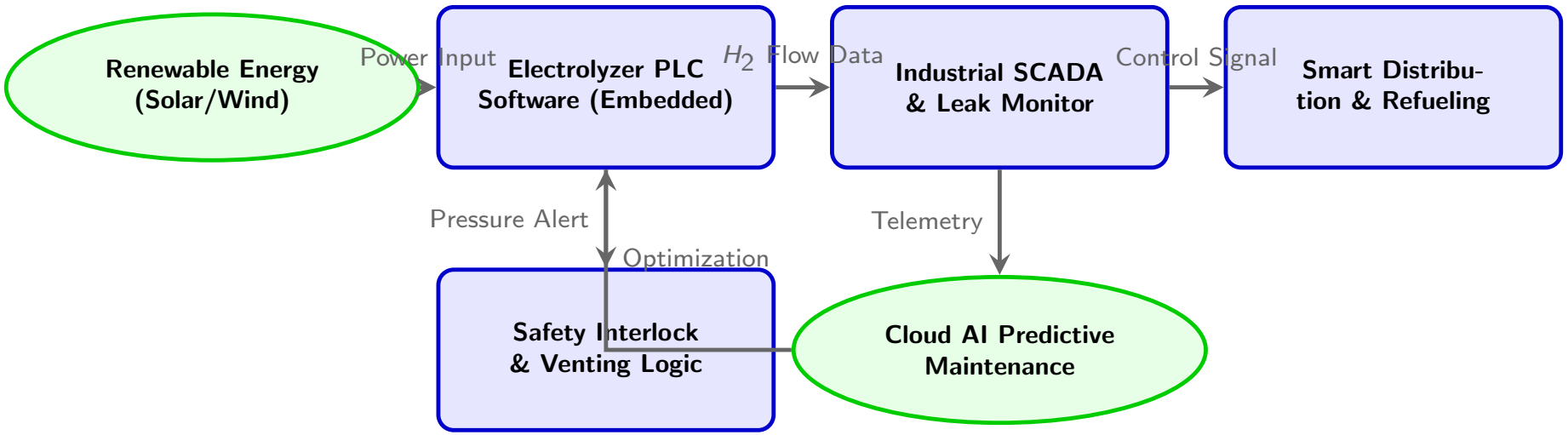
Role of Software in Hydrogen Production

- **Electrolyzer Control Systems:** Real-time algorithm control of Proton Exchange Membrane (PEM) or Alkaline electrolyzers to maximize hydrogen yield under fluctuating renewable power (solar/wind).
- **Renewable Energy Matching Logic:** Dynamic software scheduling matching fluctuating power input with optimal stack operation parameters.

Role of Software in Hydrogen Distribution

- **SCADA & Telemetry:** Leak detection software using IoT sensor nodes along high-pressure transport pipelines.
- **Compressor Station Automation:** Algorithmic management of pressure cascades during storage and refueling dispenser allocation.

Architecture of Green Hydrogen Control Software



Software System Interplay

Green Hydrogen systems require seamless convergence of embedded real-time software, industrial SCADA protocols, safety-critical loops, and cloud-based AI optimization.

Software Engineering Challenges in Green Hydrogen Systems

1. Safety-Critical Real-Time Software Requirements

- Hydrogen has a wide flammability range (4%–75% in air) and low ignition energy.
- Software must adhere to strict functional safety standards (e.g., IEC 61508 / SIL 3).
- Fail-safe software logic must execute emergency shutdown (ESD) within milliseconds upon anomaly detection.

2. Cyber-Physical Integration & Grid Stability

- Integrating intermittent renewable energy grids requires fast software loop execution to adjust stack power draw.
- Security protocols must prevent cyber-attacks on critical energy infrastructure SCADA software.

Summary of Software Domains & Process Implications

Key Takeaways

1. **Software Evolution:** Software has transitioned from simple batch processing programs to complex, safety-critical cyber-physical ecosystems.
2. **Domain Specialization:** Engineering practices differ significantly across domains—embedded software demands resource efficiency, while WebApps emphasize agility and scale.
3. **Green Hydrogen Integration:** Modern energy systems showcase the convergence of embedded controllers, real-time telemetry, AI optimization, and ultra-high reliability software engineering.
4. **Process Model Selection:** The choice of Software Process Model (Waterfall, Agile, Spiral, DevOps) depends heavily on domain constraints like safety certifications and volatility.

Lecture Objectives

- Understand the formal engineering definition and dual role of software.
- Explore traditional software application domains (System, Application, Embedded, Web).
- Examine modern intelligent application domains (Artificial Intelligence Software).
- Analyze specialized industrial domains: **Green Hydrogen Ecosystem Software** across production and distribution.

Context within Software Process Models

Selecting an appropriate software process model (e.g., Waterfall, Agile, Spiral) depends heavily on the target software application domain and its unique operational constraints.

Defining Software in Software Engineering

Formal Definition (IEEE / Pressman)

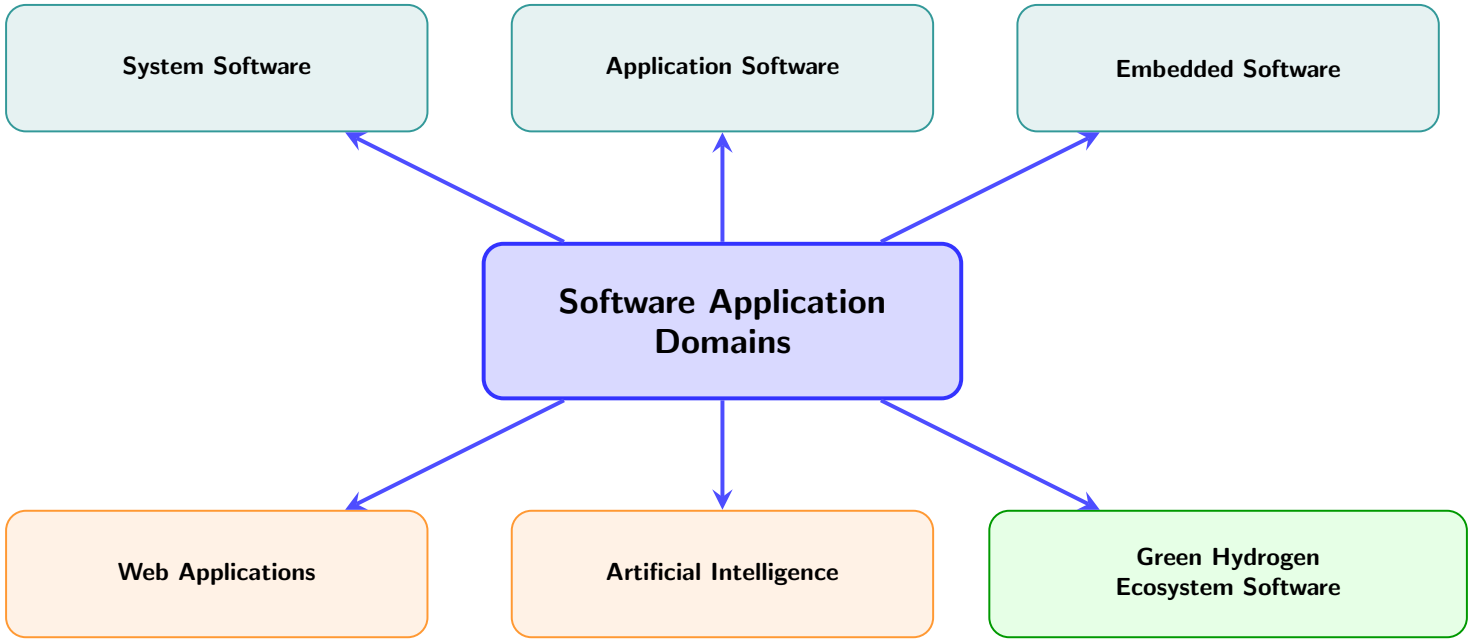
Software is defined as a composite entity comprising three core elements:

1. **Instructions (Computer Programs):** Executable code that provides desired features, function, and performance when executed.
2. **Data Structures:** Schema and structures that enable programs to adequately manipulate and process information.
3. **Documentation:** Descriptive technical manuals, specifications, user guides, and architecture blueprints.

Dual Role of Software

- **As a Product:** Delivers computing potential, processes data, and provides utility to users (e.g., CAD tools, enterprise databases).
- **As a Vehicle / Infrastructure:** Acts as the control mechanism for operating systems, networks, communication pipelines, and complex physical assets.

Taxonomy of Software Application Domains



Domain Drivers in Software Engineering

Classification governs non-functional requirements such as **safety, deterministic execution, latency, fault tolerance, and security**.

Traditional Domains: System & Application Software

System Software

- **Definition:** Programs designed to serve and manage physical computer hardware and execution environments.
- **Key Components:** Operating Systems, Compilers, Device Drivers, Linkers, Memory Managers.
- **SE Characteristics:**
  - Low-level hardware access.
  - High computational efficiency.
  - Strict performance budgets.
  - Concurrent memory operations.

Application Software

- **Definition:** Standalone programs that perform specific end-user business or technical requirements.
- **Key Components:** Enterprise Resource Planning (ERP), CAD/CAM suites, Word Processors.
- **SE Characteristics:**
  - User-centric workflow logic.
  - Data transaction integrity.
  - Rich graphical interface (GUI).
  - High maintainability needs.

Traditional Domains: Embedded Software & Web Applications

Embedded Software

- Resides in Read-Only Memory (ROM) or Microcontrollers inside non-computing hardware.
- Controls physical systems (automotive engine control units, medical devices, avionics).
- **Key SE Challenges:** Hard/soft real-time performance constraints, strict memory limits, fault recovery.

Web Applications (WebApps)

- Client-server software accessed via web browser interfaces over HTTP/HTTPS protocols.
- Hyper-distributed architectures with microservices, cloud deployments, and dynamic client UIs.
- **Key SE Challenges:** Scalability, global availability, security against web vectors, cross-browser compatibility.

Modern Domains: Artificial Intelligence (AI) Software

Nature of AI Software

Software that incorporates non-algorithmic pattern processing, statistical machine learning models, and deep learning algorithms to solve non-linear problems.

Key Software Engineering Paradigms for AI

- **Data-Driven Logic:** Functionality is learned from dataset training rather than explicitly coded step-by-step logic.
- **Non-Deterministic Output:** Identical inputs may yield probabilistic or continuous model updates requiring probabilistic testing frameworks.
- **MLOps & Lifecycle Pipeline:** SE processes extend beyond code lifecycle to include data ingestion, feature engineering, model training, validation, and continuous drift monitoring.

Specialized Domain: Green Hydrogen Ecosystem Software

Context in Clean Energy Engineering

Green Hydrogen (H_2) production via water electrolysis powered by dynamic renewable sources (solar, wind) introduces complex industrial cyber-physical challenges requiring specialized software.

Core Architecture of Green Hydrogen Software Systems

- **Industrial Control & SCADA:** Programmable Logic Controller (PLC) interface layer for physical stack operation.
- **Real-Time Telemetry & Edge Computing:** High-frequency sensor ingestion (pressure, temperature, flow rates, purity).
- **Grid Integration Middleware:** Real-time optimization interfacing with electrical power grids and renewable forecasting engines.

Software Engineering in Green Hydrogen Production

Role of Software in Hydrogen Generation

- **Electrolyser Process Control:** Software algorithms dynamically adjust current/voltage across Proton Exchange Membrane (PEM) or Alkaline stacks based on fluctuating renewable input.
- **Safety-Critical Automation:** Embedded software continuously monitors explosive lower explosive limit (LEL) oxygen/hydrogen ratios, executing safety emergency shutdown (ESD) sequences within milliseconds.
- **Predictive Maintenance Engines:** ML-assisted software evaluates electrode degradation, predicting stack re-stacking requirements to minimize plant downtime.

Software Engineering in Hydrogen Storage & Distribution

Role of Software in Logistics & Supply Chain

- **Compressor & Liquefaction Control:** Software manages multi-stage gas compression (700 bar) or cryogenic liquefaction (-253°C) under precise thermodynamic state models.
- **Pipeline & Fleet Dispatch Optimization:** Computational graph algorithms optimize tube-trailer routing, pipeline flow pressures, and refueling station inventory.
- **Digital Twin & Carbon Verification:** Software generates cryptographic / blockchain audit trails certifying green energy origin and compliance with international clean energy standards.

Comparative Matrix of Software Application Domains

Software Domain	Primary Focus	Key Quality Attribute	Typical Engineering Stack
System	Hardware control & resource abstraction	Computational efficiency & Reliability	C, C++, Assembly, Rust
Application	User task execution & workflow automation	Usability & Data integrity	Java, C#, Python, C++
Embedded	Dedicated hardware control	Determinism & Real-time latency	Embedded C, Ada, RTOS
Web Apps	Distributed internet services	Scalability & Availability	Node.js, React, Java, Go, SQL
AI / ML	Pattern inference & data learning	Accuracy, Robustness & Adaptability	Python, PyTorch, TensorFlow, CUDA
Green Hydrogen	Cyber-physical power & gas control	Safety, Reliability & Energy Efficiency	C/C++, Python, SCADA/PLC, ROS, IoT

Summary

Software Engineering principles apply universally, but process selection and non-functional priorities adapt strictly to the target application domain.

Course Context: GTU DI04000011 (Software Engineering)

This lecture focuses on the foundational architecture of software development methodologies, establishing how engineering principles govern software creation.

Key Learning Objectives:

- Understand Software Engineering as a **Layered Technology**.
- Explore the structure of the **Generic Process Model**.
- Examine the five core **Framework Activities**.
- Analyze the cross-cutting nature of **Umbrella Activities**.

Core Takeaway

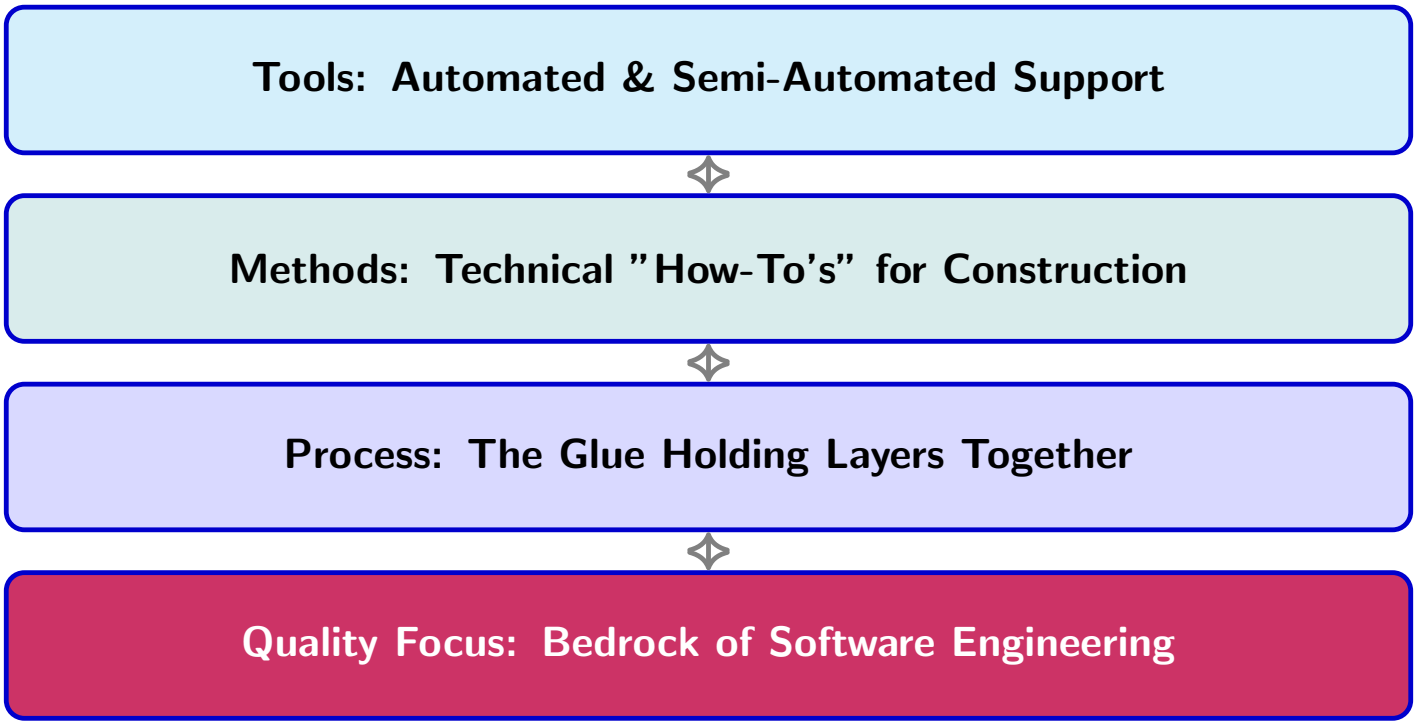
A software process provides the stability, control, and organization necessary to transform complex user requirements into high-quality software products.

Software Engineering: A Layered Technology

Software engineering is built on an organizational commitment to quality. It is structured into four distinct layers:

1. **Quality Focus (Bedrock):**
   - Fosters a continuous improvement culture (TQM, Six Sigma).
   - Establishes organizational standards and excellence metrics.
2. **Process Layer (The Glue):**
   - Defines the roadmap for timely development.
   - Controls milestone management, work products, and quality assurance.
3. **Methods Layer (The Technical "How-To's"):**
   - Covers requirements analysis, design modeling, code generation, and testing.
4. **Tools Layer (Automated Support):**
   - Provides automated and semi-automated toolchains (CASE tools, IDEs, CI/CD pipelines).

Visualizing the Layered Technology



The Generic Process Model Framework

A software process model is an abstract representation of the software development lifecycle (SDLC).

Process Framework Structure

A **Process Framework** establishes the foundation for a complete software engineering process by identifying a small number of **framework activities** that are applicable to all software projects regardless of size or complexity.

Key Elements of the Generic Model:

- **Framework Activities:** Core sequential or iterative steps.
- **Task Sets:** Specific work tasks, milestones, deliverables, and Quality Assurance (QA) checkpoints.
- **Umbrella Activities:** Continuous tasks spanning the entire SDLC.
- **Process Adaptability:** Customization based on project characteristics and risk profiles.

Framework Activities vs. Umbrella Activities

Table: Comparison of Process Components in Software Engineering

Dimension	Framework Activities	Umbrella Activities
Execution	Sequential or iterative phases	Continuous execution throughout SDLC
Focus	Direct creation of software artifacts	Oversight, quality control, and management
Scope	Phase-specific (e.g., coding, testing)	Cross-cutting across all phases
Examples	Communication, Planning, Construction	SQA, SCM, Risk Management, Reviews
Primary Goal	Transform requirements into code	Ensure process adherence, quality, and control

Generic Framework Activities (1–3)

The generic framework consists of five fundamental activities:

1. Communication

- Heavy customer and stakeholder collaboration.
- Requirements elicitation, discovery, and specification gathering.

2. Planning

- Defines the software engineering work plan (map).
- Describes technical risks, resource allocation, work tasks, and schedule.

3. Modeling

- Creation of models to better understand requirements and design.
- Includes Analysis Models (data/functional) and Design Models (architectural/UI).

Generic Framework Activities (4–5) & Task Sets

4. Construction

- Combines code generation (manual or automated programming) with testing.
- Unit testing, integration testing, and verification to reveal errors in code.

5. Deployment

- Delivery of the operational software product to the customer.
- Evaluation, feedback collection, ongoing support, and maintenance updates.

Concept of Task Sets: Each framework activity is populated by a **Task Set** comprising actual work tasks, work products (deliverables), quality assurance points, and project milestones tailored to project scope.

Umbrella Activities: Concept & Core Scope

Umbrella activities apply throughout a software project and help the team manage and control progress, quality, change, and risk.

- **Software Project Tracking and Control:** Assesses progress against the schedule plan; takes corrective action if delayed.
- **Risk Management:** Assesses risks that may affect project outcomes or quality (identification, analysis, mitigation).
- **Software Quality Assurance (SQA):** Defines and conducts activities to ensure software quality and standard compliance.
- **Formal Technical Reviews (FTR):** Evaluates software engineering work products to uncover and remove errors before propagation.

Essential Umbrella Activities Detailed

- **Measurement & Metrics:** Defines and collects process, project, and product measures to assist the team in delivering software that meets requirements.
- **Software Configuration Management (SCM):** Manages the effects of change throughout the software process (version control, change control).
- **Reusability Management:** Defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
- **Work Product Preparation and Production:** Includes activities required to create deliverables such as documentation, user manuals, and training materials.

Lecture Summary & Review Questions

Summary of Key Concepts

- Software engineering is a **layered technology**: Quality Focus, Process, Methods, Tools.
- The **Generic Process Model** blends Framework Activities (technical progress) with Umbrella Activities (management/quality).
- The 5 framework activities are Communication, Planning, Modeling, Construction, and Deployment.

GTU Exam Review Questions:

1. Explain Software Engineering as a Layered Technology with a neat diagram.
2. List and describe the 5 generic framework activities of the process model.
3. Explain the role of Umbrella Activities in software development.

Lecture Overview

This lecture introduces fundamental concepts of software process engineering within the GTU Software Engineering curriculum. It details the layered technology approach, generic process framework, core framework activities, and supporting umbrella activities.

Learning Objectives

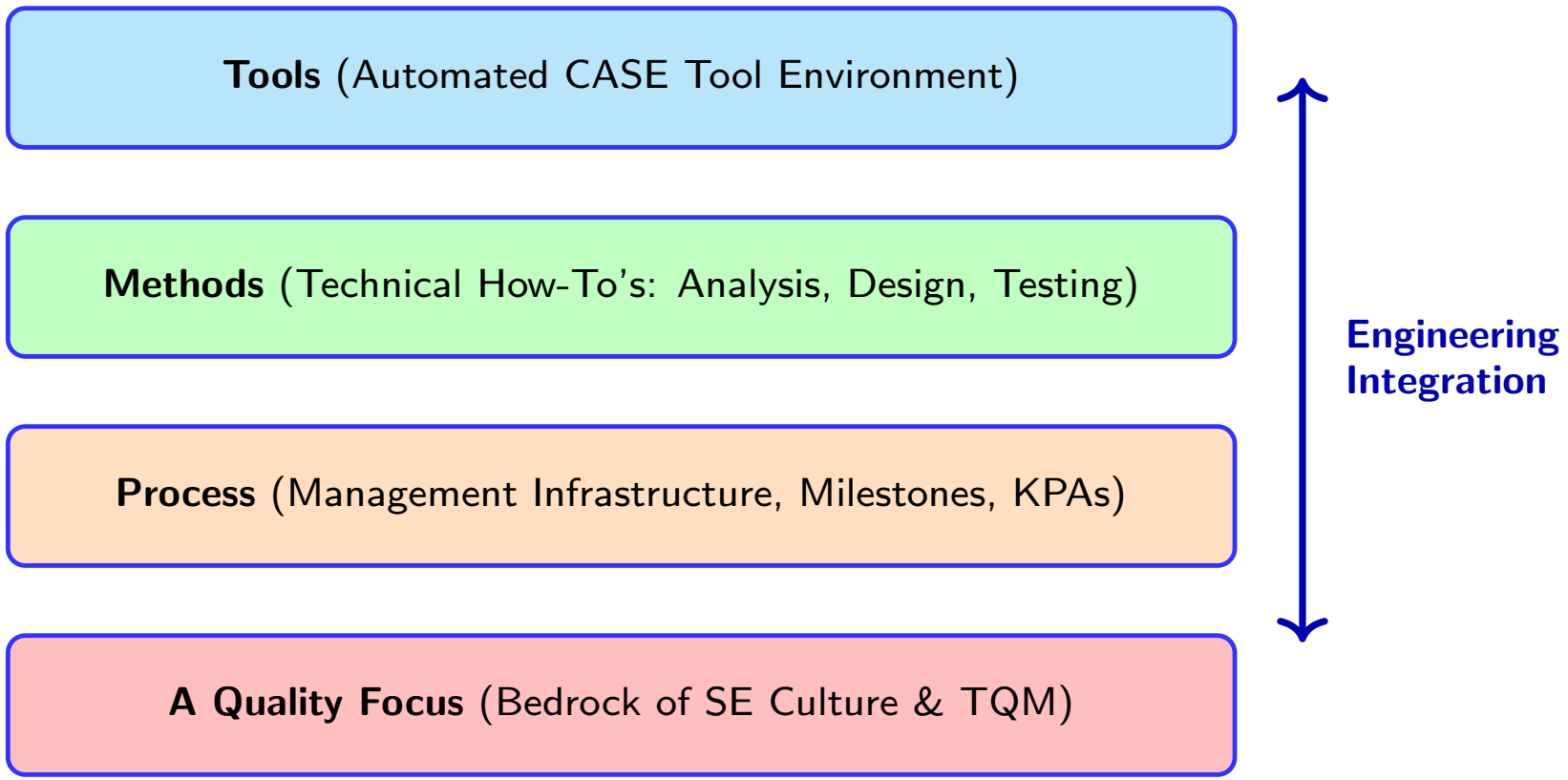
- Understand the **Layered Approach** to Software Engineering.
- Identify the structural elements of a **Generic Process Model**.
- Analyze the five **Generic Framework Activities**.
- Master the essential **Umbrella Activities** governing the software engineering lifecycle.

Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality.

- **Quality Focus:** The bedrock of software engineering. Continuous process improvement, Total Quality Management (TQM), and Six Sigma foster a quality-centric culture.
- **Process Layer:** The foundation for management control of software projects. Defines key process areas (KPAs), project milestones, deliverables, and quality checkpoints.
- **Methods Layer:** Provides technical "how-to's" for building software. Encompasses requirements engineering, architectural design, modeling, coding, testing, and maintenance.
- **Tools Layer:** Provides automated and semi-automated support for the process and methods (Computer-Aided Software Engineering - CASE tools).

The Layered Approach Diagram

Visualizing the 4 Layers of Software Engineering



Software Process Definition

A software process is defined as a structured set of activities, actions, and tasks executed to produce high-quality software work products.

- **Activity:** Strives to achieve a broad objective (e.g., stakeholder communication) regardless of application domain, project size, or complexity.
- **Action:** Encompasses a set of technical tasks that produce a major work product (e.g., creating an architectural design model).
- **Task:** Focuses on a small, well-defined objective (e.g., conducting a unit test or writing a specific class interface) that produces a tangible outcome.

A generic process framework for software engineering encompasses five fundamental activities applicable to all software projects:

1. Communication

- Requires heavy customer collaboration and stakeholder requirement gathering.
- Focuses on eliciting project scope, business requirements, domain constraints, and system objectives.

2. Planning

- Establishes a software engineering map describing technical tasks, potential risks, resource allocation, and work product schedules.
- Defines milestones, tracking strategies, and project effort estimation.

3. Modeling

- Creation of models to better understand requirements and design (Analysis & Design models).
- Establishes abstract representations of software architecture, data structures, and user interfaces.

4. Construction

- Code generation (manual programming or automated tools).
- Multi-level testing (unit, integration, and validation testing) to uncover errors.

5. Deployment

- Delivery of complete or incremental software releases to the customer for evaluation.
- Provisioning of operational support and gathering customer feedback.

Table: Comparison of Generic Process Flow Types

Process Flow	Execution Pattern	Typical Application
Linear Flow	Executes 5 framework activities in sequential order.	Well-defined, stable software requirements.
Iterative Flow	Repeats one or more activities before proceeding to the next.	Requirements needing continuous refinement.
Evolutionary Flow	Executes activities in a circular manner producing incremental releases.	Modern agile / rapid prototyping development.
Parallel Flow	Executes one or more activities concurrently with other activities.	Complex multi-component enterprise systems.

Umbrella activities occur throughout the software engineering process and apply continuously across all generic framework activities.

- **Software Project Tracking and Control:** Assesses progress against the project schedule and takes corrective action to maintain timelines and budget.
- **Risk Management:** Identifies, analyzes, prioritizes, mitigates, and monitors software risks that could impact quality or project delivery.
- **Software Quality Assurance (SQA):** Defines and conducts quality verification activities to ensure compliance with software standards.
- **Formal Technical Reviews (FTR):** Evaluates software engineering work products to uncover and correct defects early in the lifecycle.

- **Software Configuration Management (SCM):** Manages the impact of change throughout the software lifecycle (versioning, baseline control, and change requests).
- **Measurement & Metrics:** Defines and collects process, project, and product metrics to guide engineering decisions and quality assessment.
- **Reusability Management:** Establishes criteria for reusable software components and encourages asset reuse across projects.
- **Work Product Preparation and Production:** Encompasses creation of project deliverables such as technical documentation, user manuals, and installation guides.

Summary: Framework vs. Umbrella Activities

Lecture Conclusion & Comparative View

Table: Structural Comparison of Framework and Umbrella Activities

Dimension	**Framework Activities**	**Umbrella Activities**
Scope	Sequential or iterative development phases.	Pervasive across all development phases.
Primary Goal	Direct creation of system architecture & software products.	Project control, quality assurance, and risk management.
Examples	Communication, Planning, Modeling, Construction, Deployment.	Risk Management, SCM, SQA, FTR, Project Tracking.
Key Deliverable	Executable software and technical models.	Control metrics, audit logs, configuration baselines.

Lecture 7: Software Process Models – Overview

Subject Context

GTU Course Code: DI04000011 – Software Engineering
Unit: Software Process Models

Lecture Objectives

- Understand Software Engineering as a layered technology stack.
- Comprehend the structure and components of a Generic Process Model.
- Analyze the five core Generic Framework Activities.
- Identify essential Umbrella Activities and their role throughout the software lifecycle.

Key Core Concept

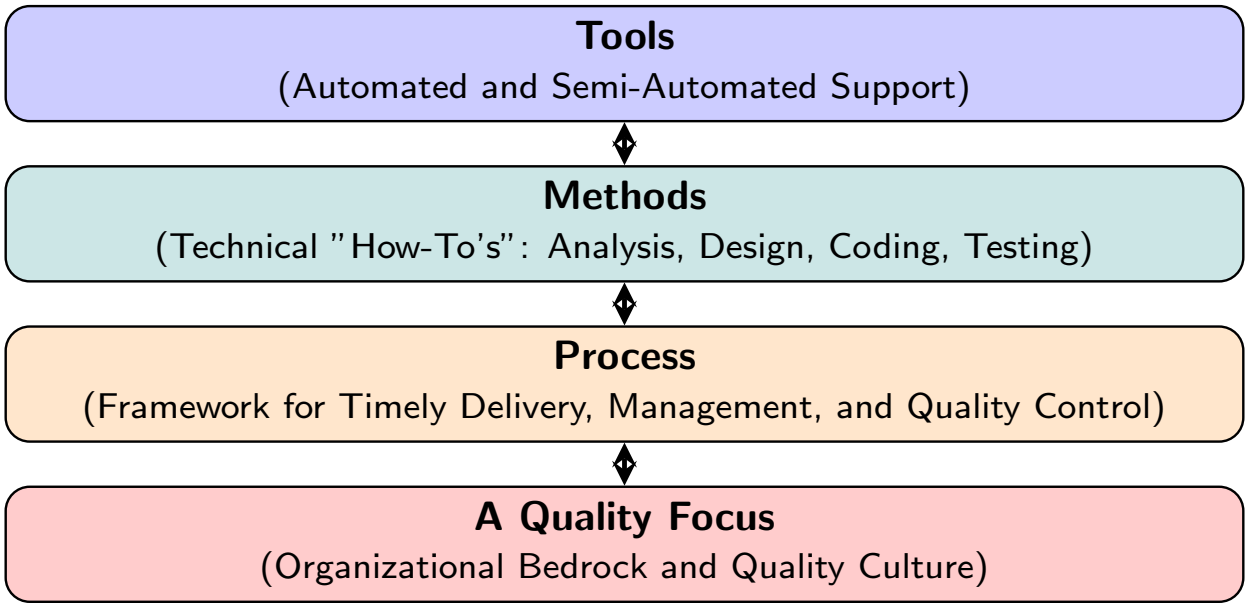
A software process provides the stability, control, and organization required to build complex software systems efficiently and with predictable quality.

Software Engineering: A Layered Approach

Software engineering is a layered technology. To achieve high quality, each layer builds upon the underlying discipline:

- **Quality Focus (Bedrock):** Any engineering approach must rest on an organizational commitment to quality. Total Quality Management (TQM), Six Sigma, and continuous improvement cultures form the baseline.
- **Process (Foundation):** The glue that binds the technology layers together. Defines a framework for timely delivery, management control, and technical risk mitigation.
- **Methods (The "How-to"):** Provide technical instructions for building software. Includes requirements analysis, design modeling, code generation, testing, and maintenance.
- **Tools (Automated Support):** Provide automated or semi-automated support for the process and methods (e.g., CASE tools, IDEs, automated testing suites, CI/CD pipelines).

Layered Technology Architecture of Software Engineering



Detailed Breakdown of Software Engineering Layers

Layer	Primary Objective	Key Deliverables / Focus
Tools	Automate framework and technical tasks	CASE Tools, Version Control, IDEs, Defect Trackers
Methods	Technical execution of development steps	Architecture Diagrams, Data Models, Test Cases, Code
Process	Establish milestones, controls, and workflows	Process Framework, Schedules, Risk Plans, Quality Audits
Quality Focus	Foster continuous process improvement	TQM, CMMI, ISO 9001 Compliance, Quality Metrics

Table: Summary of Software Engineering Layered Architecture

The Generic Process Model

A software process is defined as a collection of **activities**, **actions**, and **tasks**:

- Activity: Strives to achieve a broad objective (e.g., Communication with stakeholders) regardless of the application domain, project size, or complexity.
- Action: Encompasses a set of technical tasks that produce a major work product (e.g., Architectural Design Action within the Modeling activity).
- Task: Focuses on a small, well-defined objective that produces a specific, tangible outcome (e.g., Conducting a unit test for a specific module).

Process Framework Architecture

Process Framework = Generic Framework Activities + Umbrella Activities

Generic Framework Activities: Communication & Planning

1. Communication

- Critical initial activity to initiate project goals.
- Stakeholder identification and engagement.
- Requirements gathering and elicitation.
- Definition of domain boundary and project scope.

2. Planning

- Creates a map for the software engineering journey.
- Technical risk analysis and estimation.
- Work breakdown structure (WBS) creation.
- Resource allocation and project scheduling.

Generic Framework Activities: Modeling, Construction & Deployment

3. Modeling

Creation of models to better understand requirements and design.

- **Analysis Modeling:** Structural, behavioral, and functional representations.
- **Design Modeling:** System architecture, interface design, and component design.

4. Construction

Combines code generation (manual or automated) and rigorous testing (Unit, Integration, System testing) to uncover errors in the code.

5. Deployment

Software is delivered to the customer who evaluates the product and provides feedback based on evaluation.

Umbrella Activities: Concept and Significance

What are Umbrella Activities?

Umbrella activities occur **throughout the entire software process** and apply uniformly across all framework activities.

- They ensure project velocity, product quality, risk mitigation, and progress monitoring.
- Unlike framework activities which occur sequentially or iteratively in phases, umbrella activities run continuously from project inception to retirement.
- They protect the project against chaos, budget overruns, and scope creep.

Key Umbrella Activities in Software Engineering

- **Software Project Tracking & Control:** Assessing progress against planned schedules and taking corrective actions.
- **Risk Management:** Assessing, identifying, monitoring, and mitigating technical and managerial risks.
- **Software Quality Assurance (SQA):** Conducting activities to guarantee software quality.
- **Formal Technical Reviews (FTR):** Evaluating work products to uncover errors before propagation.
- **Measurement & Metrics:** Collecting process, project, and product measurements.
- **Software Configuration Management (SCM):** Managing change across artifacts.
- **Reusability Management:** Defining work product reuse guidelines.
- **Work Product Preparation & Production:** Creating documentation, user manuals, and logs.

Process Framework Integration & Summary

Interaction Matrix

Framework Activities (Communication, Planning, Modeling, Construction, Deployment) represent **what** we execute in stages.
Umbrella Activities (SQA, SCM, Risk Management, Tracking) represent **how** we maintain control and quality during execution.

Lecture 7 Takeaways

1. Software Engineering relies on a bedrock of **Quality** supported by Process, Methods, and Tools.
2. A process framework adapts to project characteristics through framework activity iteration.
3. Continuous **Umbrella Activities** are vital to successful, predictable software delivery.

Unit Overview: Software Process Models

A software process provides the framework for building high-quality software in a predictable, manageable, and repeatable manner.

Key Learning Objectives:

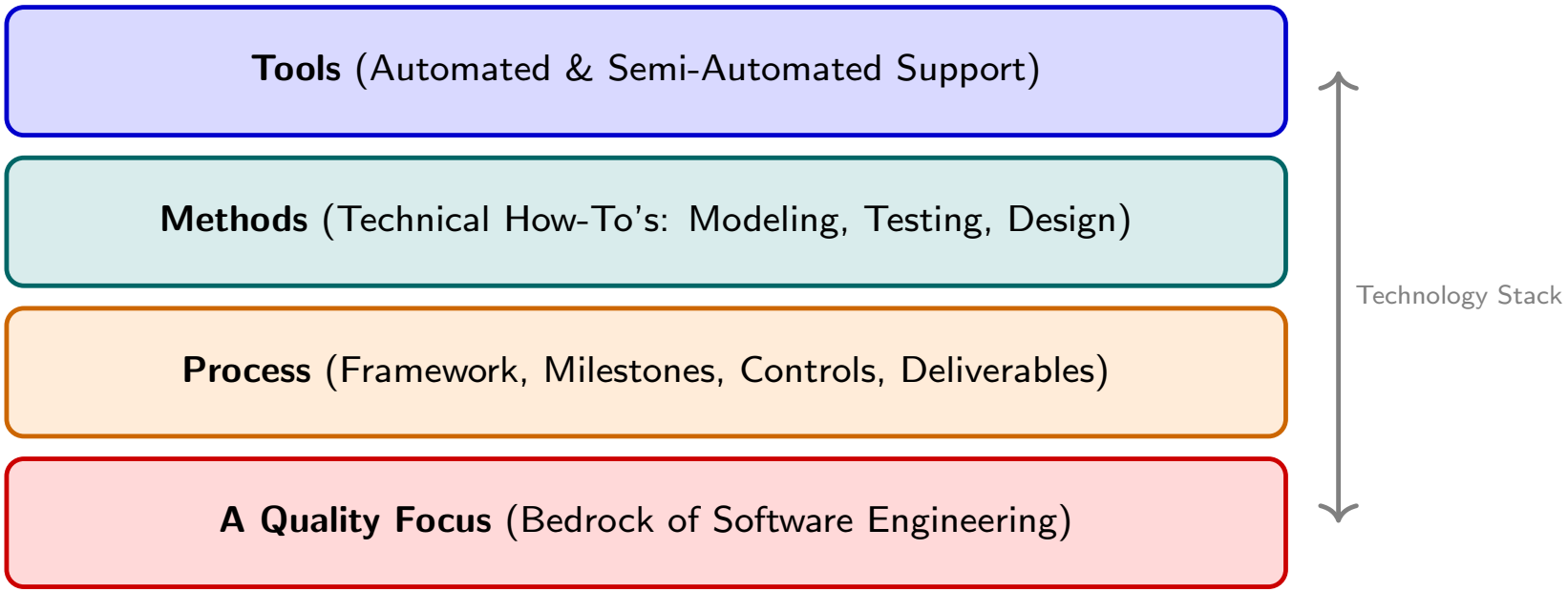
- Understand **Software Engineering as a Layered Technology**.
- Analyze the components of the **Generic Process Model**.
- Examine the five **Generic Framework Activities**.
- Explore **Umbrella Activities** and their role throughout software development.

Software engineering is an engineering discipline composed of four distinct layers built on top of organizational commitment.

1. **A Quality Focus:** The foundation/bedrock of any engineering approach. Fosters a continuous improvement culture.
2. **Process Layer:** The glue holding technology layers together. Defines framework, milestones, deliverables, and controls.
3. **Methods Layer:** Provides technical "how-to's" for building software (e.g., requirements analysis, design modeling, code generation, testing).
4. **Tools Layer:** Provides automated or semi-automated support for the process and methods (e.g., CASE tools, IDEs, CI/CD pipelines).

Layered Architecture of Software Engineering

Visual Representation



Key Takeaway

Without a solid **Quality Focus**, process models and advanced tools cannot guarantee reliable software delivery.

Definition of Process Model

A software process model is an abstract representation of a process architecture, establishing the workflow for software development tasks.

Structural Breakdown of the Generic Model:

- **Framework Activities:** Core engineering stages present in every software project regardless of size or complexity.
- **Task Sets:** Collection of software engineering work tasks, project milestones, work products, and quality assurance checkpoints for a given activity.
- **Umbrella Activities:** Operational controls executed across the entire process lifecycle.

1. Communication

Initiates project work through direct collaboration with stakeholders.

- **Requirements Elicitation:** Gathering business goals and software capabilities.
- **Collaboration:** Defining project scope, system boundary, and constraints.

2. Planning

Establishes a map for the software execution journey.

- **Estimation:** Calculating effort, cost, and human resources required.
- **Risk Analysis:** Identifying potential technical and management risks.
- **Scheduling & Tracking:** Defining milestones and timelines.

3. Modeling

Creation of representations to better understand requirements and design architecture.

- **Analysis Model:** Visualizing data, flow, behavior, and functions.
- **Design Model:** Establishing system architecture, interfaces, and component specs.

4. Construction

Combines code generation and comprehensive testing.

- **Coding:** Translating design models into operational executable code.
- **Testing:** Executing unit, integration, and system test suites to uncover defects.

5. Deployment

Delivery of software to the customer, ongoing support, and collecting user feedback.

Umbrella activities persist across all framework activities throughout the project lifecycle.

- **Software Project Tracking and Control:** Assesses progress against the plan; takes corrective action when schedule or budget deviates.
- **Risk Management:** Assesses risks that may impact outcome quality or project completion.
- **Software Quality Assurance (SQA):** Defines and conducts activities required to ensure software quality standards (IEEE/ISO) are maintained.

Umbrella Activities (Part 2)

Quality & Configuration Controls

- **Formal Technical Reviews (FTR):** Evaluates work products to uncover and correct errors before propagation into subsequent stages.
- **Measurement & Metrics:** Defines and collects process, project, and product measures to assist engineering management.
- **Software Configuration Management (SCM):** Manages and tracks changes to work products (code, documentation, models) across project versions.
- **Reusability Management:** Defines criteria for work product reuse and creates reusable component repositories.
- **Work Product Preparation and Production:** Creation of deliverables such as user manuals, technical specs, and installation guides.

Table: Framework Activities vs. Umbrella Activities

Dimension	Framework Activities	Umbrella Activities
Scope	Sequential or iterative phase steps	Continuous across entire lifecycle
Primary Goal	Produce software work products	Ensure quality, control change & risk
Execution Flow	Phase-by-phase development steps	Executed concurrently with framework
Core Examples	Communication, Planning, Modeling, Construction, Deployment	SQA, SCM, Technical Reviews, Risk Management, Measurement
Deliverables	Design models, Source code, Executable releases	SQA audit reports, Change logs, Review summary reports

Lecture Summary

Software engineering is a layered technology anchored on quality. A generic process model unites framework activities (phase-based) with umbrella activities (pervasive control).

GTU Exam Review Questions:

1. Explain Software Engineering as a Layered Technology with a suitable diagram. *[7 Marks]*
2. List and describe the five Generic Framework Activities in detail. *[7 Marks]*
3. What are Umbrella Activities? Explain any four umbrella activities with their importance. *[7 Marks]*

Software Process Model Definition

A **Software Process Model** is a simplified abstract representation of a software engineering process, defining distinct phases, task sets, milestones, and deliverables.

Classic Waterfall Model (Linear Sequential):

- **Requirements Analysis**: System specification & feasibility.
- **System Design**: Architectural & detailed design.
- **Implementation**: Source coding & unit testing.
- **Integration & Testing**: System verification.
- **Deployment & Maintenance**: Release & updates.

Key Characteristics & Fit:

- Rigid phase gates; output of one phase forms input for the next.
- Comprehensive, heavy documentation requirement.
- **Best Suited For**: Well-understood, stable requirements with low technical risk (e.g., embedded control systems, banking infrastructure).

V-Model (Verification and Validation Model)

Concept

An extension of the classic waterfall model that highlights the direct relationship between each development phase and its associated phase of software testing.

Verification Phase (Down Stream):

- **Requirements Spec** $\rightarrow$ Acceptance Test Design
- **System Architecture** $\rightarrow$ System Test Design
- **Module Design** $\rightarrow$ Integration Test Design
- **Coding** $\rightarrow$ Unit Test Execution

Validation Phase (Up Stream):

- Executes test suites corresponding to verification artifacts.
- Ensures early defect detection via design reviews.
- **Advantage**: High quality assurance for safety-critical domains (e.g., aerospace, medical devices).

Prototyping Model and Incremental Process Model

Prototyping Model

Applied when customer requirements are incomplete or ambiguous. Enables stakeholders to evaluate early mockups.

- **Throwaway Prototyping**: Prototype is discarded after requirement clarification.
- **Evolutionary Prototyping**: Prototype is incrementally refined into the final software product.

Incremental Process Model

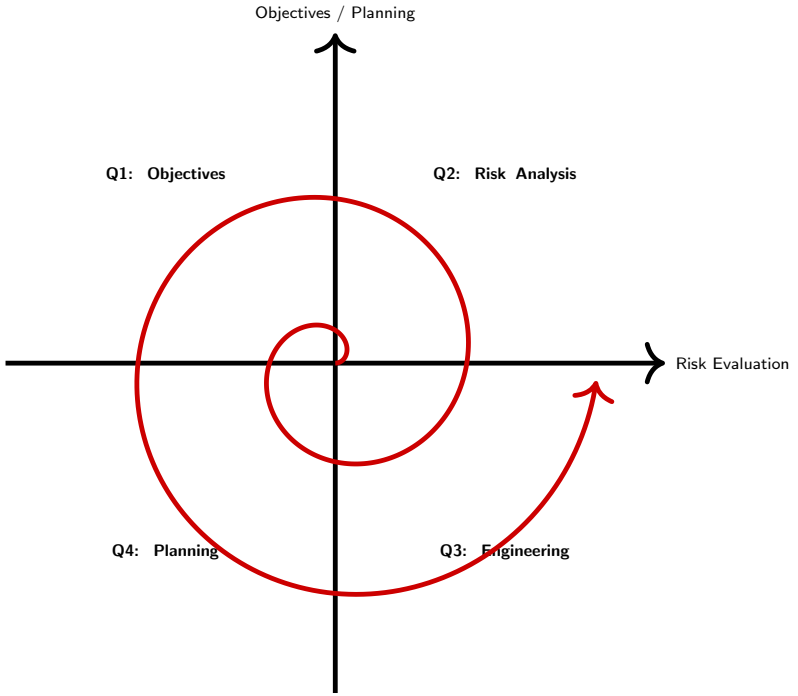
Combines linear sequential elements with iterative philosophy.

- Software is delivered in functional releases (increments), prioritizing core capabilities.
- Each increment passes through Requirements $\rightarrow$ Design $\rightarrow$ Code $\rightarrow$ Test.
- **Benefit**: Early availability of operational software and flexible risk mitigation.

Boehm's Spiral Model (Risk-Driven Approach)

Four Core Quadrants:

1. **Determine Objectives**: Identify goals, constraints, and alternative solutions.
2. **Risk Analysis & Evaluation**: Identify technical/business risks and construct prototypes.
3. **Engineering & Development**: Code, test, and verify the current release.
4. **Planning**: Review cycle progress and plan the next spiral iteration.



Comparative Analysis of Traditional Software Process Models

Dimension	Waterfall	V-Model	Prototype	Incremental	Spiral
Requirement Stability	High	Very High	Low	Medium	Low to Medium
Risk Management	Poor	Low	Medium	Medium	Explicit / Excellent
Customer Involvement	Low	Low	High	High	High
Flexibility to Change	Rigid	Rigid	Flexible	Moderate	Highly Flexible
Cost of Changes	High	High	Medium	Moderate	High
Development Style	Linear	Linear/V-shaped	Iterative	Incremental	Spiral-Iterative
Primary Application	Well-defined	Safety-Critical	UI / New Domains	Commercial Software	Large / High-Risk Enterprise

Agile Software Development: Values and Principles

Agile Manifesto Core Values

1. **Individuals and interactions** over processes and tools.
2. **Working software** over comprehensive documentation.
3. **Customer collaboration** over contract negotiation.
4. **Responding to change** over following a plan.

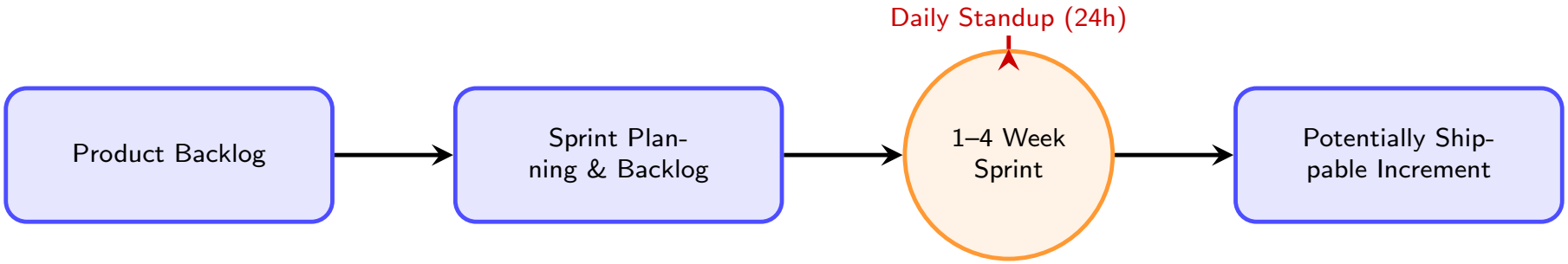
Key Agile Principles

- Satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development phases.
- Deliver working software frequently (every few weeks to months).
- Promote daily interaction between business stakeholders and developers.

Agile Frameworks: Scrum Methodology

1. **Scrum Roles:**
   - **Product Owner**: Maintains vision, prioritizes product backlog.
   - **Scrum Master**: Facilitates events, removes impediments.
   - **Development Team**: Cross-functional, self-organizing team.
2. **Scrum Events:**
   - **Sprint**: 1–4 week iteration cycle.
   - **Sprint Planning**: Define goal and backlog items.
   - **Daily Scrum**: 15-minute sync.
   - **Sprint Review & Retro**: Inspect and adapt.
3. **Scrum Artifacts:**
   - **Product Backlog**: Prioritized master feature list.
   - **Sprint Backlog**: Committed tasks for current sprint.
   - **Increment**: Shippable software release slice.

Scrum Process Flow (Process Lifecycle)



Tracking Metrics in Scrum

- **Sprint Velocity**: Rate of story points completed per sprint iteration.
- **Burn-down Chart**: Graphical representation of remaining work versus elapsed sprint time.

Agile Frameworks: Extreme Programming (XP)

Core Values of XP

Communication, **Simplicity**, **Feedback**, **Courage**, and **Respect**.

Engineering Practices:

- **Pair Programming**: Two developers work at one station (Driver & Navigator).
- **Test-Driven Development (TDD)**: Write automated tests prior to implementation code.
- **Refactoring**: Continual restructuring of code without altering external behavior.

Integration & Delivery Practices:

- **Continuous Integration**: Integrate source code multiple times daily with automated builds.
- **Small Releases**: Deploy small, working releases frequently.
- **On-Site Customer**: Real-time user feedback embedded within the development team.

Selection Framework: Traditional vs. Agile Models

Model Selection Decision Matrix

- **Requirements Volatility**: Select **Agile (Scrum/XP)** when requirements evolve dynamically; select **Waterfall/V-Model** when requirements are strictly fixed.
- **Project Scale & Complexity**: High-risk, complex architectures benefit from **Spiral/V-Model**; fast-paced web and mobile solutions fit **Agile**.
- **Team Structure & Culture**: **Agile** requires self-organizing, highly collaborative teams; **Traditional** models rely on role specialization and structured documentation.
- **Customer Availability**: **Agile** requires active on-site/continuous customer involvement; **Traditional** models focus customer involvement at initial requirements and final acceptance.

What is a Software Process Model?

A **Software Process Model** is an abstract representation of a software engineering process. It establishes a systematic framework for the activities, actions, tasks, milestones, and deliverables required to build high-quality software.

Generic Process Framework Activities:

- **Communication:** Customer collaboration & requirement gathering.
- **Planning:** Estimation, scheduling, & risk analysis.
- **Modeling:** Architectural design & analysis.
- **Construction:** Code generation & quality testing.
- **Deployment:** Delivery, evaluation, & feedback.

Primary Paradigms:

- **Prescriptive / Traditional Models:** Strive for structure, predictability, detailed documentation, and fixed planning (e.g., Waterfall, V-Model).
- **Agile / Adaptive Models:** Emphasize flexibility, continuous customer feedback, working software, and rapid response to change (e.g., Scrum, XP).

Traditional Models: Classic Waterfall Model

Linear Sequential Life Cycle (Winston Royce, 1970)

The **Waterfall Model** suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and support.

Key Characteristics:

- Rigid stage gates: A phase must be completed before the next begins.
- Heavy emphasis on baseline documentation at each stage.
- Quality assurance via formal milestone reviews.

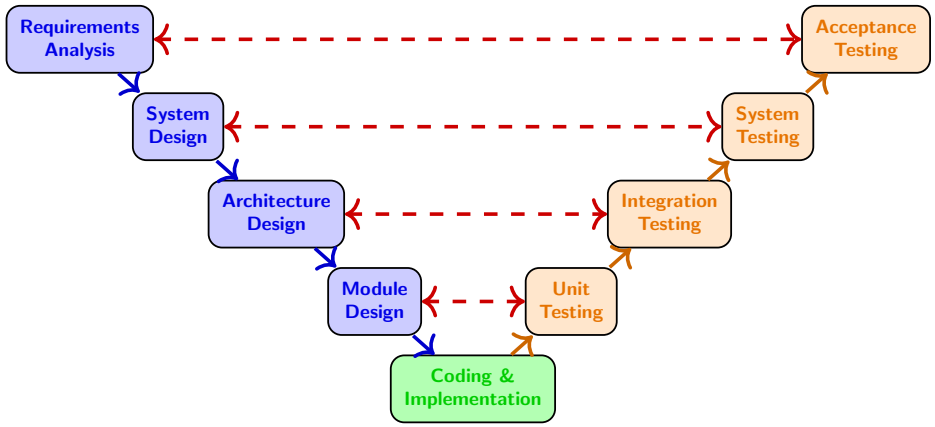
Trade-offs & Challenges:

- **Inflexibility:** Difficult to accommodate change requests late in the cycle.
- **Blocking States:** Team members wait for predecessor deliverables.
- **Late Working Code:** Operational software is unavailable until late in the life cycle.

Traditional Models: The V-Model

Verification & Validation (V&V)

- **Verification (Left Arm):** Ensuring the software is built correctly according to specifications.
- **Validation (Right Arm):** Ensuring the right software is built to satisfy user needs.
- **Early Test Planning:** Test cases are designed concurrently with their corresponding specification phases.



Evolutionary Models: Prototype Model

Concept

When customer requirements are vague or technical feasibility is uncertain, a **Prototype** (a working mock-up of the software) is developed to elicit precise requirements and validate software designs.

Prototyping Workflow:

1. **Quick Communication:** Identify known objectives.
2. **Quick Plan & Design:** Focus on user interface & visible aspects.
3. **Build Prototype:** Create working model.
4. **User Evaluation:** Gather stakeholder feedback.
5. **Refinement:** Iterate until requirements freeze.

Types & Risks:

- **Throwaway Prototyping:** Prototype is discarded after requirement clarity.
- **Evolutionary Prototyping:** Prototype becomes the core system baseline.
- **Pitfalls:** Stakeholders may confuse mock-ups with final products; developers may adopt poor sub-optimal code fixes permanently.

Iterative Models: Incremental Process Model

Definition

The **Incremental Model** combines elements of linear and parallel process flows. It delivers operational software functionality in a series of manageable releases called **Increments**.

- **First Increment (Core Product):** Basic requirements are addressed, providing fundamental functionality to the client.
- **Subsequent Increments:** Add secondary features, optimizations, and customer-requested modifications based on operational feedback.
- **Parallel Activity:** Development of Increment $N + 1$ can overlap with deployment of Increment N.

Advantages over Waterfall

- Early delivery of valuable, usable core software to end users.
- Reduces overall project risk by breaking large systems into smaller sub-projects.
- Facilitates accommodating changing requirements during later increments.

Risk-Driven Models: Boehm's Spiral Model

Overview

Proposed by Barry Boehm (1988), the **Spiral Model** is an evolutionary process model that couples the iterative nature of prototyping with the controlled, systematic aspects of the Waterfall model, adding explicit **Risk Analysis**.

Four Quadrants per Circuit (Loop):

1. **Determine Objectives:** Define goals, constraints, and alternative solutions.
2. **Identify & Resolve Risks:** Conduct risk evaluations, benchmarking, and prototyping.
3. **Development & Testing:** Build and test current level increment.
4. **Plan Next Phase:** Review results and plan the next spiral iteration.

Strategic Applicability:

- Ideal for high-risk, large-scale, complex mission-critical systems.
- Requires significant expertise in risk assessment and mitigation.
- Continuous refinement prevents major system-level failures downstream.

Agile Approach: Principles and Philosophy

The Agile Manifesto (2001)

Agile software development emphasizes adaptive planning, evolutionary development, early delivery, and continual improvement, encouraging flexible responses to change.

Four Core Agile Values:

- **Individuals & Interactions** over processes and tools.
- **Working Software** over comprehensive documentation.
- **Customer Collaboration** over contract negotiation.
- **Responding to Change** over following a plan.

Key Agile Principles:

- Satisfy customer through continuous delivery of valuable software.
- Welcome changing requirements, even late in development.
- Deliver working software frequently (weeks rather than months).
- Business people and developers must work together daily.

Scrum Architecture

Scrum is an iterative, incremental Agile framework for managing complex software development. It relies on fixed-duration iterations called **Sprints** (typically 1 to 4 weeks).

Three Core Roles:

- **Product Owner:** Manages Product Backlog & prioritizes business values.
- **Scrum Master:** Facilitates process, removes blockers, enforces Scrum rules.
- **Development Team:** Cross-functional, self-organizing engineering group.

Key Scrum Ceremonies:

- **Sprint Planning:** Select items for Sprint Backlog.
- **Daily Standup:** 15-minute daily synchronization meeting.
- **Sprint Review:** Demonstrate working product increment.
- **Sprint Retrospective:** Process review & continuous team improvement.

Agile Frameworks: Extreme Programming (XP)

Definition

Created by Kent Beck, **Extreme Programming (XP)** is an Agile software development framework designed to improve software quality and responsiveness to changing customer requirements through high-discipline technical practices.

Core Values of XP:

- Communication, Simplicity, Feedback, Courage, and Respect.

Key Technical Practices:

- **Pair Programming:** Two developers code at a single workstation (Driver & Navigator).
- **Test-Driven Development (TDD):** Write automated unit tests before writing code.

Engineering Rigor:

- **Continuous Integration (CI):** Integrate and test software code several times a day.
- **Refactoring:** Continuous design optimization without changing external functionality.
- **On-site Customer:** Dedicated user representative for immediate requirement clarification.

Comparative Analysis: Process Models Evaluation

Table: Comprehensive Comparison of Software Process Models

Model	Req. Flexibility	Risk Management	Customer Involvement	Delivery Type	Best Suited For
Waterfall	Very Low	Low (Late risk identification)	Initial & Final stages	Single final release	Well-defined, stable requirements
V-Model	Low	Medium (Early test planning)	Initial & Final stages	Single final release	Safety-critical, high-reliability systems
Prototype	High	Medium (Unclear user needs)	High (Continuous feedback)	Working prototype → final	Unclear/evolving user requirements
Spiral	High	Very High (Explicit risk assessment)	High at end of each loop	Incremental releases	Large, complex, high-risk projects
Incremental	Medium	Medium	Moderate per increment	Staged working software	Projects with urgent core features
Scrum (Agile)	Very High	High (Iterative inspection)	Constant (Product Owner)	Sprint increments (1–4 wks)	Dynamic, fast-changing requirements
XP (Agile)	Extremely High	High (Continuous testing)	Constant (On-site customer)	Frequent mini-releases	Small-medium teams, rapid code changes

What is a Software Process?

A **Software Process** is a structured set of activities, actions, and tasks required to transform user requirements into an operational software system. It defines *who* is doing *what*, *when*, and *how* to reach a target goal.

Core SDLC Activities

- **Specification:** Defining system functions and constraints.
- **Design & Implementation:** Structuring and coding software.
- **Validation:** Ensuring software meets user requirements.
- **Evolution:** Modifying software to meet changing needs.

Process Taxonomy

- **Prescriptive / Traditional:** Plan-driven processes (e.g., Waterfall, V-Model) with linear, defined execution paths.
- **Evolutionary / Iterative:** Incremental & risk-driven paths (e.g., Spiral, Prototype).
- **Agile Frameworks:** Adaptive, iterative processes prioritizing rapid deployment (e.g., Scrum, XP).

Traditional Prescriptive Models: Waterfall & V-Model

1. Classic Waterfall Model (Linear Sequential Model)

Proposed by Winston Royce (1970). Progress flows steadily downward through defined phases:

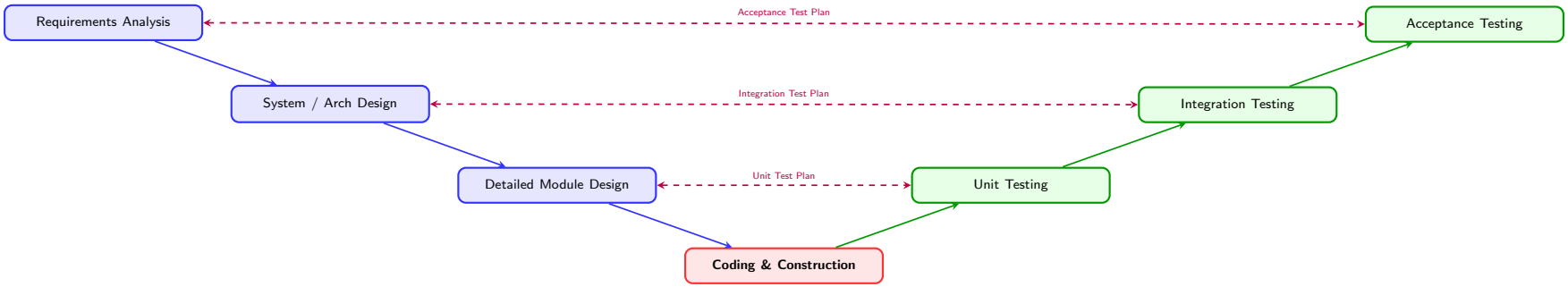
- **Requirement Analysis** → **System Design** → **Implementation** → **Integration & Testing** → **Deployment** → **Maintenance**.
- **Key Trait:** Rigid phase gating; each phase must complete with formal deliverables before the next begins. High documentation overhead.

2. V-Model (Verification and Validation Model)

An extension of the Waterfall model emphasizing test execution mapped directly to early lifecycle phases:

- **Verification Branch (Down-V):** Requirements Analysis, Architectural Design, Detailed Design, Code Construction.
- **Validation Branch (Up-V):** Unit Testing, Integration Testing, System Testing, Acceptance Testing.
- **Advantage:** Early test planning prevents defect propagation down the execution pipeline.

Architectural Diagram: The V-Model Lifecycle



Prototyping Model

Ideal when customer requirements are fuzzy or poorly understood.

- **Workflow:** Quick Requirements → Quick Design → Prototype Construction → Customer Evaluation → Prototype Refinement.
- **Types:** Throwaway Prototyping (discarded after validation) vs. Evolutionary Prototyping (refined into final system).
- **Drawback:** Risk of client misinterpreting prototype as final product; poor architectural foundations if rushed.

Incremental Process Model

Combines elements of linear and iterative process flows.

- Software is partitioned into operational releases called **Increments**.
- **Increment 1:** Core module (essential requirements).
- **Subsequent Increments:** Additional functional modules built and integrated sequentially.
- **Benefits:** Early deployment of functional baseline, reduced risk of total project failure.

Risk-Driven Engineering: Boehm's Spiral Model

The Spiral Model Architecture

Designed by Barry Boehm (1988). Integrates the iterative nature of prototyping with the controlled, systematic aspects of the Waterfall model, explicitly driven by **Risk Analysis**.

Four Cyclic Quadrants

1. **Objective Setting:** Identify quadrant goals, alternative paths, and constraints.
2. **Risk Assessment & Resolution:** Evaluate risks, construct prototypes, perform simulation.
3. **Engineering & Development:** Code, test, and validate software release.
4. **Planning Next Phase:** Review progress and plan the next spiral turn.

Critical Spiral Aspects

- **Angular Dimension:** Represents cumulative progress in completing SDLC steps.
- **Radial Dimension:** Represents cumulative cost incurred.
- **Best Suited For:** Large, high-risk, mission-critical systems requiring continuous risk mitigation.

Comparative Matrix of Traditional Process Models

Table: Evaluation of Prescriptive Software Process Models

Feature	Waterfall	V-Model	Incremental	Prototyping	Spiral
Requirement Stability	Highly Stable	Highly Stable	Moderate / Evolving	Unclear / Dynamic	Unclear / Complex
Risk Management	Low (Late testing)	Moderate (Early planning)	Moderate	Low to Medium	High (Explicit Phase)
User Involvement	Low (Beginning & End)	Low (Requirements/UAT)	Medium (Per release)	High (Continuous feedback)	High (Every iteration)
Cost & Complexity	Low / Predictable	Low to Medium	Medium	Low to Medium	High
Product Delivery	Late (At final stage)	Late	Early core, gradual add-ons	Prototypes early, code late	Incremental releases
Project Size	Small to Medium	Critical systems	Medium to Large	Medium with uncertain specs	Large, high-risk

Agile Approach: Philosophy & Principles

The Agile Manifesto (2001)

Agile is an iterative, adaptive paradigm emphasizing customer satisfaction and software delivery over process rigidity.

4 Core Agile Values

1. **Individuals and interactions** over processes and tools.
2. **Working software** over comprehensive documentation.
3. **Customer collaboration** over contract negotiation.
4. **Responding to change** over following a plan.

Key Agile Principles

- Deliver working software frequently (weeks rather than months).
- Welcome changing requirements, even late in development.
- Build projects around motivated, self-organizing teams.
- Working software is the primary measure of progress.

Agile Frameworks: Scrum Architecture

Scrum Overview

An agile framework for managing complex software development through short, fixed-length timeboxes called **Sprints** (typically 1 to 4 weeks).

Scrum Roles

- **Product Owner:** Manages product backlog and prioritizes business value.
- **Scrum Master:** Facilitator, removes impediments.
- **Dev Team:** Cross-functional, self-organizing.

Scrum Artifacts

- **Product Backlog:** Prioritized list of all desired features.
- **Sprint Backlog:** Tasks selected for current sprint.
- **Increment:** Potentially shippable software unit.

Scrum Events

- **Sprint Planning**
- **Daily Standup (15m)**
- **Sprint Review**
- **Sprint Retrospective**

Extreme Programming (XP) Paradigm

Created by Kent Beck. Focuses on software engineering best practices pushed to "extreme" levels to achieve customer satisfaction and ultra-high code quality.

Core XP Values

- **Communication:** Continuous interaction between dev & customer.
- **Simplicity:** Do what is needed, nothing more (YAGNI principle).
- **Feedback:** Instant feedback via automated tests.
- **Courage & Respect:** Refactor code boldly.

Key XP Engineering Practices

- **Pair Programming:** Two developers write code together at one workstation.
- **Test-Driven Development (TDD):** Write automated test before writing code.
- **Continuous Integration:** Code integrated multiple times daily.
- **Refactoring:** Restructure code without changing behavior.
- **Collective Ownership:** Anyone can edit any part of the codebase.

Traditional vs. Agile Paradigm: Selection Guidelines

Traditional / Plan-Driven

- **Best for:** Systems with strict safety/regulatory compliance (medical, aerospace, defense).
- **Requirements:** Clear, fixed, well-understood up front.
- **Architecture:** Heavy upfront design; low tolerance for rework.
- **Team Structure:** Hierarchical with specialized roles.

Agile / Adaptive

- **Best for:** Dynamic commercial web/mobile apps, startups, evolving markets.
- **Requirements:** Rapidly changing or emerging user needs.
- **Architecture:** Incremental design; refactoring as part of daily flow.
- **Team Structure:** Cross-functional, collaborative, self-organizing.

GTU DI04000011 Process Selection Summary

No single process model fits all project contexts. Selection depends on: (1) System Risk Level, (2) Requirement Volatility, (3) Customer Availability, and (4) Team Size and Expertise.

Unit: Software Requirement Analysis and Design

Requirements Engineering (RE) is the systematic process of discovering, analyzing, documenting, and maintaining software requirements throughout the Software Development Life Cycle (SDLC).

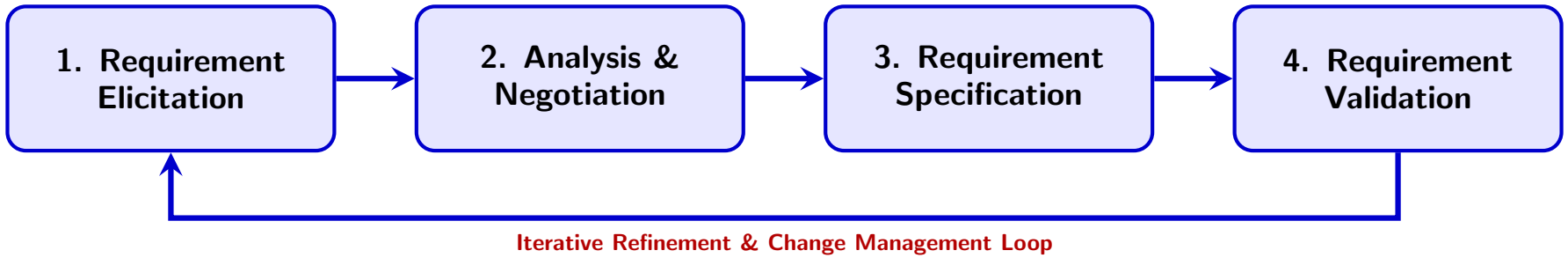
Core Objectives of Lecture 12:

- Understand the Fundamentals of Requirement Elicitation.
- Classify Requirements: Functional vs. Non-functional.
- Analyze Techniques for Requirement Gathering.
- Identify Common Challenges in Requirement Analysis.

Why Requirement Analysis Matters?

- Prevents scope creep and costly rework.
- Establishes a formal baseline for testing and validation.
- Bridges communication gaps between stakeholders and developers.

The Requirements Engineering Process Pipeline



- **Elicitation:** Gathering raw requirements from users, domain experts, and system constraints.
- **Analysis:** Resolving conflicts, checking feasibility, and prioritizing requirements.
- **Specification:** Formulating the Software Requirements Specification (SRS) document.
- **Validation:** Ensuring requirements match real user expectations and compliance rules.

Classification of Software Requirements

Software Requirements are broadly categorized into two fundamental domains:

Functional Requirements (FR)

- Describe **WHAT** the system must do.
- Focus on features, functions, and behavioral responses to specific inputs.
- Directly mapped to business tasks and user operations.
- Verified through functional and system testing.

Non-Functional Requirements (NFR)

- Describe **HOW WELL** the system performs.
- Focus on quality attributes, operational constraints, and system characteristics.
- Dictate architecture, deployment, and infrastructure.
- Verified via specialized performance/stress testing.

Functional Requirements (FR) - Deep Dive

Definition

Statements of capabilities, services, and functions that a system must provide to enable users to complete specific tasks.

Key Components of Functional Requirements:

1. **Business Rules:** Automated calculations, workflow logic, access levels.
2. **User Interactions:** Form submissions, search operations, authentication processes.
3. **External Interfaces:** Integration with payment gateways, third-party APIs, hardware devices.
4. **Data Operations:** CRUD (Create, Read, Update, Delete) operations and transaction history.

Concrete Examples in Software Engineering

- "The system shall send an automated email confirmation within 5 seconds of order placement."
- "The portal shall allow admin users to revoke role access permissions."

Non-Functional Requirements (NFR) & Quality Attributes

NFRs define the system's operational parameters across several key quality dimensions (URPS/FURPS+ model):

- **Performance & Scalability:** Throughput (TPS), response latency, bandwidth capacity.
- **Reliability & Availability:** Mean Time Between Failures (MTBF), system uptime percentage (e.g., 99.99% SLA).
- **Security & Compliance:** Data encryption (AES-256), OAuth2/JWT authentication, GDPR/ISO-27001 standards.
- **Usability & Accessibility:** Navigation efficiency, WCAG 2.1 compliance, UI responsiveness.
- **Maintainability & Portability:** Modular architecture, containerization (Docker), platform compatibility.

NFR Measurement Metric Example

Bad NFR: "The system must be fast and secure."
Good NFR: "The search query API must return responses in under 200 ms for up to 10,000 concurrent active users."

Comparison: Functional vs Non-Functional Requirements

Table: Comparative Analysis Matrix

Dimension	**Functional Requirements (FR)**	**Non-Functional Requirements (NFR)**
Definition	Specifies functions and behaviors the system must execute.	Specifies quality attributes, performance, and constraints.
Primary Focus	*What* the system does.	*How well* the system operates.
Origin	User tasks, business processes, and domain requirements.	Technical constraints, architectural targets, regulatory standards.
Verification	Checked via Pass/Fail functional testing and Use Case validation.	Evaluated using quantitative metrics (load, security, stress tests).
Failure Impact	System fails to execute a required business operation.	System executes operations, but becomes slow, insecure, or unstable.
Example	"Process credit card transactions via payment gateway."	"Complete payment processing within 1.5 seconds under peak load."

Techniques for Requirement Gathering: Interactive Methods

Requirement Elicitation relies on structured communication channels to extract domain knowledge.

Interviews

- **Structured:** Pre-defined questionnaire for systematic data gathering.
- **Unstructured:** Open-ended discussion to discover hidden requirements.
- **Pros/Cons:** High detail, but time-intensive and subject to bias.

Joint Application Design (JAD)

- Highly structured, facilitated workshops bringing together stakeholders, developers, and users.
- Accelerates consensus and reduces requirement drift.
- Ideal for complex enterprise systems.

Techniques for Requirement Gathering: Empirical & Analytical

When direct interviews are insufficient or user groups are distributed, analytical techniques are employed.

- **Surveys and Questionnaires:**
  - Effective for gathering quantitative feedback from a large, geographically dispersed user base.
  - Requires carefully designed, non-ambiguous questions (Likert scale, multiple choice).
- **Document Analysis / Legacy Reverse Engineering:**
  - Reviewing existing process documents, user manuals, system logs, and legacy source code.
  - Helps identify baseline business rules and legacy system constraints.
- **Interface Analysis:**
  - Inspecting data interchange formats (JSON, XML) across external APIs, hardware peripherals, and database schemas.

Techniques for Requirement Gathering: Observation & Prototyping

Observational and model-driven techniques help resolve ambiguous stakeholder requests.

On-Site Observation (Ethnography)

- Engineers observe end-users in their operational environment.
- **Passive Observation:** Watching workflow without intervention.
- **Active Observation:** Interacting with users while they perform tasks.
- Uncovers unstated "implicit" requirements.

Software Prototyping

- **Throwaway Prototyping:** Quick UI wireframes to confirm requirements, then discarded.
- **Evolutionary Prototyping:** Incremental working builds refined into final product.
- Provides immediate visual feedback to stakeholders.

Challenges in Requirement Elicitation & Lecture Summary

Common Challenges in Requirement Gathering

- **Ambiguity:** Vague language leading to misinterpretation between client and developers.
- **Requirement Volatility:** Uncontrolled changes during development causing budget and timeline slips.
- **Conflicting Stakeholder Needs:** Differing priorities between business executives and end-users.
- **Tacit Knowledge Problem:** Users know how to do work but struggle to articulate specific rules.

Lecture 12 Key Takeaway

Successful software design begins with rigorous Requirement Gathering. Combining interactive techniques (JAD/Interviews) with prototyping ensures both Functional and Non-functional expectations are accurately specified in the SRS.

Subject: Software Engineering (GTU Code: DI04000011)

Unit: Software Requirement Analysis and Design
Topic Focus: Requirement Gathering and Analysis

Learning Objectives:

- Understand the role of Requirements Engineering in the Software Development Life Cycle (SDLC).
- Differentiate clearly between **Functional Requirements (FRs)** and **Non-Functional Requirements (NFRs)**.
- Explore standard software engineering techniques for requirement gathering and elicitation.
- Analyze stakeholder requirements to resolve ambiguities, conflicts, and scope boundaries.

What is a Software Requirement?

A condition or capability needed by a user to solve a problem or achieve an objective, or a condition/capability that must be met by a system to satisfy a contract, standard, or specification (IEEE Std 610.12).

The Requirement Engineering (RE) Process Cycle:

1. **Requirement Elicitation:** Gathering raw requirements from stakeholders.
2. **Requirement Analysis:** Refining, categorizing, and checking for feasibility.
3. **Requirement Specification:** Documenting requirements formally in the SRS.
4. **Requirement Validation:** Ensuring requirements reflect user expectations.
5. **Requirement Management:** Controlling changes throughout project lifecycle.

Functional Requirements (FRs)

Definition of Functional Requirements

Functional Requirements specify the **specific behaviors, functions, and operations** that the software system must perform. They define *what* the system should do in response to specific inputs and conditions.

Key Characteristics of Functional Requirements:

- Describes user interactions, automated data processing, and system workflows.
- Directly verifiable through functional, unit, and integration testing.

Examples in Software Systems:

- *Authentication:* "The system shall allow users to log in using OTP authentication."
- *Transaction:* "The system shall calculate total order cost including taxes and shipping."
- *Reporting:* "The system shall generate a PDF monthly sales report at 23:50 daily."

Non-Functional Requirements (NFRs)

Definition of Non-Functional Requirements

Non-Functional Requirements define the **quality attributes, performance criteria, and system constraints**. They specify *how well* the system must perform its functions rather than *what* functions it performs.

Core NFR Categories (ISO/IEC 25010 Quality Model):

- **Performance Efficiency:** Throughput, response time, resource utilization.
- **Reliability & Availability:** Mean Time Between Failures (MTBF), system uptime (e.g., 99.99%).
- **Security:** Data encryption (AES-256), role-based access control (RBAC).
- **Usability:** Accessibility compliance, user interface learnability time.
- **Maintainability & Scalability:** Modularity, horizontal load scaling capability.

Comparative Analysis: Functional vs. Non-Functional

Dimension	Functional Requirements (FR)	Non-Functional Requirements (NF
Primary Focus	System capabilities, commands, and inputs/outputs.	System qualities, operation straints, and performance.
Question Answered	*What* does the system do?	*How well* does the system perform
Origin	Derived from user tasks and business logic requirements.	Derived from technical architecture curity, and quality needs.
Testing Method	Functional, System, and Black-box testing.	Load, Stress, Security, Usability, Benchmark testing.
Failure Impact	Specific feature fails to execute correctly.	System performance degrades, secu breached, or system crashes.
Example	"System allows user to transfer funds."	"Fund transfer completes in < 1.5 onds."

Why is Requirement Gathering Difficult?

Requirement elicitation is often called the hardest part of software engineering due to communication barriers, evolving requirements, and domain complexity.

Major Elicitation Challenges:

- **The Tacit Knowledge Problem:** Stakeholders struggle to articulate routine tasks.
- **Scope Creep:** Continuous unmanaged addition of features during elicitation.
- **Conflicting Requirements:** Different user groups have contradictory priorities.
- **Ambiguity & Vagueness:** Usage of imprecise terms like "fast", "user-friendly", or "robust".
- **Domain Gap:** Communication disconnect between technical team and domain experts.

Traditional Elicitation Techniques

Established methods used to gather baseline software requirements from documentation and individuals.

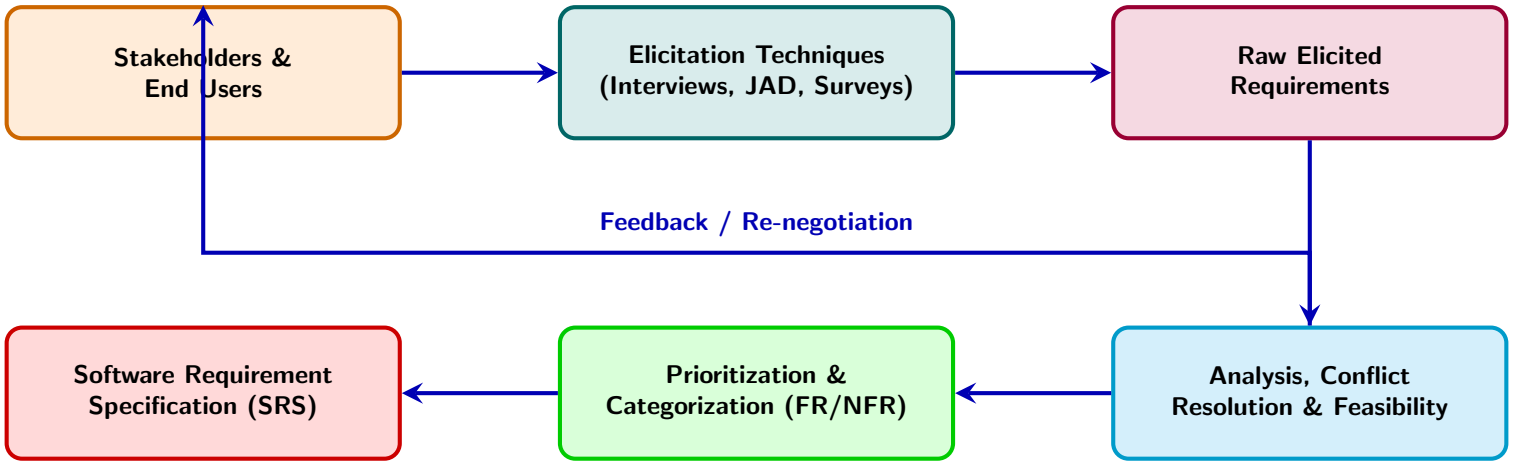
- **Interviews:**
  - *Structured:* Predefined questionnaire to get specific data points.
  - *Unstructured:* Open-ended discussion to discover unexpected domain insights.
- **Questionnaires & Surveys:**
  - Efficient for gathering feedback from a large, geographically dispersed user base.
  - Useful for statistical validation of user preferences.
- **Document Analysis / Study of Existing Systems:**
  - Reviewing legacy code, business policies, forms, user manuals, and competitor systems.

Modern & Collaborative Techniques

Interactive techniques designed to align diverse stakeholders and uncover hidden system needs.

- **Joint Application Development (JAD) Workshops:**
  - Intensive structured sessions bringing developers, users, and management together to reach consensus rapidly.
- **Brainstorming Sessions:**
  - Open creative sessions to generate novel features without immediate critique.
- **Observation / Ethnography:**
  - Analysts observe users in their actual work environment to understand real operational workflows.
- **Prototyping:**
  - Building low-fidelity mockups or wireframes to get immediate feedback.

Workflow of Requirement Elicitation & Analysis



Best Practices for Effective Requirement Gathering

- Apply the **SMART** criteria: Requirements must be **S**pecific, **M**easurable, **A**ttainable, **R**ealistic, and **T**raceable.
- Maintain a **Requirements Traceability Matrix (RTM)** to map requirements to design, code, and test cases.
- Resolve conflicting requirements early through stakeholder prioritization techniques (e.g., MoSCoW method).

Lecture 13 Summary:

- Requirements Engineering forms the foundation of system quality.
- FRs define system functions; NFRs enforce system operational qualities.
- Combining traditional (interviews) and collaborative (JAD/Prototyping) methods yields complete and accurate SRS specifications.

What is a Requirement?

A **software requirement** is a condition or capability to which a software system must conform. It represents a documented representation of a condition or capability needed by a user to solve a problem or achieve a business objective.

- **Requirement Engineering (RE):** The systematic process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- **Primary Goal:** Translate ambiguous, incomplete stakeholder needs into precise, unambiguous, and verifiable system specifications.
- **Core Phases:** Elicitation/Gathering, Analysis & Negotiation, Specification (SRS creation), and Validation & Management.

Key Objectives of Requirement Analysis

- **Conflict Resolution:** Detect and resolve conflicting requirements among diverse stakeholders.
- **Boundary Definition:** Elucidate software boundaries and operational environment constraints.
- **Feasibility Assessment:** Analyze technical, operational, and economical feasibility early in the SDLC.

Cost of Defect Correction

Requirement errors discovered late in the SDLC (e.g., during integration testing or maintenance) can cost up to **100 times more** to fix compared to errors identified during the requirement gathering phase.

Functional Requirements (FR)

- Specify inputs, outputs, and behaviors.
- Express specific functions/services system must perform.
- Directly tied to user use cases and software features.

Non-Functional Requirements (NFR)

- Specify quality attributes and system constraints.
- Define performance, reliability, security, and scalability.
- Apply to the system as a whole rather than a single feature.

Domain Requirements

Requirements derived from the application domain (e.g., medical devices, banking algorithms) that reflect fundamental domain rules and regulatory compliance standards.

Characteristics of Functional Requirements

- **High Level of Specificity:** Explicitly states system response to specific inputs under defined operational conditions.
- **Directly Testable:** Easily translated into concrete test cases with deterministic Pass/Fail criteria.
- **User-Centric:** Captured via User Stories, Use Case Diagrams, and Feature Trees.

Examples of Functional Requirements in E-Commerce

1. The system shall allow users to search products by category, price range, and customer rating.
2. The system shall generate an automated email receipt within 30 seconds of transaction completion.
3. The payment gateway shall process credit card payments via SSL encrypted channels.

FURPS+ Quality Model for NFRs

- **Usability:** User interface aesthetics, accessibility, consistency, human factors.
- **Reliability:** Mean Time Between Failures (MTBF), fault tolerance, recoverability.
- **Performance:** Response time, throughput, memory footprint, resource utilization.
- **Supportability:** Maintainability, testability, extensibility, serviceability.
- **Plus (+):** Design, implementation, interface, and physical constraints.

Sample NFR Statement

"The system database shall maintain 99.99% uptime availability and handle up to $10,000$ concurrent user connections with response latency < 200 ms."

Functional vs. Non-Functional Requirements

Comparative Analysis Table

Parameter	Functional Requirements (FR)	Non-Functional Requirements (NFR)
Focus	Specifies *what* the system should do.	Specifies *how well* the system performs.
Scope	Localized to individual features or modules.	Global attribute affecting the entire system.
Elicitation	Derived from user goals and workflow analysis.	Derived from technical rules and quality goals.
Testing	Verified via Black-box / Functional testing.	Evaluated via Performance, Load, & Security testing.
Capture Format	Use cases, user stories, functional specs.	Quality matrices, SLA agreements, benchmarks.
Criticality	Essential for software completeness.	Essential for software usability and commercial success.

Table: Detailed Comparison Matrix of FRs and NFRs

Understanding Requirement Elicitation

Elicitation is the practice of discovering requirements by communicating with stakeholders, users, and domain experts.

- **Interactive Techniques:** Direct engagement with stakeholders (Interviews, Workshops, Surveys).
- **Observational Techniques:** Passive or active monitoring of existing workflows (Ethnomethodology, Job Shadowing).
- **Analytical Techniques:** Document analysis, reverse engineering, competitive market research.
- **Prototyping Techniques:** Rapid creation of UI mockups to discover latent user requirements.

Interviews & Questionnaires

- **Structured/Unstructured:** Deep qualitative insights from key individual stakeholders.
- **Surveys:** Quantitative gathering from a broadly distributed user base.

Joint Application Development (JAD)

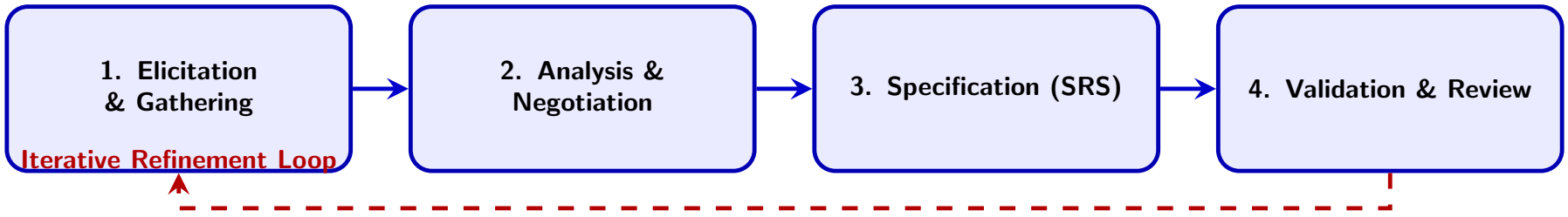
- Intensive structured workshops.
- Brings developers, business analysts, and end users together for rapid consensus building.

Observation & Document Analysis

- **Ethnography / Observation:** Uncovers implicit knowledge that users forget to articulate.
- **Document Analysis:** Mining existing business process manuals, legacy system specs, and forms.

Requirement Engineering Process Workflow

Iterative Requirement Elicitation & Analysis Lifecycle



Phase Description

- **Elicitation:** Collect raw user needs and business requirements.
- **Analysis:** Disambiguate, categorize, prioritize, and resolve conflicts.
- **Specification:** Formalize into Software Requirement Specification (SRS).
- **Validation:** Review SRS with stakeholders for completeness, consistency, and correctness.

Requirement Validation Checks

- **Validity:** Does the software satisfy real stakeholder business needs?
- **Consistency:** Are there any contradictory requirements within the specification?
- **Completeness:** Are all real-world operational scenarios and constraints included?
- **Realism:** Can the requirement be implemented within budget and technical constraints?
- **Verifiability:** Can test cases be written to demonstrate compliance?

Requirements Traceability Matrix (RTM)

A grid document linking individual requirements to design components, code modules, and test cases to ensure all requirements are fulfilled and scope creep is prevented.

Lecture 15: Overview & Learning Objectives

Course: GTU DI04000011 – Software Engineering
Unit: Software Requirement Analysis and Design
Topic: Software Requirement Specification (SRS), Customer Requirements, and Functional Requirements

Learning Objectives

- Understand the definition, role, and structure of a Software Requirement Specification (SRS).
- Distinguish between Customer (User) Requirements and Technical Functional Requirements.
- Analyze the core characteristics of a high-quality SRS as per IEEE 830 standards.
- Learn how functional requirements form the foundation of software architecture and testing.

Understanding Software Requirement Specification (SRS)

Definition

A **Software Requirement Specification (SRS)** is a comprehensive document that describes what a software system should do and how it is expected to perform. It serves as a formal contract between the customer and the software development team.

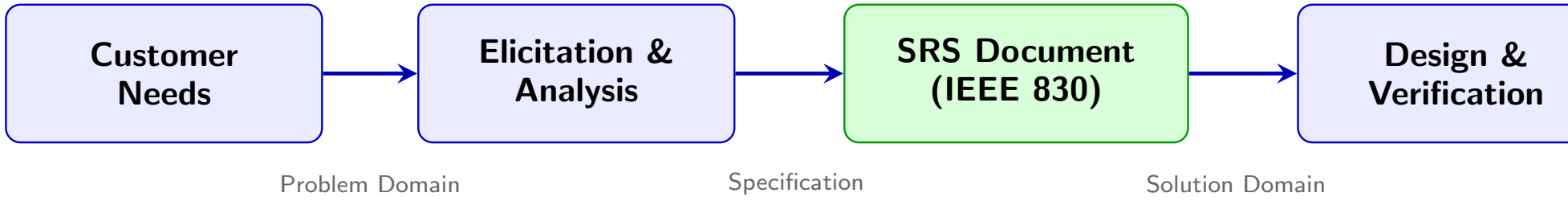
- **Primary Purpose:** Captures complete external behavior of the system without defining internal implementation details.
- **Standardization:** Governed by guidelines such as IEEE 830 / ISO/IEC/IEEE 29148.
- **Dual Audience:** Written to be understandable by non-technical stakeholders (customers) while providing exact technical directives for system architects and developers.

Role and Purpose of SRS in SDLC

Key Functions of an SRS Document

- **Agreement Baseline:** Establishes a clear agreement between the customer and the vendor on system capabilities.
- **Design Input:** Acts as the primary input document for architectural design and software modeling.
- **Cost & Schedule Estimation:** Provides precise scope boundaries for calculating effort, budget, and timelines.
- **Verification & Validation:** Serves as the reference benchmark for System Testing and Acceptance Testing.
- **Maintenance Reference:** Helps future developers understand original design intent and functional boundaries.

Requirements Engineering Process Flow



Core Process Transition

Requirements Engineering transforms unstructured **Customer Requirements** into structured, precise **Functional Requirements** formalized within the SRS.

Customer Requirements (User Requirements)

Concept

Customer Requirements express the goals, business needs, and operational constraints from the perspective of system users and business owners.

- **Format:** Written in natural language, often using high-level use-case descriptions or user stories.
- **Focus:** Focuses on *"WHAT the user needs to accomplish"* rather than technical operations.
- **Characteristics:**
  - May contain ambiguities or domain-specific jargon.
  - Focuses on user goals, workflow outcomes, and business value.
  - Formulated during initial feasibility and elicitation phases.

Functional Requirements

Concept

Functional Requirements specify the explicit behavioral statements detailing how the system must transform inputs into outputs, execute workflows, and handle exceptional conditions.

- **Technical Precision:** Must be actionable, testable, and detailed for software developers.
- **Core Components:**
  - **Input Processing:** Data entry formats, validations, and state updates.
  - **System Operations:** Business logic, calculations, and automated workflows.
  - **Output Generation:** Screen displays, reports, API responses, and database writes.
  - **Exception Handling:** Error detection, messaging, and recovery actions.

Characteristics of a Good SRS (Part 1)

According to the **IEEE 830 Standard**, a high-quality SRS must possess specific qualities:

Correct: Every stated requirement must accurately reflect a capability that the software is required to meet.

Unambiguous: Every requirement has exactly one interpretation. Language must be precise, avoiding terms like *"user-friendly"* or *"efficient"*.

Complete: Contains all essential requirements (functional, non-functional, interface), responses to all inputs, and defined responses to invalid data.

Consistent: No requirement conflicts with another (e.g., conflicting delivery dates, mutually exclusive logical conditions).

Characteristics of a Good SRS (Part 2)

Verifiable: A requirement is verifiable if there exists a finite, cost-effective process by which a machine or human can prove that the software meets the requirement.

Modifiable: The structure and style of the document allow changes to be made easily, completely, and consistently without ruining redundant sections.

Traceable: The origin of each requirement is clear (Backward Traceability), and each requirement can be referenced in design and testing artifacts (Forward Traceability).

Ranked for Importance/Stability: Requirements are prioritized (e.g., Essential, Conditional, Optional) to aid decision-making during resource constraints.

Comparison of Requirement Types

Dimension	Customer Requirements	Functional Requirements
Target Audience	Business Clients, End-Users	System Architects, Developers, Testers
Language	Natural language, business terms	Structured, precise technical statements
Primary Focus	User goals & business outcomes	System behavior, logic, and data flow
Example	"The user can pay for orders online."	"System shall process credit card payments via Payment API and return a 16-digit transaction ID."
Testing Baseline	User Acceptance Testing (UAT)	System & Integration Testing

Table: Comparison between Customer and Functional Requirements

Summary & Review Questions

Lecture Summary

- SRS acts as the bridge between problem domain and solution domain.
- Customer requirements articulate business needs, whereas functional requirements detail precise system behaviors.
- Adhering to IEEE 830 characteristics ensures a robust, testable, and maintainable software product.

GTU Review Questions

1. Define SRS. List and explain any four characteristics of a good SRS document as per IEEE 830.
2. Differentiate between Customer Requirements and Functional Requirements with suitable software engineering examples.

Lecture 16: Overview & Learning Objectives

Course: Software Engineering (DI04000011)
Unit: Software Requirement Analysis and Design
Topic: Software Requirement Specification (SRS), Customer Requirements, and Functional Requirements

Learning Objectives

- Define the role and importance of a Software Requirement Specification (SRS) document.
- Distinguish between Customer Requirements and System/Functional Requirements.
- Analyze the characteristics of a high-quality, professional SRS document.
- Understand the requirement engineering workflow converting stakeholder needs into formal engineering artifacts.

What is an SRS?

An **SRS** is a formal document that describes what a proposed software system must do, its constraints, and its interaction with external environments. It serves as a contract between customers/users and developers.

Primary Functions of SRS:

- Establishes complete agreement on project scope.
- Provides a baseline for software validation and testing.
- Serves as the foundation for software design and estimation.

Target Audience:

- *Customers/Users:* Validate business requirements.
- *Architects/Developers:* Guide system design & code.
- *QA/Testers:* Generate test plans & test cases.
- *Project Managers:* Plan schedule & budget.

Customer Requirements (User Requirements)

Definition

Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written primarily for domain experts, business owners, and end-users.

- **Origin:** Elicited from interviews, domain research, workshops, and user stories.
- **Expression Level:** High-level abstract concepts avoiding technical jargon (e.g., "The system must process online payments securely").
- **Challenges in Elicitation:**
  - Ambiguity in domain language.
  - Conflicting requirements across different stakeholder groups.
  - Implicit/Unstated assumptions by subject matter experts.

Functional Requirements

Definition

Detailed descriptions of the system's functions, inputs, outputs, processes, and behavior under specific conditions. They define *what* the software must explicitly execute.

Key Elements of Functional Requirements:

1. **System Services:** Specific calculations, data manipulation, business logic (e.g., "The system shall calculate sales tax based on zip code").
2. **User Inputs & Responses:** Behavior on valid and invalid input data.
3. **Exception Handling:** Defined actions during system errors or network failures.
4. **State Transitions:** System response to operational mode shifts.

Comparison: Customer vs. Functional Requirements

Table: Detailed Comparison of Requirement Types

Attribute	**Customer Requirement**	**Functional Requirement**
Focus	Business problem & end goal	Technical mechanism & execution
Target Audience	Clients, Users, Executives	Engineers, Architects, Testers
Language	Natural, domain-specific language	Precise technical specifications
Detail Level	Abstract / High-level	Rigorous / Explicit / Granular
Example	"Allow patients to view medical records online."	"System shall query DB for PatientID and return JSON payload within 500ms."

Characteristics of a Good SRS (Part 1)

To minimize project risks and costly rework, IEEE 830 / ISO 29148 standards dictate that a quality SRS must possess key attributes:

1. **Correctness:** Every requirement stated in the SRS represents a true requirement of the system to be built.
2. **Unambiguous:** Every statement has exactly one interpretation. Language must be precise and free of vague terms (e.g., avoid "fast", "user-friendly", "robust").
3. **Completeness:** Contains all significant requirements (functional, performance, design constraints, and external interfaces) and defined responses to all realization inputs.
4. **Consistency:** No two requirements conflict with each other (e.g., conflicting data types, operational rules, or terminology).

Characteristics of a Good SRS (Part 2)

5. **Verifiable (Testable):** There exists a cost-effective, finite process by which a machine or person can prove the final software meets the requirement.
6. **Modifiable:** Structure and style allow changes to be made easily, completely, and consistently without breaking overall cohesion.
7. **Traceable:** Origin of each requirement is clear (backward traceability), and each requirement facilitates reference in future development artifacts (forward traceability).
8. **Ranked for Importance/Stability:** Requirements are prioritized (e.g., High/Medium/Low or MoSCoW) to guide release planning and resource allocation.

Requirements Flow in Software Engineering

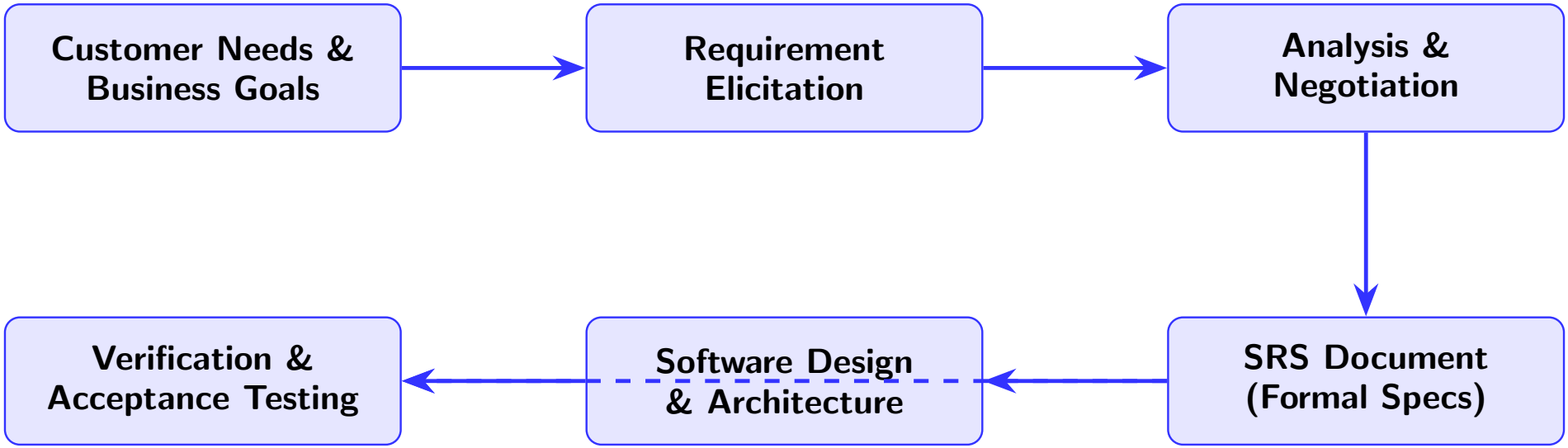


Figure: Workflow: Eliciting Requirements to Verifying SRS

Structure of a Standard SRS (IEEE 830 Standard)

A standardized template ensures readability, maintainability, and thoroughness.

Standard SRS Outline

1. **Introduction**
   - Purpose, Scope, Definitions, Acronyms, and References
2. **Overall Description**
   - Product Perspective, User Classes, Operating Environment, Design Constraints
3. **Specific Requirements**
   - External Interface Requirements (User, Hardware, Software, Communication)
   - Functional Requirements (Detailed behavioral specs)
   - Performance, Security, Reliability, and Maintainability Requirements

Summary & Review Checklist

Key Takeaways

- The SRS is the foundational specification bridge connecting business requirements with engineering implementation.
- Customer requirements reflect business goals; Functional requirements detail exact software behaviors.
- Adhering to good SRS characteristics (unambiguous, verifiable, complete, traceable) prevents requirement leakage and scope creep.

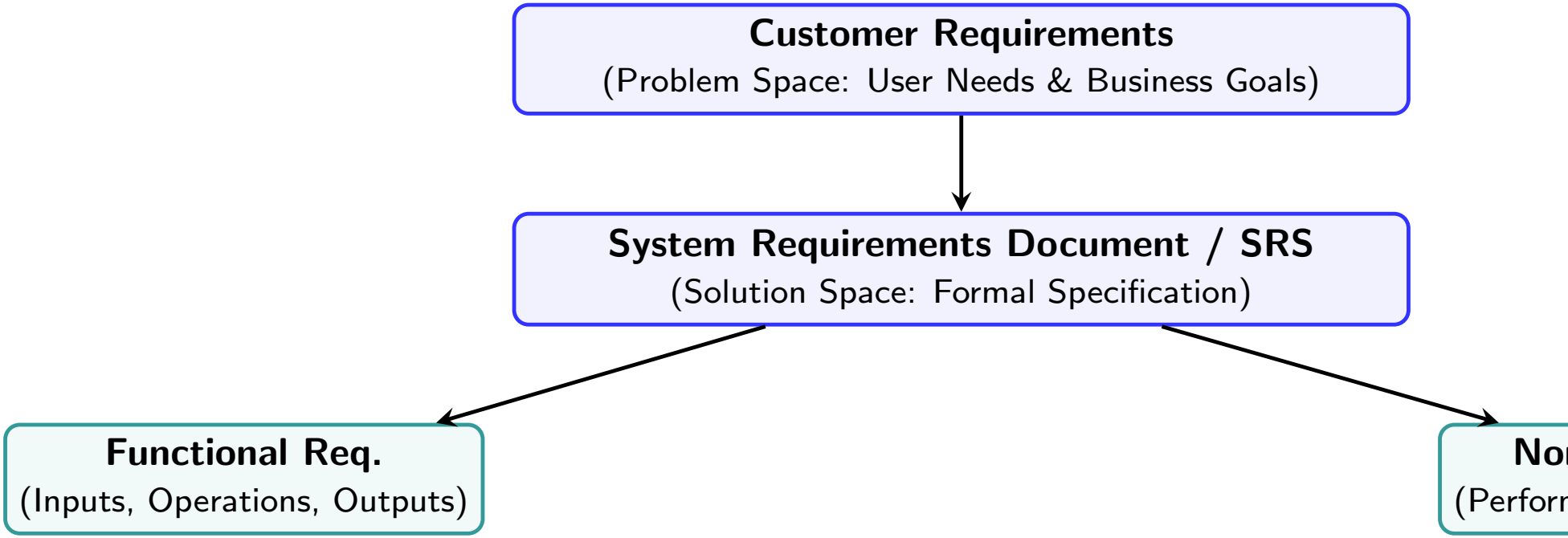
SRS Quality Checklist Question

Can a QA engineer design an automated test case directly from your requirement statement without asking for clarification? If no, the requirement lacks clarity or verifiability!

Introduction to Software Requirement Specification (SRS)

- **Definition of SRS**: A Software Requirement Specification (SRS) is a formal, comprehensive document that defines the functional, non-functional, and operational requirements of a software system before development begins.
- **Role in Software Engineering Lifecycle**:
  - Serves as a contract/agreement between the customer (client) and the development team.
  - Acts as the authoritative baseline for project scope, software design, testing, and validation.
  - Minimizes costly rework by disambiguating expectations early in the Software Development Life Cycle (SDLC).
- **Key Stakeholders**:
  - **Customers/Users**: Validate business rules and usability expectations.
  - **System Architects & Developers**: Use SRS to design system architecture and implement code.
  - **QA/Test Engineers**: Build test suites and acceptance criteria directly from SRS assertions.

Requirement Hierarchy in Software Engineering



Key Distinction

Customer requirements capture *what the user needs to accomplish*, whereas System Requirements (SRS) specify *how the system must behave* to satisfy those needs.

Customer Requirements (User Requirements)

- **Definition**: High-level statements written in natural language (and supplemented with domain diagrams) that describe the services the system is expected to provide to end-users and the operational constraints under which it must operate.
- **Primary Characteristics**:
  - Expressed from the perspective of non-technical stakeholders and domain experts.
  - Free of implementation details, technical jargon, architectural choices, and programming frameworks.
  - Focused on business value, user goals, workflows, and environmental constraints.
- **Elicitation Techniques**:
  - Stakeholder Interviews & Surveys.
  - User Stories & Use Case scenarios (e.g., "As a GTU Student, I want to view my semester CPI...").
  - Brainstorming workshops and rapid paper prototyping.

Functional Requirements (FR)

- **Definition**: Functional Requirements specify the explicit services, operations, behaviors, state transformations, and data processing functions that the software system must perform in response to specific inputs.
- **Core Components of a Functional Requirement**:
  - **Input Data**: Data items provided by users, external hardware sensors, or third-party APIs.
  - **System Processing**: Logic, algorithms, business rules, and state changes executed by software.
  - **Output Response**: Display output, stored data records, generated files, or outbound system triggers.
- **Examples in GTU Academic Management System**:
  - *FR-01*: "The system shall calculate the SPI of a student using GTU grading formula upon marks entry."
  - *FR-02*: "The system shall automatically lock exam form submission after midnight of the published deadline."

Standard Format for Functional Requirement Specification

Each functional requirement should be uniquely identifiable and structured rigorously:

- **Requirement ID**: Unique alphanumeric key (e.g., `REQ-FR-102`).
- **Title**: Concise name describing the function.
- **Trigger**: Event initiating the system action.
- **Pre-conditions**: System states required before execution.
- **Functional Logic**: Step-by-step description of system processing.
- **Post-conditions**: Guaranteed system state after completion.

Avoid Ambiguity in Writing Functional Requirements

Poor: "The system should quickly process student marks."
Good: "Upon receiving student marks CSV, the system shall compute grade points within 2.0 seconds and update the database record."

Characteristics of a Good SRS - Part 1

- **1. Correctness**:
  - Every requirement stated in the SRS accurately represents a real capability required by the software to meet user goals.
- **2. Unambiguity**:
  - Every requirement has *exactly one interpretation*. Terms with multiple meanings must be defined in a formal glossary.
- **3. Completeness**:
  - The SRS must include all significant requirements (functional, quality, performance, constraints), define system response to invalid inputs, and include full label/reference definitions.
- **4. Modifiability**:
  - The document structure and style must allow changes to be made easily, completely, and consistently without breaking traceability.

Characteristics of a Good SRS - Part 2 (IEEE 830 Standards)

Table: Key SRS Quality Characteristics & Verification Methods

Characteristic	**IEEE 830 Standard Definition**	**Verification / Audit Method**
Consistency	Requirements do not conflict with each other or external standards.	Cross-requirement verification matrix.
Verifiability	Quantifiable criteria exist such that a cost-effective test can check compliance.	Automated test case generation & acceptance testing.
Traceability	Origin of each requirement is clear, and forward/backward tracing is possible.	Requirement Traceability Matrix (RTM).
Feasibility	Requirements can be implemented within technology, budget, and time limits.	Proof of concept / Technical risk analysis.

Standard IEEE 830 Structure of an SRS Document

1. **1. Introduction**
   - 1.1 Purpose & 1.2 Scope
   - 1.3 Definitions, Acronyms, and Abbreviations
   - 1.4 References & 1.5 Overview
2. **2. Overall Description**
   - 2.1 Product Perspective & Product Functions
   - 2.2 User Characteristics & Constraints
   - 2.3 Assumptions and Dependencies
3. **3. Specific Requirements**
   - 3.1 External Interface Requirements (User, Hardware, Software, Communication)
   - 3.2 Functional Requirements (Detailed Input/Output/Processing)
   - 3.3 Performance & Non-functional Attributes (Security, Maintainability)

Common Pitfalls & Risk Mitigation in SRS Drafting

Common Pitfalls

- **Design Contamination**: Specifying design solutions instead of requirement needs.
- **Over-generalization**: Using vague terms like "etc.", "user-friendly", "fast".
- **Silence on Failures**: Omitting error handling scenarios.
- **Dangling Requirements**: Requirements without customer origins.

Mitigation Strategies

- Formal Requirement Reviews with Domain Experts.
- Requirement Verification against standard checklists.
- Utilizing Requirement Traceability Matrix (RTM).
- Prototyping risky or ambiguous requirements early.

Summary & GTU Review Questions

Lecture Summary

- SRS is the foundational document bridging user expectations to actual system design.
- Customer requirements reflect user goals; Functional requirements define technical operations.
- A good SRS is Correct, Unambiguous, Complete, Consistent, Verifiable, Modifiable, and Traceable.

GTU Exam Review Questions

1. Differentiate between Customer Requirements and Functional Requirements with suitable software examples. [4 Marks]
2. Explain the characteristics of a good SRS document as per IEEE standard. [7 Marks]
3. Why is ambiguity in SRS dangerous for software development? Explain with an example. [3 Marks]

Lecture 18: Overview & Learning Objectives

- **Course**: GTU Course DI04000011 – Software Engineering
- **Unit**: Software Requirement Analysis and Design
- **Primary Topics Covered**:
  - Fundamentals of Software Design and Key Characteristics
  - Systematic Comparison: Requirement Analysis vs. Software Design
  - Principle of Modularization: Cohesion and Coupling
  - Classification of Cohesion (Coincidental to Functional)
  - Classification of Coupling (Content to Data/Uncoupled)
- **GTU Exam Relevance**: High frequency of 7-mark questions on design characteristics, Analysis vs. Design distinction, and detailed classification of Cohesion and Coupling with diagrams.

Characteristics of a Good Software Design

- **Correctness & Completeness**: Faithfully implements all functional and non-functional requirements specified in the Software Requirement Specification (SRS).
- **Understandability & Simplicity**: Clear structure, consistent naming conventions, and manageable complexity enable developers to comprehend module roles effortlessly.
- **High Modularity**: Well-defined, independent components with clear interfaces, promoting separation of concerns.
- **Maintainability & Extensibility**: Minimal effort required to fix bugs, adapt to environment changes, or incorporate new functionality without side-effects.
- **Reusability**: Modules are decoupled and general enough to be repurposed across different sub-systems or projects.
- **Efficiency & Performance**: Optimal allocation and consumption of system resources (CPU, Memory, I/O, Network).
- **Traceability**: Direct mapping between SRS requirements, design artifacts, and target source code components.

Requirement Analysis vs. Software Design

Dimension	Requirement Analysis	Software Design
Primary Goal	Understand and capture user problem domain	Conceptualize technical solution domain
Focus	**WHAT** the system must do	**HOW** the system will satisfy requirements
Input	Customer statements, Domain knowledge	SRS Document, Use Cases, Domain Models
Output	SRS Document, Use Case Models, DFDs	Design Doc (SDD), Architecture, Class Diagrams
Abstraction	High (Problem/Logical Level)	Low to Medium (System/Structural Level)
Key Stakeholder	End-users, Business Analysts, Clients	Software Architects, Lead Developers, Testers
Validation	Validated against client expectation	Validated against SRS requirements

Fundamentals of Modular Design: Cohesion & Coupling

- **Modularization**: Breaking a software system into smaller, manageable, self-contained units (modules, packages, classes).
- **Cohesion**:
  - Measures the **intra-module strength** (how tightly related elements within a single module are).
  - **Golden Rule**: Aim for **High Cohesion** (elements work together toward a single, well-defined goal).
- **Coupling**:
  - Measures the **inter-module interdependence** (how strongly one module relies on another).
  - **Golden Rule**: Aim for **Low (Loose) Coupling** (modules communicate via minimal, well-defined interfaces).
- **Fundamental Design Axiom**: *"High Cohesion and Low Coupling yield robust, maintainable, and extensible software architectures."*

Classification of Cohesion: Low to Medium Cohesion

1. **Coincidental Cohesion (Worst / Low)**:
   - Tasks are grouped arbitrarily without any meaningful relationship (e.g., a utility function containing ‘printReport()‘, ‘calculateTax()‘, and ‘parseXML()‘).
2. **Logical Cohesion**:
   - Tasks are logically related (e.g., all input processing routines grouped together), but selected conditionally via a flag parameter during runtime.
3. **Temporal Cohesion**:
   - Tasks are combined because they execute at the same time phase during program life cycle (e.g., ‘systemInitialization()‘ opening files, clearing buffers, initializing variables).
4. **Procedural Cohesion**:
   - Tasks execute in a specific sequential order to accomplish a process, though they operate on different data entities.

Classification of Cohesion: Medium to High Cohesion

5. **Communicational Cohesion**:
   - Tasks perform different operations, but operate on the same shared input data or produce the same output structure (e.g., ‘updateAndPrintStudentRecord()‘).
6. **Sequential Cohesion**:
   - Output of one internal task serves as the direct input to the next task within the same module (e.g., ‘parseJSON()‘ $\rightarrow$ ‘validateData()‘ $\rightarrow$ ‘saveToDatabase()‘).
7. **Functional Cohesion (Best / High)**:
   - All elements in the module execute towards achieving a **single, well-defined task** (e.g., ‘computeSquareRoot()‘, ‘encryptData()‘). Highly desirable for maintainability.

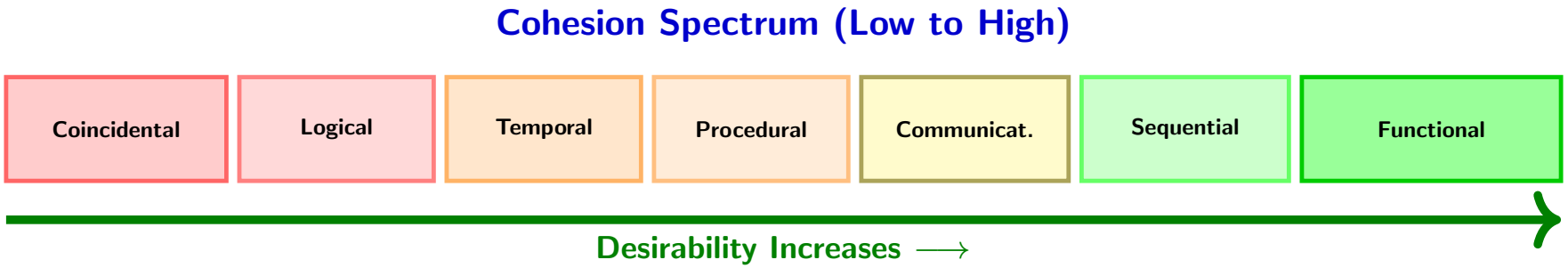
Classification of Coupling: High to Medium Coupling

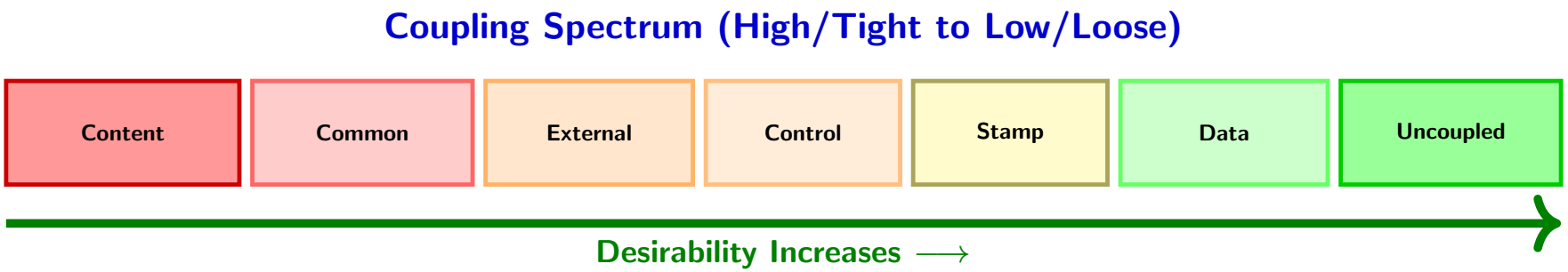
1. **Content Coupling (Worst / High / Tight)**:
   - One module directly modifies or accesses internal data/code of another module (violates encapsulation).
2. **Common (Global) Coupling**:
   - Multiple modules share global data stores or global variables. Modifying shared data impacts all dependent modules.
3. **External Coupling**:
   - Modules share an externally imposed format, communication protocol, or hardware interface (e.g., OS interface, I/O device structure).
4. **Control Coupling**:
   - One module controls the execution flow of another by passing control information (flags, switches, option codes).

Classification of Coupling: Low (Loose) Coupling

5. **Stamp (Data-Structure) Coupling**:
   - Modules communicate by passing composite data structures (e.g., ‘Student‘ object), but the receiving module only uses a subset of fields (e.g., ‘studentId‘).
6. **Data Coupling (Best / Low / Loose)**:
   - Modules communicate strictly by passing elementary data items as individual arguments (e.g., ‘calculateSimpleInterest(principal, rate, time)‘).
7. **Uncoupled (Ideal Zero-Coupling)**:
   - Modules operate completely independently without sharing any data, control, or state information.

Visualizing Cohesion & Coupling Spectra





Lecture Summary & GTU Review Questions

- **Summary**: Good design transitions "WHAT" (Analysis) to "HOW" (Design). Maximizing Functional Cohesion and achieving Data Coupling ensures maintainable and modular systems.
- **GTU Exam Review Questions**:
  1. State and explain the characteristics of a good software design. [7 Marks]
  2. Differentiate between Requirement Analysis and Software Design. [4 Marks]
  3. Explain different types of Cohesion with suitable real-world examples. [7 Marks]
  4. Define Coupling. Differentiate between Content, Common, Control, Stamp, and Data Coupling. [7 Marks]
  5. Why is "High Cohesion and Low Coupling" desirable in Software Engineering? [3 Marks]

Software Design Overview & Characteristics

What is Software Design?

Software design is the process of defining software architecture, components, modules, interfaces, and characteristics to satisfy specified requirements. It transforms the **"WHAT"** (requirements) into the **"HOW"** (technical architecture).

Characteristics of a Good Software Design

- **Correctness & Completeness:** Accurately implements all functional and non-functional requirements specified in the SRS.
- **Understandability & Clarity:** Logical organization making code easy to navigate and comprehend for maintenance teams.
- **Maintainability & Modularity:** Highly modular structure allowing modifications without ripple effects across the system.
- **Reusability & Extensibility:** Components designed to be reusable in other modules or future system extensions.
- **Efficiency & Reliability:** Optimal utilization of memory, CPU, and storage while gracefully handling errors.

Software Requirement Analysis v/s Software Design

Table: Comparative Analysis: Software Analysis vs. Software Design

Parameter	**Requirement Analysis Phase**	**Software Design Phase**
Primary Focus	Understanding the problem domain (**WHAT** the system must do).	Engineering the solution domain (**HOW** the system will execute).
Input Artifacts	Customer requirements, feasibility study, business objectives.	Software Requirements Specification (SRS) document.
Output Artifacts	SRS Document, Use Cases, Data Flow Diagrams (DFDs), ER Diagrams.	Design Document (SDD), Architecture Diagrams, Module Schemas.
Perspective	User and domain expert centric.	Developer and software architecture centric.
Key Activities	Elicitation, modeling, validation, conflict resolution.	Architectural partitioning, interface design, data structure selection.
Evaluation Metric	Requirement coverage and customer satisfaction.	Coupling, cohesion, maintainability, and execution efficiency.

Foundations of Modularity: Cohesion and Coupling

Modularity Concept

Modularity divides a system into independent, smaller units (modules) to manage software complexity via decomposition and information hiding.

Cohesion (Intra-Module)

- Measures internal functional strength of a module.
- Degree to which elements inside a module belong together.
- **Goal: HIGH Cohesion**

Coupling (Inter-Module)

- Measures degree of interdependence between different modules.
- Extent of connections and shared data between modules.
- **Goal: LOW Coupling**

Golden Rule of Software Engineering: *"High Cohesion and Low (Loose) Coupling lead to robust, maintainable, and reusable software systems."*

Classification of Cohesion (Low to Medium Levels)

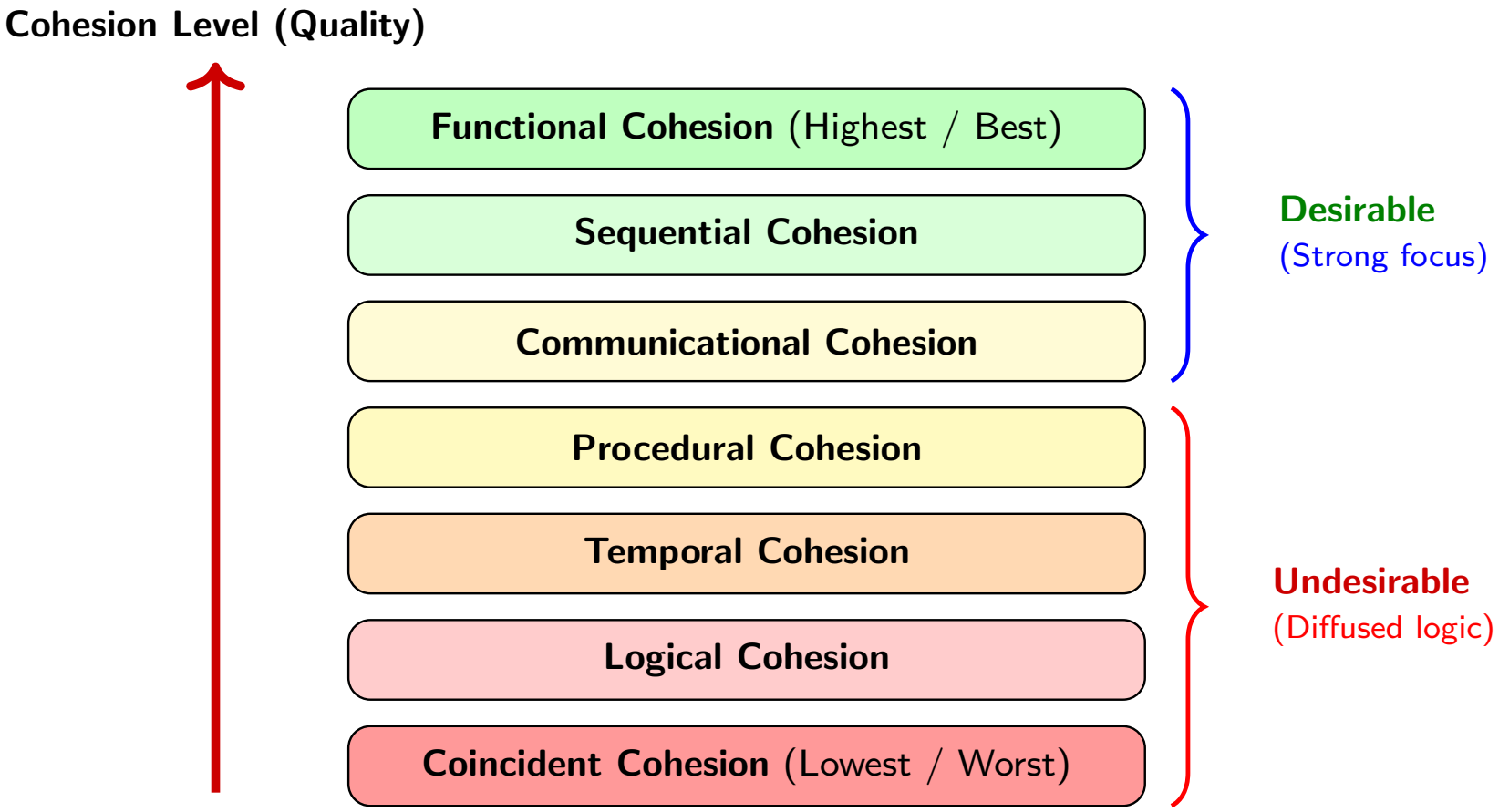
Cohesion is ranked from lowest (least desirable) to highest (most desirable):

1. **Coincident Cohesion (Worst / Lowest):**
   - Operations are grouped completely randomly without any logical relationship.
   - Example: A utility module containing `printReport()`, `calculateTax()`, and `sortArray()`.
2. **Logical Cohesion:**
   - Functions perform logically similar activities but operate on different data based on a control flag.
   - Example: A single input routine handling keyboard, mouse, and file input streams.
3. **Temporal Cohesion:**
   - Functions are grouped because they must execute within the same time window.
   - Example: System startup initialization routine (`initDB()`, `loadConfig()`, `openLogs()`).
4. **Procedural Cohesion:**
   - Elements execute in a specific sequence or workflow, but do not share data.
   - Example: Read file permissions $\rightarrow$ calculate checksum $\rightarrow$ write log record.

Classification of Cohesion (High Levels)

5. **Communicational Cohesion:**
   - Functions operate on the same input data or produce the same output data stream.
   - Example: A module that reads customer data, validates customer record, and formats customer output.
6. **Sequential Cohesion:**
   - Output of one element acts as the direct input to the next element in an assembly-line chain.
   - Example: Raw data parser $\rightarrow$ Data validator $\rightarrow$ Database formatter.
7. **Functional Cohesion (Best / Highest):**
   - All elements within the module contribute to performing a single, well-defined task.
   - Highly reusable, easy to test, and isolated from side-effects.
   - Example: `calculateMatrixDeterminant()`, `computeTaxAmount()`.

Visualizing Cohesion Spectrum



Classification of Coupling (High to Medium Coupling)

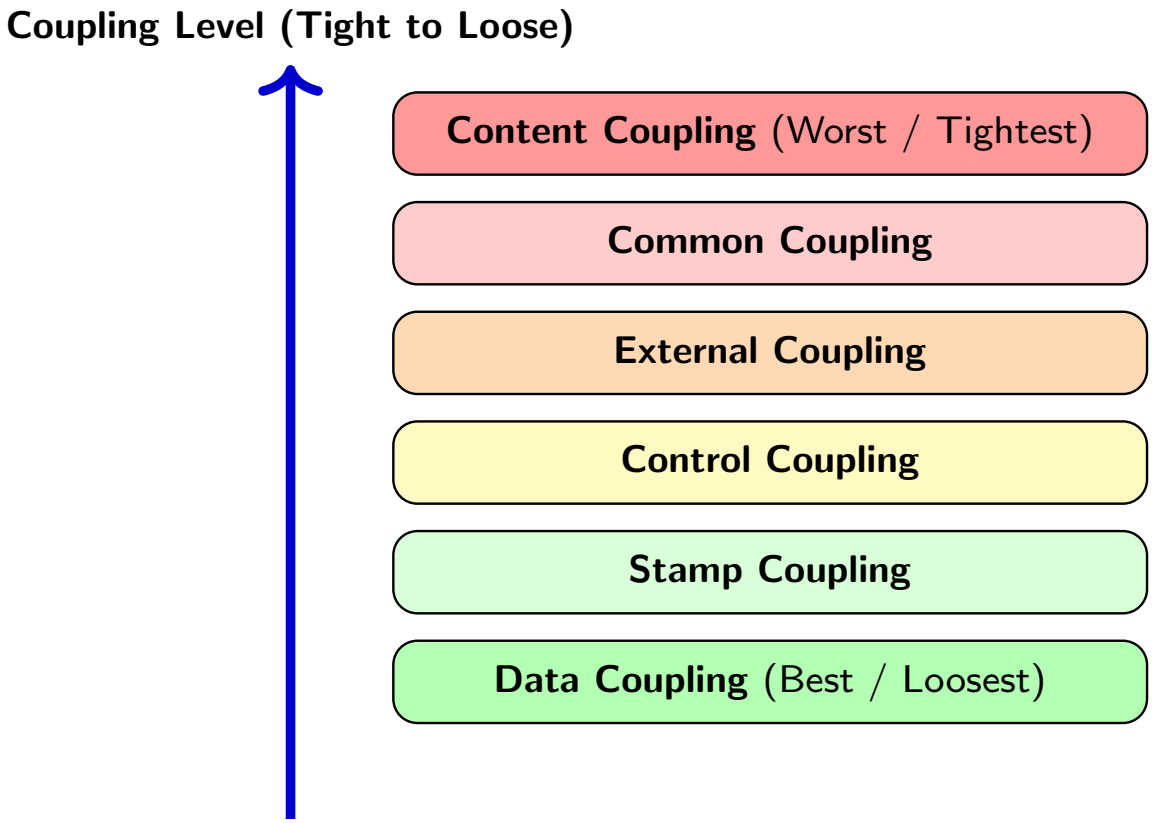
Coupling measures interdependence between modules, ranked from worst (tightest) to best (loosest):

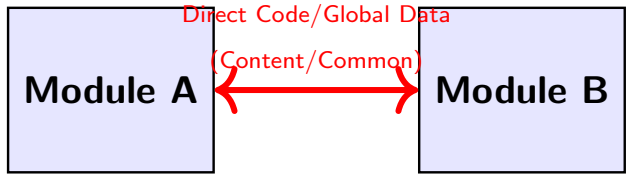
1. **Content Coupling (Worst / Tightest):**
   - One module directly accesses or modifies the internal code/data of another module.
   - Completely breaks encapsulation and information hiding.
   - Example: Branching (jumping) directly into another module's local instructions.
2. **Common Coupling:**
   - Multiple modules share access to global data structures or variables.
   - Changes to global variables propagate errors across all connected modules.
   - Example: Multiple functions modifying a global `system_config` struct.
3. **External Coupling:**
   - Modules share an externally imposed data format, communication protocol, or hardware interface.
   - Example: External sensor protocol, I/O device formatting.

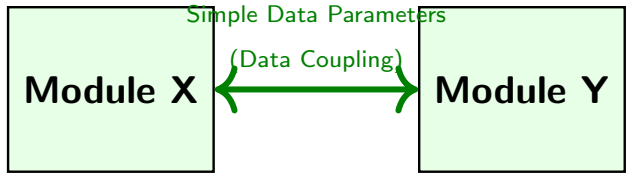
Classification of Coupling (Medium to Loose Coupling)

4. **Control Coupling:**
   - One module passes control parameters (flags/switches) to dictate the internal execution logic of another module.
   - Example: Passing a boolean `isAdmin` flag to alter function execution paths.
5. **Stamp (Data Structure) Coupling:**
   - Complete composite data structures (e.g., Objects, Structs) are passed as parameters, even if the target module needs only a few fields.
   - Example: Passing an entire `Employee` record to a function that only uses `salary`.
6. **Data Coupling (Best / Loosest):**
   - Modules communicate strictly by passing primitive parameters (data values) via function arguments.
   - Highly desirable; change in one module does not impact the internal structure of others.
   - Example: `calculateInterest(double principal, float rate)`.

Visualizing Coupling Spectrum & Inter-Module Coupling







Summary: Principles of Effective Software Design

Core Trade-off Matrix

- **High Cohesion:** Keeps related responsibilities together; maximizes single-purpose module focus.
- **Low Coupling:** Minimizes dependencies across module boundaries; isolates code changes.

GTU Exam Key Summary Points

1. Software Design bridges Analysis (Requirements) with Coding (Implementation).
2. Ideal design target: **Functional Cohesion** + **Data Coupling**.
3. Avoid **Content Coupling** and **Coincident Cohesion** completely.
4. Information hiding and abstraction directly foster low coupling and high cohesion.
5. Modular design significantly reduces software maintenance costs and bug propagation.

Software Engineering (GTU Course: DI04000011)

Unit: Software Requirement Analysis and Design

- **Software Design Definition:** The process of transforming user requirements captured in the SRS document into a structural blueprint for constructing the software system.
- **Key Learning Objectives:**
  - Understand the essential characteristics of a good software design.
  - Differentiate clearly between Requirement Analysis and Software Design.
  - Master Module Cohesion and classify its seven distinct levels.
  - Master Module Coupling and classify its five major levels.
- **Core Engineering Goal:** Achieve high intra-module cohesion and low inter-module coupling to maximize maintainability, reusability, and scalability.

Characteristics of a Good Software Design

A software design must satisfy functional, non-functional, and architectural criteria:

1. **Correctness & Completeness:** Accurately implements all requirements specified in the SRS document without omissions.
2. **Understandability & Simplicity:** Maintains a clean architectural structure allowing developers to easily trace components to requirements.
3. **Efficiency:** Optimizes system resource utilization including execution runtime, memory footprint, and I/O operation.
4. **Maintainability & Modularity:** Structured into independent modules so modifications in one section do not cause unexpected side effects.
5. **Testability & Traceability:** Facilitates straightforward unit, integration, and system level testing procedures.
6. **Reusability:** Components are decoupled enough to be repurposed across different software applications.

Software Analysis vs. Software Design

Table: Comparative Analysis: Requirement Analysis vs. Software Design

Parameter	**Requirement Analysis**	**Software Design**
Primary Goal	Understand and document user needs	Propose architectural solution
Core Question	**WHAT** system must do	**HOW** system will execute
Primary Input	Customer vision & domain rules	Software Requirement Spec (SRS)
Primary Output	SRS Document & Use Cases	Architecture, Modules, Schemas
Abstraction	Problem Domain view	Solution Domain view
Target Audience	Clients, End-users, Analysts	Developers, Testers, Architects

Fundamentals of Modularity: Cohesion & Coupling

Modularization

The process of decomposing a software system into smaller, self-contained, and manageable functional blocks called **modules**.

Cohesion (Intra-module)

- Measures the **functional strength** of elements bound within a single module.
- High cohesion implies a module performs one single, focused task.
- **Engineering Target:** Maximize Cohesion.

Coupling (Inter-module)

- Measures the **degree of interdependence** between distinct modules.
- Low coupling implies modules interact through clean, minimal interfaces.
- **Engineering Target:** Minimize Coupling.

Classification of Cohesion (Part 1: Low to Moderate)

Cohesion ranges from lowest (least desirable) to highest (most desirable):

1. Coincidental Cohesion (Worst): Elements are grouped arbitrarily with no meaningful relationship (e.g., a random utility file mixing math, file I/O, and string operations).
2. Logical Cohesion: Elements perform logically similar tasks, but the specific execution path is selected by a control parameter passed at runtime (e.g., a single function handling all input types via a switch-case).
3. Temporal Cohesion: Elements are grouped together because they execute during the same time phase of execution (e.g., an initialization module setting up database connections, loggers, and UI components).
4. Procedural Cohesion: Elements are grouped because they execute in a specific sequence to accomplish a goal (e.g., read file permissions, validate user token, parse header).

Classification of Cohesion (Part 2: High Cohesion)

Higher levels of cohesion ensure cohesive responsibility and simplified maintenance:

5. Communicational Cohesion: Elements operate on the same input data or produce the same output data, though tasks perform different functions (e.g., print report summary and write report to file using identical dataset).
6. Sequential Cohesion: Elements are arranged in a strict processing pipeline where the output of one element serves as the direct input to the next (e.g., raw data extraction $\rightarrow$ data sanitization $\rightarrow$ report generation).
7. Functional Cohesion (Best): Every element within the module contributes directly to executing a single, well-defined mathematical or logical function (e.g., `calculateSquareRoot()`, `computeTaxAmount()`).

Classification of Coupling (Part 1: Tight/High Coupling)

Coupling measures module interdependency. High coupling makes systems fragile.

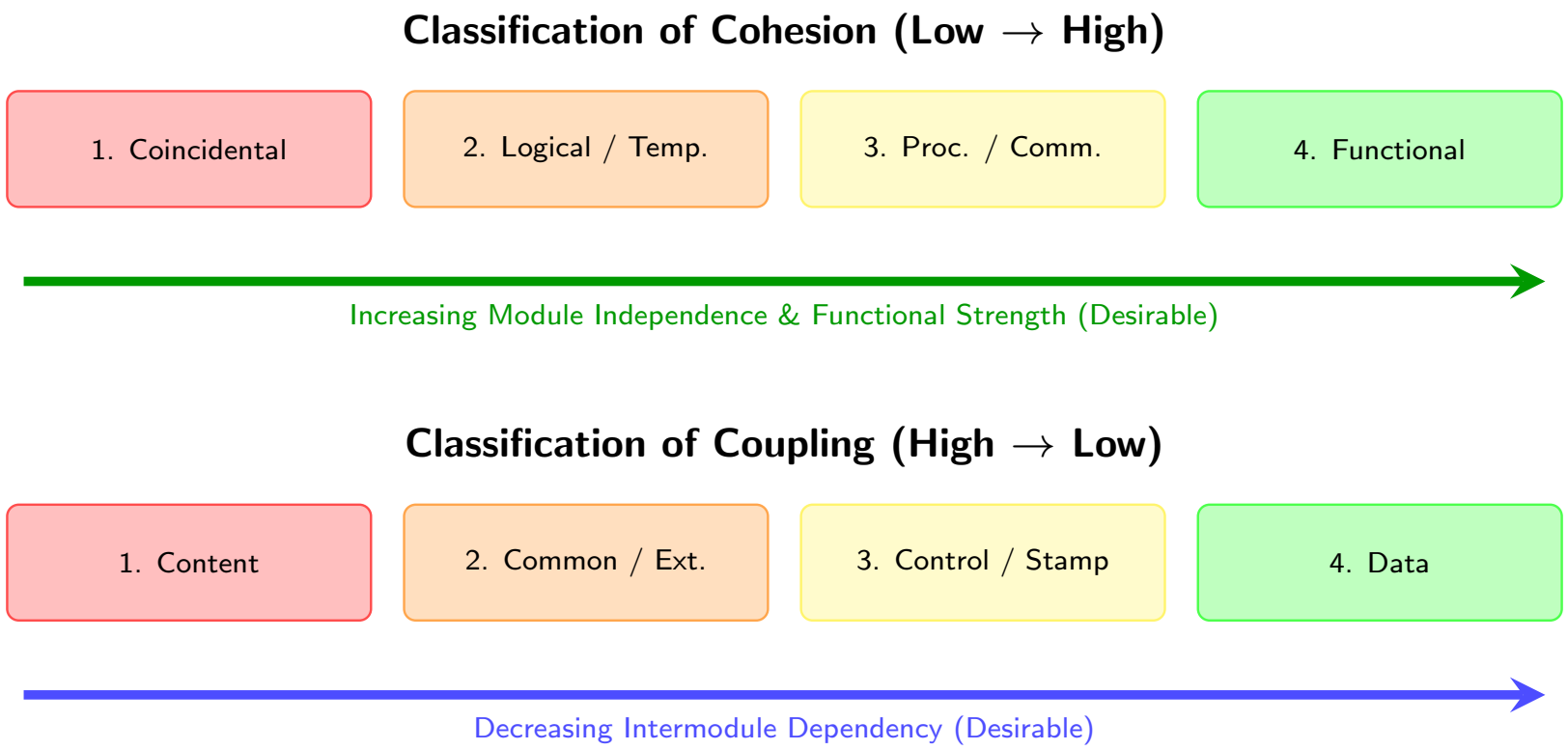
1. Content Coupling (Worst): One module directly modifies or references the internal data or non-public implementation details of another module (completely breaks encapsulation).
2. Common (Global) Coupling: Multiple modules share access to a global data structure. Modifying the global data schema forces simultaneous updates across all coupled modules.
3. External Coupling: Modules share an externally imposed format, protocol, or hardware interface specification (e.g., external device drivers or OS signals).

Classification of Coupling (Part 2: Loose/Low Coupling)

Lower coupling enhances maintainability, isolated testing, and reusability.

4. Control Coupling: One module directs the internal logic flow of another module by passing control flags or command switches (e.g., passing a boolean `isSortDescending` flag).
5. Stamp (Data Structure) Coupling: Modules communicate by passing a composite data structure (e.g., passing a complete `Student` object), but the target module uses only a small fraction of the fields.
6. Data Coupling (Best): Modules communicate strictly by passing minimal, essential primitive data values as parameters through public function calls (e.g., `computeInterest(principal, rate, time)`).

Visualizing Cohesion and Coupling Spectrum



Summary & GTU Review Questions

Summary of Software Design Best Practices

- Software Design transforms requirements into actionable developer blueprints.
- Design target: **High Functional Cohesion** and **Low Data Coupling**.
- High cohesion ensures focused responsibilities and easier testing.
- Low coupling prevents side effects and cascading failures during maintenance.

GTU Exam Frequently Asked Questions (DI04000011)

1. Differentiate between Analysis and Design phases of software development. [7 Marks]
2. List and explain the characteristics of a good software design. [4 Marks]
3. Define Cohesion. Explain different types of Cohesion with suitable examples. [7 Marks]
4. Define Coupling. Classify and describe all types of Coupling. [7 Marks]

Lecture 21: Function-Oriented Software Design

- **Course**: GTU DI04000011 – Software Engineering
- **Unit**: Software Requirement Analysis and Design
- **Lecture Focus**: Function-Oriented Design & Data Flow Diagrams (DFD)

Overview of Lecture Topics

- Fundamentals of Function-Oriented Software Design (FOSD)
- Core Concepts and Characteristics of Data Flow Diagrams (DFD)
- DFD Notations, Symbols, and Structural Rules
- Context-Level DFD (Level 0) Architecture & System Boundaries
- Level-1 DFD Functional Decomposition & Data Stores
- DFD Balancing Rules and Common Design Anomalies

Function-Oriented Software Design (FOSD)

Core Design Paradigm

Function-Oriented Design decomposes a complex system into a hierarchy of functional units (modules/functions), where each module transforms input data streams into output data streams.

Key Characteristics:

- Top-down functional decomposition.
- System expressed as high-level functions invoking sub-functions.
- Centralized data management passed across function boundaries.
- Focus on processes rather than objects/data models.

Primary Analysis Artifacts:

- **DFD**: Data Flow Diagram (Data pipeline)
- **SC**: Structure Chart (Control hierarchy)
- **DD**: Data Dictionary (Data definitions)
- **PSPEC**: Process Specification (Logic)

Data Flow Diagram (DFD): Fundamentals

Definition of Data Flow Diagram

A Data Flow Diagram (DFD) is a graphical modeling technique that depicts the flow of data through an information system, illustrating data inputs, processing steps, storage repositories, and output destinations.

- **Data-Centric Abstraction**: Focuses purely on data transformations without representing control logic, loops, or execution sequences.
- **Multi-Level Refinement**: Enables stepwise refinement from a high-level context view to detailed sub-system processes.
- **Communication Tool**: Provides a clear non-technical representation easily understood by users, analysts, and developers.
- **Core Components**: External Entities, Processes, Data Stores, and Data Flows.

DFD Symbols and Notations

Table: Comparison of Standard DFD Notations

DFD Element	Yourdon & DeMarco Notation	Gane & Sarson Notation
Process	Circle (Bubble)	Rounded Rectangle with top section
Data Flow	Labelled Directed Arrow	Labelled Directed Arrow
Data Store	Parallel Lines (Open ended)	Rectangle with open right end
External Entity	Rectangle / Square	Shadowed Rectangle / Double Square

Fundamental Rule of Data Flow

Every data flow arrow **must** connect to at least one Process block. Direct connectivity between External Entities, or between an External Entity and a Data Store, is strictly invalid.

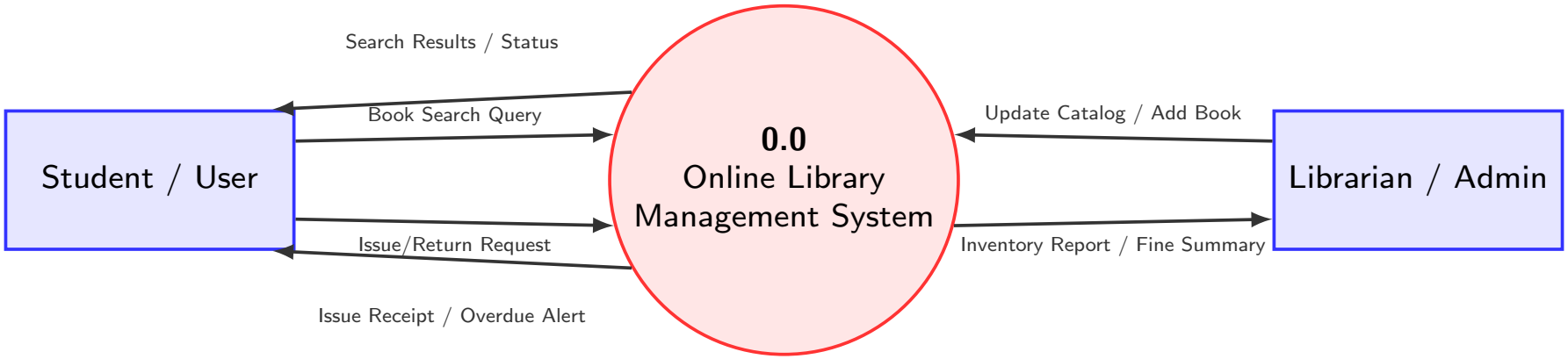
Context-Level DFD (Level 0 DFD)

Definition and Purpose

The Context-Level DFD represents the highest level of abstraction in functional modeling. It defines the complete system as a single central process interacting with surrounding external entities.

- **System Perimeter**: Clearly demarcates what is inside vs. outside the system boundary.
- **Process 0**: The entire system is modeled as a single bubble labeled Process **0.0** (e.g., *Software Engineering LMS*).
- **Abstraction of Internal Storage**: Internal data stores are hidden at Level 0 to maintain architectural overview.
- **External Interfaces**: Explicitly captures external actors (Users, External APIs, Hardware) and main inputs/outputs.

Context-Level DFD Example



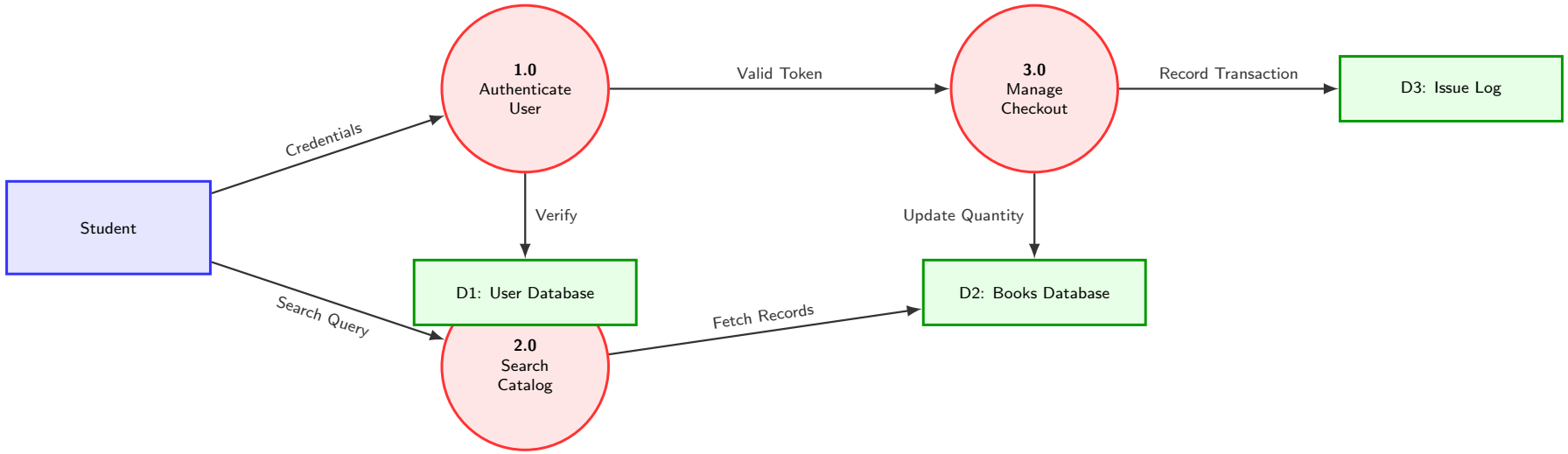
Level-1 DFD: Functional Decomposition

Exploding the Context Diagram

Level-1 DFD decomposes Process 0.0 into its main functional sub-processes, exposing key internal data stores and intermediate data interactions.

- **Sub-process Numbering**: Major modules are assigned decimal identifiers (**1.0, 2.0, 3.0**, etc.).
- **Data Stores Exposure**: Introduces internal files and database tables (labeled **D1, D2, D3**, etc.).
- **Functional Partitioning**: Separates concerns into modules (e.g., User Authentication, Catalog Search, Transaction Management).
- **Interface Preservation**: All external entities and flows from Level 0 must be preserved at Level 1.

Level-1 DFD Architecture Example



DFD Rules, Balancing, and Common Errors

Rule of DFD Balancing

A parent DFD and its exploded child DFD are **balanced** if the net input and output data flows connecting the parent process match precisely with the boundary flows of the child diagram.

Constructive DFD Rules:

- Processes must transform input into output (Verb-Noun naming).
- Data cannot spontaneous generate or disappear inside a store.

Common Design Anomalies:

- **Black Hole**: Process with inputs but no outputs.
- **Miracle**: Process with outputs but no inputs.
- **Grey Hole**: Inputs insufficient to generate output.

Comparison: Context DFD vs. Level-1 DFD

Table: Architectural Comparison of DFD Levels

Feature / Aspect	**Context Level DFD (Level 0)**	**Level-1 DFD**
Abstraction Level	Global system overview	Decomposed functional view
Process Count	Exactly 1 process (Process 0.0)	Multiple sub-processes (1.0, ...)
Internal Data Stores	Hidden / Not depicted	Explicitly shown (D1, D2, ...
Primary Objective	Define system scope & external boundaries	Map functional data transfo tions
Target Audience	Clients, Business Analysts, Executives	System Architects, Software D opers

Function-Oriented Software Design (FOSD)

Core Philosophy

Function-Oriented Software Design focuses on decomposing a complex software system into a hierarchy of functional units or modules. Each module performs a specific, well-defined data transformation.

- **Top-Down Decomposition**: The high-level system function is progressively partitioned into smaller, manageable sub-functions.
- **Data-Centric Flow**: Emphasizes how data inputs move through functional transformations to produce desired outputs.
- **State Independence**: Functions ideally operate independently, relying primarily on input parameters and minimizing global state side-effects.
- **Key Artifacts**: Data Flow Diagrams (DFDs), Data Dictionaries, Structure Charts, and Decision Tables.

Contrast with Object-Oriented Design (OOD)

While OOD packages data and operations together into entities (objects), FOSD treats functions and data as separate abstractions.

Data Flow Diagram (DFD): Fundamentals

Definition

A Data Flow Diagram (DFD) is a graphical modeling tool used in Structured Analysis to visualize how data flows through an information system, the processes that transform data, and the storage locations for data.

Key Characteristics:

- Implementation-independent view.
- Shows *what* system does, not *how*.
- Non-procedural (no loops or control flow logic).
- Easily understood by non-technical stakeholders.

Primary Objectives:

- Define system boundaries.
- Specify functional requirements clearly.
- Map input-to-output data transformations.
- Provide a foundation for software architectural design.

DFD Notations and Symbols

Table: Standard DFD Notations: Yourdon-DeMarco vs. Gane-Sarson

DFD Element	Yourdon & DeMarco	Gane & Sarson
Process	Circle (Bubble)	Rounded Rectangle
Data Store	Parallel Horizontal Lines	Open-ended Rectangle
External Entity	Rectangle	Double-bordered Rectangle / Square
Data Flow	Named Directed Arrow	Named Directed Arrow

Description of DFD Components

- **Process**: Transforms incoming data flows into outgoing data flows.
- **Data Store**: Repository of data at rest (database, file, paper archive).
- **External Entity**: Source or sink of data outside system boundary (users, external hardware, third-party software).
- **Data Flow**: Pipeline transporting data packets with meaningful titles.

Rules and Constraints for DFD Construction

Core Construction Rules

To maintain consistency and validity, DFD models must satisfy strict structural rules:

1. **Process Rules**:
   - Every process must have at least one input flow and one output flow.
   - *Black Hole*: Process with inputs but no outputs (Invalid).
   - *Miracle*: Process producing outputs without inputs (Invalid).
2. **Data Store Rules**:
   - Data cannot move directly between two data stores; it must pass through a process.
   - External entities cannot interact directly with data stores.
3. **Entity Rules**:
   - External entities cannot connect directly to other external entities.

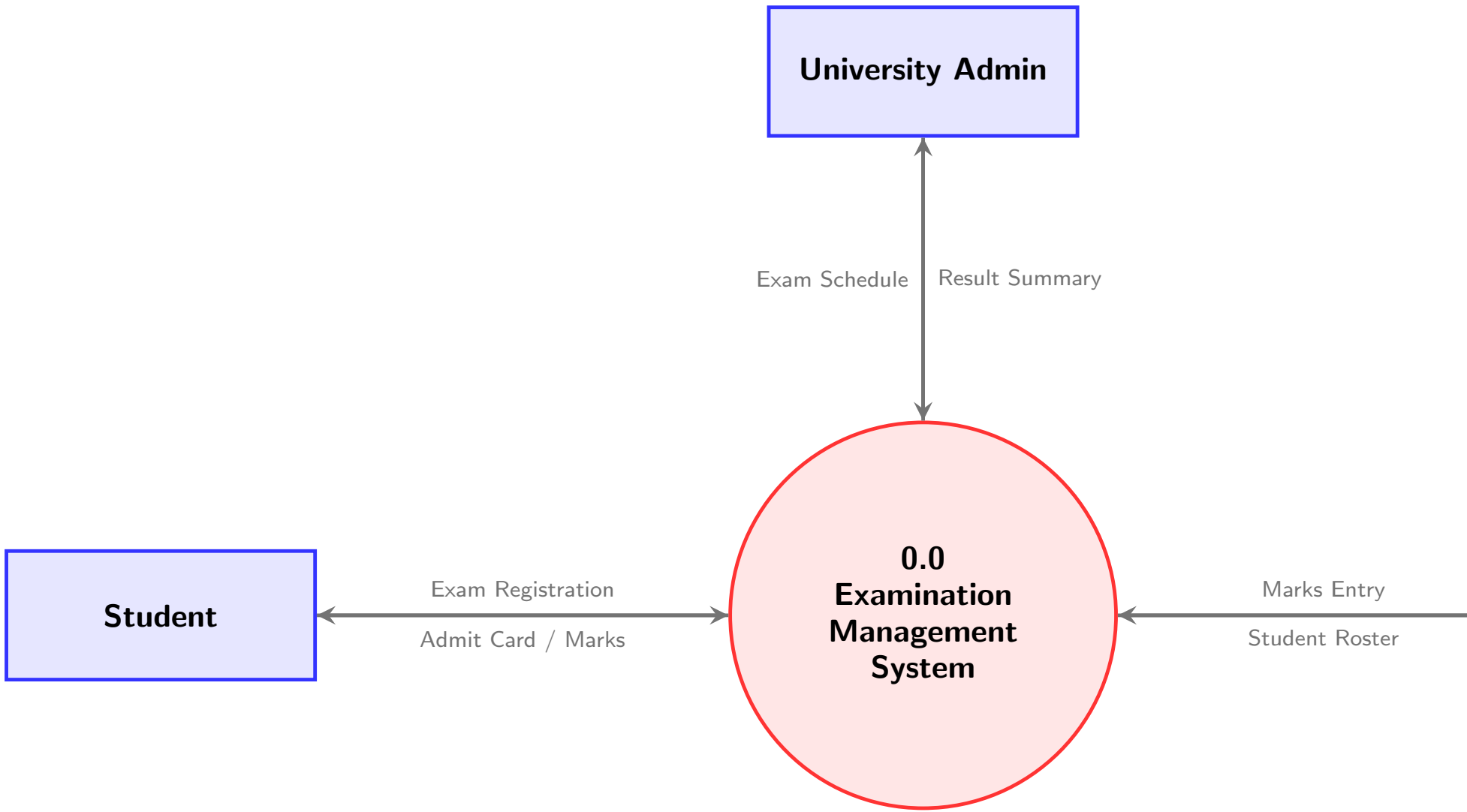
Context Level DFD (Level-0 DFD)

Concept

The Context Level DFD represents the entire software application as a **single central process** (labeled Process 0.0), establishing the boundary between the system and its environment.

- **Purpose**: Identifies external interfaces, external actors (sources/sinks), and global inputs/outputs.
- **Characteristics**:
  - Contains exactly **one** process node representing the entire system.
  - Displays **no internal data stores** (data stores are hidden inside Process 0.0).
  - Clearly delineates what is inside versus outside the system boundary.
- **GTU Application Domain**: Online University Examination System.

Context Level DFD: University Examination System



Level-1 Data Flow Diagram (Level-1 DFD)

Decomposition of Process 0.0

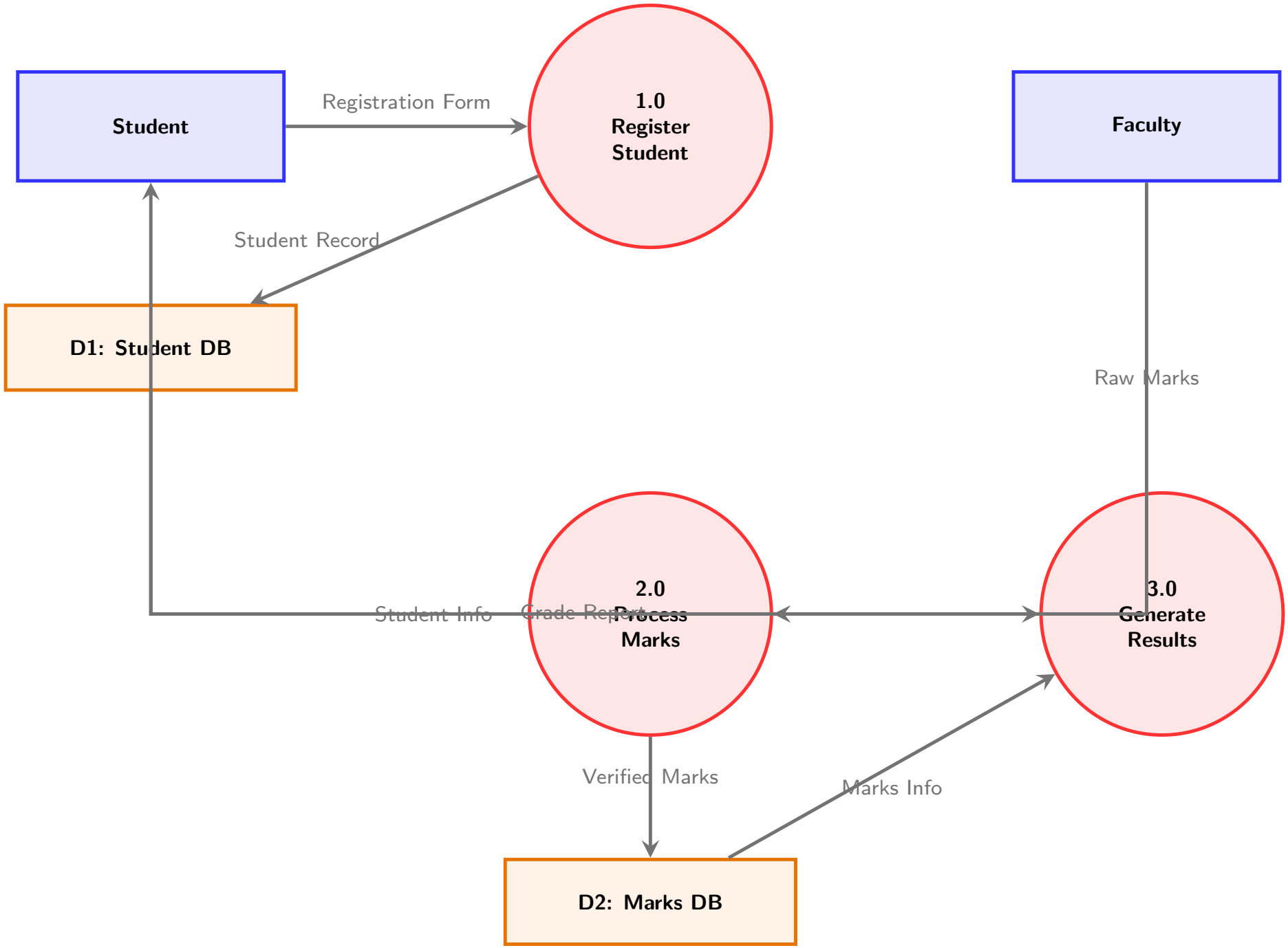
A Level-1 DFD explodes the single process from the Context Diagram into its major functional sub-processes, exposing core internal operations and persistent data stores.

- **Functional Partitioning**: Breaks down Process 0.0 into numbered sub-processes (e.g., 1.0, 2.0, 3.0).
- **Introduction of Data Stores**: Internal data repositories (e.g., Student DB, Exam DB) appear for the first time.
- **Data Flow Balancing Rule**:

Rule of Conservation

All input and output data flows entering or leaving the Context Diagram **must** appear with identical names and directions at Level-1.

Level-1 DFD Architecture



DFD Design Guidelines & Common Errors

Best Practices

- Limit processes per level to 7 ± 2 to prevent cognitive clutter.
- Use clear verb-noun phrases for processes (e.g., *Calculate Grade*).
- Label all data flows with explicit noun names.
- Maintain continuous traceability between levels.

Common Errors

- **Unbalanced DFD**: Mismatch of boundary flows between Context and Level-1.
- **Control Flow Fallacy**: Adding decision arrows (if/else) into DFD.
- **Direct Entity-Store Link**: Bypassing processes to access databases.
- **Spaghetti Flow**: Crossing data lines due to poor spatial layout.

Summary & GTU Review Questions

Lecture Summary

Function-Oriented Design organizes software around transformations. DFDs provide a multi-level abstract visual model of data movement, progressing systematically from Context Level (boundary) to Level-1 (functional modules) and beyond.

GTU Exam Questions (DI04000011)

1. Explain the significance of Data Flow Diagrams in Structured Analysis. Differentiate between Context Level DFD and Level-1 DFD. **[7 Marks]**
2. Draw Context Level DFD and Level-1 DFD for an Automated Teller Machine (ATM) system or Library Management System. **[7 Marks]**
3. List and illustrate the rules for drawing a valid DFD. What are "Black Hole" and "Miracle" processes? **[4 Marks]**

Overview & Module Context

Software Requirement Analysis and Design → Function-Oriented Software Design (FOD)

- **Core Philosophy**: Views a software system as a collection of functional modules that transform inputs into desired outputs.
- **Top-Down Decomposition**: High-level system functions are recursively decomposed into smaller, manageable, and independent sub-functions.
- **Primary Artifact**: Data Flow Diagram (DFD) — a graphical modeling tool representing data transformations across system boundaries.

Key Objective of Lecture 23

Master the principles of Data Flow Diagrams (DFDs), Context-Level (Level 0) DFDs, Level-1 DFDs, and structural decomposition rules in Structured Analysis.

Principles of Function-Oriented Software Design

Fundamental Principles

1. **Functional Abstraction**: Focuses on *what* functions perform rather than how data objects are encapsulated.
2. **Data Flow Focus**: Data flows sequentially through processing steps (pipelined architecture).
3. **Centralized Control**: Higher-level modules control execution flow of lower-level functional subroutines.

FOD vs. Object-Oriented Design

- **FOD**: Functions are primary entities; data streams pass between functions.
- **OOD**: Objects combining data and operations are primary.
- **Best Suited For**: Computational systems dominated by data transformations (e.g., compilers, report generators, signal processors).

Fundamentals of Data Flow Diagrams (DFD)

What is a Data Flow Diagram?

A graphical representation of the flow of data through an information system, modeling its process aspects without revealing procedural logic or control structures (no loops or conditionals).

Four Core Components of a DFD

- **Process**: Transforms incoming data flows into outgoing data flows.
- **Data Flow**: Represents data in motion (labeled directed arrows).
- **Data Store**: Represents data at rest (repositories, databases, or files).
- **External Entity (Source/Sink)**: External actors/systems that supply or consume data.

Standard DFD Notations & Components Comparison

Table: Comparison of Gane & Sarson vs. Yourdon & DeMarco DFD Notations

Component	Yourdon & DeMarco	Gane & Sarson	Semantic Function
Process	Circle (Bubble)	Rounded Rectangle	Data transformation
Data Flow	Directed Arrow	Directed Arrow	Movement of data
Data Store	Parallel Lines	Open-ended Box	Repository / Data a
External Entity	Rectangle	Double-bordered Rect	External Source or tion

DFD Hierarchy and Concept of Leveling

Why DFD Leveling?

Complex software systems cannot be effectively modeled in a single diagram. **Leveling** (functional decomposition) breaks system complexity into a hierarchy of diagrams.

Hierarchy Levels

- **Context Level (Level 0)**: Single process, system boundary, external actors.
- **Level 1 DFD**: Major functional subsystems and data stores.
- **Level 2+ DFD**: Detailed sub-process breakdown.
- **Primitive DFD**: Atomic processes specified via Mini-Specs.

Balancing Rule

- All input and output data flows connected to a parent process **must** match the net inputs and outputs of its child DFD diagram.
- Guarantees data flow consistency across abstraction levels.

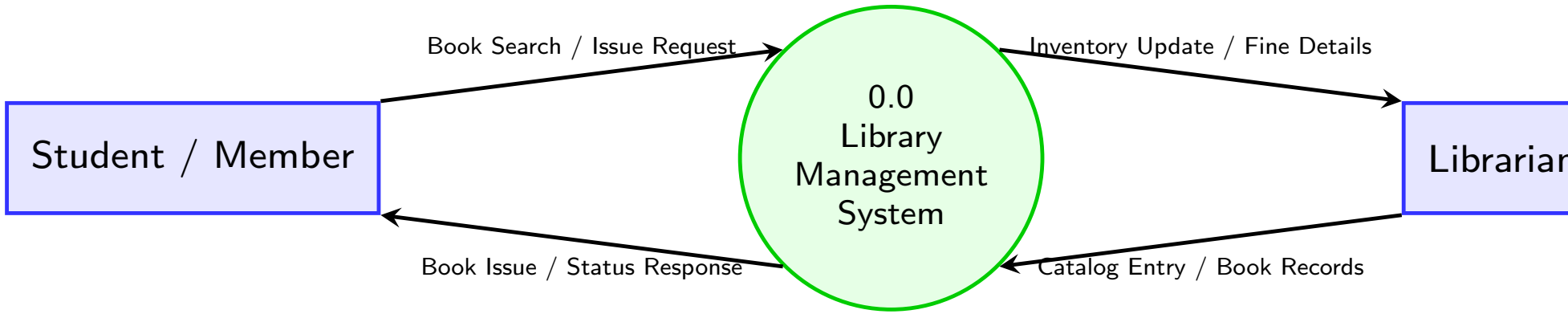
Context-Level DFD (Level 0 DFD)

Definition & Scope

The highest-level abstraction of a software system in Structured Analysis, establishing system boundaries.

- **Single Bubble**: The entire software system is represented as a single central process labeled *"0"* or *"System Name"*.
- **No Internal Data Stores**: Internal data stores are hidden at this level to maintain high-level scope focus.
- **System Boundary Definition**: Explicitly defines interaction between the software system and external entities (users, hardware devices, external APIs).
- **High-Level Data Streams**: Depicts primary input requests and primary output results/reports.

Context-Level DFD Example: Library Management System



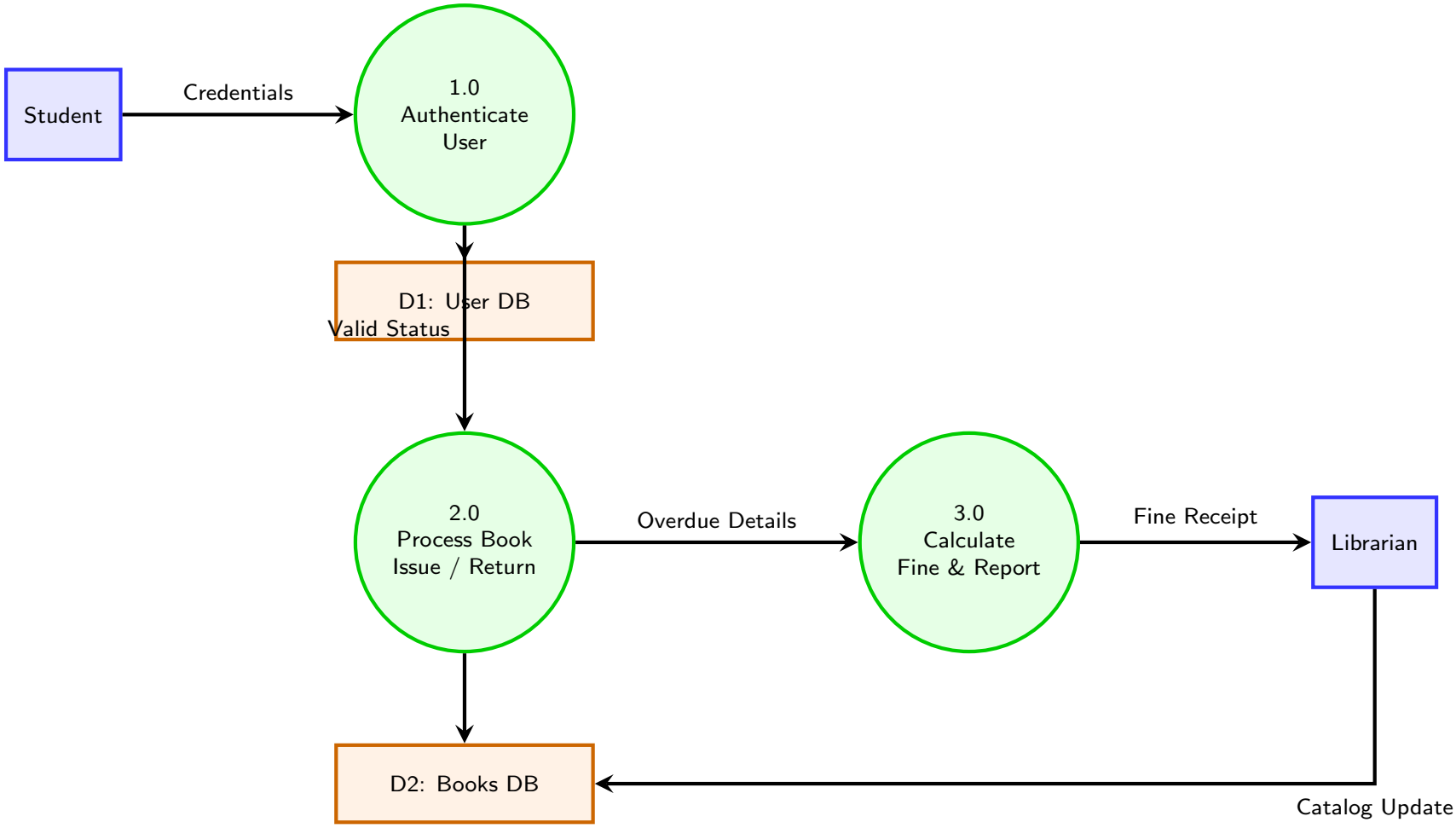
Level-1 Data Flow Diagram (Functional Decomposition)

Purpose of Level-1 DFD

Decomposes the Context-Level single process into major functional sub-processes, revealing major internal workflows and data repositories.

- **Process Numbering**: Processes are identified with integer IDs: 1.0, 2.0, 3.0, etc.
- **Introduction of Data Stores**: Internal repositories (D1, D2, ...) are introduced for data persistence between processes.
- **Preservation of External Entities**: External entities remain identical to Level 0 to maintain boundary consistency.
- **Subsystem Identification**: Each process handles a distinct major task (e.g., Authentication, Processing, Reporting).

Level-1 DFD Architectural Breakdown



DFD Construction Rules & Level Balancing Principles

Essential DFD Structural Rules

- **No Black Holes**: A process must not have input flows without producing output flows.
- **No Miracles**: A process cannot generate output flows without receiving input flows.
- **No Gray Holes**: Inputs to a process must be sufficient to produce its designated output flows.
- **No Direct Entity-to-Store Flow**: External entities cannot interact directly with a Data Store or another Entity without an intermediate process.

Summary of Lecture 23

Function-Oriented Design provides a structured top-down decomposition framework using Context-Level (Level 0) and Level-1 DFDs to establish clear functional boundaries, data transformations, and storage architectures.

GTU DI04000011: Software Engineering

Unit: Software Requirement Analysis and Design
Topic: Object-Oriented Modeling with Unified Modeling Language (UML)

- **Object-Oriented Analysis and Design (OOAD):**
  - Models real-world software entities as *objects* combining state (attributes) and behavior (methods).
  - Promotes foundational software engineering principles: Abstraction, Encapsulation, Modularity, and Hierarchy.
- **Role of UML in Software Engineering:**
  - Industry-standard visual language standardized by the Object Management Group (OMG).
  - Serves as a architectural blueprint for specification, visual modeling, construction, and documentation.
  - Facilitates communication between business analysts, software architects, developers, and QA engineers.
- **Core UML Building Blocks:** Structural/Behavioral Things, Relationships, and Diagrams.

UML Diagram Taxonomy & Classification

Table: Classification of Core UML 2.x Diagrams

Diagram Type	Category	Primary Purpose in SE	Development Phase
Use Case Diagram	Behavioral	Captures functional requirements & actor interactions	Requirements Analysis
Class Diagram	Structural	Models static structure, entity attributes, & relationships	Design Phase
Sequence Diagram	Interaction	Visualizes temporal message exchange between objects	Detailed Design
Activity Diagram	Behavioral	Represents procedural workflows, control, & data flow	Requirements & Logic
Component Diagram	Implementation	Models physical software components & interface dependencies	Architectural Design

- **Structural Views:** Capture the static organization of system elements (Classes, Components, Nodes).
- **Behavioral Views:** Capture dynamic execution patterns, state changes, and operational workflows over time.

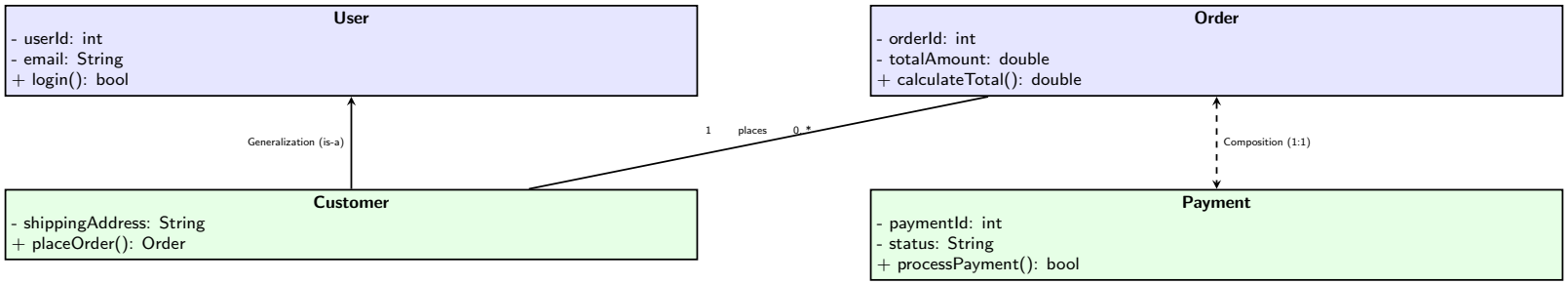
Use Case Diagram: Requirements Modeling

- **Objective:** Defines system boundaries and functional scope from the perspective of external actors.
- **Core Components:**
  - **Actor:** External entity (User, Hardware, External Subsystem) interacting with the application.
  - **Use Case:** High-level unit of business function yielding a measurable outcome.
  - **System Boundary:** Enclosing boundary defining the operational scope of the software.
- **Key Stereotypes & Relationships:**
  - **Association:** Structural communication link between Actor and Use Case.
  - <<**include**>>**:** Mandatory dependency where a base use case explicitly incorporates sub-routine logic.
  - <<**extend**>>**:** Optional dependency executing conditionally at specified extension points.
  - **Generalization:** Inheritance hierarchy between actors or between use cases.

Class Diagram: Structural Foundations

- **Definition:** Central structural artifact depicting system classes, attributes, methods, and static relationships.
- **Class Structure:** Three-part box containing Class Name, Attributes (Data), and Operations (Methods).
- **Visibility Notation:** + Public, – Private, # Protected, ~ Package/Default.
- **Structural Relationships:**
  - **Association:** Structural connection with defined multiplicity (e.g., $1 \cdots *$).
  - **Aggregation (Shared):** Weak "has-a" ownership; child entity lifespan is independent of parent.
  - **Composition (Composite):** Strong "has-a" ownership; child lifecycle is tied directly to parent.
  - **Generalization:** "Is-a" inheritance link from specialized class to generalized superclass.
  - **Realization:** Contractual implementation of an Interface by a concrete class.

Class Diagram Visual Modeling (TikZ Structural Model)



Structural Model Insights

Demonstrates class hierarchy (`Customer` extends `User`), multiplicity association between `Customer` and `Order`, and composite relationship with `Payment`.

Sequence Diagram: Dynamic Interaction Modeling

- **Purpose:** Captures temporal order of message exchanges among object instances for specific execution scenarios.
- **Core Notational Elements:**
  - **Lifeline:** Vertical dashed line representing participant existence over chronological time.
  - **Activation Bar:** Narrow box overlaying lifeline showing active focus of control.
  - **Messages:** Horizontal directed arrows modeling communication invocations.
- **Message Call Types:**
  - **Synchronous Message:** Caller blocks awaiting response (Solid line, filled arrow).
  - **Asynchronous Message:** Caller continues without blocking (Solid line, open arrow).
  - **Return Message:** Explicit return value sent back to caller (Dashed line, open arrow).
- **Combined Fragments:** Control structures for conditional execution (`alt`, `opt`) and looping (`loop`).

Activity Diagram: Workflow & Control Flow

- **Purpose:** Models algorithmic logic, business process workflows, and parallel operational tasks.
- **Essential Diagram Elements:**
  - **Initial Node:** Solid black circle indicating process starting point.
  - **Activity/Action Node:** Rounded rectangle representing discrete executable work steps.
  - **Decision Point:** Diamond evaluating guard conditions (`[condition]`) to branch flow.
  - **Fork & Join Bars:** Synchronization bars splitting execution into concurrent threads or joining them.
  - **Final Node:** Encased black dot marking workflow termination.
- **Swimlanes (Partitions):** Columns or rows organizing activities by responsible system module or organizational entity.

Component Diagram: Architectural Implementation

- **Purpose:** Visualizes physical organization, modular code packaging, and subsystem interfaces.
- **Core Concepts:**
  - **Component:** Modular, replaceable piece of system code encapsulating implementation details.
  - **Provided Interface ("Lollipop"):** Operations exposed by a component for external consumers.
  - **Required Interface ("Socket"):** External dependencies required by a component to function.
  - **Assembly Connector:** Wiring mechanism matching provided and required interface contracts.
- **Engineering Application:** Supports Component-Based Software Engineering (CBSE), enterprise application integration, and microservice decoupling.

Comparative Analysis of UML Modeling Techniques

Table: Comprehensive Comparison of Core UML Diagrams

Diagram	Perspective	Abstraction	Primary Focus	Key Stakeholders
Use Case	Requirements	High / Business	System Scope & Functional Goals	Clients, Business Analysts
Class	Static Structure	Medium / Logical	Data Entities, Operations, & Relations	Software Architects, Developers
Sequence	Dynamic Interaction	Low / Detailed	Message Timing & Call Flow	Developers, QA Engineers
Activity	Dynamic Workflow	Medium / Procedural	Logic Control & Concurrency	System Analysts, Process Designers
Component	Physical Architecture	Low / Implementation	Software Modules & Interfaces	System Architects, DevOps Engineers

Integrated Modeling Strategy

Comprehensive Software Engineering demands coupling structural models (Class/Component) with behavioral models (Use Case/Sequence/Activity) for complete requirement traceability.

Best Practices & OOAD Summary for GTU DI04000011

- **Software Engineering Best Practices in UML Modeling:**
  - Maintain inter-diagram consistency (e.g., sequence messages must correspond to class operations).
  - Incorporate **SOLID** design principles during structural class modeling.
  - Avoid over-modeling; target appropriate levels of abstraction relative to project scale.
- **Lecture 24 Key Takeaways:**
  - UML standardizes visual communication across the software development lifecycle (SDLC).
  - Mastered 5 core UML diagrams: Use Case, Class, Sequence, Activity, and Component diagrams.
- **GTU Examination Review Questions:**
  1. Differentiate between `<<include>>` and `<<extend>>` relationships with suitable examples.
  2. Contrast Aggregation and Composition with respect to memory lifecycle management.

Overview of Object-Oriented Modeling (OOM)

Object-Oriented Modeling uses objects as the primary abstraction for analyzing and designing software systems. It bridges the gap between real-world problem domains and software implementation.

- **Unified Modeling Language (UML):** Standardized visual modeling language maintained by the Object Management Group (OMG).
- **Dual Perspectives in UML:**
  - **Structural Modeling:** Defines static architecture, classes, interfaces, and physical software components.
  - **Behavioral Modeling:** Captures dynamic behavior, object interactions, state changes, and activity workflows.
- **Role in GTU DI04000011:** Crucial during the Software Requirement Analysis and Design phases of the Software Development Life Cycle (SDLC).

Overview of Essential UML Diagrams

UML Diagram	Category	Primary Focus	Key Elements
Use Case Diagram	Behavioral	Functional Requirements	Actors, Use Cases, Include/Extend
Class Diagram	Structural	Static Architecture	Classes, Attributes, Operations, Links
Component Diagram	Structural	Software Subsystems	Components, Provided/Required Interfaces
Sequence Diagram	Behavioral	Time-ordered Interactions	Lifelines, Messages, Activation Bars
Activity Diagram	Behavioral	Workflow & Business Logic	Actions, Decisions, Fork/Join, Swimlanes

Table: Comparison of Key UML Diagrams in Software Engineering

Use Case Diagram - Capturing System Functionality

Purpose

Specifies the external behavioral view of the system by identifying system boundaries, actors, and functional interactions.

- **Actor:** External entity (User, System, Hardware) interacting with the application.
- **Use Case:** High-level system feature representing a complete unit of useful work.
- **Relationships:**
  - **≪include≫:** Mandatory sub-routine/dependency required by a base use case.
  - **≪extend≫:** Optional or conditional enhancement to a base use case under specific conditions.
  - **Generalization:** Inheritance relationship between actors or between use cases.

Class Diagram - Structural Blueprint

Core Composition

Represents object structure with attributes, operations (methods), and visibility modifiers (+, −, #, ∼).

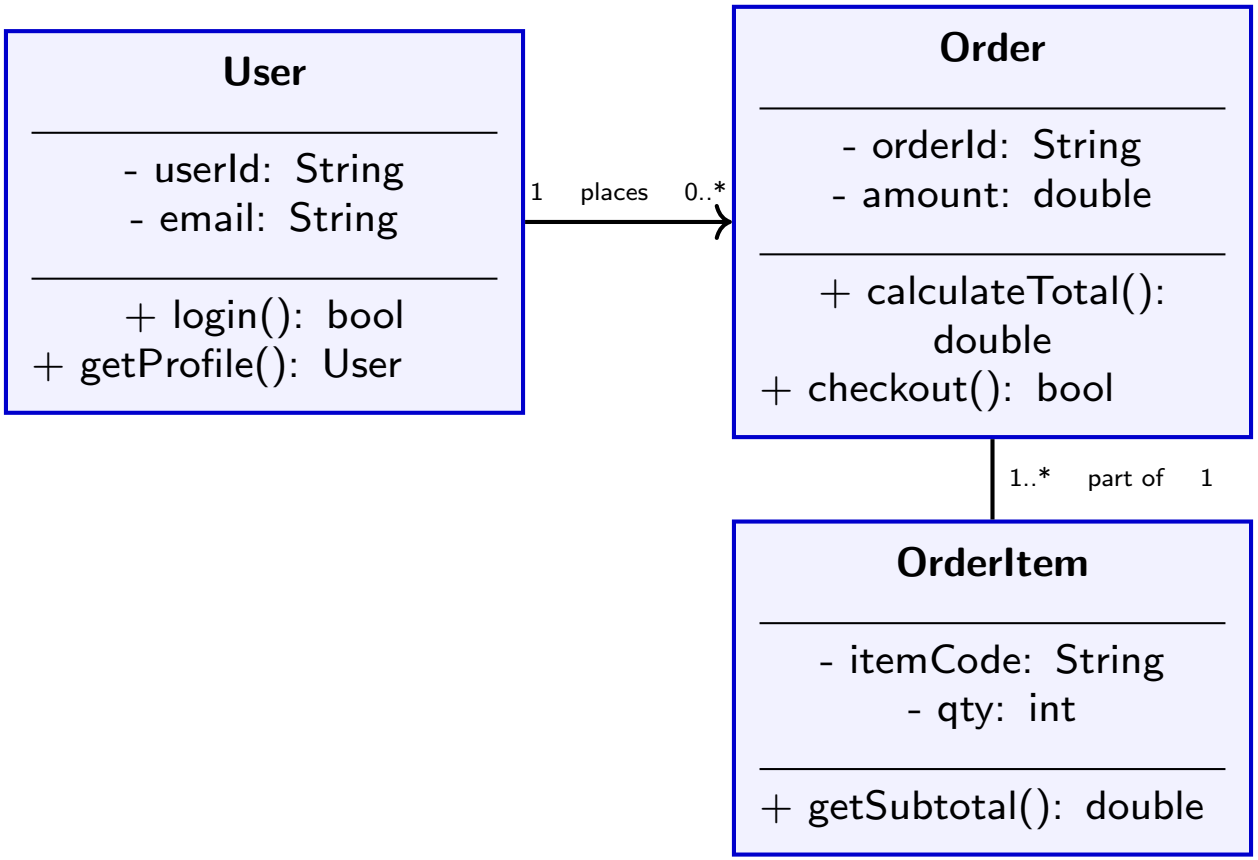
Relationships:

- **Association:** Simple structural connection between classes.
- **Aggregation:** Weak "has-a" relationship (part can exist independently).
- **Composition:** Strong "has-a" ownership (part lifecycle bound to whole).

Advanced Concepts:

- **Generalization:** Inheritance ("is-a" relationship).
- **Realization:** Interface implementation.
- **Multiplicity:** Specifies numerical bounds (e.g., 1, 0..∗, 1..∗).

TikZ UML Representation - Class Diagram Example



Structural Summary

Models static relationships: User-to-Order Association ($1 : N$) and Order-to-OrderItem Composition ($1 : N$).

Component Diagram - Physical Subsystems

Definition

Illustrates the organization and dependencies among software components, supporting Component-Based Software Engineering (CBSE).

- **Component:** Autonomous, reusable module encapsulating implementation details (e.g., DLL, JAR, Web Service).
- **Provided Interface (Lollipop notation):** Services offered by the component to external modules.
- **Required Interface (Socket notation):** Services required by the component from external modules.
- **Assembly Connector:** Connects provided interface of one component to required interface of another.
- **Application in SE:** System architecture specification, deployment planning, and modular maintenance.

Sequence Diagram - Time-Ordered Message Passing

Overview

Interaction diagram emphasizing the chronological order of messages exchanged between objects to achieve a use case scenario.

Structural Notation:

- **Lifeline:** Vertical dashed line representing participant existence over time.
- **Activation Bar:** Narrow rectangle showing active execution of an operation.

Message Types:

- **Synchronous (Solid Arrowhead):** Caller waits for response.
- **Asynchronous (Open Arrowhead):** Caller continues without waiting.
- **Return Message (Dashed Arrow):** Value returned from call.

Combined Fragments

Used for complex control logic: `alt` (conditional branching), `opt` (optional execution), `loop` (iteration).

Activity Diagram - Procedural Flow of Control

Definition

Specialized behavioral diagram used to model operational workflows, algorithms, and business logic execution paths.

- **Control Nodes:**
  - **Initial Node (Filled Circle):** Starting point of activity execution.
  - **Decision / Merge (Diamond):** Conditional branch points and path mergers.
  - **Fork & Join Bars (Solid Line):** Splitting execution into concurrent parallel threads and synchronizing them.
  - **Final Node (Bullseye):** Termination of workflow execution.
- **Swimlanes (Partitions):** Visual grouping representing organizational responsibilities (e.g., Customer, Payment Gateway, Warehouse).

UML Diagrams Selection Matrix in Software Lifecycle

SDLC Phase	Primary Diagram	Perspective	Engineering Artifact
Requirements Analysis	Use Case Diagram	External / Functional	Requirement Specification
Domain / System Design	Class Diagram	Static / Structural	Class Structures & Schemas
High-Level Architecture	Component Diagram	Physical / Modular	Component Interfaces
Detailed Design	Sequence Diagram	Dynamic / Temporal	API Method Invocations
Business Process Modeling	Activity Diagram	Behavioral / Control	Workflow & Algorithm Logic

Table: Mapping UML Diagrams to Software Engineering SDLC Phases

Summary & Key GTU DI04000011 Exam Points

Summary of Key Concepts

- Object-Oriented Modeling organizes software into modular, manageable, and reusable components using UML.
- Structural diagrams (Class, Component) define system architecture, while Behavioral diagrams (Use Case, Sequence, Activity) model system execution.

Frequent GTU Examination Questions

1. Differentiate between ≪`include`≫ and ≪`extend`≫ relationships in Use Case Diagrams with suitable examples.
2. Compare Aggregation vs. Composition with real-world class diagram examples.
3. Explain Fork, Join, and Swimlanes in Activity Diagrams.
4. Draw a Sequence Diagram for User Authentication or E-Commerce Checkout.

Introduction to Object-Oriented Modeling (OOM)

- **Object-Oriented Modeling**: A paradigm that visualizes software systems as a collection of interacting objects combining data (attributes) and behavior (methods).
- **Role of UML**: Unified Modeling Language (UML) provides a standardized, visual language to specify, construct, visualize, and document software system artifacts.

Key Objectives for Lecture 26

- Understand the dual architectural perspective of UML: **Structural** vs. **Behavioral** modeling.
- Master notation, modeling rules, and practical application of 5 core UML diagrams:
  1. **Use Case Diagram**: User-system interaction boundaries.
  2. **Class Diagram**: Static structural backbone.
  3. **Sequence Diagram**: Dynamic message-passing logic over time.
  4. **Activity Diagram**: Workflow procedural control flow and parallelism.
  5. **Component Diagram**: Physical software architecture and modules.

UML Diagram Classification & Architectural Views

Structural Diagrams

Focus on static elements of the software system without considering temporal state changes.

- **Class Diagram**: Structural blueprint (Classes, Interfaces, Relationships).
- **Component Diagram**: Physical modules, libraries, runtime components, and interfaces.
- *Additional*: Package, Deployment, and Object diagrams.

Behavioral Diagrams

Focus on dynamic execution, state transitions, temporal interaction, and process workflows.

- **Use Case Diagram**: High-level functional requirements and actor interaction.
- **Sequence Diagram**: Time-ordered object interaction and message passing.
- **Activity Diagram**: Step-by-step procedural workflow and decision logic.

GTU Exam Note

Always justify UML diagram selection based on whether the requirement specifies **static system structure** or **dynamic procedural/temporal behavior**.

Core Architectural Elements

- **Actor**: External entity (User, Hardware, External System) interacting with the system. Represented as a stick figure.
- **Use Case**: A set of actions performed by the system yielding an observable result of value to an actor. Represented as an ellipse.
- **System Boundary**: Rectangle enclosing use cases, delimiting system scope.

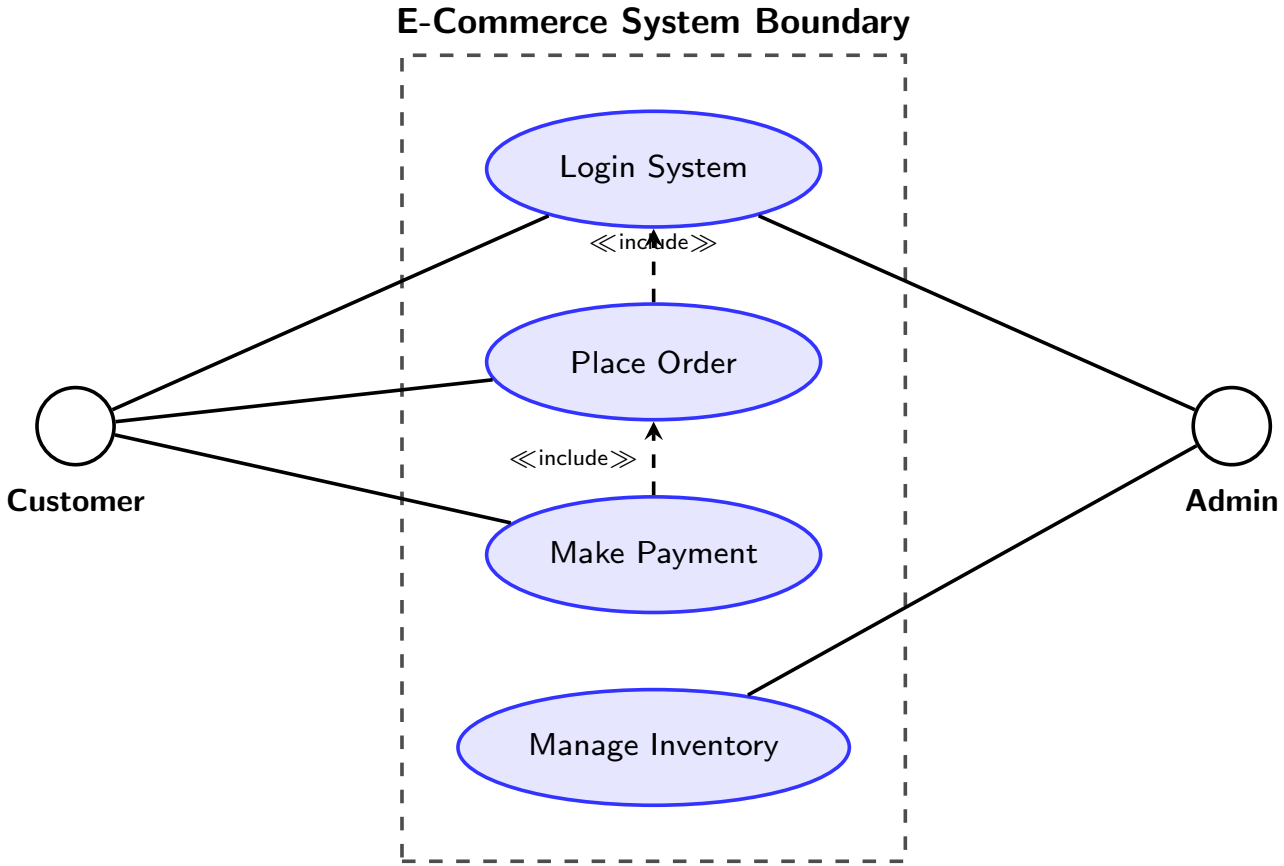
Relationships in Use Case Diagrams

`<<include>>`: Mandatory sub-routine relationship. Target use case MUST be executed during source execution (e.g., *Checkout* $\xrightarrow{\ll\text{include}\gg}$ *Authenticate*).

`<<extend>>`: Optional/Conditional behavior executed under explicit extension points (e.g., *Apply Coupon* $\xrightarrow{\ll\text{extend}\gg}$ *Checkout*).

Generalization: Inheritance relationship between actors or use cases (is-a relationship).

Use Case Diagram: Architectural TikZ Model



Class Structure (3 Compartments)

- **Class Name**: Top compartment, centered, bold (Abstract classes in italics).
- **Attributes**: `[visibility] name : type [= default]`
- **Operations/Methods**: `[visibility] name(parameterList) : returnType`

Visibility Specifiers

- + Public (Accessible anywhere)
- - Private (Class internal only)
- # Protected (Class & subclasses)
- ~ Package/Default (Package scope)

Multiplicity Notation

- `1`: Exactly one — `0..1`: Zero or one — `*`, `0..*`: Zero or more — `1..*`: One or more

Class Diagram Relationships Taxonomy

Table: **Comparison of Class Diagram Relationships**

Relationship	Semantics & Description	UML Line Symbol	Lifetime Dependency
Association	Structural connection between classes ("uses a")	Solid line	Independent
Aggregation	Weak whole-part relationship ("has a")	Solid line with open diamond	Part survives Whole
Composition	Strong whole-part relationship ("owns a")	Solid line with filled diamond	Part dies with Whole
Generalization	Inheritance hierarchy ("is a")	Solid line with hollow arrowhead	Subclass depends on Superclass
Dependency	Temporary usage relationship ("depends on")	Dashed line with open arrowhead	Transient parameter usage

Key Elements of Sequence Diagrams

- **Lifeline**: Vertical dashed line representing the existence of an object over time.
- **Activation Box**: Vertical thin rectangle on lifeline indicating active object execution.
- **Object Header**: Formatted as `objectName : ClassName`.

Message Types & Notations

Synchronous Message: Solid line with filled arrowhead ($\longrightarrow$). Sender waits for completion.

Asynchronous Message: Solid line with open arrowhead ($\longrightarrow$). Sender continues without waiting.

Return Message: Dashed line with open arrowhead ($\dashrightarrow$). Returns control or value.

Self Call: Message arrow looping back to the originating lifeline.

Combined Fragments: `alt` (if-else), `loop` (iteration), `opt` (optional execution).

Fundamental Nodes & Symbols

- **Initial Node**: Solid filled circle (●) initiating execution.
- **Final Node**: Filled circle inside hollow circle (⊙) terminating workflow.
- **Action State**: Rounded rectangle describing an executable step or computation.
- **Decision / Merge Node**: Diamond symbol (◇) evaluating guard conditions `[condition]`.

Concurrency & Organizational Swimlanes

- **Fork Bar**: Solid horizontal/vertical line splitting 1 input flow into concurrent parallel threads.
- **Join Bar**: Synchronization bar merging multiple parallel threads back into 1 single flow.
- **Swimlanes (Partitions)**: Columns grouping activities by responsible business role or system component.

Purpose of Component Diagrams

- Model physical software components (libraries, executables, source code files, databases).
- Illustrate high-level software modularity, packaging, and interface dependencies.

Interfaces & Connectors

Provided Interface: Services offered by component. Represented by a **Lollipop** symbol (circle on line).

Required Interface: Services required from external components. Represented by a **Socket** symbol (half-circle on line).

Assembly Connector: Ball-and-socket connection linking provided and required interfaces directly.

GTU Architectural Rule

Component diagrams model **executable modules and software build units**, distinct from Class diagrams which model abstract domain object structures.

Summary & GTU Exam Practice Questions

Summary of UML Diagram Selection

Software Engineering Phase	Primary UML Diagram	Core Modeling Objective
Requirements Elicitation	Use Case Diagram	System scope & Actor boundaries
Domain Analysis	Class Diagram	Static entity structure & attributes
Dynamic System Modeling	Sequence Diagram	Time-ordered message passing
Workflow & Business Logic	Activity Diagram	Control flow & concurrency
Implementation Architecture	Component Diagram	Physical modules & interfaces

GTU Exam Practice Questions (DI04000011)

1. Differentiate between Aggregation and Composition with suitable UML notation and a real-world Software Engineering example. [7 Marks]
2. Draw a Use Case Diagram and Sequence Diagram for an Automated Teller Machine (ATM) Banking System. [7 Marks]

Lecture 27: Software Project Estimation & Scheduling

Course: GTU DI04000011 – Software Engineering
Unit: Software Project Estimation & Scheduling

Lecture Objectives

- Understand the pivotal role and core responsibilities of a **Software Project Manager (SPM)**.
- Master direct size estimation metrics: **Lines of Code (LOC)**, KLOC, and their practical applications.
- Analyze indirect size estimation metrics: **Function Point (FP)** analysis and its 5 core information domain characteristics.
- Compare LOC and FP metrics to select suitable size metrics for software projects.

Responsibilities of Software Project Manager (SPM)

Role of a Software Project Manager

A Software Project Manager (SPM) is responsible for planning, executing, monitoring, and closing software projects within time, budget, and quality constraints.

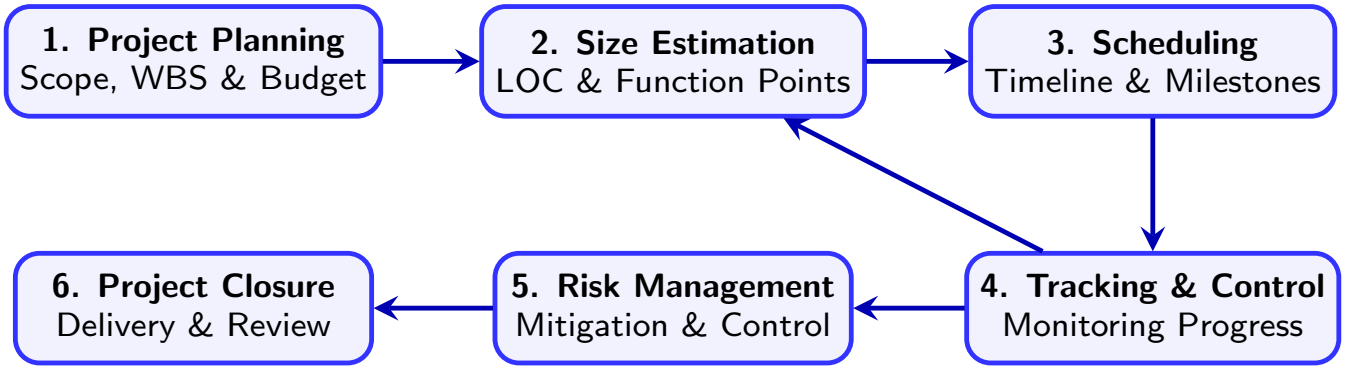
Core Managerial Responsibilities:

- **Project Planning:** Defining scope, objectives, project deliverables, and WBS.
- **Cost & Size Estimation:** Estimating required effort, duration, and resource cost.
- **Risk Management:** Identifying, analyzing, mitigating, and monitoring risks.

Core Operational Responsibilities:

- **Staffing & Leadership:** Recruiting, assigning tasks, and resolving team conflicts.
- **Tracking & Control:** Monitoring milestones via Gantt and PERT charts.
- **Client Interaction:** Managing stakeholder expectations and status reporting.

SPM Responsibilities in Project Management Lifecycle



Key Managerial Insight

Project estimation directly influences scheduling and tracking. Inaccurate size estimation propagates errors across the entire software project lifecycle.

Why Estimate Software Size?

Software size is the primary determinant of project cost, effort, duration, and staffing requirement. It forms the base input for estimation models such as COCOMO.

- **Direct Metrics:** Measure quantifiable physical properties of software code directly (e.g., Lines of Code - LOC, KLOC).
- **Indirect Metrics:** Measure functionality provided to the user rather than raw code volume (e.g., Function Points - FP, Object Points).

Challenges in Size Estimation

- Requirements uncertainty during the early planning phase.
- Technology and programming language dependency of direct metrics.
- Subjectivity in evaluating functional complexity.

Metrics for Size Estimation: Lines of Code (LOC)

Definition of LOC

Lines of Code (LOC) is a direct measure of software size calculated by counting the number of source code lines written in a program. Often expressed in KLOC (Kilo Lines of Code, 1 KLOC = 1000 LOC).

Standard LOC Counting Rules:

- **Included:** Executable source code statements, data definition statements, and declaration statements.
- **Excluded:** Blank lines, comment lines, auto-generated code, header boilerplate, and external library code.

Common Productivity Metrics Derived from LOC

$$\text{Productivity} = \frac{\text{KLOC}}{\text{Person-Months}}$$

$$\text{Quality/Defect Density} = \frac{\text{Defects}}{\text{KLOC}}$$

LOC Metric: Evaluation & Comparison

Table: Comprehensive Evaluation of Lines of Code (LOC) Metric

Aspect	Description / Characteristics
Advantages	• Simple, easy to calculate, and universally understood. • Direct measure of program size after coding is complete. • Widely supported in classic estimation models (COCOMO I).
Disadvantages	• Highly language-dependent (Assembly vs Python). • Penalizes expressive, concise programming languages. • Cannot be measured early in the requirements phase. • Encourages verbose, inefficient coding practices.
Best Suited For	Procedural languages, post-implementation code analysis.

Function Point (FP) Analysis: Fundamentals

Concept of Function Points

Introduced by Allan Albrecht at IBM (1979), Function Point Analysis measures software size based on the **functionality requested and delivered** to the end user, independent of implementation technology.

Key Objectives of FP Analysis:

- Measure user-facing functional requirements directly.
- Estimate size early in the Software Development Life Cycle (SDLC) during requirements specification.
- Provide a language-independent metric for comparing productivity across different technologies.

FP Formula

$$\text{FP} = \text{UFP} \times \text{VAF}$$

Where **UFP** = Unadjusted Function Points, and **VAF** = Value Adjustment Factor.

Five Information Domain Characteristics of FP

Function Point Analysis classifies system functionality into 5 parameters, each rated as **Low**, **Average**, or **High** complexity:

1. **External Inputs (EI):** Elementary user inputs that update internal system files or state (e.g., input forms, transactions).
2. **External Outputs (EO):** Processed data generated for user output (e.g., reports, dynamic visual displays, invoices).
3. **External Inquiries (EQ):** Interactive input/output combinations that query data without altering system state.
4. **Internal Logical Files (ILF):** Groups of logically related data maintained inside system boundary (e.g., database tables).
5. **External Interface Files (EIF):** Files maintained by external systems referenced for reading/lookup only.

Function Point Calculation Methodology

Step 1: Compute Unadjusted Function Points (UFP)

$$\text{UFP} = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} \cdot N_{ij}$$

Where w_{ij} is the weight associated with parameter i at complexity level j, and N_{ij} is the count.

Step 2: Calculate Value Adjustment Factor (VAF)

Evaluate 14 General System Characteristics (GSCs) (e.g., data communication, performance, reusability) on a scale of 0 (no influence) to 5 (strong influence):

$$\text{Degree of Influence (DI)} = \sum_{k=1}^{14} F_k$$

$$\text{VAF} = 0.65 + (0.01 \times \text{DI})$$

Note: VAF ranges between 0.65 and 1.35 ($\pm 35\%$ adjustment).

Comparative Analysis: LOC vs Function Points

Comparative Summary

- **Language Dependency:** LOC is technology-dependent; FP is language-independent.
- **SDLC Phase Availability:** LOC is available only post-implementation; FP can be calculated during requirements design phase.
- **Accuracy & Effort:** LOC is automated but inconsistent across tech stacks; FP requires subjective expert judgment but yields consistent functional sizing.

Lecture 27 Summary

- Software Project Managers must select accurate size metrics for successful project planning and scheduling.
- Combined usage (converting FP to LOC using language expansion ratios) bridges early estimation and code analysis.

Unit: Software Project Estimation & Scheduling

Software estimation is the process of predicting the effort, time, and cost required to build a software system based on measured or estimated size.

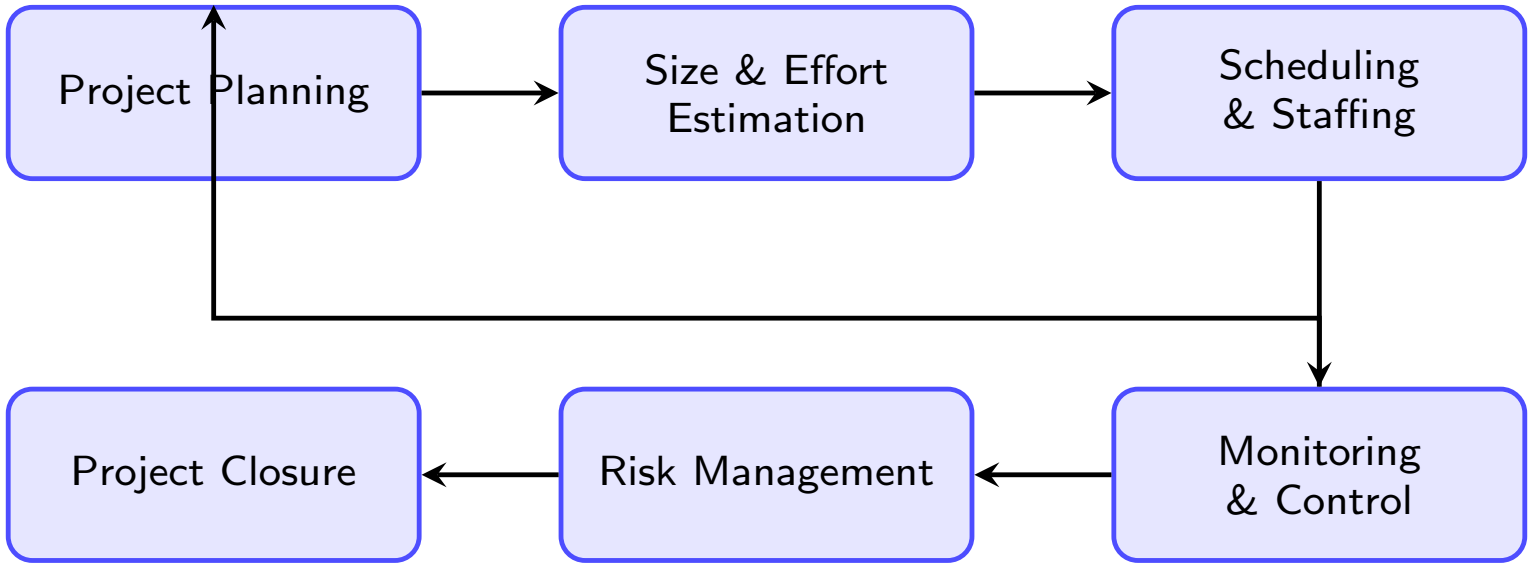
Key Topics Covered in Lecture 28:

- **Responsibilities of Software Project Manager (SPM):** Core managerial role, project control, and risk oversight.
- **Software Size Estimation Metrics:** Concept of software size measurement in Software Engineering.
- **Lines of Code (LOC):** Definition, application, advantages, and limitations.
- **Function Point (FP) Analysis:** Information domain parameters, weighting factors, VAF, and FP computation.
- **Comparative Analysis:** LOC vs. FP metrics for software size estimation.

Responsibilities of Software Project Manager (SPM)

- **Project Planning:** Defining scope, project goals, milestones, deliverables, project charter, and Work Breakdown Structure (WBS).
- **Effort & Cost Estimation:** Estimating required person-months, financial budget, and schedule timelines.
- **Resource Allocation:** Assigning software engineers, hardware resources, development tools, and infrastructure.
- **Risk Management:** Identifying technical, operational, and financial risks, evaluating probability/impact, and planning mitigation strategies.
- **Project Monitoring & Control:** Tracking progress against baseline schedules using Earned Value Analysis (EVA) or Gantt charts.
- **Team Leadership & Communication:** Facilitating stakeholder communication, inter-team coordination, and conflict resolution.

SPM Activity Lifecycle



Key Insight

Project management is an iterative control loop where estimation, risk monitoring, and tracking continuously refine project plans.

Why Measure Software Size?

Software size is the baseline parameter used to derive effort, duration, team size, cost, and defect density in Software Engineering.

Categories of Size Metrics:

1. **Direct Metrics (Physical Measurement):**
   - Measure tangible artifacts produced directly during coding.
   - Primary Metric: **Lines of Code (LOC)** / Kilo Lines of Code (KLOC).
2. **Indirect Metrics (Functional Measurement):**
   - Measure functionality delivered to the user independent of technology stack.
   - Primary Metric: **Function Point (FP)** Analysis.

Size Metric 1: Lines of Code (LOC)

Definition

LOC measures software size by counting the total number of executable source code lines written (typically expressed as KLOC $= 10^3$ lines of code).

Advantages:

- Simple and intuitive to measure post-implementation.
- Widely supported in classical estimation models (e.g., COCOMO).
- Easy to automate using source code counting tools.

Disadvantages:

- Programming language dependent (Assembly vs. Python).
- Cannot be measured early in Requirement Analysis phase.
- Penalizes efficient, concise, and re-usable code.
- Ignores non-coding efforts (design, testing, documentation).

Size Metric 2: Function Point (FP) Analysis

Concept (Allan Albrecht, IBM - 1979)

Function Point Analysis measures software size based on functional user requirements (FUR), making it completely independent of programming languages.

5 Information Domain Characteristics:

1. **External Inputs (EI):** Data entering the application boundary from users or external systems (e.g., input forms).
2. **External Outputs (EO):** Derived data generated by application for external consumption (e.g., reports, screens).
3. **External Inquiries (EQ):** Online input-output combinations resulting in immediate data retrieval.
4. **Internal Logical Files (ILF):** User-identifiable logical data stored inside system boundary.
5. **External Interface Files (EIF):** Data maintained by other systems referenced by the application.

Function Point Calculation Formula

Step 1: Compute Unadjusted Function Points (UFP)

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} (Count_{ij} \times Weight_{ij})$$

Weighting Factor Matrix for Information Domain:

Information Domain Parameter	Low	Average	High
External Inputs (EI)	3	4	6
External Outputs (EO)	4	5	7
External Inquiries (EQ)	3	4	6
Internal Logical Files (ILF)	7	10	15
External Interface Files (EIF)	5	7	10

Step 2: Calculate Value Adjustment Factor (VAF)

$$VAF = 0.65 + 0.01 \times \sum_{k=1}^{14} F_k$$

where F_k (scale 0–5) represents 14 General System Characteristics (GSC).
Step 3: Final Function Point: $FP = UFP \times VAF$

Comparison: LOC vs. Function Points

Parameter	Lines of Code (LOC)	Function Point (FP)
Metric Type	Direct physical measurement	Indirect functional measurement
Language Dependency	Highly dependent on programming language	Completely language independent
Estimation Phase	Late stage (requires code implementation or design)	Early stage (can be estimated from SRS / Requirements)
Focus	Developer perspective (code volume)	User perspective (functionality delivered)
Object-Oriented & Modern Frameworks	Poor suitability (code generators skew line count)	High suitability across modern software paradigms
Ease of Calculation	Easy after code implementation	Requires functional analysis of parameters

Numerical Example: FP Calculation

Problem Statement

A GTU Exam Management System has: 10 EIs (Low), 5 EOs (Average), 4 EQs (High), 2 ILFs (Average), and 1 EIF (Low). Total score for 14 GSC factors $\sum F_k = 35$. Calculate the final Function Point (FP).

Solution:

1. **Calculate UFP:**
   - $EI = 10 \times 3 = 30$
   - $EO = 5 \times 5 = 25$
   - $EQ = 4 \times 6 = 24$
   - $ILF = 2 \times 10 = 20$
   - $EIF = 1 \times 5 = 5$
   - **Total UFP** $= 30 + 25 + 24 + 20 + 5 = 104$
2. **Calculate VAF:** $VAF = 0.65 + 0.01 \times (35) = 0.65 + 0.35 = 1.00$
3. **Calculate Final FP:** $FP = UFP \times VAF = 104 \times 1.00 =$ **104 FP**

Summary & Review Questions

Summary

- Software Project Managers handle planning, sizing, scheduling, and risk mitigation.
- LOC is a direct metric, simple to count post-implementation but language dependent.
- Function Point (FP) is a language-independent indirect metric evaluated from 5 information domain characteristics and 14 GSCs.

GTU Examination Questions for Practice:

1. Explain the key responsibilities of a Software Project Manager during the software development life cycle. [7 Marks]
2. Differentiate between LOC and Function Point size estimation metrics. Explain the 5 information domain characteristics of FP analysis with a numerical example. [7 Marks]

Lecture Focus

This lecture explores project management principles, focusing on managerial responsibilities and primary software size estimation metrics essential for project scheduling and effort computation.

Key Learning Objectives

- Understand the core **responsibilities of a Software Project Manager (SPM)**.
- Analyze **Lines of Code (LOC)** as a direct software size metric, including its benefits and limitations.
- Master **Function Point (FP) Analysis**, its 5 fundamental functional types, and mathematical computation.
- Evaluate LOC vs. FP metrics for software project effort and cost estimation.

- **Project Planning & Scope Management:**
  - Defines project boundaries, objectives, deliverable milestones, and WBS (Work Breakdown Structure).
  - Prevents scope creep through formal change control processes.
- **Resource & Cost Estimation:**
  - Estimates effort (person-months), cost, hardware/software infrastructure, and staffing requirements.
- **Risk Management & Mitigation:**
  - Identifies technical, operational, and financial risks; formulates risk mitigation, monitoring, and contingency plans (RMMM).
- **Project Scheduling & Tracking:**
  - Develops Gantt charts/PERT networks, monitors critical paths, and tracks milestones.
- **Team Leadership & Stakeholder Communication:**
  - Manages inter-team dynamics, performance evaluation, and client status reporting.

Table: Managerial Responsibilities by SDLC Phase

SDLC Phase	Core SPM Responsibilities	Key Outputs / Artifacts
Inception	Feasibility analysis, stakeholder alignment, high-level scope definition	Project Charter, Feasibility Report
Planning	Size/effort estimation (LOC/FP), scheduling, team allocation, risk planning	Project Management Plan (PMP), Schedule
Execution	Task assignment, quality assurance oversight, configuration management	Sprint Backlog, Progress Reports
Monitoring	Variance analysis (EVMA), tracking cost/schedule performance metrics	Earned Value Charts, Risk Logs
Closure	Client sign-off, post-mortem analysis, organizational process asset updating	Project Audit, Retrospective Report

Concept of Size Estimation

Software size is the primary determinant of project effort, cost, duration, and team size. Estimation metrics are categorized into **Direct Metrics** (LOC) and **Indirect Metrics** (FP).

- **Lines of Code (LOC / KLOC):**
  - Measures software size by counting total delivered source lines of code excluding comments and blank lines.
  - Standardized unit: **KLOC** (Thousands of Lines of Code).
- **Common Derived Productivity Metrics:**
  - **Productivity:** $\text{Productivity} = \frac{\text{KLOC}}{\text{Person-Months}}$
  - **Quality:** $\text{Defect Density} = \frac{\text{Number of Defects}}{\text{KLOC}}$
  - **Cost Metric:** $\text{Cost per LOC} = \frac{\text{Total Cost}}{\text{LOC}}$

LOC Metric: Evaluation & Critical Issues

Strengths and Limitations in Estimation

Advantages of LOC

- **Simplicity:** Universally understood, easy to automate counting using standard tools.
- **Direct Metric:** Directly reflects artifact volume once source code is written.
- **Historical Data:** Widely supported in legacy empirical models like COCOMO.

Disadvantages of LOC

- **Language Dependence:** High-level languages (Python, Java) require fewer lines than Assembly/C for identical functionality.
- **Late Availability:** Cannot be accurately determined during early design/planning.
- **Disincentivizes Efficiency:** Penalizes concise, refactored code structure.

Function Point Analysis (FPA)

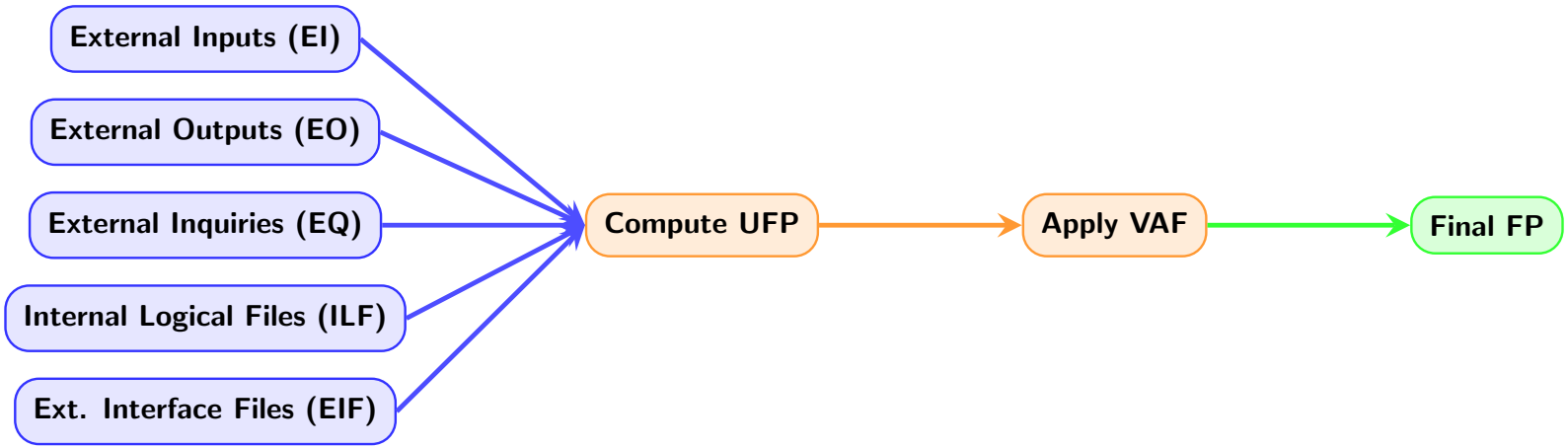
Developed by Allan Albrecht (IBM), Function Point (FP) is an **indirect measure** of software size based on user-visible functional requirements independent of programming language or technology stack.

Five Information Domain Components:

1. **External Inputs (EI):** User inputs that process data or control signals (e.g., input forms).
2. **External Outputs (EO):** Derived data outputs provided to the user (e.g., reports, messages).
3. **External Inquiries (EQ):** Interactive queries requiring an immediate data response without modifying database state.
4. **Internal Logical Files (ILF):** Logical groups of user-identifiable data maintained within application boundaries (e.g., database tables).
5. **External Interface Files (EIF):** Data files maintained by another application but referenced for data/control.

Function Point Computation Model

Functional Information Processing Pipeline



Functional Complexity Weighting

Each component is classified as **Low**, **Average**, or **High** complexity based on Data Element Types (DETs), Record Element Types (RETs), or File Types Referenced (FTRs).

Mathematical Formulation of Function Points

UFP, GSC, VAF, and Final FP Formulas

1. Unadjusted Function Point (UFP) Calculation

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} (W_{ij} \times Z_{ij})$$

- W_{ij}: Weight assigned to functional type i at complexity level j (Low, Avg, High).
- Z_{ij}: Count of functional units for type i at complexity level j.

2. Value Adjustment Factor (VAF)

Evaluates 14 General System Characteristics (GSCs) scored from 0 (No Influence) to 5 (Strong Influence):

$$VAF = 0.65 + 0.01 \times \sum_{k=1}^{14} F_k \quad \text{where } F_k \in [0, 5]$$

Note: VAF ranges strictly between **0.65** and **1.35** ($\pm 35\%$ adjustment).

3. Final Function Point (FP)

$$FP = UFP \times VAF$$

Table: Systematic Comparison of LOC and FP Metrics

Attribute	Line of Code (LOC)	Function Point (FP)
Metric Type	Direct physical measurement	Indirect functional measurement
Language Dependency	Highly dependent on language	Independent of technology stack
Availability Phase	Late SDLC (Post Coding/Implementation)	Early SDLC (Requirements/SRS phase)
User Viewpoint	Developer/Technical view	End-User functional view
Estimation Accuracy	Subject to high variance early on	Standardized (IFPUG guidelines)
Non-coding Effort	Does not measure documentation/design	Captures overall functional system scope

Worked Example

A system has 4 Low EIs ($4 \times 3 = 12$), 2 Average EOs ($2 \times 4 = 8$), 3 High EQs ($3 \times 6 = 18$), 2 Average ILFs ($2 \times 10 = 20$), and 1 Low EIF ($1 \times 7 = 7$).
Sum of 14 GSC factors $\sum F_k = 45$.

1. $UFP = 12 + 8 + 18 + 20 + 7 = \mathbf{65}$
2. $VAF = 0.65 + (0.01 \times 45) = \mathbf{1.10}$
3. $FP = 65 \times 1.10 = \mathbf{71.5}$ FPs

Summary Key Takeaways

- SPM oversees project scope, estimation, risks, schedule, and deliverables.
- LOC is a direct, code-centric metric with language dependency drawbacks.
- Function Points enable technology-independent size estimation early in SRS.

Constructive Cost Model (COCOMO)

- Developed by **Barry W. Boehm** in 1981 as an empirical algorithmic cost estimation model.
- Uses **KLOC** (Kilo Lines of Code) as the primary size metric to estimate project metrics.
- **Core Estimation Outputs**:
  - **Effort (E)**: Measured in Person-Months (PM).
  - **Development Time (D)**: Measured in Calendar Months (M).
  - **Staffing (P)**: Average number of persons required (E/D).
- **Hierarchy of COCOMO Models**:
  1. **Basic COCOMO**: Quick, static single-variable model.
  2. **Intermediate COCOMO**: Incorporates Cost Drivers via Effort Adjustment Factors (EAF).
  3. **Detailed COCOMO**: Multi-level phase-sensitive estimation (Module $\rightarrow$ Subsystem $\rightarrow$ System).

COCOMO Software Development Modes

Software projects are classified into three complexity modes based on domain clarity and team experience:

1. Organic Mode

- Small team working in a highly familiar, stable environment.
- Well-understood software requirements, flexible interface constraints.
- *Example*: Simple business systems, small utility tools, internal payroll.

2. Semi-detached Mode

- Medium-sized team with mixed levels of domain experience.
- Mix of rigid and flexible operational requirements.
- *Example*: Transaction processing system, DBMS design, compilers.

3. Embedded Mode

- Complex operational constraints, tight coupling with hardware/interfaces.
- Rigid operational rules and non-negotiable performance targets.
- *Example*: Real-time flight control system, medical care equipment.

Basic COCOMO Model: Mathematical Formulas

Mathematical Formulations for Basic COCOMO:

$$\text{Effort } (E) = a_b \times (\text{KLOC})^{b_b} \quad \text{[Person-Months (PM)]} \tag{1}$$

$$\text{Development Duration } (D) = c_b \times (E)^{d_b} \quad \text{[Months (M)]} \tag{2}$$

$$\text{Recommended Team Size } (P) = \frac{E}{D} \quad \text{[Persons]} \tag{3}$$

Key Parameters

- **KLOC**: Size of the software project in thousands of delivered source code lines.
- a_b, b_b: Effort coefficients reflecting project complexity mode.
- c_b, d_b: Schedule duration coefficients reflecting project mode.
- 1 Person-Month = 152 working hours (21.5 working days).

Basic COCOMO Coefficient Matrix

Basic COCOMO Model Parameters & Characteristics Table:

Development Mode	**Project Characteristics**	a_b	b_b	c_b	d_b
Organic	Small size, relaxed requirements, familiar environment	2.4	1.05	2.5	0.38
Semi-detached	Medium size, mixed constraints, moderate complexity	3.0	1.12	2.5	0.35
Embedded	Large size, tight hardware/software interface constraints	3.6	1.20	2.5	0.32

Table: Basic COCOMO Model Coefficients Matrix

- Note: The exponent coefficient $b_b > 1.0$ accounts for super-linear scaling of communication overhead in larger software systems.

Intermediate COCOMO Model & Cost Drivers

Intermediate COCOMO Formula:

$$E = a_i \times (\text{KLOC})^{b_i} \times \text{EAF} \tag{4}$$

Where EAF is the **Effort Adjustment Factor**, computed as $\prod_{i=1}^{15}$ Cost Driver$_i$.
15 Cost Drivers across 4 Categories:

1. **Product Attributes**: Software reliability (RELY), Database size (DATA), Product complexity (CPLX).
2. **Hardware Attributes**: Execution time constraint (TIME), Main storage constraint (STOR), Virtual machine volatility (VIRT), Computer turn-around time (TURN).
3. **Personnel Attributes**: Analyst capability (ACAP), Application experience (AEXP), Programmer capability (PCAP), VMS experience (VEXP), Programming language experience (LEXP).
4. **Project Attributes**: Modern programming practices (MODP), Use of software tools (TOOL), Required development schedule (SCED).

Detailed (Complete) COCOMO Model

Characteristics of Detailed COCOMO:

- Accounts for variations in cost driver impacts across different software development phases.
- Applies estimations hierarchically to 3 levels:
  - **Module Level**: Lowest software components.
  - **Subsystem Level**: Aggregation of related modules.
  - **System Level**: Total integrated software project.

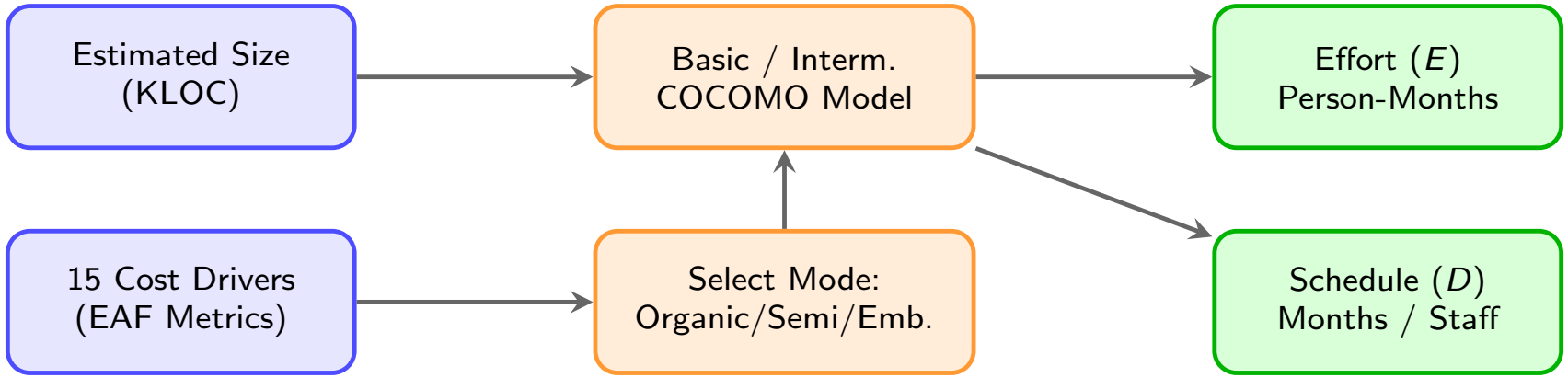
Phase Distribution of Effort

Detailed COCOMO evaluates effort allocation across 5 phases:

1. Requirements Analysis & System Architecture Planning
2. System Design & Detailed Component Specification
3. Module Coding, Unit Testing & Documentation
4. Integration Testing & Subsystem Verification
5. Final System Acceptance Testing & Deployment

TikZ Architecture: COCOMO Estimation Process

COCOMO Workflow & Architectural Pipeline:



Numerical Worked Example: COCOMO Estimation

Problem Statement: Estimate Effort (E), Development Time (D), and Average Staffing (P) for a **Semi-detached** software project estimated at **32 KLOC**.
Given Parameters for Semi-detached Mode:
$a_b = 3.0, \quad b_b = 1.12, \quad c_b = 2.5, \quad d_b = 0.35$

Step-by-Step Solution

1. **Calculate Effort (E)**:

$$E = a_b \times (\text{KLOC})^{b_b} = 3.0 \times (32)^{1.12} \approx 3.0 \times 48.44 = \mathbf{145.32} \text{ Person-Months}$$

2. **Calculate Development Time (D)**:

$$D = c_b \times (E)^{d_b} = 2.5 \times (145.32)^{0.35} \approx 2.5 \times 5.71 = \mathbf{14.28} \text{ Months}$$

3. **Calculate Average Staff Size (P)**:

$$P = \frac{E}{D} = \frac{145.32}{14.28} \approx \mathbf{10.18} \text{ Persons} \approx \mathbf{10} \text{ Engineers}$$

Evolution to COCOMO II

Transition from COCOMO 81 to COCOMO II: To handle modern software development paradigms (Object-Oriented, Agile, Component-Based Software Engineering), COCOMO II was introduced.

Three Sub-models in COCOMO II

1. **Application Composition Model**: Used during early prototyping stage. Uses **Object Points** as the primary metric.
2. **Early Design Model**: Used during system architecture phase when detailed requirements are emerging. Uses **Unadjusted Function Points (UFP)**.
3. **Post-Architecture Model**: Used during system construction. Uses **SLOC** or **Function Points** with 17 cost drivers and 5 scale factors.

- **5 Scale Factors** in COCOMO II: Precedentedness (PREC), Development Flexibility (FLEX), Architecture/Risk Resolution (RESL), Team Cohesion (TEAM), Process Maturity (PMAT).

Limitations & Summary of COCOMO

Key Advantages

- Mathematically rigorous empirical model.
- Industry standard benchmark for software cost estimation.
- Clear classification into development modes.

Limitations

- Highly sensitive to early KLOC inaccuracies.
- Ignores customer communication quality.
- Cost driver assignments can be subjective.

GTU Exam Summary Note

In GTU Software Engineering exams:

- Remember formulas for Basic COCOMO: $E = a_b(\text{KLOC})^{b_b}$, $D = c_b(E)^{d_b}$.
- State distinctions between Organic, Semi-detached, and Embedded modes clearly.
- List the 4 categories of Intermediate COCOMO Cost Drivers.

Software Project Estimation & COCOMO Overview

Software Project Estimation

Software project estimation is the process of predicting the effort, time, and cost required to design, construct, and test a software system. Accurate estimation prevents budget overruns, unrealistic scheduling, and poor software quality.

What is the COCOMO Model?

- **COCOMO** stands for **COnstructive COst MOdel**.
- Developed by **Dr. Barry W. Boehm** in 1981, derived from the statistical analysis of 63 historical software projects.
- It is an empirical algorithmic cost model that uses **KLOC** (Kilo Lines of Code) as the fundamental sizing metric.

Hierarchy of COCOMO Models

- **Basic COCOMO**: Static single-valued model for fast, macro-level estimation.
- **Intermediate COCOMO**: Extends basic model by introducing 15 Cost Drivers.
- **Detailed COCOMO**: Incorporates phase-sensitive multipliers for individual modules.

Classification of Software Development Modes

Software projects are categorized into three development modes based on complexity, team size, and domain familiarity:

Organic Mode Small teams working in a highly familiar environment with well-understood, flexible software requirements.

- Example: Internal payroll system, inventory tracker, simple utility scripts.
- Typical size: < 50 KLOC.

Semi-Detached Mode Intermediate team size with mixed software engineering experience. Requirements consist of a blend of rigid and flexible operational constraints.

- Example: Database management systems (DBMS), compilers, transaction processors.
- Typical size: $50 - 300$ KLOC.

Embedded Mode Software coupled tightly with complex hardware, strict interfaces, and rigid operational regulations.

- Example: Air traffic control system, missile guidance systems, medical equipment.

Basic COCOMO Estimating Formulas

Core Estimation Equations

$$\text{Effort } (E) = a \times (\text{KLOC})^b \quad \text{[Person-Months (PM)]}$$

$$\text{Development Time } (D) = c \times (E)^d \quad \text{[Months (M)]}$$

$$\text{Persons Required } (P) = \frac{E}{D} \quad \text{[Persons]}$$

Meaning of Variables

- **KLOC**: Estimated code size in Thousands of Delivered Source Instructions (KDSI).
- **Person-Month (PM)**: Units of effort equal to 152 working hours of engineering effort per month.
- **a, b, c, d**: Empirical constants specific to each software development mode.

Empirical Constants & Project Mode Characteristics

Table: Basic COCOMO Empirical Coefficients Table

Software Mode	a	b	c	d	Team & System Nature
Organic	2.4	1.05	2.5	0.38	Small team, familiar, relaxed deadlines
Semi-Detached	3.0	1.12	2.5	0.35	Medium team, mixed experience
Embedded	3.6	1.20	2.5	0.32	Hard constraints, complex interfaces

Key Insights

- The exponent $b > 1.0$ captures **diseconomy of scale**: as lines of code double, communication overhead increases, requiring more than double the effort.
- Embedded mode has the highest exponent ($b = 1.20$) due to heavy verification and hardware integration.

Intermediate COCOMO & Effort Adjustment Factor (EAF)

Basic COCOMO relies solely on code size. Intermediate COCOMO refines effort using 15 Cost Drivers across 4 software domains.

Intermediate Effort Equation

$$E = a_i \times (\text{KLOC})^{b_i} \times \text{EAF}$$

where $\text{EAF} = \prod_{k=1}^{15} F_k$ is the product of all 15 Effort Multipliers (cost drivers).

15 Cost Driver Categories

1. **Product Attributes**: Software reliability, Database size, Product complexity.
2. **Hardware Attributes**: Execution time constraint, Main storage constraint, Platform volatility.
3. **Personnel Attributes**: Analyst capability, SE capability, Programming language experience.
4. **Project Attributes**: Modern practices, Software tool usage, Development schedule constraints.

Phase-Sensitive Effort Allocation

Concept of Detailed COCOMO

Detailed COCOMO extends Intermediate COCOMO by recognizing that cost drivers affect different phases of the Software Development Life Cycle (SDLC) unequally.

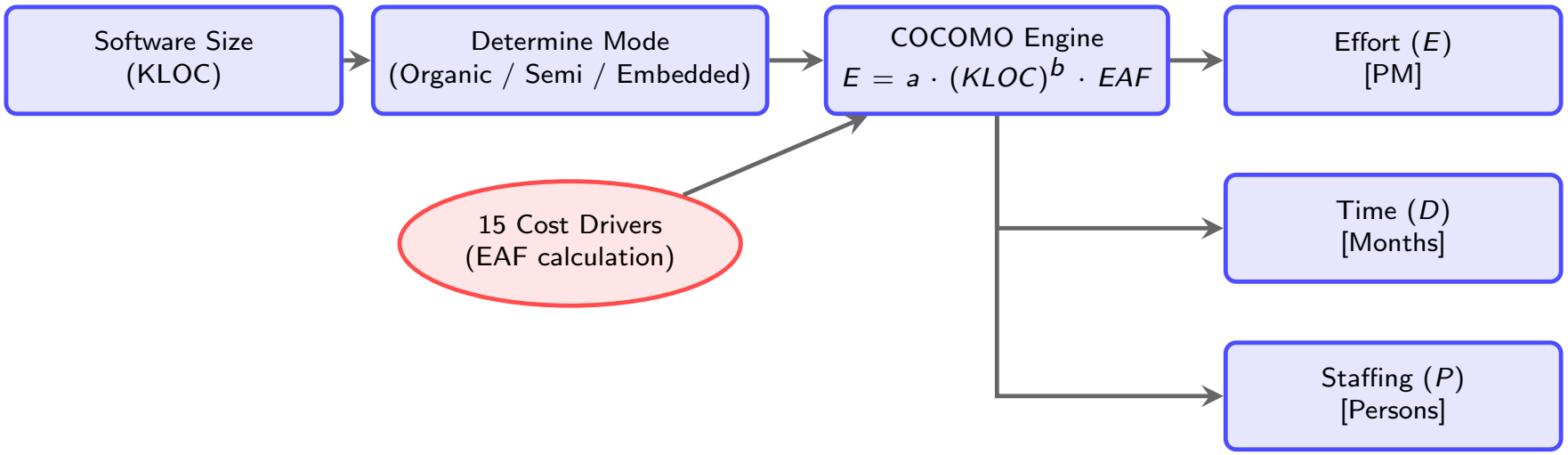
Three-Level Architecture

- **Module Level**: Estimates effort for individual source code modules.
- **Subsystem Level**: Aggregates module estimates into functional subsystems.
- **System Level**: Combines subsystems to form the integrated final system.

Typical SDLC Phase Effort Distribution

- **System Architecture & Planning**: $\sim 6\% - 8\%$ of total effort.
- **Detailed Design**: $\sim 16\% - 18\%$ of total effort.
- **Coding & Unit Testing**: $\sim 48\% - 52\%$ of total effort.
- **Integration & System Testing**: $\sim 16\% - 24\%$ of total effort.

TikZ Structural Flow of COCOMO Estimation Process



Step-by-Step Calculation Example

Problem Statement

A software team is building an **Organic mode** system estimated at **32 KLOC**. Calculate the total Effort (E), Development Time (D), and Average Staff Size (P).

Given Data & Constants

Mode = Organic ($a = 2.4, b = 1.05, c = 2.5, d = 0.38$), KLOC = 32.

Step-by-Step Solution

1. **Calculate Effort (E):**

$$E = 2.4 \times (32)^{1.05} = 2.4 \times 37.89 \approx \mathbf{90.93} \text{ Person-Months (PM)}$$

2. **Calculate Development Time (D):**

$$D = 2.5 \times (90.93)^{0.38} = 2.5 \times 5.55 \approx \mathbf{13.88} \text{ Months}$$

3. **Calculate Staffing Level (P):**

$$P = \frac{E}{D} = \frac{90.93}{13.88} \approx \mathbf{6.55} \rightarrow \mathbf{7} \text{ Software Engineers}$$

Modernizing Cost Estimation - COCOMO II Framework

Why COCOMO II?

COCOMO 81 assumed traditional waterfall lifecycle and greenfield development. Modern software development heavily emphasizes reuse, object-oriented paradigms, dynamic component integration, and Agile practices.

Three Sub-Models of COCOMO II

1. **Application Composition Model**: Used in early prototyping phase; relies on Object Points (screens, reports, components).
2. **Early Design Model**: Used when requirements are stabilized but system architecture is uncommitted; uses Unadjusted Function Points (UFP).
3. **Post-Architecture Model**: Used during main development phase; incorporates 17 cost drivers and 5 scale factors.

Lecture 31 Summary & Takeaways

Comparative Overview of COCOMO Levels

- **Basic COCOMO**: Quick, macro-level estimation based strictly on size (KLOC).
- **Intermediate COCOMO**: Refines estimates using 15 Effort Adjustment Factor (EAF) cost drivers.
- **Detailed COCOMO**: Applies phase-wise cost drivers to individual system modules.
- **COCOMO II**: Modernized framework catering to object-oriented, component-based software development.

Best Practices for Project Managers

- Never rely on a single algorithmic model; cross-validate COCOMO results with Function Point Analysis (FPA) and Expert Judgment.
- Re-estimate project effort continuously as software requirements mature throughout the SDLC.

Lecture 32: Introduction to COCOMO Model

GTU Course: DI04000011 - Software Engineering
Unit: Software Project Estimation & Scheduling

- **Overview**: The **COCOMO** (*Constructive Cost Model*) is an algorithmic software cost estimation model developed by Barry Boehm in 1981.
- **Primary Input Metric**: Size of the software product measured in **KLOC** (*Kilo Lines of Code*).
- **Key Objectives**:
  - Predict total effort required in **Person-Months (PM)**.
  - Predict total calendar duration in **Months (M)**.
  - Estimate average team staffing size and total financial cost.
- **Three Hierarchical Levels**:
  1. **Basic COCOMO**: Static single-variable model for quick estimates.
  2. **Intermediate COCOMO**: Incorporates 15 Effort Adjustment Factors (Cost Drivers).
  3. **Detailed COCOMO**: Incorporates phase-wise and subsystem-level cost drivers.

Software Development Modes in COCOMO

Boehm categorized software projects into **three software development modes** based on complexity and environment rigidity:

- **Organic Mode**:
  - Small, experienced teams working in familiar in-house environments.
  - Well-understood software requirements, high flexibility.
  - Example: Standard payroll systems, business data processing tools.
- **Semi-detached Mode**:
  - Intermediate team size with mixed levels of domain experience.
  - Mix of rigid and flexible requirements with moderate constraints.
  - Example: Database management systems, compilers, inventory systems.
- **Embedded Mode**:
  - Tight hardware, software, and operational constraints.
  - High degree of innovation required; software strongly coupled with target hardware.
  - Example: Real-time air traffic control, missile guidance, avionics systems.

Basic COCOMO Model: Mathematical Formulas

Basic COCOMO provides quick, order-of-magnitude estimates using power-law equations based solely on size (KLOC):

1. Effort Equation

$$E = a_b \times (\text{KLOC})^{b_b} \quad \text{[Person-Months (PM)]}$$

2. Development Time Equation

$$D = c_b \times (E)^{d_b} \quad \text{[Months]}$$

3. Staffing & Productivity Metrics

$$\text{Average Staff Size (SS)} = \frac{E}{D} \quad \text{[Persons]}$$

$$\text{Productivity (P)} = \frac{\text{KLOC}}{E} \quad \text{[KLOC / PM]}$$

Where a_b, b_b, c_b, d_b are mode-specific empirical constants.

Basic COCOMO Constants & Mode Comparison

Empirically derived coefficients for the Basic COCOMO model:

Table: Basic COCOMO Empirical Coefficients

Software Mode	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

- **Observations**:
  - $b_b > 1.0$ reflects non-linear scaling penalty due to communication overhead.
  - Embedded systems experience highest effort scale exponent ($b_b = 1.20$).
  - Organic projects have lower initial coefficient ($a_b = 2.4$) and closer to linear scaling ($b_b = 1.05$).

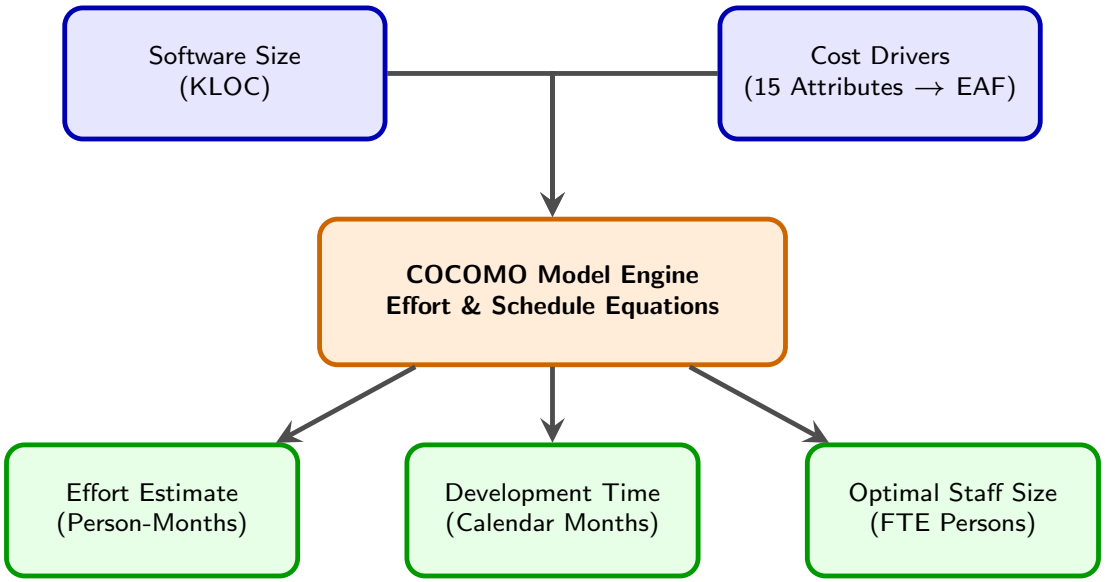
Intermediate COCOMO & Cost Drivers

Intermediate COCOMO adjusts effort using an **Effort Adjustment Factor (EAF)** derived from 15 Cost Drivers:

$$E = a_i \times (\text{KLOC})^{b_i} \times \text{EAF} \quad \text{where EAF} = \prod_{k=1}^{15} F_k$$

1. **Product Attributes**:
   - RELY (Required Reliability)
   - DATA (Database Size)
   - CPLX (Product Complexity)
2. **Hardware Attributes**:
   - TIME (Execution Time Constraint)
   - STOR (Main Storage Constraint)
   - VIRT (Virtual Machine Volatility)
   - TURN (Memory Turnaround Time)
3. **Personnel Attributes**:
   - ACAP (Analyst Capability)
   - AEXP (App. Experience)
   - PCAP (Programmer Capability)
   - VEXP (VM Experience)
   - LEXP (Language Experience)
4. **Project Attributes**:
   - MODP (Modern Prog. Practices)
   - TOOL (Use of SW Tools)
   - SCED (Required Schedule)

COCOMO Architectural Flow



- **Process Workflow**: Software size (KLOC) and domain parameters pass into the COCOMO engine to compute total effort, schedule duration, and recommended staffing.

Detailed COCOMO Model

Detailed (Advanced) COCOMO extends Intermediate COCOMO by accounting for phase-wise effort distribution and component-level variations.

- **Phase-Wise Decomposition**: Effort is distributed across major software development lifecycle (SDLC) phases:
  1. Requirements Analysis & System Design
  2. Detailed Structural Design
  3. Coding & Unit Testing
  4. Integration & System Testing
- **Multi-Level Cost Drivers**:
  - Cost drivers are evaluated at **subsystem** or **module level** rather than applying globally to the entire project.
  - Enables accurate estimation for large heterogeneous software systems combining real-time, database, and UI modules.

Numerical Problem Walkthrough: Basic COCOMO

Problem Statement: Estimate effort, duration, and staffing for a software project of size **32 KLOC** developed in **Semi-detached Mode**.

- **Step 1: Identify Parameters**: For Semi-detached mode: $a_b = 3.0$, $b_b = 1.12$, $c_b = 2.5$, $d_b = 0.35$.
- **Step 2: Calculate Effort (E)**:

$$E = 3.0 \times (32)^{1.12} = 3.0 \times 48.50 \approx \mathbf{145.5} \text{ Person-Months}$$

- **Step 3: Calculate Development Time (D)**:

$$D = 2.5 \times (145.5)^{0.35} = 2.5 \times 5.71 \approx \mathbf{14.28} \text{ Months}$$

- **Step 4: Calculate Average Staffing Size (SS)**:

$$SS = \frac{E}{D} = \frac{145.5}{14.28} \approx \mathbf{10.19} \text{ Persons } (\approx \mathbf{10} \text{ full-time engineers})$$

Modern Evolution: COCOMO II

COCOMO II (Boehm et al., 2000) addresses modern software engineering practices (Agile, OOD, Reuse, COTS integration):

- **Three Lifecycle Sub-Models**:
  1. **Application Composition Model**: Used during early prototyping; based on Object Points.
  2. **Early Design Model**: Used when requirements are exploring architecture; uses Function Points / Unadjusted KLOC.
  3. **Post-Architecture Model**: Used during actual development; uses KLOC/Function Points with 17 cost drivers and 5 scale factors.
- **Exponent Modeling via Scale Factors**: Exponent b is not static; it is calculated dynamically using 5 Scale Factors (SF_i):

$$b = 0.91 + 0.01 \sum_{i=1}^{5} SF_i$$

Scale factors include Precedentedness, Development Flexibility, Architecture/Risk Resolution, Team Cohesion, Process Maturity.

Comparative Summary & Practical Guidelines

Table: Comparison of COCOMO Models

Model	Primary Input	Key Characteristics & Use Case
Basic	KLOC	Quick estimate, single variable, small/early projects
Intermed.	KLOC + 15 EAF	Account for team skill, hardware, product complexi
Detailed	Subsystem KLOC	Phase-wise decomposition, module-level estimation
COCOMO II	KLOC / FP / OP	Supports Agile, component reuse, 5 Scale Factors.

GTU Exam Key Takeaway

Always verify project mode (*Organic*, *Semi-detached*, *Embedded*) and state correct formula constants (a_b, b_b, c_b, d_b) when solving estimation problems in GTU examinations.

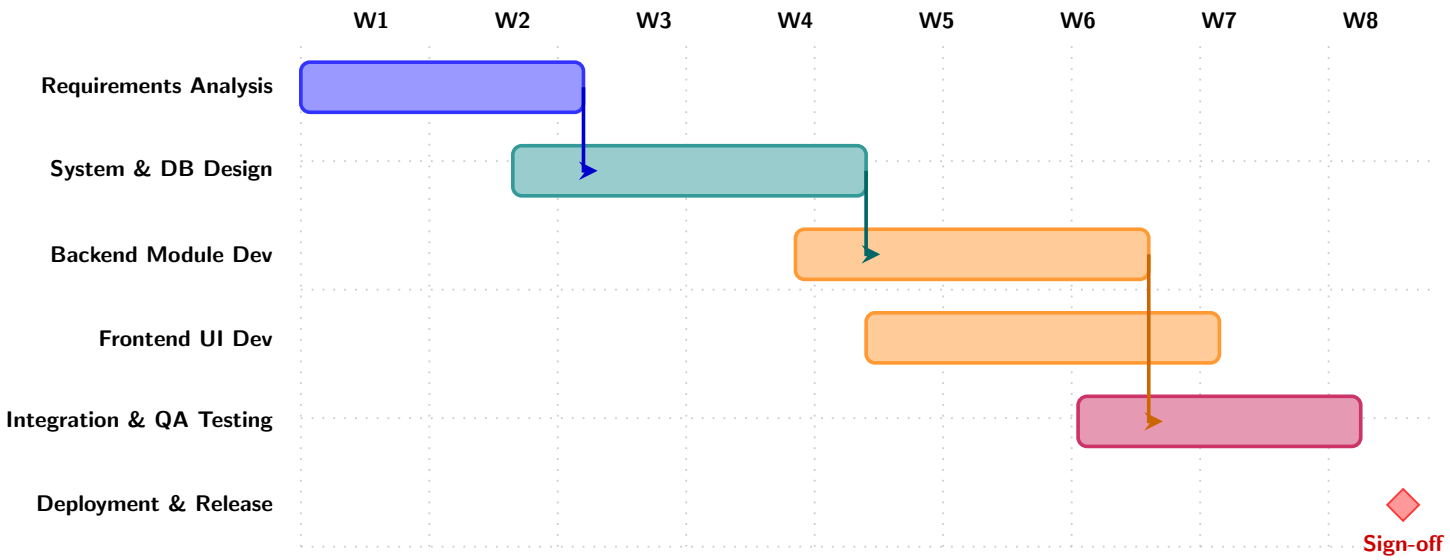
Introduction to Software Project Scheduling

- **Definition**: Software project scheduling involves decomposing the total engineering effort into discrete tasks, establishing dependency networks, and mapping work packages against calendar time.
- **Importance in Software Engineering**:
  - Mitigates schedule slippage and cost overruns during the software development lifecycle (SDLC).
  - Enables optimal resource allocation (engineers, QA testers, cloud build environments).
  - Establishes a baseline for measuring project progress against target milestones.
- **Core Scheduling Principles**:
  - **Work Breakdown Structure (WBS)**: Hierarchical decomposition of system deliverables.
  - **Task Interdependency**: Identification of sequential, parallel, and coupled activities.
  - **Effort Allocation**: Assigning staff-days/hours based on estimation models (e.g., COCOMO, Function Points).
  - **Milestone Definition**: Defining verifiable checkpoints (e.g., SRS Sign-off, Architecture Freeze).

Gantt Charts in Software Engineering

- **Overview**: A Gantt Chart is a bar chart representation of a software project schedule displaying task timelines, durations, and resource assignments over time.
- **Key Structural Components**:
  - **Horizontal Axis**: Time scale (Divided into Days, Weeks, or Sprints).
  - **Vertical Axis**: List of WBS software engineering tasks.
  - **Horizontal Bars**: Represent task start time, duration, and completion percentage.
  - **Milestone Markers**: Diamond symbols representing key completion events.
  - **Dependency Links**: Directed lines showing finish-to-start relationships.
- **Advantages & Limitations**:
  - **Advantages**: Highly intuitive visual summary of overall project status and overlapping phases.
  - **Limitations**: Struggles to visualize complex multi-branch logic compared to PERT/CPM diagrams.

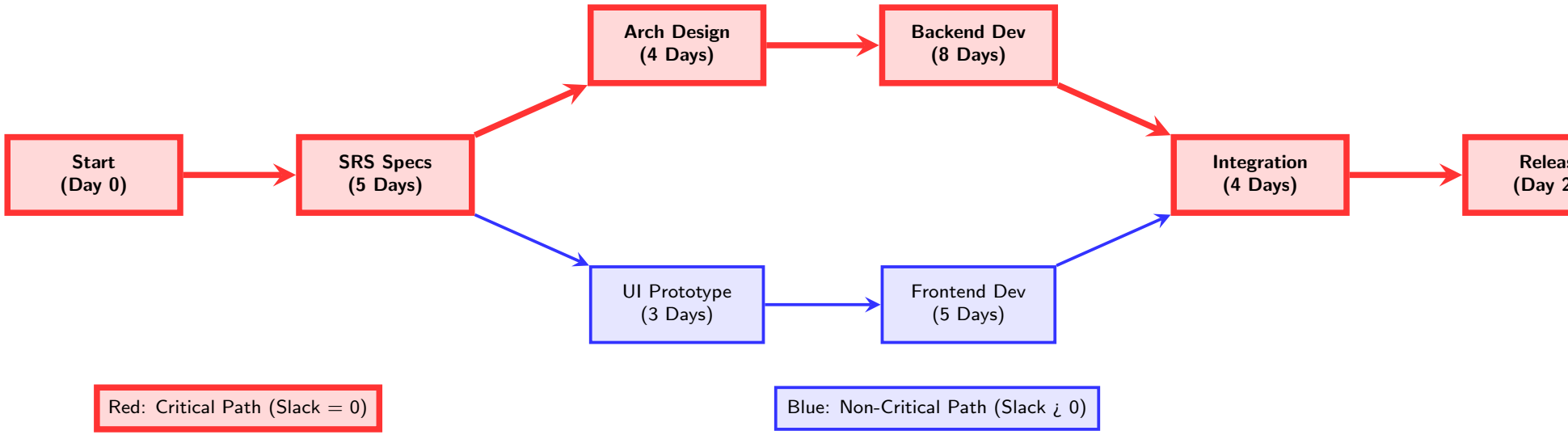
TikZ Diagram: Software Project Gantt Chart



Flow Charts & Network Diagrams in Scheduling

- **Role of Flow Charts in Software Scheduling**:
  - Maps execution sequences, branching logic, and concurrent engineering tasks.
  - Forms the foundation of PERT (Program Evaluation and Review Technique) and CPM (Critical Path Method).
- **Software Task Dependency Types**:
  - **Finish-to-Start (FS)**: Database Schema design must finish before ORM coding starts.
  - **Start-to-Start (SS)**: Backend API dev and API documentation start simultaneously.
  - **Finish-to-Finish (FF)**: System Testing finishes when Bug Verification finishes.
- **Critical Path Analysis (CPM)**:
  - The longest path of dependent activities determining minimum total project duration.
  - Tasks on the critical path have **Zero Float / Slack Time**.

TikZ Diagram: Activity Network Flowchart (PERT/CPM)



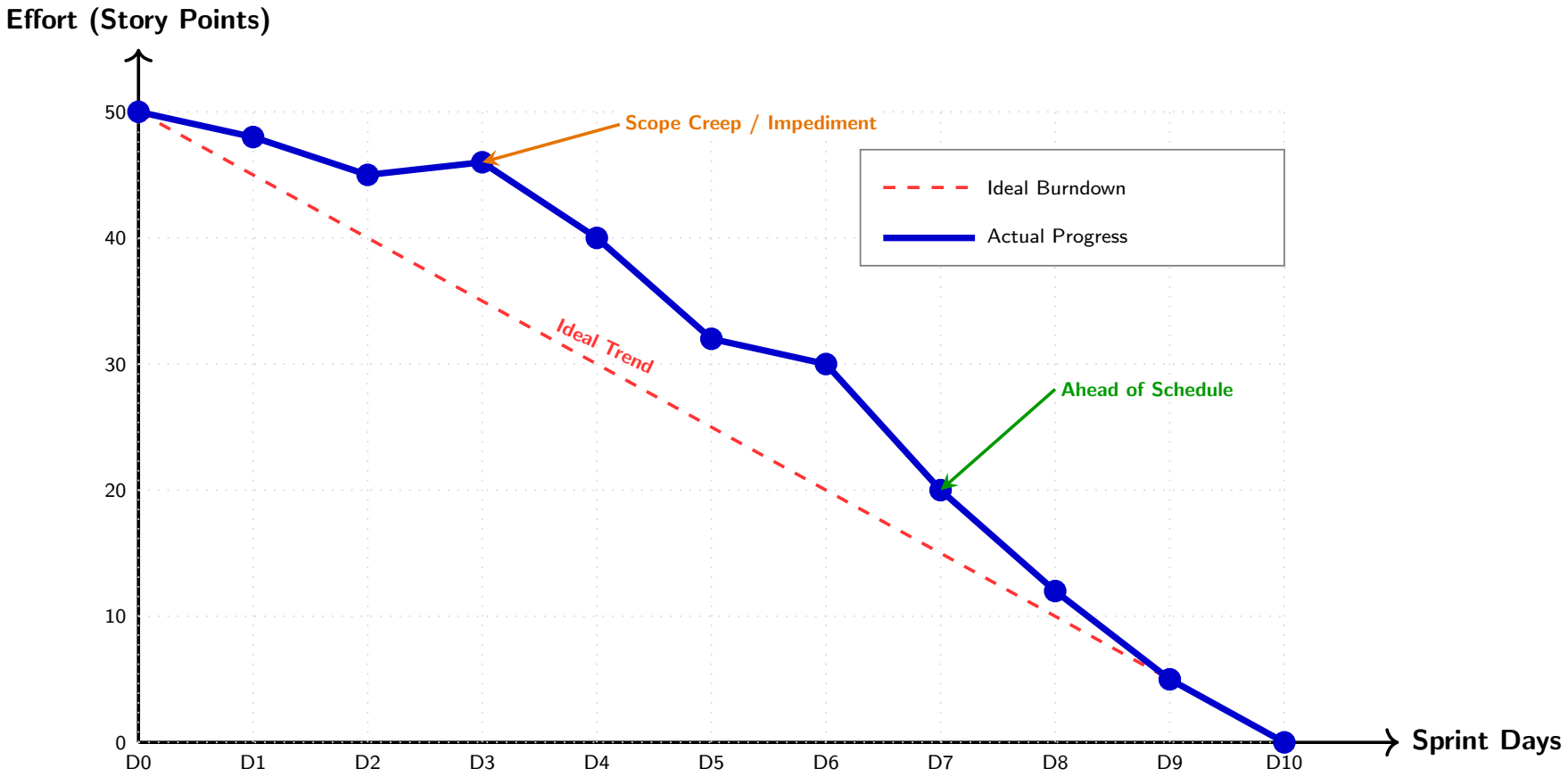
Comparison of Project Scheduling Techniques

Feature / Metric	**Gantt Chart**	**Network Flowchart (PERT/CPM)**	**Sprint Burndown Chart**
Primary Focus	Timeline & Resource Allocation	Task Dependencies & Critical Path	Daily Work Remaining in Iteration
Target Methodology	Waterfall / Hybrid Models	Plan-Driven Systems	Agile / Scrum Frameworks
Time Horizon	Macro Level (Months/Quarters)	Entire Project Lifecycle	Micro Level (1–4 Week Sprints)
Dependency Visual	Low–Moderate (Bar Overlaps)	High (Explicit Directed Graph)	None (Aggregated Metrics)
Metric Tracked	Calendar Start/Finish Dates	Task Slack & Total Duration	Remaining Story Points / Hours
Adaptability	Moderate	Low (Recalculation Required)	High (Updated Daily)

Agile Estimation & Sprint Burndown Charts

- **Context in Agile Software Engineering**:
  - Agile substitutes monolithic long-term schedules with short, fixed-length iterations called **Sprints**.
  - User Stories are estimated in relative units called **Story Points (SP)** or **Ideal Hours**.
- **Definition of Sprint Burndown Chart**:
  - A graphical tool displaying the rate at which an Agile team burns through work (remaining story points) during a Sprint.
- **Key Elements**:
  - **X-Axis**: Working days of the Sprint (e.g., Day 1 to Day 10).
  - **Y-Axis**: Total committed effort (Story Points or Task Hours).
  - **Ideal Burndown Line**: Straight line from total commitment down to 0 at Sprint end.
  - **Actual Burndown Line**: Real-time daily plot of actual effort remaining.

TikZ Diagram: Agile Sprint Burndown Chart



Analyzing Sprint Burndown Chart Patterns

- **Diagnostic Patterns in Burndown Curves**:
  - **Curve Above Ideal Line**: Team is behind schedule; potential risk of incomplete commitment.
  - **Curve Below Ideal Line**: Team is burning points faster than planned; can pull in extra backlog items.
  - **Horizontal Plateau**: Blocked stories due to external dependencies, technical blockers, or slow QA.
  - **Upward Spike**: Mid-sprint scope creep (adding new user stories) or re-estimation of task complexity.
  - **End-of-Sprint Cliff**: Work done in bulk at the end; indicates delayed testing or large un-decomposed stories.
- **Agile Velocity Link**:
  - **Velocity** = Average Story Points completed per Sprint.
  - Historical velocity informs future sprint capacity planning and project release dates.

Summary & Review Questions

- **Summary**:
  - Gantt Charts offer timeline and resource views suited for high-level plan-driven scheduling.
  - Network Flow Charts (PERT/CPM) define precise task interdependencies and critical execution paths.
  - Sprint Burndown Charts provide daily progress transparency and early risk detection in Agile teams.
- **GTU Examination Review Questions**:
  1. Differentiate between Gantt Chart and PERT Network Diagram in software project scheduling. (4 Marks)
  2. Explain Critical Path Method (CPM) with a neat activity network flowchart example. (7 Marks)
  3. Draw a Sprint Burndown Chart and explain how scope creep and blocked tasks are identified. (7 Marks)

Lecture 34: Project Scheduling Overview

Course: GTU DI04000011 – Software Engineering
Unit: Software Project Estimation & Scheduling
Topic: Project Scheduling – Gantt Chart, Flow Chart, Sprint Burndown Chart

Learning Objectives

- Understand the core principles and necessity of project scheduling in software development.
- Learn how to construct and interpret **Gantt Charts** for timeline tracking and task dependencies.
- Analyze project workflows and critical paths using **Flow Charts** and activity networks (PERT/CPM).
- Master **Sprint Burndown Charts** for agile iteration control, velocity measurement, and scope monitoring.
- Evaluate structural trade-offs between traditional and agile project scheduling models.

Fundamentals of Software Project Scheduling

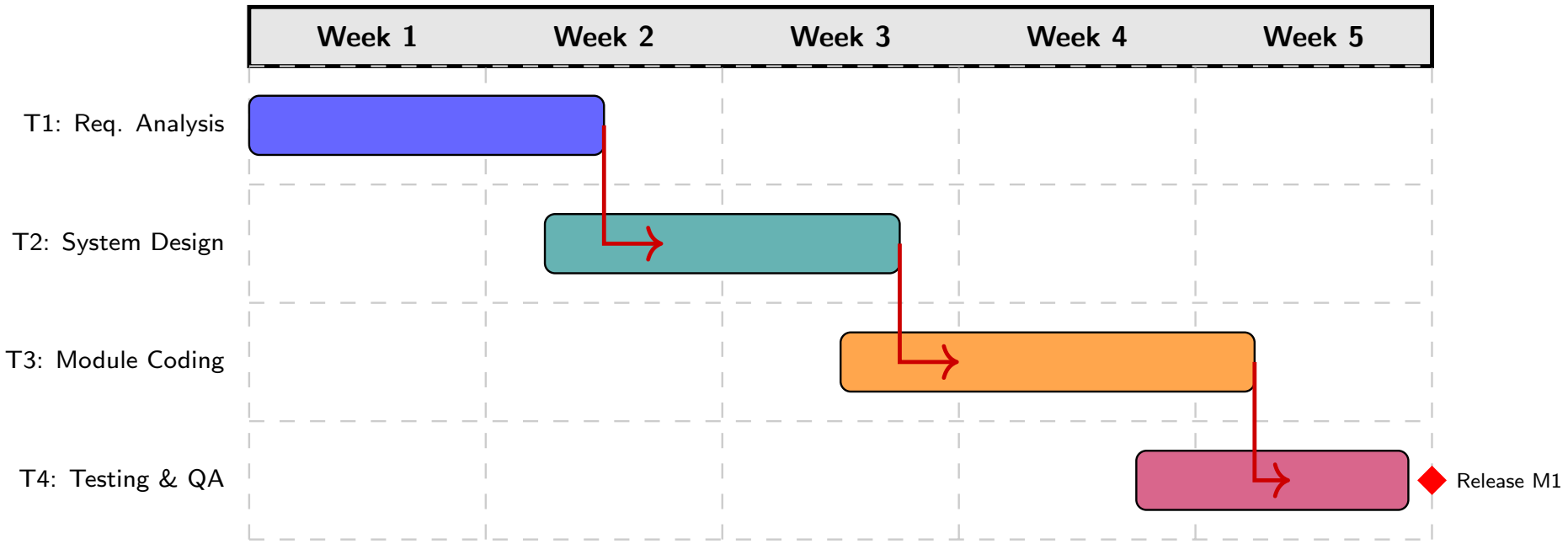
- **Definition**: Software project scheduling involves dividing the project into discrete tasks, establishing dependencies, allocating resources, and assigning timeframes to ensure timely delivery.
- **Work Breakdown Structure (WBS)**:
  - Hierarchical decomposition of project deliverables into manageable work packages.
  - Serves as the foundational input for creating task schedules and effort allocations.
- **Key Principles of Effective Scheduling**:
  - **Compartmentalization**: Breaking work into distinct, measurable activities.
  - **Interdependency Identification**: Mapping relationships (Finish-to-Start, Start-to-Start).
  - **Effort Allocation**: Assigning staff-months/hours based on estimation models (COCOMO, Story Points).
  - **Milestone Definition**: Establishing clear validation points throughout the software lifecycle.

Gantt Charts in Software Engineering

- **Overview**: First developed by Henry Gantt, it is a horizontal bar chart used to visually track project schedules over a continuous timeline.
- **Core Components**:
  - **Horizontal Axis (X-axis)**: Represents time duration (Days, Weeks, Months, or Sprints).
  - **Vertical Axis (Y-axis)**: Lists software development activities/tasks derived from the WBS.
  - **Task Bars**: Length represents planned duration; color fill indicates percent completion.
  - **Milestones**: Represented by diamond symbols ($\Diamond$) to mark significant delivery points.
  - **Dependency Links**: Arrows connecting tasks to denote precedence constraints.
- **Advantages**: Highly intuitive visually; easy to communicate progress to stakeholders.
- **Limitations**: Can become overly cluttered for large-scale enterprise software systems with hundreds of interdependent tasks.

Gantt Chart Construction & Visualization

Software Development Phase Schedule (TikZ Representation):



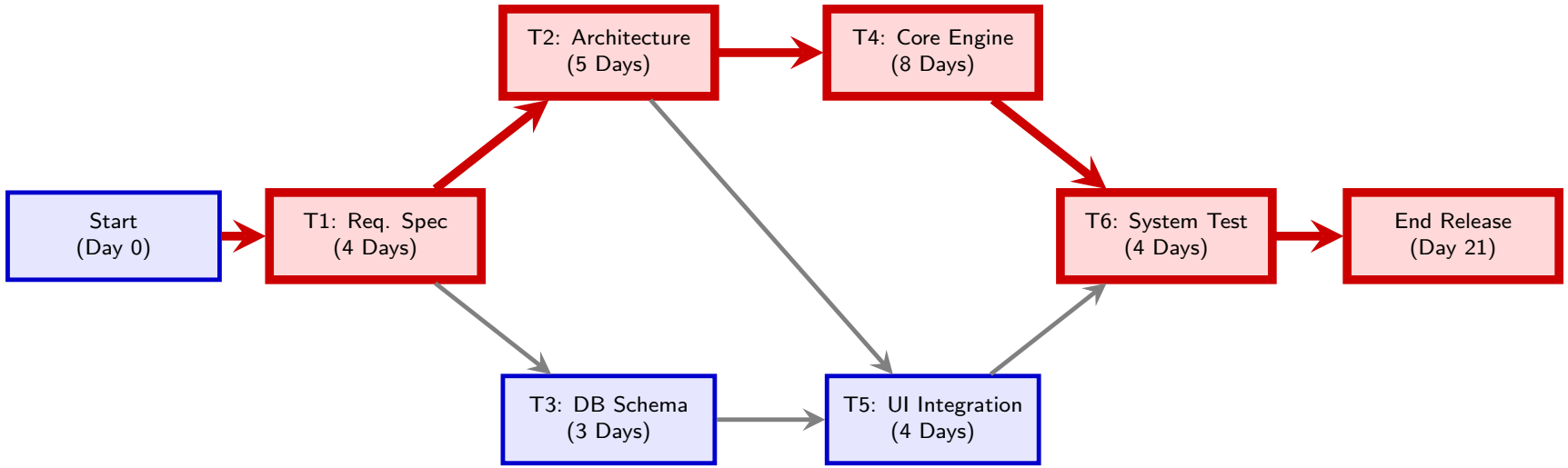
- Red arrows depict **Finish-to-Start** task dependencies.
- The red diamond highlights the **Deployment Milestone** upon QA sign-off.

Flow Charts & Activity Networks in Scheduling

- **Overview**: Flow Charts and Task Networks (PERT/CPM charts) model project workflows as directed graphs to illustrate sequence, parallelism, and logical control flow.
- **Core Concepts**:
  - **Nodes**: Represent activities, milestones, or decision points.
  - **Edges (Arrows)**: Represent execution paths, task sequencing, and dependencies.
- **Critical Path Method (CPM)**:
  - **Critical Path**: The longest continuous path through the activity network, determining the minimum possible project duration.
  - **Slack/Float Time**: The amount of time a non-critical task can be delayed without extending the overall project completion date.
  - Total Float = Late Start (LS) − Early Start (ES).
- **Purpose**: Essential for detecting bottlenecks and optimizing resource allocation.

Scheduling Flow Chart & Critical Path Network

PERT/CPM Activity Flow Network for Software System:



Critical Path Analysis

- **Critical Path (Highlighted Red)**: Start $\rightarrow$ T1 $\rightarrow$ T2 $\rightarrow$ T4 $\rightarrow$ T6 $\rightarrow$ End.
- **Duration**: $4 + 5 + 8 + 4 = 21$ Days. Any delay in T1, T2, T4, or T6 delays project release.

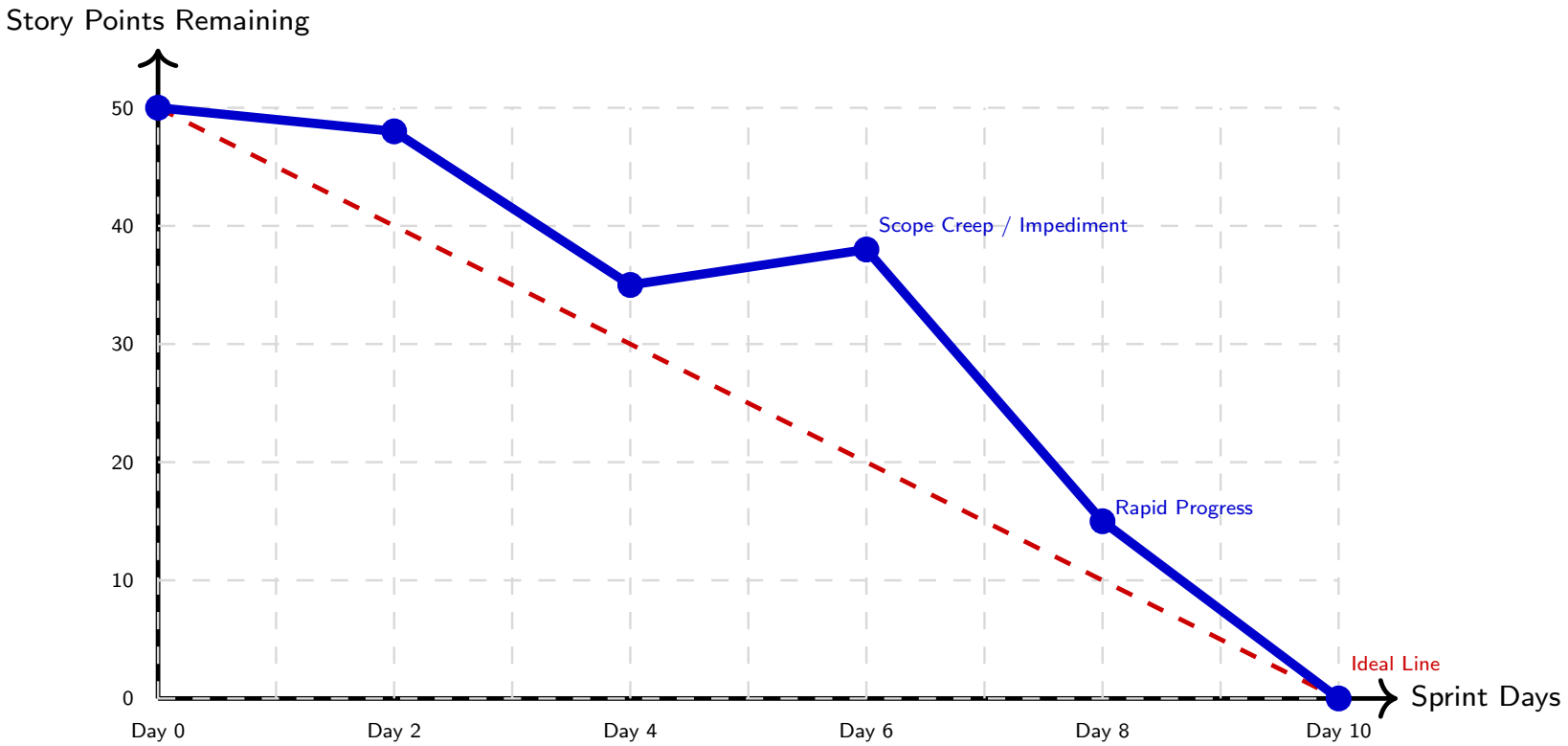
Agile Project Scheduling & Sprint Dynamics

- **Shift to Agile Scheduling**: Traditional predictive scheduling (Gantt/PERT) assumes static requirements. Agile replaces multi-month static plans with dynamic, empirical timeboxing.
- **Timeboxing & Sprints**:
  - Projects are divided into fixed-length iterations (typically 1 to 4 weeks) called **Sprints**.
  - Scope is fixed during a Sprint; uncompleted items return to the Product Backlog.
- **Story Points & Estimation**:
  - Relative units representing effort, complexity, and uncertainty (Planning Poker / Fibonacci series).
- **Team Velocity**:
  - The average number of Story Points completed per Sprint:

$$\text{Velocity} = \frac{\sum \text{Completed Story Points across Sprints}}{\text{Number of Sprints}}$$

Sprint Burndown Chart in Agile Models

- **Definition**: Graphical display of remaining work vs. remaining time in a Sprint.



- **Ideal Line**: Linear degradation rate assuming constant daily burn.
- **Actual Line Deviation**: Above line = Behind schedule; Below line = Ahead of schedule.

Comparative Analysis of Scheduling Tools

Table: Comparison of Software Project Scheduling Techniques

Feature / Dimension	Gantt Chart	Flow Chart / PERT	Sprint Burndown
Primary Paradigm	Traditional / Waterfall	Waterfall / Predictive	Agile / Scrum
Primary Focus	Timeline & Task Dates	Task Logic & Dependencies	Work Remaining in Sprint
Unit of Measurement	Calendar Days / Weeks	Task Duration / Slack	Story Points / Hours
Critical Path View	Implicit (requires arrows)	Explicit (highlights CPM)	N/A (Focus on velocity)
Adaptability	High overhead for changes	Complex re-graphing	Highly adaptive per Sprint
Target Audience	Management / Client	Project Planners / Leads	Developers / Scrum Team

Lecture Summary & Review Questions

Key Takeaways

- **Gantt Charts** provide timeline clarity, showing who is working on what and when.
- **Flow Charts / PERT** expose hidden task dependencies and pinpoint the **Critical Path**.
- **Sprint Burndown Charts** empower Agile teams to track real-time story point resolution and detect scope creep instantly.

GTU Review Questions

1. Differentiate between Gantt charts and PERT network diagrams with neat sketches.
2. Explain the significance of the Critical Path in software project scheduling. How is slack time computed?
3. Draw a Sprint Burndown chart for a 50 Story Point sprint across 10 days, demonstrating scenario analysis where a team encounters an impediment on Day 5.

Overview & Learning Objectives

Unit: Software Project Estimation & Scheduling

Software project scheduling transforms estimated effort into a detailed execution plan by allocating work across defined timeline milestones.

- **Core Objective:** Track development progress, allocate human resources, and detect schedule delays early in the SDLC.
- **Key Topics Covered in Lecture 35:**
  1. **Gantt Charts:** Visualizing task timelines, effort distribution, and overlapping milestones.
  2. **Flow Charts / Task Networks:** Mapping activity dependencies, critical path, and task sequencing.
  3. **Sprint Burndown Charts:** Monitoring agile iteration progress, work remaining, and team velocity.

Foundations of Task Allocation

Software Scheduling Rules

- **Compartmentalization:** Decompose project into manageable tasks via Work Breakdown Structure (WBS).
- **Interdependency:** Identify parallel vs. sequential software activities.
- **Time Allocation:** Assign effort (person-months) to explicit calendar dates.
- **Effort Validation:** Ensure developer load does not exceed capacity.

Key Project Artifacts

- **Milestones:** Defined completion checkpoints (e.g., SRS Signoff, Architecture Freeze).
- **Work Packages:** Specific software deliverables assigned to engineering sub-teams.
- **Slack Time:** Permissible delay of a task without slipping project deadline.

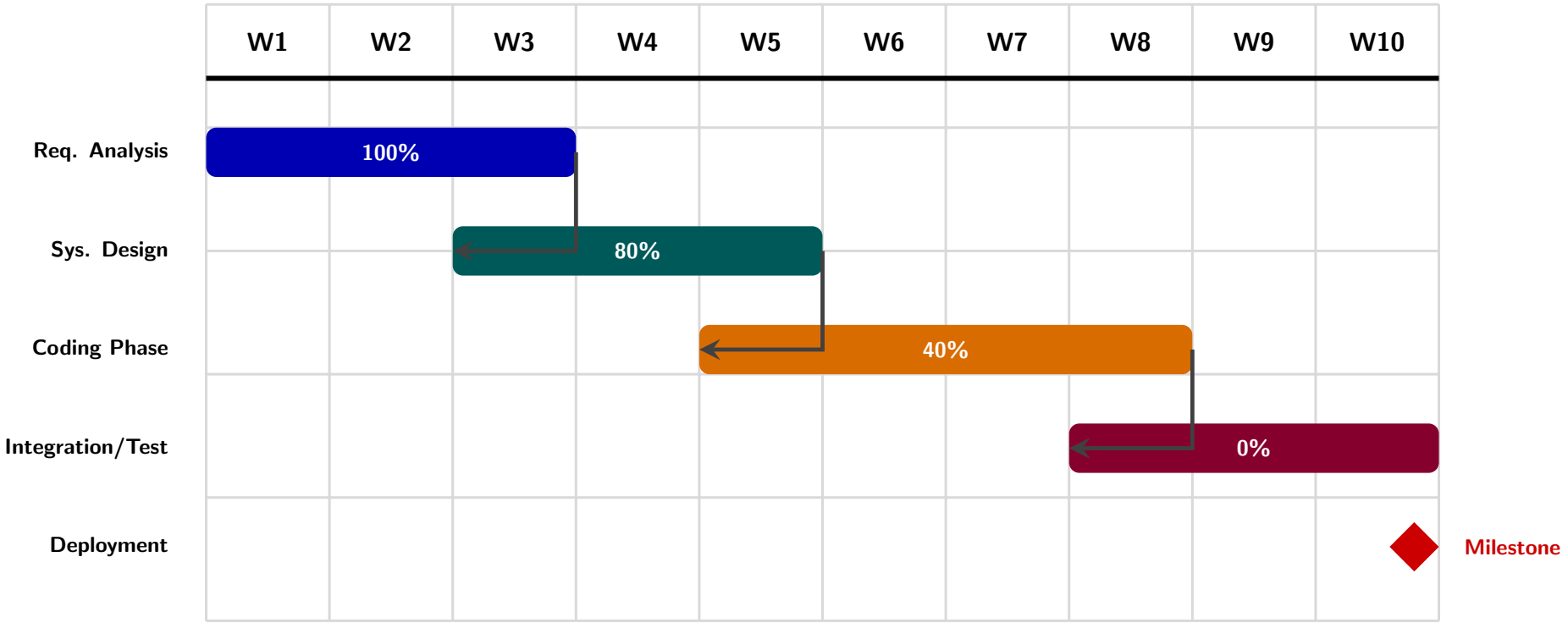
Timeline-Based Project Scheduling

What is a Gantt Chart?

A matrix display listing software engineering tasks along the vertical axis and calendar time units along the horizontal axis. Horizontal bars depict task start dates, durations, and end dates.

- **Key Features for Software Projects:**
  - Depicts overlapping engineering phases (Requirements, Design, Coding, Testing).
  - Visualizes team resource allocation across software modules.
  - Displays progress tracking using shaded progress fill indicators.
  - Highlights project milestones using diamond markers.
- **Limitation:** Less effective than PERT/CPM activity networks at representing complex non-linear task dependencies.

Gantt Chart for Software Module Development



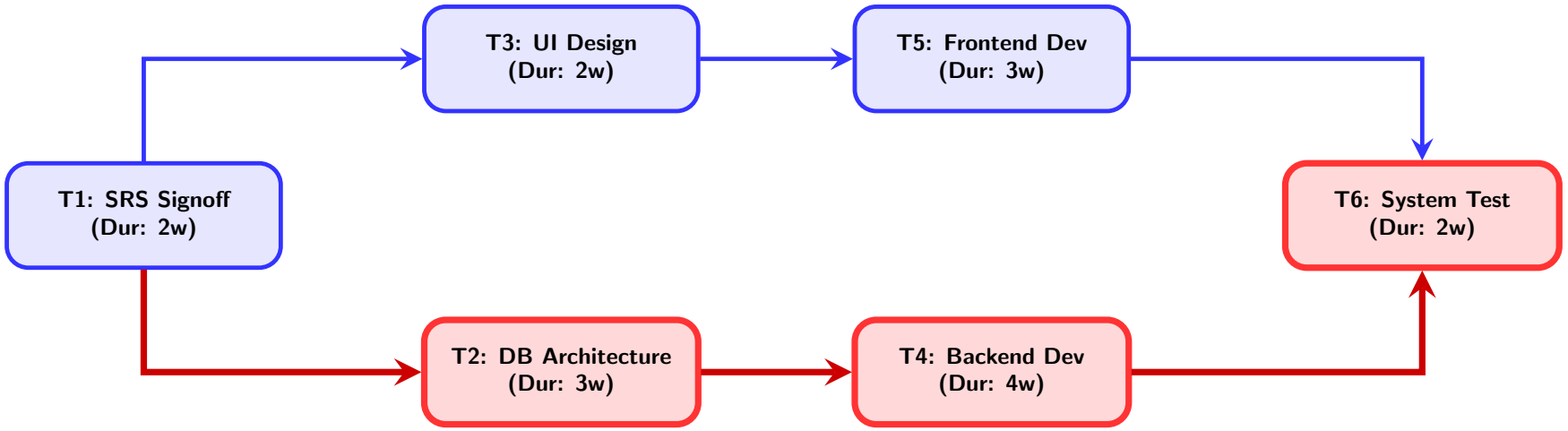
Activity Flow Networks in Software Scheduling

Flow Chart / Activity Network Model

A graphic representation of task sequences, interdependencies, and concurrency across software engineering activities (Activity-on-Node structure).

- **Core Scheduling Concepts:**
  - **Predecessor/Successor Logic:** Task B (Coding) cannot start until Task A (Design) completes.
  - **Parallel Execution:** Independent activities performed concurrently (e.g., UI Mockup and Database Schema design).
  - **Critical Path Method (CPM):** The longest sequence of dependent tasks that determines the minimum total project duration.
- **Engineering Value:** Identifies bottleneck tasks where any delay directly slips the target software release date.

Activity Flow Chart & Critical Path Analysis



Red Path: Critical Path (T1 $\rightarrow$ T2 $\rightarrow$ T4 $\rightarrow$ T6 = 11 Weeks Minimum Duration)

Agile Iterative Scheduling vs. Traditional Scheduling

Agile Scheduling Paradigm

In Agile software development (e.g., Scrum), time is fixed into short time-boxed iterations (*Sprints* of 1–4 weeks), while functional scope remains flexible.

- **Sprint Planning:** Selecting high-priority User Stories measured in *Story Points* or *Ideal Developer Hours*.
- **Team Velocity:** The rate at which an agile team completes story points per sprint, used to forecast future sprint capacity.
- **Adaptive Execution:** Continuous re-prioritization via daily Scrum standups and sprint reviews.
- **Agile Tracking Metric:** Sprint Burndown Chart.

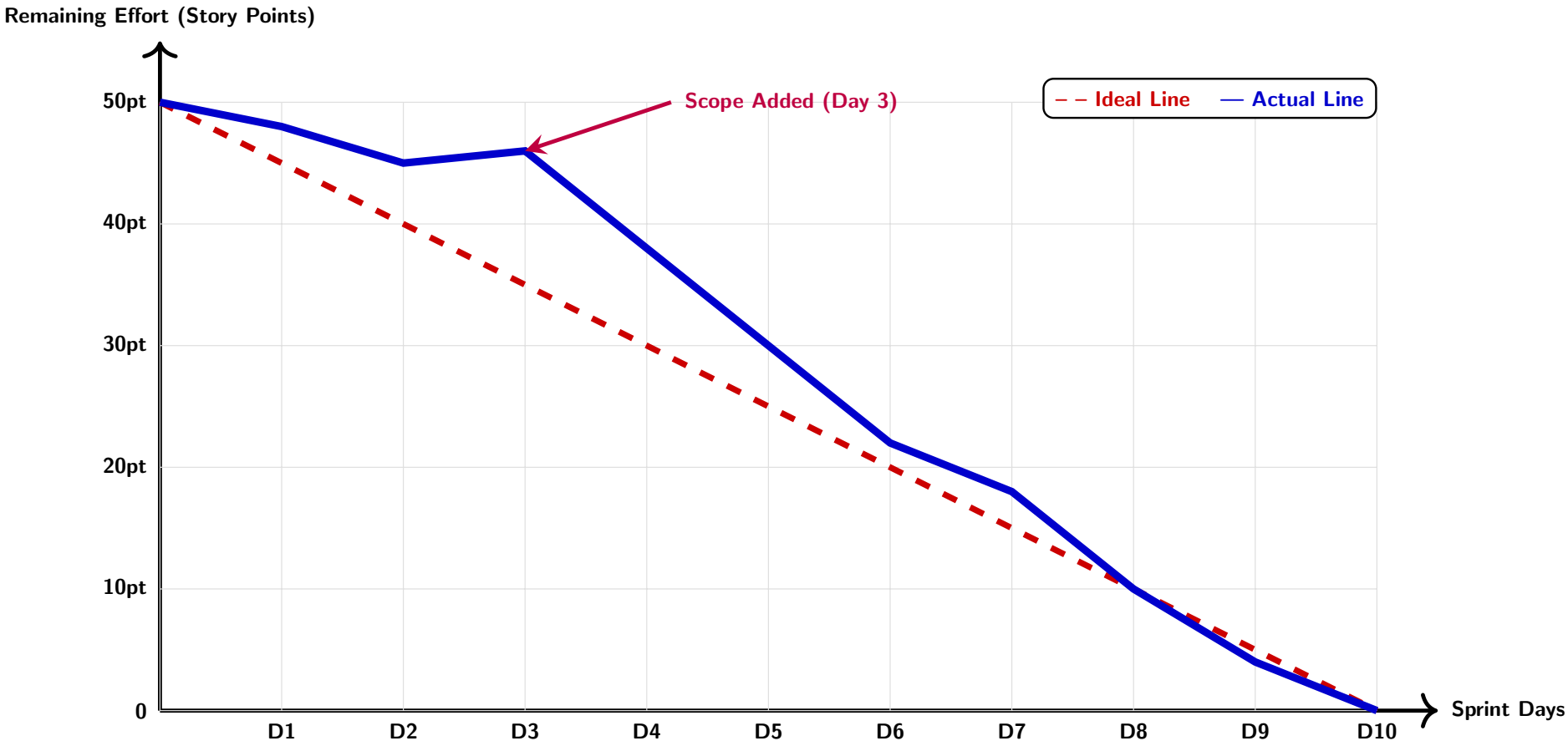
Monitoring Sprint Progress & Scope

Definition & Dynamic Tracking

A graphical depiction of remaining sprint effort (Y-axis: Story Points/Hours) plotted against time (X-axis: Sprint Working Days).

- **Ideal Burndown Line:** A linear diagonal reference path drawn from total committed effort on Day 0 down to 0 effort on the final day.
- **Actual Burndown Line:** Updated daily during Scrum standup:
  - **Above Ideal Line:** Team is behind schedule (tasks taking longer or blocked).
  - **Below Ideal Line:** Team is ahead of schedule (tasks finished faster).
- **Scope Creep Spike:** Upward step jumps indicate mid-sprint addition of unassigned user stories.

Sprint Burndown Plot (10-Day Sprint)



Gantt Chart vs Flow Chart vs Sprint Burndown Chart

Table: **Comparative Analysis of Software Scheduling Tools**

Feature	**Gantt Chart**	**Flow Chart (CPM)**	**Sprint Burndown**
Primary Focus	Timeline & task duration	Dependency & Critical Path	Work remaining velocity
SDLC Model	Waterfall / Hybrid	Sequential Projects	Agile / Scrum Sprints
Visual Form	Horizontal bar matrix	Network flow graph	2D Line progress plot
Key Benefit	Resource & milestone view	Spot delays & slack time	Real-time sprint tracking
Update Frequency	Weekly / Bi-weekly	Phase milestones	Daily Scrum Standup

Lecture 36: Overview of Software Risk Management

Course: GTU DI04000011 – Software Engineering
Unit: Software Project Estimation & Scheduling
Topic Focus: Software Risk Management Fundamentals

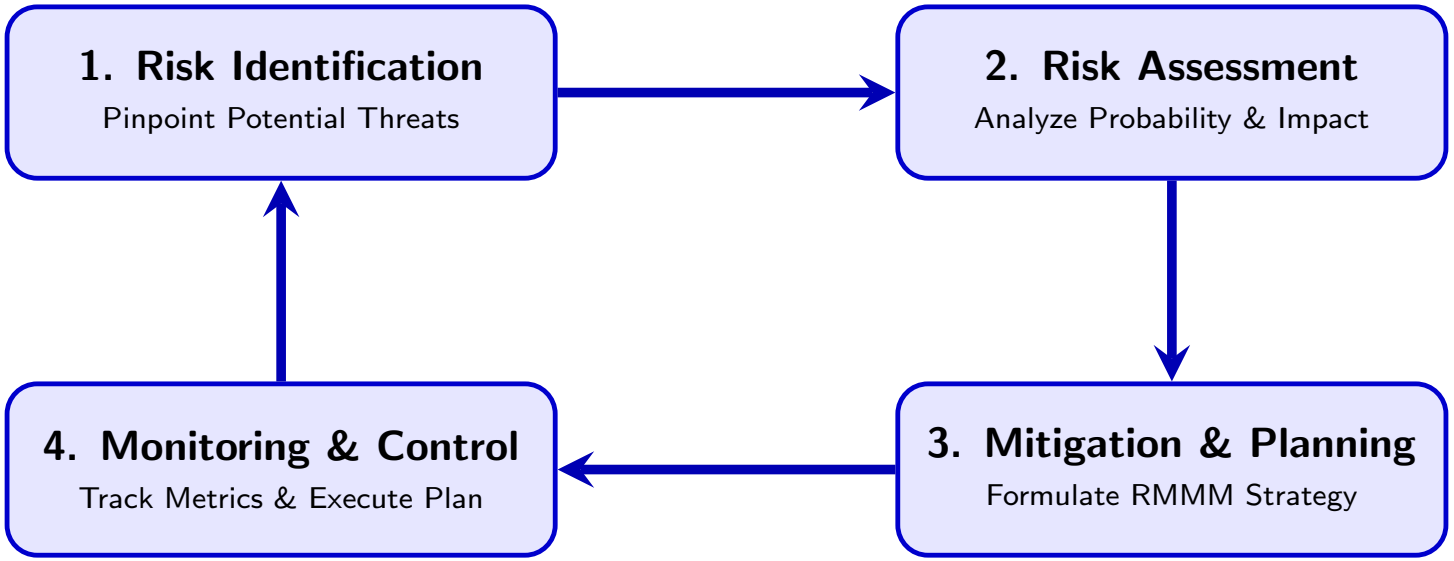
What is a Software Risk?

A software risk is a potential future event or condition that has a probability of occurrence and a negative consequence on the software project's cost, schedule, or product quality.

- **Dual Characteristics of Risk:**
  - **Uncertainty:** The event may or may not happen ($0 < P < 1$).
  - **Loss:** If the event occurs, unwanted consequences or financial/technical losses follow.
- **Software Management Strategies:**
  - **Reactive Risk Strategy:** "Firefighting" mode – reacting to problems only when they occur.
  - **Proactive Risk Strategy:** Formal risk management – anticipating risks, evaluating impacts, and preparing mitigation plans in advance.

The Software Risk Management Process Cycle

The proactive risk management lifecycle consists of four interconnected, iterative phases:



Iterative Process

Software risk management is continuous throughout the Software Development Life Cycle (SDLC) as new risks emerge and project context evolves.

Risk Identification: Categorization & SEI Taxonomy

Risk Identification is a systematic attempt to specify threats to the project plan.

Primary Risk Categories in Software Engineering

- **Project Risks:** Threaten the project schedule or resources (e.g., staff turnover, budget cuts).
- **Technical Risks:** Threaten the quality and timeliness of the software (e.g., complex algorithms, unproven frameworks).
- **Business Risks:** Threaten the viability of the software product (e.g., market changes, loss of upper management commitment).

SEI Software Risk Taxonomy Factors

- Product Size & Complexity
- Business Impact
- Customer Characteristics
- Process Definition & Rigor
- Development Environment
- Staff Size & Experience

Software Risk Classification & Common Project Risks

Software risks can also be classified based on predictability:

- **Known Risks:** Uncovered after careful evaluation of project plan, business environment, and technical specifications (e.g., unrealistic delivery date).
- **Predictable Risks:** Extrapolated from past project experience (e.g., staff turnover, poor communication with client).
- **Unpredictable Risks:** Extremely difficult to identify in advance (e.g., unexpected vendor failure, sudden regulatory changes).

Top Common Software Engineering Risks

1. **Personnel Shortages:** Key team members leaving mid-project.
2. **Unrealistic Schedules & Budgets:** Over-optimistic estimation.
3. **Requirement Volatility:** Scope creep and continuous requirement changes.
4. **Developing the Wrong User Interface:** Mismatch with end-user expectations.
5. **Gold Plating:** Adding unnecessary features that delay delivery.

Risk Assessment: Probability, Impact & Exposure

Risk Assessment evaluates the likelihood and severity of each identified risk to prioritize management efforts.

Risk Exposure (RE) Formula

Quantitative risk evaluation uses Risk Exposure (RE), also known as Risk Impact or Risk Risk Value:

$$RE = P \times L$$

Where:

- P = Probability of occurrence of the risk ($0 < P \leq 1.0$)
- L = Loss / Impact if the risk occurs (measured in cost, time, or severity)

Impact Categorization:

- **Catastrophic:** Project failure.
- **Critical:** Major schedule delay/cost overrun.
- **Marginal:** Moderate disruption.
- **Negligible:** Minor inconvenience.

Assessment Steps:

1. Estimate probability P for each risk.
2. Determine impact magnitude L.
3. Compute Risk Exposure RE.
4. Sort risks by RE in descending order.

Software Project Risk Assessment Table

A Risk Table prioritizes risks based on likelihood and severity for targeted mitigation.

Table: Software Engineering Risk Assessment & Exposure Matrix

Software Risk Event	**Category**	**Prob. (P)**	**Impact (L)**	**Risk Exposure (RE)**
Key Architect Resigns	Project	0.30	\$80,000	\$24,000 (High)
Unrealistic Delivery Deadline	Project	0.60	\$50,000	\$30,000 (Critical)
3rd-Party API Incompatibility	Technical	0.40	\$30,000	\$12,000 (Medium)
Database Bottleneck at Scale	Technical	0.25	\$40,000	\$10,000 (Medium)
Scope Creep (Uncontrolled Req.)	Business	0.70	\$60,000	\$42,000 (Critical)

Cut-off Line Concept

A threshold line is drawn across the Risk Table. Risks above the line receive formal mitigation planning; risks below are merely monitored.

RMMM Plan: Mitigation, Monitoring & Management

The RMMM (Risk Mitigation, Monitoring, and Management) Plan is a key artifact in software project management.

The Three Pillars of RMMM

- **Risk Mitigation (Prevention):** Actions taken *before* the risk occurs to reduce the probability of occurrence or lower the impact.
- **Risk Monitoring (Tracking):** Continuous tracking of project indicators to detect whether a risk is becoming more or less likely.
- **Risk Management / Contingency (Action):** Pre-planned execution steps taken *after* the risk actually occurs to minimize damage.

Cost-Benefit Trade-off

Mitigation costs money, time, and resources. Software managers must ensure that the cost of mitigation is significantly less than the expected Risk Exposure (RE).

Risk Mitigation Strategies in Software Projects

Practical mitigation strategies for core software engineering risks:

Mitigating High Staff Turnover

- Meet with current staff to determine causes for turnover.
- Mitigate impact by cross-training team members.
- Enforce modular code structures and comprehensive inline documentation.
- Maintain detailed architecture design documents for fast onboarding.

Mitigating Requirement Volatility & Scope Creep

- Conduct formal customer review workshops and prototype sign-offs.
- Establish a strict Change Control Board (CCB) and change impact analysis process.
- Adopt iterative/Agile development cycles to accommodate planned incremental changes.

Risk Monitoring and Dynamic Risk Control

Risk monitoring is an ongoing project tracking activity aimed at detecting early warning signs.

Key Software Metrics to Monitor

- **Team Morale & Turnover Indicators:** Job satisfaction, interpersonal conflicts.
- **Development Velocity & Burn-down Variation:** Schedule slip on milestone tasks.
- **Defect Density Spikes:** Unusually high bug rates in specific modules signaling technical risk.
- **Requirement Changes Count:** Frequency of change requests submitted per sprint.

Risk Control & Re-evaluation

- Regularly review and update the Risk Table during weekly status meetings.
- If a risk materializes, invoke the pre-scripted contingency plan immediately.
- Conduct post-mortem analyses after risk resolution to update organizational risk checklists.

Summary & Academic Review (GTU Course DI04000011)

Lecture 36 Summary Checklist

1. Software risks combine **uncertainty** and **loss**. Proactive management is essential.
2. The risk management cycle spans **Identification** $\rightarrow$ **Assessment** $\rightarrow$ **Mitigation/Planning** $\rightarrow$ **Monitoring/Control**.
3. Risk Exposure calculation: $RE = P \times L$.
4. The RMMM plan handles proactive prevention (Mitigation), metric tracking (Monitoring), and contingency response (Management).

GTU Exam Orientation Questions

- **Q1:** Define Software Risk. Differentiate between Reactive and Proactive risk strategies.
- **Q2:** Explain the RMMM plan with a suitable software engineering scenario (e.g., staff turnover).
- **Q3:** Calculate RE given $P = 0.4$ and financial loss $L = \$50,000$, and discuss its placement in a Risk Table.

Context in Software Engineering

Software Risk Management is a critical sub-discipline of Software Project Estimation and Scheduling (GTU Course DI04000011). It aims to identify, analyze, and address potential project threats before they cause schedule slippage, budget overruns, or software quality degradation.

Core Attributes of Risk

- **Uncertainty**: The risk event may or may not occur ($0 < P < 1$).
- **Loss**: The negative consequence or impact (C) if the risk materializes.

Primary Objectives

- Avoid software project failure.
- Minimize impact of unforeseen events.
- Provide realistic project scheduling.
- Enhance decision-making under uncertainty.

Management Strategies

- **Reactive Strategy ("Firefighting"):** The project team reacts to problems only when they occur. Resources are spent coping with crisis management rather than prevention.
- **Proactive Strategy:** Formal risk management plan established before coding begins. Potential risks are identified, evaluated, prioritized, and mitigated systematically.

Taxonomy of Software Risks

Project Risks	Threaten the project plan, schedule, or cost (e.g., staff turnover, loss of experienced developers).
Technical Risks	Threaten software quality and timeliness (e.g., complex architecture, unproven technologies, hard design constraints).
Business Risks	Threaten the viability of the software product (e.g., building a product nobody wants, shifts in market demand).

Definition

Risk Identification is a systematic attempt to specify threats to the project plan. By creating a checklist of known and predictable risks, team leads can anticipate failure modes.

Generic vs. Product-Specific

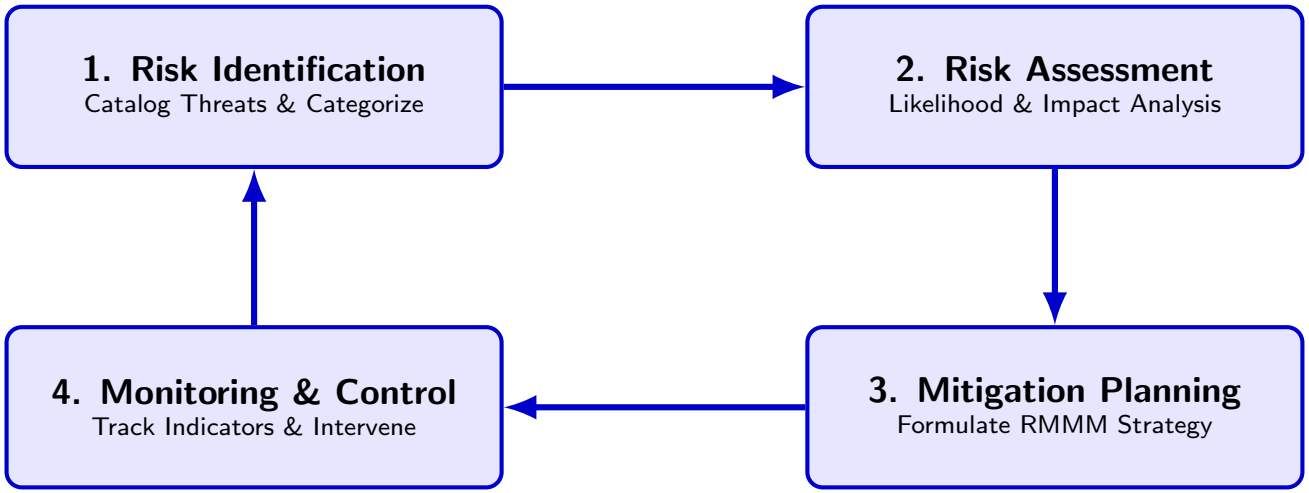
- **Generic Risks:** Potential threats to every software project (e.g., requirement changes, team communication breakdown).
- **Product-Specific Risks:** Hazards unique to the specific domain, architecture, or technology stack.

Identification Categories

- **Size Risk:** Lines of Code, function points.
- **Business Impact:** Constraints by executive management.
- **Customer Characteristics:** Client sophistication and responsiveness.
- **Process Definition:** Adherence to SEI CMM / ISO standards.

Risk Management Process Workflow

Iterative Steps in Software Risk Lifecycle



Continuous Feedback Loop

Risk management is not a static document created at project kickoff. It functions as a dynamic process integrated across all phases of the Software Development Life Cycle (SDLC).

Assessment Steps

1. Establish a scale reflecting the likelihood of a risk (P).
2. Delineate the consequences of the risk (C).
3. Estimate the impact of the risk on the project and product.
4. Calculate the overall risk exposure score.

Mathematical Model

Risk Exposure (RE) is computed as:

$$RE = P \times C$$

where:

- P = Probability of occurrence ($0 < P < 1$).
- C = Cost/Impact to the project if risk occurs.

Impact Category Scale

- **Catastrophic:** Project termination.
- **Critical:** Major schedule delay.
- **Marginal:** Moderate delay/overrun.
- **Negligible:** Minor inconvenience.

Table: Software Project Risk Prioritization & Exposure Analysis

Identified Risk Item	**Category**	**Prob. (P)**	**Impact (C)**	**Exposure**
Key developer turnover during critical phase	Project	0.30	$40,000	$12,00
Unmanaged scope creep / requirement shifts	Technical	0.50	$50,000	$25,00
Third-party component delivery delay	Technical	0.20	$60,000	$12,00
Client delivery deadline compression	Business	0.40	$35,000	$14,00
Target hardware specification change	Technical	0.15	$20,000	$3,00

Interpretation

Risks are sorted by Risk Exposure (RE). The team establishes a risk cutoff threshold; risks above the threshold receive explicit resource allocation for mitigation planning.

The RMMM Framework

An RMMM Plan documents all risk analysis operations and serves as part of the overall software project plan.

Risk Mitigation (Avoidance): Proactive steps taken to prevent the risk from occurring. Focuses on reducing the probability (P) or potential impact (C).

Risk Monitoring: Continuous tracking of project metrics to detect early warning signs or triggers indicating that a risk is escalating.

Risk Management (Contingency): Action plan executed when mitigation efforts fail and the risk materializes into an active problem.

RMMM Efficiency

Mitigation actions must be cost-effective: the cost of implementing a mitigation strategy should be significantly less than the Risk Exposure (RE).

Risk Mitigation Strategies

Actionable Solutions for Common Software Risks

1. High Staff Turnover (Project Risk)

- **Mitigation:** Maintain staff redundancy, cross-train team members, enforce strict documentation standards, establish shadow roles for critical modules.

2. Requirements Instability (Technical / Scope Risk)

- **Mitigation:** Build prototypes early, implement formal Change Control Boards (CCB), use Agile incremental deliveries to baseline requirements.

3. Performance Degradation (Technical Risk)

- **Mitigation:** Conduct early architectural stress tests, execute simulation benchmarks, perform code reviews focused on complexity metrics.

Monitoring Indicators

- Team morale and interpersonal communication quality.
- Defect discovery rate vs. resolution rate.
- Milestone achievement variance (SPI/CPI in Earned Value Management).
- Availability of hardware and third-party tools.

Trigger Signals

Predefined conditions that alert the project manager to execute contingency plans:

- Schedule slip exceeding 10% of total sprint time.
- Open critical defects exceeding threshold before release candidate build.
- Key staff member resignation.

Control Actions

Adjust project schedules, reallocate engineering staff, compress scope via feature deferral, or invoke emergency contingency reserve funds.

Summary of the 4 Key Risk Steps

1. **Identification:** Systematic cataloging of generic and product risks.
2. **Assessment:** Quantifying exposure ($RE = P \times C$) and prioritizing risks.
3. **Mitigation & Planning:** Structuring proactive RMMM plans.
4. **Monitoring & Control:** Tracking warning triggers and executing contingency actions.

GTU Exam Preparation Notes

- Understand the difference between Reactive and Proactive strategies.
- Be prepared to calculate Risk Exposure (RE) and rank software risks in a tabular format.
- Explain the components of an RMMM Plan with real-world software engineering examples.

Course Context: GTU DI04000011 – Software Engineering

This lecture focuses on the implementation and static verification phase of the Software Development Life Cycle (SDLC), addressing code quality, static review techniques, and software documentation principles.

Key Learning Objectives

- Understand the necessity and elements of **Coding Standards** and **Coding Guidelines**.
- Analyze static code review mechanisms: **Code Walkthroughs** vs. **Formal Code Inspections**.
- Distinguish between **Internal Documentation** (code-level) and **External Documentation** (system/user-level).

Coding Standards vs. Coding Guidelines

Ensuring Uniformity and Quality

Fundamental Definitions

- **Coding Standards**: Mandatory rules enforced by an engineering organization to guarantee uniform style, safety, portability, and compliance across a codebase.
- **Coding Guidelines**: Recommended best practices and style suggestions that assist developers in writing clean, expressive, and maintainable software.

Importance in Software Engineering

- **Maintainability**: Reduces code comprehension effort during the maintenance phase, which accounts for over 70% of total software cost.
- **Team Interoperability**: Enables seamless code transfer between developers by eliminating personal stylistic idiosyncrasies.
- **Defect Avoidance**: Restricts anti-patterns, dangerous language features (e.g., uninitialized pointers, implicit type conversions), and unhandled side effects.

Key Elements of Coding Standards

Structural, Naming, and Control Constraints

Naming & Formatting

- **Naming Conventions**: Strict usage of PascalCase (classes), camelCase (variables/methods), and UPPER_CASE (constants).
- **Layout & Indentation**: Standard tab/space widths, explicit brace placement rules, maximum line length (e.g., 80–120 chars).
- **Header Structure**: Mandatory standard header block for copyright, module description, and history.

Control & Safety Rules

- **Complexity Limits**: Capping Cyclomatic Complexity per function ($V(G) \leq 10$).
- **Control Flow**: Prohibiting unstructured control jumps (‘goto‘ statements, multiple ‘return‘ points per function).
- **Defensive Coding**: Mandatory validation of function input parameters and explicit error/exception handling.

Concept of Code Review

Code review is a **static verification technique** where software source code is systematically examined by developers before testing or merging into the primary baseline.

Primary Objectives

- Discover logical defects, edge-case flaws, and security vulnerabilities early in the SDLC.
- Verify compliance with established coding standards and architectural requirements.
- Promote shared ownership and knowledge dissemination across the development team.

Spectrum of Review Rigor

- **Informal Techniques**: Peer code reviews, pair programming, and **Code Walkthroughs**.
- **Formal Techniques**: **Code Inspections** (Fagan Inspection Framework with structured processes, roles, and metrics).

Definition

An **informal review technique** in which the author of the code leads one or more team members through a step-by-step examination of the source code, simulating execution paths using sample test scenarios.

Key Characteristics

- **Author-Driven**: The session is scheduled, guided, and explained by the author.
- **Low Formality**: Minimal pre-meeting preparation is required from the review participants.
- **Interactive Traversal**: Focuses on walking through execution logic, algorithm correctness, and edge cases interactively.

Trade-offs

- *Advantages*: Low resource overhead, fast execution, excellent for knowledge sharing.
- *Disadvantages*: Vulnerable to author bias, lacks structured defect metrics, and may miss non-obvious structural flaws.

Definition

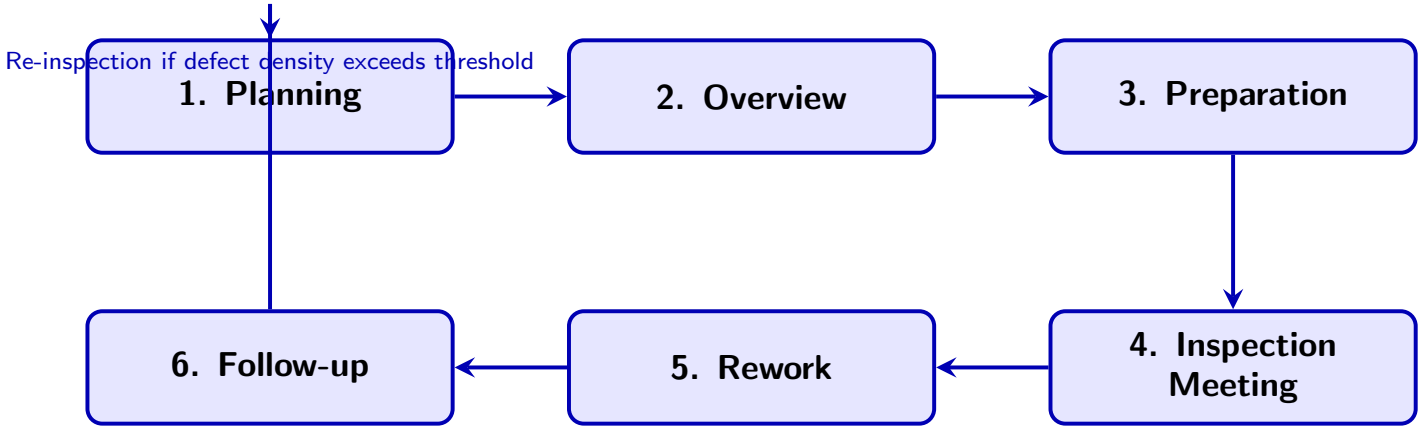
A **formal, highly structured static verification procedure** (originally defined by Michael Fagan) aimed at finding defects using predefined checklists and formal role assignments.

Defined Team Roles

- **Moderator**: Neutral leader who plans the inspection, controls meeting flow, and verifies rework.
- **Author**: Developer who created the code artifact (answers questions, does not lead).
- **Reader**: Paraphrases and reads out the code line-by-line during the meeting.
- **Inspector / Reviewer**: Examines code against functional specs and defect checklists.
- **Scribe**: Formally logs every identified defect, location, and severity during the meeting.

Code Inspection Workflow

Formal Process Stages



Rule of Execution

Inspection meetings focus exclusively on **defect discovery**, strictly prohibiting discussions regarding defect resolution or alternative solutions during the meeting session.

Comparison: Code Walkthrough vs. Code Inspection

Formal vs. Informal Review Comparison

Parameter	Code Walkthrough	Code Inspection
Formality Level	Informal to semi-formal	Formal and highly structured
Session Leader	Code Author	Neutral Trained Moderator
Primary Goal	Code understanding and scenario validation	Systematic defect detection & quality assurance
Preparation	Minimal pre-meeting prep	Intensive individual analysis using checklists
Roles	Author, Reviewers (unstructured)	Moderator, Author, Reader, Inspectors, Scribe
Metrics & Data	Not formally tracked	Defect density, inspection rate, error metrics logged
Output Artifact	Informal meeting notes	Formal Defect List and Inspection Report

Definition

Documentation provided directly within the source code files to assist maintainers and developers in understanding the internal construction of the software module.

Key Components of Internal Documentation

- **Module Header Blocks**: Summary of module purpose, author details, creation date, dependencies, and revision history.
- **Inline Comments**: Concise descriptions of non-obvious logic, complex algorithms, mathematical models, and business rule constraints.
- **Self-Documenting Code**: Descriptive naming of variables, functions, and classes to minimize unnecessary comment redundancy.
- **Interface Specifications**: Pre-conditions, post-conditions, parameter descriptions (‘@param‘), return types (‘@return‘), and raised exceptions (‘@throws‘).

Definition

Independent documentation artifacts produced separately from source code to meet the operational, technical, and managerial needs of varied stakeholders.

Categories of External Documentation

- **User Documentation**: User Manuals, Installation Guides, Troubleshooting Manuals, and FAQs for end-users.
- **System Administration Docs**: Deployment guides, server configuration specifications, database schemas, and security policies.
- **Technical / Developer Docs**: Software Requirements Specification (SRS), Software Design Document (SDD), Architecture Manuals, API references, and Test Suite Documents.

Lecture 38 Summary

High-quality software delivery requires disciplined adherence to **coding standards**, static verification via **walkthroughs** and **inspections**, and maintainable **internal and external documentation**.

GTU Course: Software Engineering (DI04000011)

Unit: Software Coding and Testing
Lecture Focus: Standards, Reviews, and Documentation Practices

Key Topics Covered:

- **Coding Standards & Guidelines:** Differentiating mandatory rules from advisory recommendations.
- **Code Review Techniques:** Comparative analysis of informal Walkthroughs and formal Code Inspections.
- **Software Documentation:** Implementation of Internal (code-level) and External (system/user-level) documentation.

Learning Objective

To master static quality assurance methods and documentation frameworks that ensure maintainable, error-free software implementations.

Coding Standards vs. Coding Guidelines

Definitions & Comparative Analysis

Dimension	Coding Standards	Coding Guidelines
Nature	Mandatory rules strictly enforced.	Recommended best practices & idioms.
Scope	Syntax, naming conventions, indentation, error structures.	Design patterns, architectural styles, performance advice.
Enforcement	Automated static linters, CI/CD gates.	Peer reviews, design discussions.
Objective	Uniformity, consistency, bug prevention.	Code elegance, maintainability, modularity.
Example	"Class names must use PascalCase."	"Prefer immutability over mutable state where feasible."

Key Takeaway

Standards eliminate personal formatting friction; guidelines elevate systemic code quality.

- **Naming Conventions:**
  - Variables and methods: `camelCase` (e.g., `calculateTax()`)
  - Classes and interfaces: `PascalCase` (e.g., `PaymentProcessor`)
  - Constants: `UPPER_SNAKE_CASE` (e.g., `MAX_RETRY_LIMIT`)
- **Layout and Formatting:**
  - Fixed indentation (e.g., 4 spaces, no mixed tabs).
  - Maximum line length boundaries (e.g., 80 or 120 characters).
  - Consistent brace placement (K&R vs. Allman style).
- **Security and Error Management:**
  - Mandatory input parameter validation and sanitization.
  - Explicit exception handling (avoid empty catch blocks).
  - Prevention of hardcoded secrets or sensitive memory leaks.

What is a Code Review?

A systematic examination of computer source code intended to find bugs, security flaws, and standard violations prior to dynamic testing.

Taxonomy of Static Verification Techniques:

- **Informal Reviews:** Ad-hoc peer feedback, pair programming, desk checking.
- **Semi-Formal Reviews:** Code Walkthroughs led by the code author.
- **Formal Reviews:** Code Inspections with defined roles, checklists, and entry/exit criteria.

Primary Benefits

Detects defects early in the SDLC (lowering fix costs), enforces team standards, and facilitates cross-training.

- **Definition:** An informal or semi-formal review where the author guides team members through source code logic line-by-line.
- **Key Characteristics:**
  - Driven and presented by the author.
  - Focuses on scenario simulation (dry running test cases manually).
  - Open, unstructured feedback session.
- **Team Roles:**
  - **Author / Presenter:** Explains code design and logic flow.
  - **Reviewers / Peers:** Question assumptions, identify logic flaws.
  - **Scribe:** Records identified issues and action items.
- **Primary Goal:** Logic validation, knowledge sharing, and gathering early feedback.

Definition

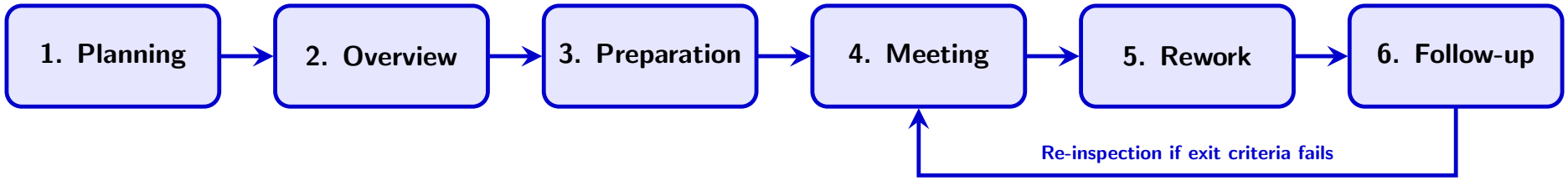
A rigorous, highly formal static analysis process introduced by Michael Fagan to systematically discover defects in code artifacts.

Key Operational Roles:

- **Moderator:** Leads the inspection process, ensures adherence to protocol, maintains objectivity (never the author).
- **Author:** Provides code artifact, clarifies technical queries, reworks defects.
- **Reader:** Reads/paraphrases the source code during the formal meeting.
- **Inspector:** Scrutinizes code against standardized defect checklists.
- **Recorder / Scribe:** Documents each logged defect accurately.

Fagan Inspection Process Workflow

Structured Phases of Formal Code Inspection



Phase Descriptions:

1. **Planning:** Moderator verifies entry criteria and schedules reviewers.
2. **Preparation:** Individual inspectors analyze code using checklists.
3. **Inspection Meeting:** Reader paraphrases code; defects are logged (no solutions discussed).
4. **Rework & Follow-up:** Author resolves defects; Moderator verifies exit criteria.

Comparative Analysis: Walkthrough vs. Inspection

Formal vs. Semi-Formal Static Reviews

Parameter	Code Walkthrough	Code Inspection
Formality	Informal to Semi-Formal	Strictly Formal
Leader	Code Author	Independent Moderator
Reader	Author presents logic	Designated Reader paraphrases
Preparation	Minimal prior study	Extensive individual review
Objective	Logic validation & learning	Defect detection & metrics logging
Checklists	Optional or informal	Mandatory structured checklists
Metrics	Rarely collected	Formally tracked (defect rate)

Definition

Documentation embedded directly within the source code to assist maintainers in understanding implementation details.

Key Elements of Internal Documentation:

- **Header Comments:** Module purpose, author details, creation date, modification logs, license.
- **Function / Interface Annotations:** Parameter contracts, return types, pre/post-conditions, exception types.
- **Inline Comments:** Clarify non-obvious algorithm choices (explaining *why*, not *what*).
- **Self-Documenting Code:** Descriptive identifier names and clean modular structures that reduce comment verbosity.

Docstring Generators

Tools like Javadoc, Doxygen, and Sphinx compile internal annotations into external HTML technical specifications.

- **Definition:** Standalone technical and operational documents designed for developers, maintainers, and end-users.
- **Developer / Technical Documentation:**
  - Software Requirements Specification (SRS)
  - Software Design Document (SDD - Architecture, ER diagrams, class schemas)
  - Test Plan and Test Execution Reports
  - API Reference and Integration Guides
- **User Documentation:**
  - User Manuals and Operational Guides
  - Installation, Setup, and Configuration Guides
  - System Administrator Reference Manuals
  - Release Notes and Known Issues

Summary

High quality coding standards, rigorous reviews, and thorough documentation together form the foundation of software maintainability.

Lecture 40: Software Coding & Testing Overview

Course: GTU DI04000011 – Software Engineering
Unit: Software Coding and Testing
Lecture Topic: Coding Standards, Code Reviews, and Software Documentation

Learning Objectives

- Understand the necessity of **Coding Standards and Programming Guidelines** in software development.
- Differentiate between informal and formal code review techniques: **Code Walkthrough** and **Code Inspection**.
- Examine the structure, roles, and procedures of **Fagan's Inspection Process**.
- Classify software documentation into **Internal** and **External Documentation** and analyze their attributes.

Coding Standards and Guidelines: Core Concepts

Definitions

- **Coding Standards:** Mandatory rules enforced across an organization or project to ensure code uniformity, portability, and defect reduction.
- **Coding Guidelines:** Recommended best practices and stylistic advice that assist developers in writing clean, readable code.

Why Enforce Standards?

- **Readability & Maintainability:** Over 70% of software lifecycle cost is spent on maintenance. Standardized code reduces cognitive load.
- **Portability:** Minimizes compiler-dependent and hardware-dependent behavior.
- **Defect Prevention:** Prevents common coding errors (e.g., memory leaks, uninitialized variables, array index out of bounds).
- **Ease of Integration:** Simplifies combining modules developed by distributed teams.

Key Programming Guidelines & Best Practices

Naming & Formatting

- **Naming Conventions:**
  - Variables: Meaningful nouns (`totalAmount`).
  - Functions: Action verbs (`calculateTax()`).
  - Constants: UPPER_CASE (`MAX_RETRY`).
- **Layout & Indentation:** Consistent indentation (2/4 spaces), explicit scope braces.
- **File Headers:** Module name, author, creation date, modification history.

Control & Architecture

- **Avoid Magic Numbers:** Replace hardcoded values with named constants.
- **Control Complexity:** Limit nesting depth of loops/conditionals (≤ 3 levels).
- **Single Responsibility:** Keep functions short (≤ 50 lines) and single-purpose.
- **Error Handling:** Use explicit return codes or structured exception handling (`try-catch`).

What is Code Review?

A systematic examination of computer source code intended to find and fix mistakes overlooked in the initial development phase, improving both software quality and developer skills.

Table: Taxonomy of Code Review Techniques

Parameter	**Informal Review**	**Code Walkthrough**	**Formal Code Inspection**
Formality	Low / Ad-hoc	Semi-formal	Highly Formal (Fagan)
Lead Role	Peer developer	Code Author	Trained Moderator
Primary Focus	Quick feedback	Code comprehension	Defect detection & metrics
Roles Defined	None	Author, Reviewers	Moderator, Author, Reader, Recorder
Data Collection	No	Optional	Mandatory (Defect density, speed)
Checklists	No	Optional	Mandatory domain checklists

Code Walkthrough: Workflow & Characteristics

Definition

An informal to semi-formal static analysis technique where the code author leads members of the development team through a segment of software code while participants ask questions and note defects.

Walkthrough Process

1. **Preparation:** Author distributes source code and design documents to participants prior to the meeting.
2. **Presentation:** Author walks through the code line-by-line or scenario-by-scenario.
3. **Dry Run / Execution Tracing:** Reviewers simulate execution with test data.
4. **Defect Logging:** Identified issues, anomalies, or standard violations are logged by the author/scribe for correction.

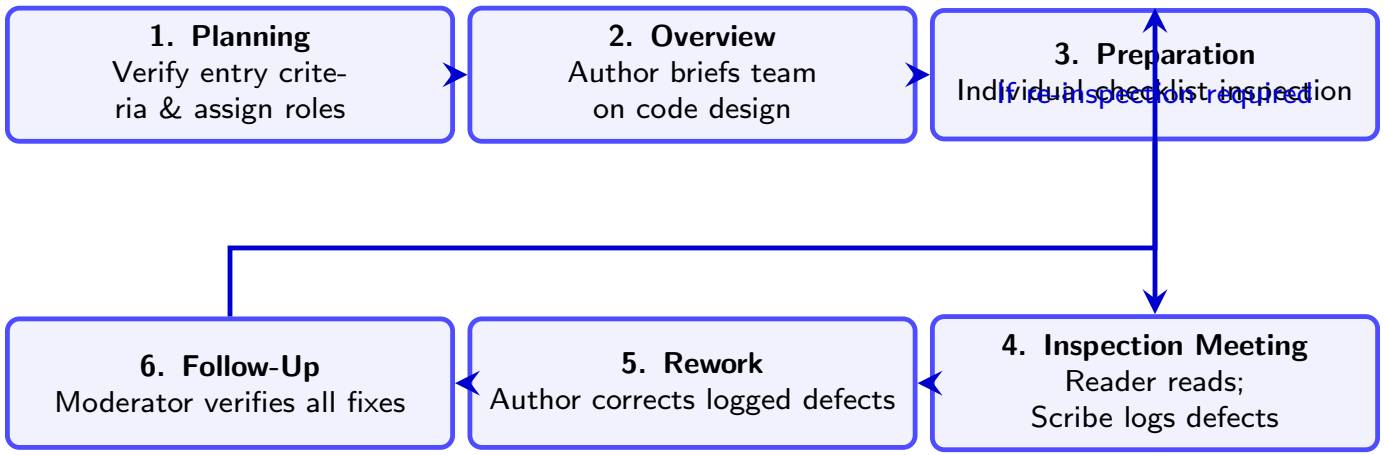
Key Advantage

Promotes knowledge sharing across the development team and educates junior developers on code design.

Formal Code Inspection: Fagan Process

Fagan's Inspection Methodology

A formal, rigorous static review technique designed by Michael Fagan to identify defects early in the SDLC without executing the software.



Fagan Inspection Roles & Performance Metrics

Inspection Team Roles

- **Moderator:** Leads the inspection, ensures adherence to rules, manages schedule.
- **Author:** Created the artifact; answers questions during meeting.
- **Reader:** Reads code aloud line-by-line during the meeting.
- **Recorder (Scribe):** Records every defect on the defect log sheet.
- **Inspector:** Entire team acts as inspectors using checklist driven defect finding.

Key Quality Metrics

- **Defect Density:**

$$\text{Defect Density} = \frac{\text{Total Defects}}{\text{KLOC}}$$

- **Inspection Rate:** Lines of code reviewed per hour (typically 150–250 LOC/hr).
- **Phase Defect Removal Efficiency (DRE):** Percentage of total errors removed before testing phase.

Software Documentation: Overview & Classification

Role of Documentation in Software Engineering

Documentation encompasses all written materials, diagrams, and media that describe the architecture, design, code, operation, and maintenance of a software system.

Internal Documentation

- Embedded directly inside the source code file.
- Aimed at maintenance programmers and developers.
- Synchronized automatically with code modifications.

External Documentation

- Maintained as separate documents from code.
- Serves users, system administrators, and architects.
- Created during various phases of SDLC.

Internal Software Documentation

Components of Internal Documentation

1. **Header Blocks:** Metadata at the top of each file/class (Purpose, Author, License, Modification History).
2. **Inline Comments:** Explanations of non-obvious algorithms, edge cases, and business logic decisions.
3. **Self-Documenting Code:** Using clear module decomposition and intuitive variable/method identifiers so code explains *what* it does.
4. **Structured API Comments:** Tool-extractable docstrings (e.g., Javadoc, Doxygen, Sphinx) detailing parameter types, return values, and exceptions.

Best Practice Rule

Comments should explain **WHY** a piece of code was written in a specific way, not **WHAT** the code is doing (which should be clear from the code itself).

External Software Documentation

Categories of External Documents

- **User Documentation:**
  - User Manuals / Guides (Step-by-step feature usage).
  - Installation & Configuration Guides.
  - Release Notes & Known Issues list.
- **System Documentation:**
  - Software Requirements Specification (SRS).
  - High-Level & Low-Level Design Documents (ADD/LLD).
  - Database Schemas, UML Class/Sequence Diagrams.
- **Operations Documentation:**
  - Deployment Scripts & Runbooks.
  - System Administrator & Troubleshooting Manuals.

Lecture 41: Overview & Learning Objectives

Unit: Software Coding and Testing

Welcome to Lecture 41 of Software Engineering (GTU Course: DI04000011). This lecture focuses on translating detailed design into high-quality code and verifying it prior to testing.

Key Learning Objectives:

- Understand the significance of **Coding Standards** and **Guidelines** in software development.
- Distinguish between informal code reviews (**Code Walkthrough**) and formal code reviews (**Code Inspection**).
- Explore Fagan's inspection process and role allocations during code reviews.
- Differentiate between **Internal Documentation** (comments, header blocks) and **External Documentation** (user/system manuals).

What are Coding Standards and Guidelines?

During software development, writing clean, uniform, and readable code is critical for maintainability and scalability across engineering teams.

Coding Standards (Mandatory)

- Rules enforced strictly across the organization.
- Non-compliance leads to build errors or review rejection.
- Focuses on naming rules, layout, indentation, and structure.

Coding Guidelines (Recommended)

- Best practices to improve code quality and design.
- Provide general direction without strict automated enforcement.
- Focuses on design patterns, complexity reduction, and style.

Key Benefits

Reduces maintenance cost, enhances team readability, facilitates code reuse, and minimizes defect injection rates during implementation.

Comparison: Coding Standards vs. Coding Guidelines

Table: Comparison of Software Coding Standards and Guidelines

Attribute	**Coding Standards**	**Coding Guidelines**
Enforcement	Mandatory compliance (Strictly enforced)	Recommendation (Advisory / tional)
Scope	Syntactic and structural uniformity	Semantic quality and design princi
Verification	Static analysis tools, linter rules	Peer code reviews, senior develo feedback
Flexibility	Rigid (No deviations permitted)	Flexible (Deviations allowed with ju fication)
Examples	CamelCase for variables, tab size = 4 spaces, mandatory header blocks	Avoid nested loops ¿ 3 levels, prefer mutability, modularize long functio

Code Review Techniques: Overview

Role of Code Review in Static Verification

Code review is a static testing technique where source code is examined by humans before execution to discover bugs, logic errors, and standard violations early in the SDLC.

Primary Objectives of Code Review:

- **Early Defect Detection:** Finding logic errors, unhandled boundary conditions, and memory leaks.
- **Standard Compliance:** Ensuring code follows organizational coding rules.
- **Knowledge Transfer:** Familiarizing team members with different modules.

Spectrum of Review Methods

Code reviews range from informal, peer-to-peer discussions (**Walkthroughs**) to highly formal, structured, defect-tracking processes (**Fagan Inspections**).

Code Review: Code Walkthrough

Definition of Code Walkthrough

An informal or semi-formal review technique led by the author of the code, where the author walks reviewers through the code using sample test cases.

Characteristics of Code Walkthroughs:

- **Author-Led:** The author directs the meeting and controls the narrative pace.
- **Scenario-Driven:** Uses simulated test data to trace execution paths.
- **Informal Preparation:** Minimal formal preparation required before the meeting.
- **Primary Goal:** Educate peers, gain consensus, and catch obvious logical flaws.

Limitation

Walkthroughs may lack rigor because the author's potential bias can guide reviewers away from hidden defects.

Code Review: Code Inspection (Fagan Inspection)

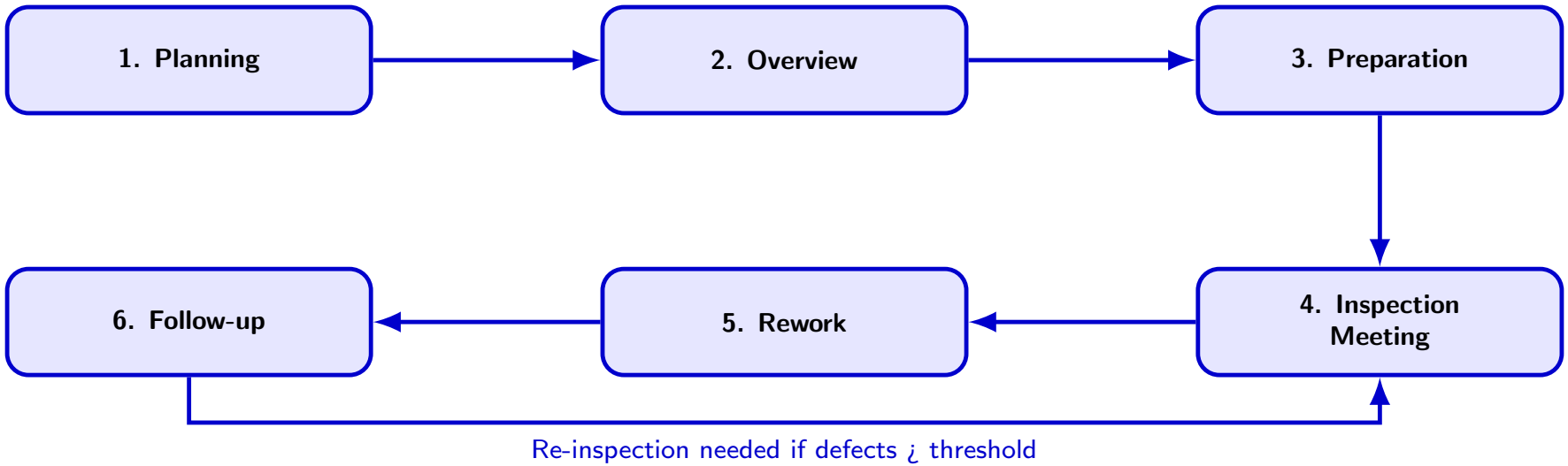
Definition of Code Inspection

A formal, highly disciplined review process invented by Michael Fagan, designed to systematically locate defects using formal checklists and role assignments.

Key Roles in Fagan Inspection:

- **Moderator:** Leads the inspection process, ensures adherence to rules, and manages meetings.
- **Author:** Developer who wrote the artifact; answers questions during review.
- **Reader:** Reads through the code line-by-line during the inspection meeting.
- **Inspector / Reviewer:** Examines code using checklists to locate defects.
- **Scribe / Recorder:** Formally records every identified defect on a bug report.

Formal Code Inspection Process Workflow



Process Highlights

Inspection metrics (defect density, inspection speed lines/hr) are logged to track process quality and decide if re-inspection is mandatory.

Code Walkthrough vs. Code Inspection

Table: Comparative Analysis: Walkthrough vs. Inspection

Feature	Code Walkthrough	Code Inspection
Formality Level	Informal to Semi-formal	Highly Formal & Structured
Meeting Leader	Code Author	Trained Independent Moderator
Roles Assigned	Flexible / Unassigned	Explicit (Moderator, Reader, Scribe, Inspector)
Preparation	Minimal pre-meeting prep	Intensive individual preparation using checklists
Data Gathering	Optional / Rare	Mandatory tracking of defect metrics and effort
Primary Goal	Code familiarization, peer feedback	Defect detection, standard adherence, quality as ance
Artifact Output	Informal list of suggestions	Formal Defect Log & Verification Sign-off

Definition of Internal Documentation

Internal documentation comprises comments, header information, and design notes embedded directly inside the source code file to assist developers and maintainers.

Components of Internal Documentation:

- **File Header Blocks:** Author name, creation date, license, module description, revision history.
- **Function Header Comments:** Purpose, input arguments (@param), return values (@return), exception types (@throws).
- **Inline / Block Comments:** Explanations for non-trivial algorithms or complex business logic.
- **Self-Documenting Code:** Meaningful variable/method naming that reduces comment reliance.

Best Practices

Document *why* code is doing something, not *what* it does (avoid stating the obvious). Keep comments synchronized with code changes.

Definition of External Documentation

Documentary artifacts produced separately from the source code to support users, system administrators, and future maintainers across the software lifecycle.

Key Types of External Documentation:

- **User Manual:** Instructions for end-users operating the application.
- **System Administrator Manual:** Deployment, environment setup, and installation steps.
- **Software Requirements Specification (SRS):** Functional & non-functional baseline requirements.
- **Software Design Document (SDD):** Architecture diagrams, module design, database schemas.

Lecture 41 Summary

Coding standards ensure consistency, code reviews (walkthroughs & inspections) guarantee quality prior to execution, and internal/external documentation ensures long-term software maintainability.

Software Testing Fundamentals

- **Definition of Software Testing:** The process of executing a program or application with the intent of finding errors, verifying that the software system meets specified requirements, and evaluating its overall quality.
- **Verification vs. Validation (V&V Model):**
  - *Verification ("Are we building the product right?"):* Static activities such as reviews, walkthroughs, and inspections to ensure code complies with design specifications.
  - *Validation ("Are we building the right product?"):* Dynamic testing execution to ensure the software satisfies customer requirements and operational needs.
- **Core Defect Terminology:**
  - **Error (Human Mistake):** A flaw in human reasoning by a software developer or designer.
  - **Fault / Defect / Bug:** A state in a software artifact resulting from an error.
  - **Failure:** An event where the system fails to perform its required function during execution.
- **Primary Objectives:** Early defect detection, risk mitigation, and establishing product confidence.

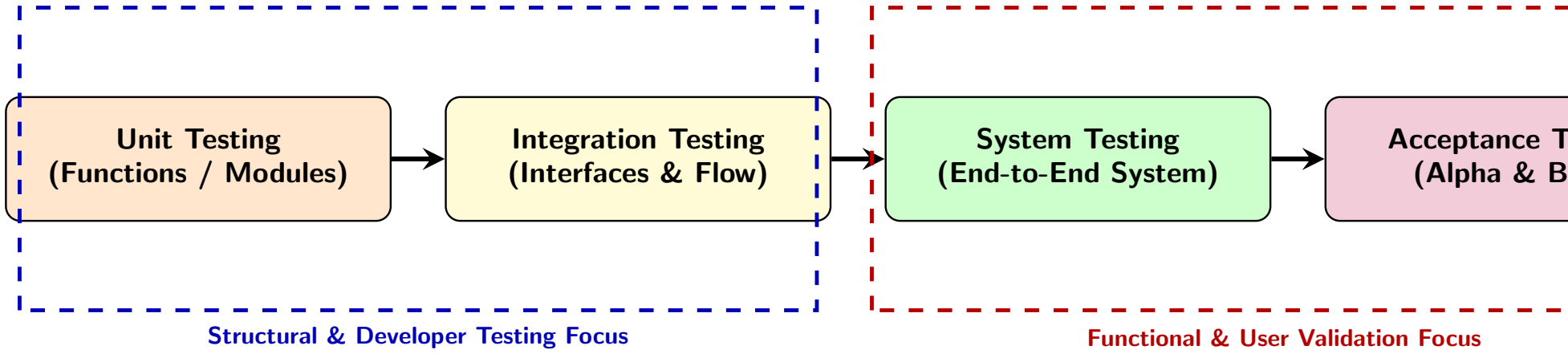
Functional Testing: Black-Box Testing

- **Concept:** A testing paradigm where the internal implementation, code structure, and logic of the software under test are unknown to the tester.
- **Key Characteristics:**
  - Evaluates system functional behavior strictly against requirement specifications.
  - Can be executed by software testers, business analysts, or end-users.
- **Major Black-Box Testing Techniques:**
  - **Equivalence Partitioning (EP):** Divides input domains into valid and invalid data classes where the program is expected to process each item within a class identically.
  - **Boundary Value Analysis (BVA):** Focuses on testing at the boundaries of equivalence partitions (min, min+, nominal, max-, max) where errors heavily concentrate.
  - **Decision Table Testing:** Maps complex logical input combinations to system actions.
  - **State Transition Testing:** Tests system behavior across state changes driven by input events.

Structural Testing: White-Box Testing

- **Concept:** A software testing method where the internal logic, code structure, data flow, and control flow paths of the application are fully visible and analyzed.
- **Key Characteristics:**
  - Requires programming proficiency and architectural understanding.
  - Identifies hidden code bugs, dead code, logic errors, and security vulnerabilities.
- **Code Coverage Criteria:**
  - **Statement Coverage:** Verifies that every executable statement in the source code is executed at least once.
  - **Branch / Decision Coverage:** Verifies that every branch (True/False outcome of conditional statements) is evaluated.
  - **Condition Coverage:** Validates that every individual condition in a compound decision evaluates to True and False.
  - **Path Coverage:** Tests all linearly independent execution paths through the control flow graph, evaluated using Cyclomatic Complexity $V(G) = E - N + 2P$.

Testing Hierarchy and Workflow



- **V-Model Alignment:** Testing phases mirror development activities, ensuring early detection of requirement and design flaws.
- **Shift-Left Paradigm:** Executing unit and integration tests early minimizes defect propagation and repair costs.

Overview of Unit and Integration Testing

- **Unit Testing:**
  - Tests individual software modules or methods in complete isolation.
  - Utilizes **Stubs** (dummy modules called by the unit under test) and **Drivers** (dummy main routines calling the unit under test).
  - Automation frameworks: JUnit, PyTest, NUnit.
- **Integration Testing:**
  - Verifies data interactions and interface integrity between integrated components.
  - **Top-Down Strategy:** Begins with main control modules, moving down the hierarchy using stubs.
  - **Bottom-Up Strategy:** Starts with low-level utility modules, moving upward using drivers.
  - **Sandwich Strategy:** Combines top-down and bottom-up approaches to test middle layers concurrently.
  - **Big Bang Approach:** Integrates all modules simultaneously (discouraged due to difficult fault localization).

Overview of Alpha and Beta Testing

- **Acceptance Testing Context:** Final phase of testing performed before product rollout to validate readiness for release.
- **Alpha Testing:**
  - Conducted at the **developer's site** by internal QA teams or select internal stakeholders.
  - Performed in a controlled environment to catch major defects prior to external release.
  - Combines both functional (black-box) and structural (white-box) inspection methods.
- **Beta Testing:**
  - Conducted at **end-user locations** by actual prospective customers.
  - Performed in an uncontrolled, real-world operational context across diverse environments.
  - Purely black-box testing focusing on usability, compatibility, and real-world robustness.
  - User feedback is collected to perform final tuning before general production release.

Comparative Analysis of Testing Methodologies

Comparison Dimension	Black-Box Testing	White-Box Testing
Primary Focus	System behavior and functional compliance	Internal code logic, structures, and paths
Knowledge Required	No source code knowledge required	Deep knowledge of programming & architecture
Tester Type	Independent QA Testers / End Users	Software Developers / Test Engineers
Key Techniques	EP, BVA, Decision Tables, State Testing	Statement, Branch, Condition, Path Coverage

Comparison Dimension	Alpha Testing	Beta Testing
Testing Environment	Developer Site (Controlled Environment)	End-User Site (Real-World Environment)
Target Audience	Internal testing staff & project team	External target users & prospective clients
Timing Phase	Pre-release phase (Before Beta)	Final pre-launch phase (Post-Alpha)

Test Documentation and Standards

- **Importance of Test Documentation:** Establishes traceability, repeatability, accountability, and standardized communication throughout the software lifecycle.
- **IEEE 829 Standard Test Artifacts:**
  - **Test Plan:** Outlines testing scope, strategy, schedule, resources, and risk management.
  - **Test Design Specification:** Details refined testing approaches and feature specifications.
  - **Test Case Specification:** Specifies inputs, execution steps, and expected outcomes.
  - **Defect / Incident Report:** Documents anomalies observed during test execution.
  - **Test Summary Report:** Evaluates testing results and overall product readiness.
- **Essential Test Case Elements:** Test Case ID, Title, Pre-conditions, Execution Steps, Input Data, Expected Results, Actual Results, Pass/Fail Status, and Severity/Priority.

Standard Test Case Specification Template

Test Case Attribute	Specification Detail
Test Case ID	TC_AUTH_042
Module / Feature	User Authentication – Login Security
Test Title	Verify account lockout after 3 consecutive failed password attempts
Pre-conditions	User account `student@gtu.ac.in` is registered and active.
Test Execution Steps	1. Open the GTU portal login screen. 2. Input valid user ID: `student@gtu.ac.in`. 3. Input an invalid password three consecutive times. 4. Click the 'Submit Login' button.
Input Data	Password = `"InvalidPass123"`
Expected Result	Account status transitions to Locked for 15 mins; error prompt displayed.
Actual Result	Account locked notification displayed correctly; access blocked.
Status & Severity	**PASS** — Severity: High — Priority: P1

Lecture 42 Summary and Key Takeaways

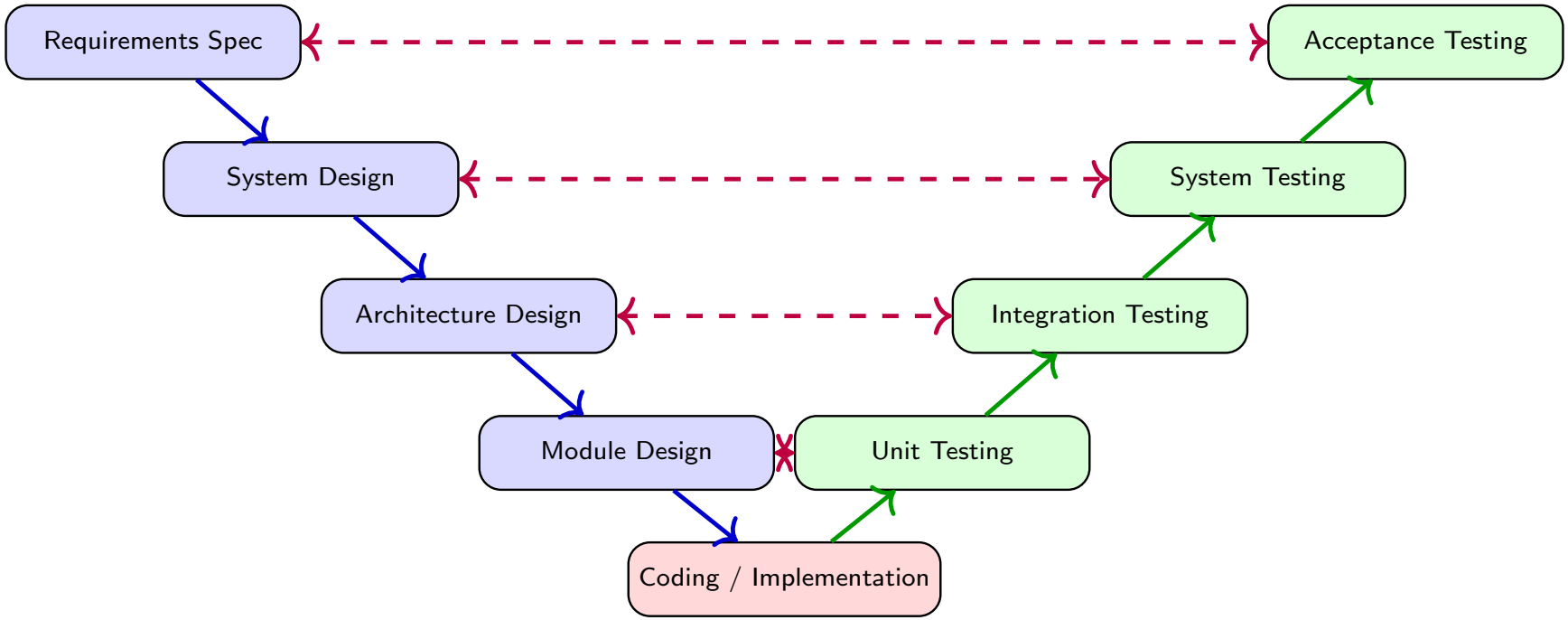
- **Core Discipline:** Software testing bridges static verification and dynamic validation to deliver defect-free software systems.
- **Dual Paradigms:**
  - *Black-Box Testing* validates functionality without code visibility.
  - *White-Box Testing* evaluates internal control structures and coverage.
- **Structured Levels:** Systematic progression from Unit Testing (isolated components) and Integration Testing (interface interactions) to Acceptance Testing (Alpha/Beta).
- **Documentation Rigor:** Utilizing standardized IEEE 829 test case templates ensures complete defect traceability and quality assurance.
- **GTU DI04000011 Examination Focus:** Master Equivalence Partitioning, Boundary Value Analysis, Cyclomatic Complexity calculations, Black-Box vs. White-Box comparison, and Test Case design.

What is Software Testing?

Software testing is the process of executing a program or application with the intent of finding errors, verifying that the software product meets specified requirements, and ensuring overall product quality.

- **Error, Fault (Bug), and Failure**:
  - **Error**: A human mistake made during coding, design, or specification.
  - **Fault / Bug**: The manifestation of an error in software code or documentation.
  - **Failure**: The inability of a software system to perform its required function during execution.
- **Key Objectives**: Defect detection, quality assurance, risk mitigation, and assessing software reliability.
- **Verification vs. Validation (V&V)**:
  - **Verification**: "Are we building the product right?" (Static analysis, reviews, inspections).
  - **Validation**: "Are we building the right product?" (Dynamic execution against requirements).

Testing Levels & V-Model Architecture



Software Testing Life Cycle (STLC) Takeaway

Testing is integrated into every phase of software development, mapping each design phase to its corresponding verification and validation level.

Concept of Black-Box Testing

Focuses on evaluating software functionality based on specifications without examining internal code structure, algorithm logic, or internal paths.

- **Equivalence Partitioning (EP)**:
  - Divides the input domain into valid and invalid data classes/partitions.
  - Test cases are chosen from each partition assuming all elements behave similarly.
- **Boundary Value Analysis (BVA)**:
  - Selects test inputs at the boundaries of equivalence partitions (min, $min+$, $nominal$, $max-$, max).
  - Defects occur predominantly at extreme input boundaries.
- **Decision Table Testing**:
  - Models complex business logic involving combinations of inputs, conditions, and actions.
- **State Transition Testing**:
  - Evaluates system behavior changes in response to input events across state changes.

Concept of White-Box (Glass-Box) Testing

Tests internal structures, control flows, data paths, and code execution logic. Requires full knowledge of source code.

- **Code Coverage Criteria**:
  - **Statement Coverage**: Ensures every executable code statement runs at least once.
  - **Branch / Decision Coverage**: Ensures every conditional outcome (True/False) evaluates at least once.
  - **Path Coverage**: Validates all independent execution paths through the control flow graph.
- **McCabe's Cyclomatic Complexity**:
  - Measures the logical complexity of source code using its Control Flow Graph (CFG).
  - Formula: $V(G) = E - N + 2P$ (where E = edges, N = nodes, P = connected components) or $V(G) = P_{dec} + 1$.
  - Defines the upper bound on the number of test cases required for basis path coverage.

Comparison: Black-Box vs. White-Box Testing

Parameter	Black-Box Testing	White-Box Testing
Focus	External functionality and user requirements	Internal logic, code structure, and control paths
Knowledge Required	No programming or code structure knowledge needed	Detailed programming and internal implementation knowledge
Performed By	Independent Software Testers, End Users	Software Developers, QA Automation Engineers
Testing Level	System and User Acceptance testing	Unit and Integration testing
Techniques	EP, BVA, Decision Tables, State Transition	Statement/Branch Coverage, Basis Path Testing
Granularity	High level (Macro perspective)	Low level (Micro perspective)

Table: Comparative Analysis of Testing Approaches

Overview of Unit Testing

Definition & Purpose

Testing individual components or modules in isolation to verify that a specific unit of source code operates correctly according to design specs.

- **Scope**: Functions, methods, classes, or modules.
- **Test Harness Components**:
  - **Driver**: A main or calling module built to pass test inputs to the Unit Under Test (UUT) and display execution results.
  - **Stub**: A dummy module called by the UUT to simulate lower-level missing or unintegrated components.
- **Test Doubles**: Mock objects, Fakes, and Spies used to isolate units from external dependencies (e.g., databases, APIs).
- **Automation Frameworks**: JUnit, PyTest, NUnit for automated test execution and regression suite building.

Overview of Integration Testing

Definition & Goal

Combines unit-tested modules and tests them as an integrated group to discover interface defects and interaction errors between components.

- **Integration Strategies**:
  - **Top-Down Integration**: Begins with main control modules. Uses *Stubs* to substitute for lower-level modules. Validates major control nodes early.
  - **Bottom-Up Integration**: Begins with leaf-level worker modules. Uses *Drivers* to invoke lower components. Useful when lower-level utility modules are completed first.
  - **Sandwich / Hybrid Integration**: Combines top-down and bottom-up approaches concurrently to test both core control and low-level modules.
  - **Big-Bang Integration**: Integrates all modules simultaneously. (High risk, difficult defect isolation).

Overview of Alpha & Beta Testing

User Acceptance Testing (UAT) Phase

Conducted to validate that the completed software system meets business requirements and satisfies user expectations prior to commercial release.

- **Alpha Testing**:
  - Conducted at the **developer's site** in a controlled lab environment.
  - Performed by internal software QA teams or selected internal staff.
  - Objective: Identify major defects, missing functionality, and performance bottlenecks before external distribution.
- **Beta Testing**:
  - Conducted at the **customer/end-user site** in an un-monitored real-world operational environment.
  - Performed by a representative sample of target end users ("pre-release" version).
  - Objective: Collect user feedback, verify real-world hardware/OS compatibility, and discover unexpected defects.

Test Documentation: Test Case Template

IEEE 829 Standard Test Case Structure

A test case specifies input parameters, execution steps, expected outcomes, and environmental conditions to test a system feature.

Field Name	Specification / Example Data
Test Case ID	TC_AUTH_001
Test Description	Verify user authentication with valid credentials
Pre-conditions	User profile exists in DB and status is set to Active
Test Steps	1. Navigate to /login 2. Enter valid Email & Password 3. Click Submit
Test Data	Email: student@gtu.edu.in, Password: GTU#Pass2026
Expected Result	System redirects user to Dashboard and displays active session
Actual Result	System successfully redirected user to Dashboard
Status	**PASS** / FAIL / BLOCKED

Lecture Summary & GTU Exam Questions

Lecture Summary

- Testing ensures software reliability through Verification and Validation.
- Functional (Black Box) tests requirements; Structural (White Box) tests code paths and logic.
- Unit testing isolates modules using Drivers and Stubs; Integration testing checks interface interactions.
- Alpha and Beta testing validate customer requirements in real environments.

GTU Exam Review Questions (Course: DI04000011)

1. Differentiate between Black-box and White-box testing with suitable examples. [7 Marks]
2. Explain Top-down and Bottom-up Integration testing strategies along with the role of Stubs and Drivers. [7 Marks]
3. Prepare an IEEE 829 standard test case document for an online GTU result portal login module. [4 Marks]

- **Definition of Software Testing**: The process of executing a program or system with the intent of finding errors, verifying that it satisfies specified requirements, and validating that it meets user needs.
- **Verification vs. Validation (V&V Model)**:
  - *Verification*: "Are we building the product right?" (Reviews, walkthroughs, inspections - static analysis).
  - *Validation*: "Are we building the right product?" (Executing code against requirements - dynamic analysis).
- **Core Objectives of Testing**:
  - Uncovering defect patterns before deployment.
  - Providing confidence in system quality and reliability metrics.
  - Preventing defect propagation across software development lifecycle (SDLC) phases.

Defect Terminology

- **Error (Mistake)**: A human action that produces an incorrect result (e.g., programming syntax error or design oversight).
- **Fault (Defect / Bug)**: Manifestation of an error in software artifacts (code, documentation).
- **Failure**: Deviation of software execution from its specified expected behavior.

Fundamental Testing Principles (Myers & Pressman)

- **Testing shows presence of defects**: Testing can show that defects are present, but cannot prove that there are no defects.
- **Exhaustive testing is impossible**: Risk analysis and priorities guide testing effort rather than testing all combinations.
- **Defect Clustering & Pesticide Paradox**: Most operational failures are found in a small number of modules (80/20 rule); repeated test suites lose effectiveness over time.

- **Concept**: Tests software functionality without internal knowledge of code structure, implementation details, or internal path logic. Focuses on inputs and expected outputs.
- **Equivalence Partitioning (EP)**:
  - Divides input domain into valid and invalid equivalence classes.
  - Assumption: System behaves identically for any representative input within a class.
  - Reduces the total number of test cases required while maintaining test coverage.
- **Boundary Value Analysis (BVA)**:
  - Complements EP by testing values at the boundaries of equivalence classes.
  - Errors tend to concentrate at boundary extremes rather than inside partitions.
  - Evaluates: Minimum, Just above minimum, Nominal, Just below maximum, and Maximum values.

Decision Table Testing

- Used for complex business logic involving combinations of conditions and corresponding actions.
- Represents conditional logic as a tabular matrix of inputs (conditions) versus outputs (actions/rules).
- Ensures complete test coverage for multi-variable boolean logic rules.

State Transition Testing

- Suitable for reactive systems or state machines where software output depends on current state and historical events.
- Analyzes valid and invalid state transitions triggered by external events/inputs.
- Verifies system robustness across state transition matrices and state diagrams.

- **Concept**: Tests internal program structure, code paths, control logic, data flows, and internal data structures. Requires source code access.
- **Control Flow Testing Levels**:
  - **Statement Coverage**: Percentage of executable statements exercised by test suite.

$$\text{Statement Coverage} = \left(\frac{\text{Number of executed statements}}{\text{Total number of executable statements}} \right) \times 100\%$$

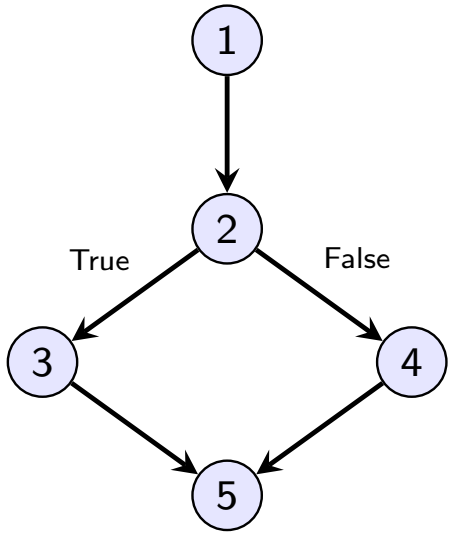
  - **Branch/Decision Coverage**: Ensures every decision outcome (TRUE and FALSE branch) is evaluated at least once.

$$\text{Branch Coverage} = \left(\frac{\text{Number of executed decision outcomes}}{\text{Total number of decision outcomes}} \right) \times 100\%$$

  - **Condition/Path Coverage**: Tests independent execution paths through the control flow graph (CFG).

- **Cyclomatic Complexity** $V(G)$: A metric measuring structural complexity of code and upper bound of independent paths.
- **Calculation Methods**:
  1. $V(G) = E - N + 2P$ (E: Edges, N: Nodes, P: Components)
  2. $V(G) = P_{pred} + 1$ (P_{pred}: Predicate nodes)
  3. $V(G) =$ Enclosed regions in CFG $+ 1$
- **Basis Set**: Minimum test set guaranteeing 100% branch coverage.

Control Flow Graph (CFG)



Unit Testing

- Focuses on individual modules, functions, or classes in isolation.
- Uses **Stubs** (dummy modules called by unit under test) and **Drivers** (dummy main routines calling unit under test).
- Typically performed by software developers using unit testing frameworks.

Integration Testing

- Tests interaction and interfaces between integrated units.
- **Top-Down Integration**: Starts from root module down; uses stubs.
- **Bottom-Up Integration**: Starts from low-level leaf components; uses drivers.
- **Sandwich (Hybrid) Integration**: Combines top-down and bottom-up approaches.
- **Big Bang Integration**: All components integrated simultaneously.

- **Acceptance Testing Context**: Final operational phase evaluating software readiness against customer business requirements.

Alpha Testing

- Conducted at **developer site**.
- Performed by internal QA teams and users in controlled environment.
- Observed by developers to record bugs and usage issues immediately.
- Performed prior to software release to client or market.

Beta Testing

- Conducted at **customer site**.
- Performed by external end-users in real-world operational environments.
- Uncontrolled environment; bugs reported periodically to developers.
- Precedes final commercial release (Release Candidate).

Table: Industry Standard Test Case Specification Format

Test Case ID	Test Description	Input Data	Expected Output	Priority
TC_LOG_001	Valid User Login Verification	User: `admin@gtu.ac.in`, Pass: `Sec#2026`	Redirect to Dashboard, Session Active	High
TC_LOG_002	Invalid Password Handling	User: `admin@gtu.ac.in`, Pass: `WrongPass`	Error: "Invalid Credentials"	High
TC_BVA_003	Age Input Boundary Test (Min)	Age = 18 (Valid Boundary)	Registration Accepted	Medium
TC_BVA_004	Age Input Boundary Test (Out)	Age = 17 (Invalid Boundary)	Error: "Must be 18 or older"	Medium

- **Key Elements**: Test ID, Module, Pre-conditions, Test Steps, Input Data, Expected vs. Actual Result, Pass/Fail Status, Execution Date, Severity.

Summary of Key Takeaways

- Testing is a multi-tiered activity combining black-box (functional) and white-box (structural) methodologies.
- Structural coverage (Basis Path, Branch) guarantees code quality, while functional methods (EP, BVA) guarantee specification adherence.
- Systematic progression from Unit $\rightarrow$ Integration $\rightarrow$ System $\rightarrow$ Alpha/Beta Acceptance ensures defect identification at lowest cost.

GTU Exam Review Questions

1. Differentiate between Equivalence Partitioning and Boundary Value Analysis with an example.
2. Calculate the Cyclomatic Complexity for a given Control Flow Graph and derive its basis paths.
3. Compare Alpha and Beta testing across testing environment, target users, and defect tracking.

Lecture 45: Software Coding and Testing

Course: GTU DI04000011 – Software Engineering
Unit: Software Coding and Testing

Lecture Objectives

- Understand software testing fundamentals, verification vs. validation.
- Explore Black-Box (Functional) and White-Box (Structural) testing techniques.
- Analyze test coverage criteria and McCabe's Cyclomatic Complexity.
- Master Unit, Integration, Alpha, and Beta testing strategies.
- Learn standardized Test Documentation and Test Case Template structure.

Definitions & Terminology

- **Error (Mistake):** A human action that produces an incorrect result (e.g., logic or syntax error).
- **Fault (Bug/Defect):** A flaw in software that can cause a component to fail in execution.
- **Failure:** Inability of a software system to perform its required functions within specified limits.

Verification ("Right Product")

- Static testing activities (Reviews, Walkthroughs, Inspections).
- Assures software conforms to specifications.

Validation ("Product Right")

- Dynamic testing activities (Executing code with test inputs).
- Assures software meets customer needs.

Functional Testing: Black-Box Testing

Core Concept

Focuses on software functional requirements without knowledge of internal code structure, logic paths, or implementation details.

Key Black-Box Techniques

- **Equivalence Partitioning (EP):**
  - Divides the input domain into valid and invalid equivalence classes.
  - Selects one representative value from each class to minimize redundant test cases.
- **Boundary Value Analysis (BVA):**
  - Focuses on values at boundaries where defects commonly concentrate.
  - For an input range $[a, b]$, test values include: $a - 1, a, a + 1, b - 1, b, b + 1$.

Advanced Black-Box Testing Techniques

Decision Table Testing

Represents complex business rules by mapping combinations of input conditions to corresponding system actions in a tabular matrix format.

Cause-Effect Graphing

Visual mapping technique that links input conditions (causes) to system outputs (effects) using boolean logical operators (AND, OR, NOT) to generate minimal test suites.

State Transition Testing

Validates system behavior across distinct states, state transitions, inputs, and outputs using State Transition Diagrams (STD), highly effective for event-driven systems.

Structural Testing: White-Box Testing

Core Concept

Examines internal program structures, control paths, logic gates, and code statements. Requires complete visibility into source code architecture.

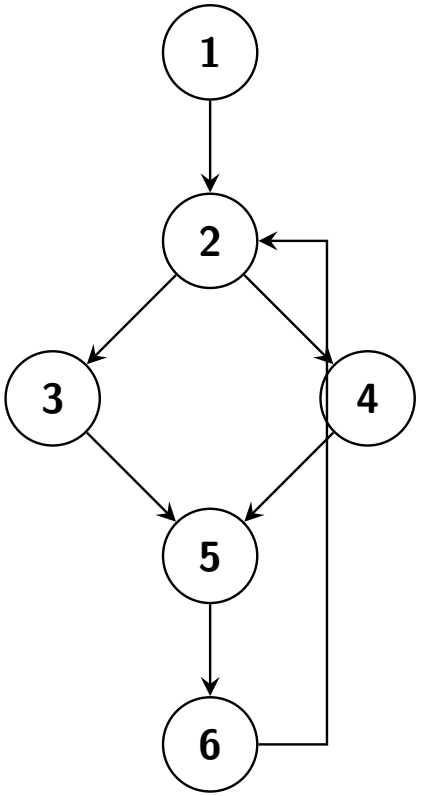
Basis Path Testing & Cyclomatic Complexity

- Developed by Thomas McCabe to measure software logical complexity.
- Computes the upper bound on the number of linearly independent paths.
- **Calculation Formulas for $V(G)$:**
  1. $V(G) = E - N + 2P$ (where E = Edges, N = Nodes, P = Connected Components)
  2. $V(G) = D + 1$ (where D = Decision Nodes)
  3. $V(G) = R$ (where R = Bounded and Unbounded Regions)

White-Box Coverage & Control Flow Graph

Coverage Criteria Hierarchy

- **Statement Coverage:** $\frac{\text{Executed Statements}}{\text{Total Statements}} \times 100\%$
- **Branch/Decision Coverage:** Evaluates both True and False outcomes of every decision node.
- **Condition Coverage:** Evaluates individual boolean sub-expressions independently.
- **Path Coverage:** Validates all possible execution paths through the Control Flow Graph.



Control Flow Graph (CFG)

$E = 7, N = 6 \implies V(G) = 7 - 6 + 2 = 3$

Levels of Testing: Unit & Integration

Unit Testing

- Verifies individual modules, components, or functions in isolation.
- Typically performed by software developers using frameworks like JUnit, PyTest, or NUnit.
- Focuses on internal algorithm logic, local data structures, and error handling paths.

Integration Testing

- Validates interaction, interfaces, and data communication between integrated units.
- Detects interface mismatches, parameter corruption, and communication protocol flaws.

Integration Testing Strategies

Top-Down Integration

Modules integrated moving down control hierarchy. Uses **Stubs** (dummy modules simulating called low-level subroutines).

Bottom-Up Integration

Atomic low-level modules integrated first. Uses **Drivers** (dummy modules simulating calling high-level components).

Sandwich / Hybrid Integration

Combines Top-Down (for control logic) and Bottom-Up (for reusable utility components).

Big-Bang Integration

All modules integrated simultaneously; difficult to isolate root cause of failures.

Acceptance Testing: Alpha & Beta Testing

Overview of Acceptance Testing

Evaluates system compliance with business requirements and determines operational release readiness.

Attribute	**Alpha Testing**	**Beta Testing**
Location	Developer's Site	End-User / Customer Site
Environment	Controlled environment	Real-world operational environment
Testers	Internal QA & selected end-users	External target user community
Timing	Pre-release internal phase	Final field testing phase
Primary Goal	Identify crash bugs & baseline quality	Evaluate usability, compatibility, & load

Test Documentation: Test Case Template

IEEE 829 Standardized Test Case Artifact

Standardized test cases document test specifications, inputs, and operational criteria to ensure test execution repeatability.

Field Name	Description / Concrete Example
Test Case ID	`TC_AUTH_001` (Unique Identifier)
Test Title	Verify User Authentication with Valid Credentials
Pre-conditions	Database connected; User account `student@gtu.ac.in` exists.
Test Steps	1. Open `/login`. 2. Enter valid email & password. 3. Click Login.
Test Data	Email: `student@gtu.ac.in`, Pass: `GTU@2026`
Expected Result	Redirect to Dashboard, display "Welcome Student".
Actual Result	Successfully redirected to Dashboard with correct banner.
Status / Priority	PASS / High (Critical Path Functionality)