Software Engineering (DI05032011)

Complete Lecture Deck – All 45 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

Course Agenda I

1. Lecture 1.1: Software, Software Application Domain
2. Lecture 1.2: Layered Approach & Products
3. Lecture 1.3: Software process and software engineering methods
4. Lecture 1.4: Generic Framework Activity & Umbrella Activity
5. Lecture 2.1: SDLC Intro
6. Lecture 2.2: Waterfall Model
7. Lecture 2.3: Incremental Model

Course Agenda II

Course Agenda III

15. Lecture 3.4: Req Engineering
16. Software Requirement Specification (SRS): Functional Requirements
17. Lecture 3.6: Non-Functional Req.
18. Lecture 3.7: Good Software Design
19. Lecture 3.8: Analysis v/s Design
20. Lecture 3.9: Classification of Cohesion
21. Lecture 3.10: Classification of Coupling

Course Agenda IV

Course Agenda V

Course Agenda VI

Course Agenda VII

Lecture Agenda

- 1. The Formal Definition of Software
- 2. The Dual Role of Software
- 3. Software Characteristics (Hardware vs. Software)
- 4. The Failure Curve (Bathtub vs. Software Curve)
- 5. Seven Broad Categories of Software Application Domains

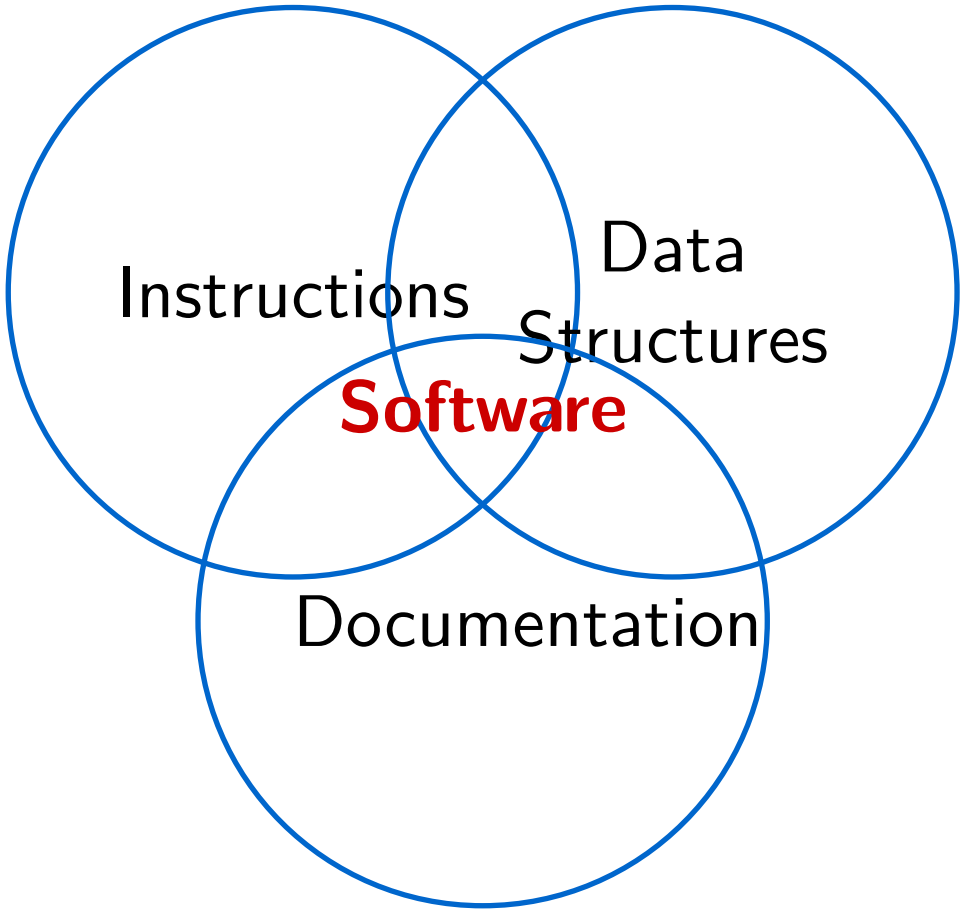
What is Software? (Beyond Code)

Software is a collection of:

1. Instructions (computer programs) that when executed provide desired features, function, and performance.
2. Data structures that enable the programs to adequately manipulate information.
3. Documentation that describes the operation and use of the programs.

Equation

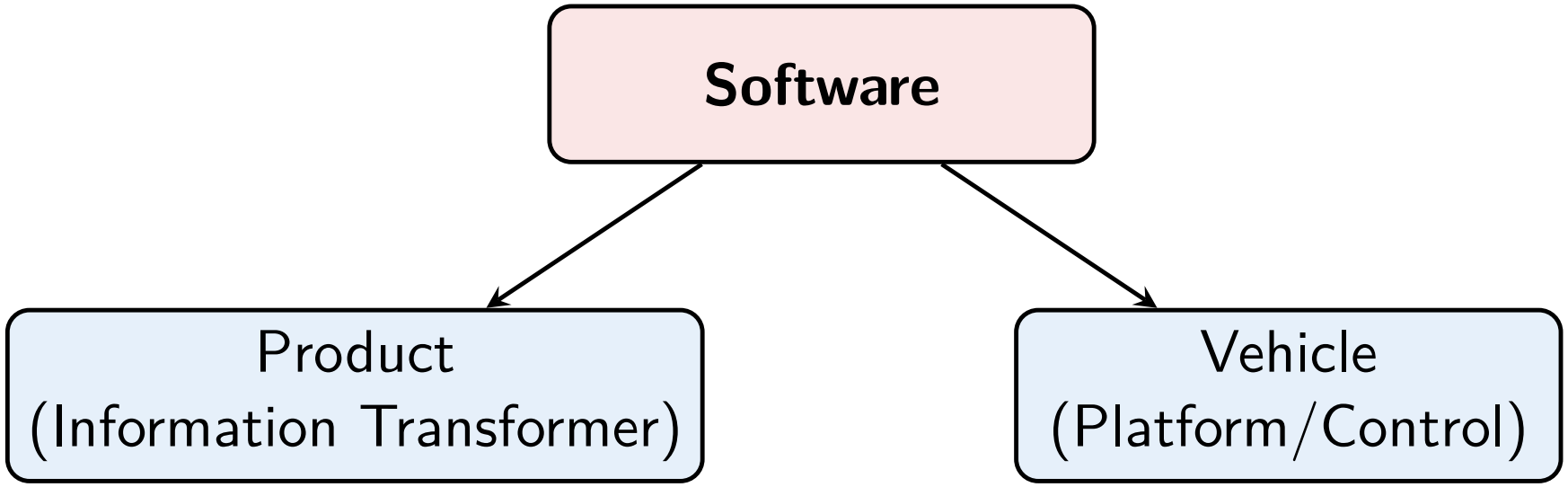
Software = Program + Data + Documentation



The Dual Role of Software

Software acts as both a Product and a Vehicle for delivering a product.

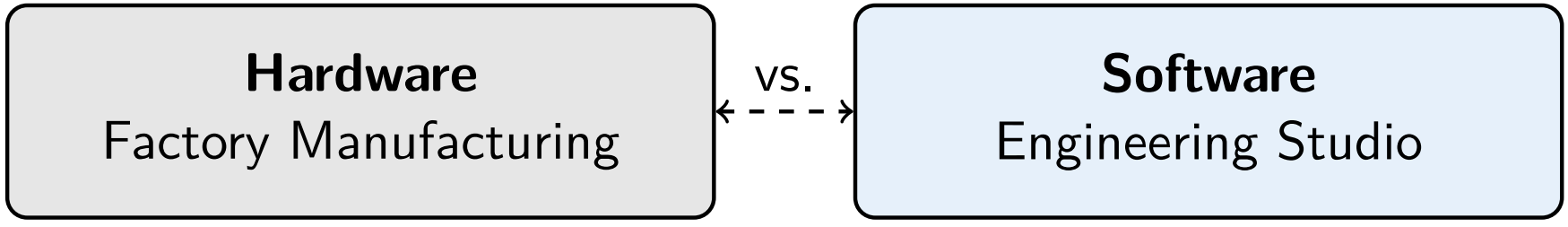
- **As a Product:** Delivers computing potential, transforms data (e.g., word processors, financial software).
- **As a Vehicle:** Acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation/control of other programs (IDEs, compilers).



Defining Software Characteristics

Unlike physical hardware, software has distinct characteristics that define its engineering process:

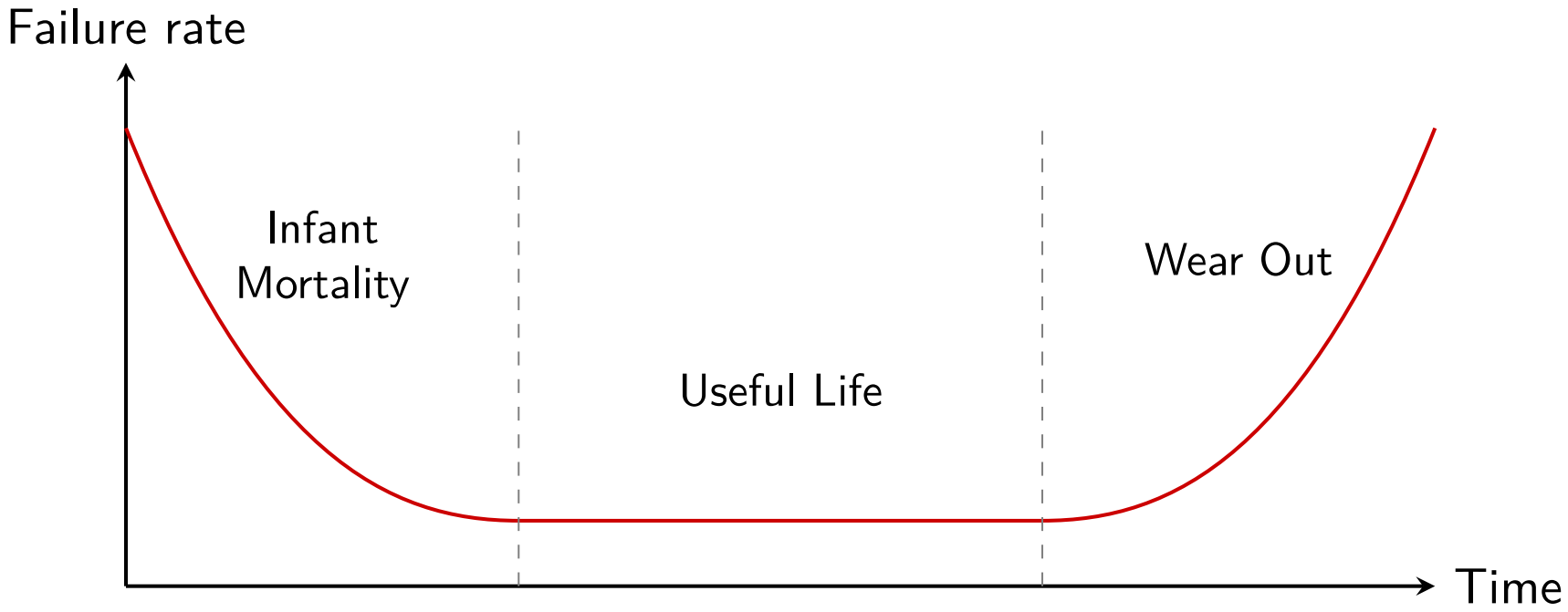
1. Software is developed or engineered, it is not manufactured in the classical sense.
2. Software doesn't 'wear out' (but it does deteriorate).
3. Although the industry is moving toward component-based construction, most software continues to be custom-built.



The Hardware Failure Curve (Bathtub Curve)

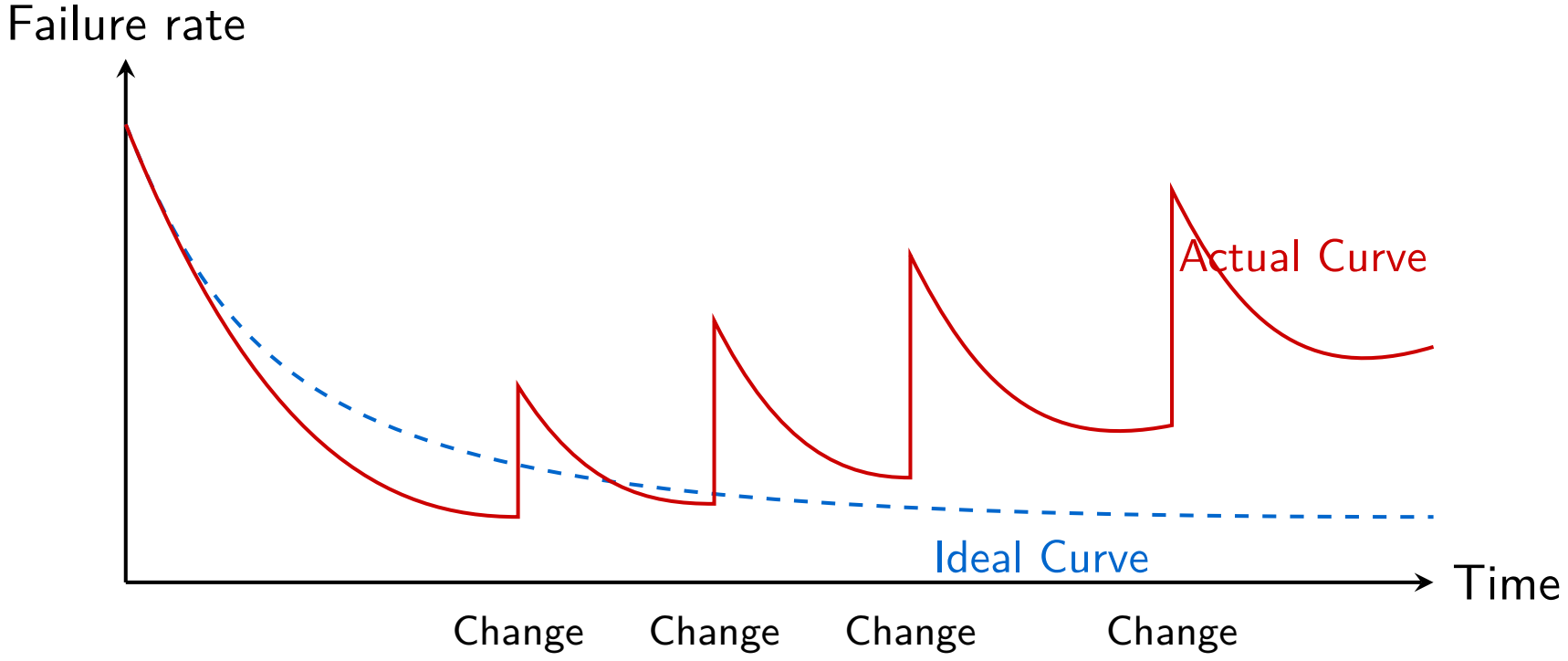
Hardware exhibits a 'bathtub curve' for failure rates over time.

- **Infant Mortality Phase:** High early failure rate due to design or manufacturing defects.
- **Useful Life Phase:** Steady-state, low failure rate.
- **Wear Out Phase:** Failure rate spikes sharply due to physical wear, environmental effects (dust, temperature, vibration).



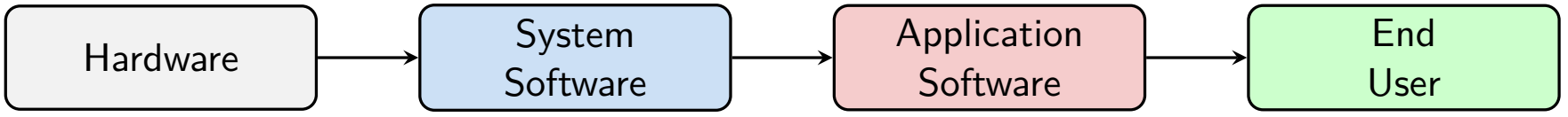
The Software Failure Curve

- Software idealized curve: Starts high (undiscovered bugs), drops asymptotically to a steady state. It does not wear out.
- Actual software curve: Features 'spikes' in failure rate during its life.
- Why spikes? Every time software is updated or modified (maintenance), new defects are introduced.
- Result: Software **deteriorates** rather than wears out due to entropy introduced by continuous changes.



Software Application Domains - Part 1

1. **System Software:** Programs written to service other programs. Characterized by heavy interaction with hardware, multiple concurrent users, complex data sharing (e.g., OS, compilers, drivers).
2. **Application Software:** Stand-alone programs that solve a specific business need. Process business or technical data to facilitate operations (e.g., point-of-sale systems, ERPs).



Software Application Domains - Part 2

3. **Engineering/Scientific Software:** Characterized by 'number crunching' algorithms. Includes astronomy, volcanology, automotive stress analysis, and orbital dynamics (e.g., CAD/CAM, MATLAB toolboxes).
4. **Embedded Software:** Resides within a product or system and is used to implement and control features and functions for the end user and the system itself (e.g., microwave keypad control, automobile fuel injection control).

Scientific Software
Heavy Math, Algorithms

Embedded Software
Hardware Integration, Real-Time

Software Application Domains - Part 3

5. **Product-Line Software:** Designed to provide a specific capability for use by many different customers (e.g., MS Office, inventory control products).
6. **Web Applications (WebApps):** Network-centric software category spanning browser-based apps to complex SOA (Service-Oriented Architectures) and cloud ecosystems.
7. **Artificial Intelligence Software:** Makes use of non-numerical algorithms to solve complex problems not amenable to computation or straightforward analysis (e.g., robotics, expert systems, LLMs).

Product-Line (e.g., Office Suites)	WebApps (e.g., Cloud Systems)	AI Software (e.g., Neural Networks)

Summary

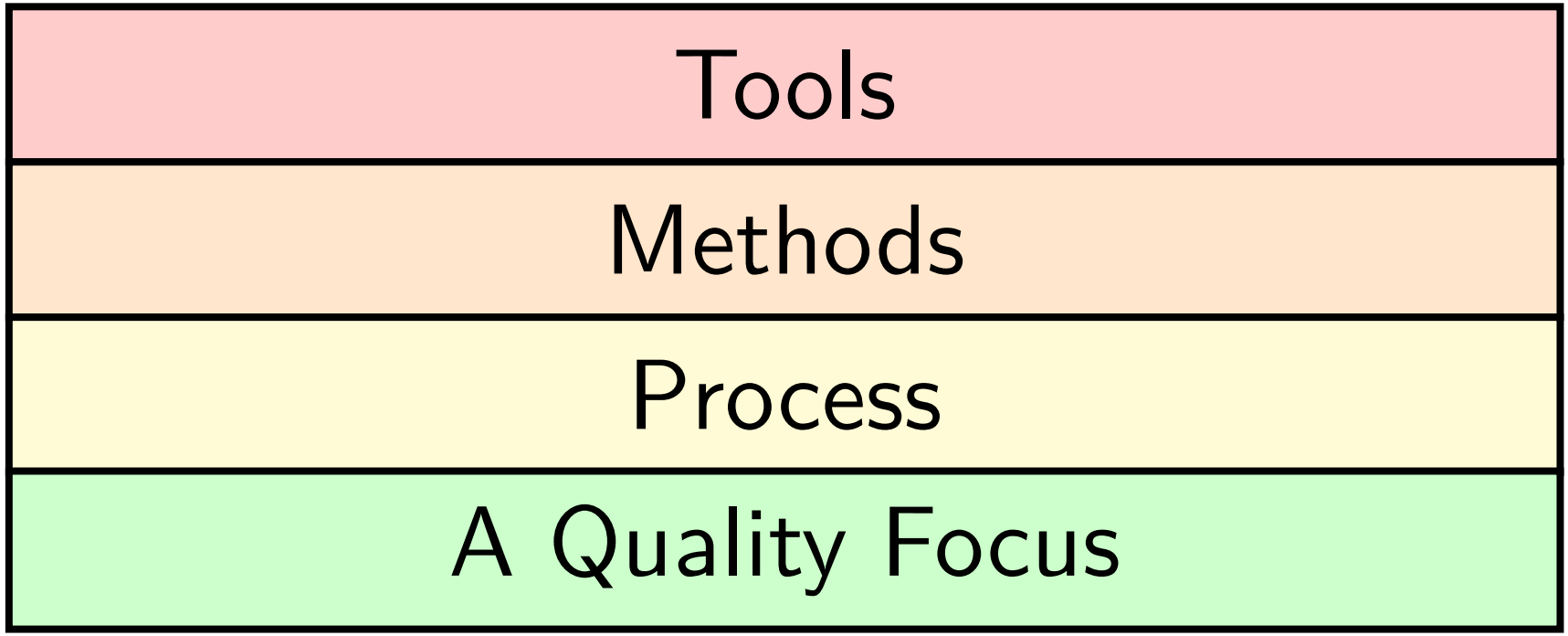
- Software is Program + Data Structures + Documentation.
- Software acts as both the product and the vehicle for delivering products.
- Software does not physically wear out like hardware; it deteriorates due to changes and maintenance.
- Application domains vary wildly in constraints (e.g., Embedded requires real-time efficiency, AI requires heuristic processing).

Next Lecture Preview

Topic: Software Engineering - A Layered Approach & Programs vs. Software Products

Key questions to ponder:

- Why isn't every program considered a software product?
- What are the distinct layers that make up the discipline of software engineering?



Questions?

Recap & Agenda

Recap: We defined software as programs + data + documentation and explored how software deteriorates rather than wears out.

Agenda:

- Program vs. Software Product
- Cost and Effort Multipliers
- Attributes of Good Software
- Formal Definition of Software Engineering (IEEE)
- The Layered Technology Approach

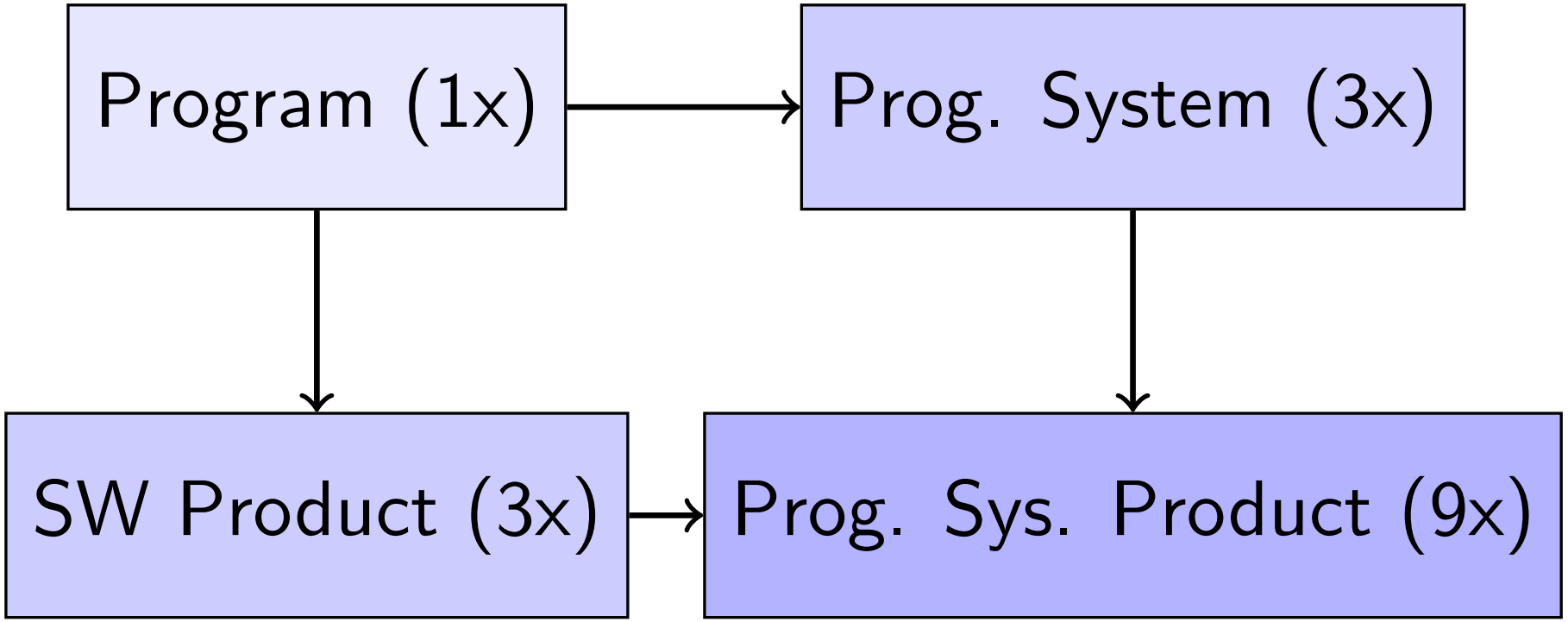
Program vs. Software Product

Program	**Software Product**
Usually developed by a single developer for personal use.	Developed by multiple engineers for multiple users/customers.
Lacks comprehensive documentation and formalized data structures.	Requires extensive documentation (User manuals, System design, API docs).
Rarely subjected to rigorous testing or QA.	Rigorous testing, version control, and maintenance plans.

Cost and Effort Multipliers

Converting a Program into a Software Product increases effort exponentially.

- **Programming System** (Integration): Requires 3x effort.
- **Software Product** (Generalization): Requires 3x effort.
- **Programming Systems Product**: 9x the cost and effort.



Attributes of Good Software

A software product must exhibit key quality attributes:

1. **Maintainability:** Must evolve to meet changing needs.
2. **Dependability & Security:** Reliability, security, and safety.
3. **Efficiency:** Should not waste system resources.
4. **Acceptability:** Understandable, usable, and compatible.

What is Software Engineering?

IEEE Definition (Standard 610.12)

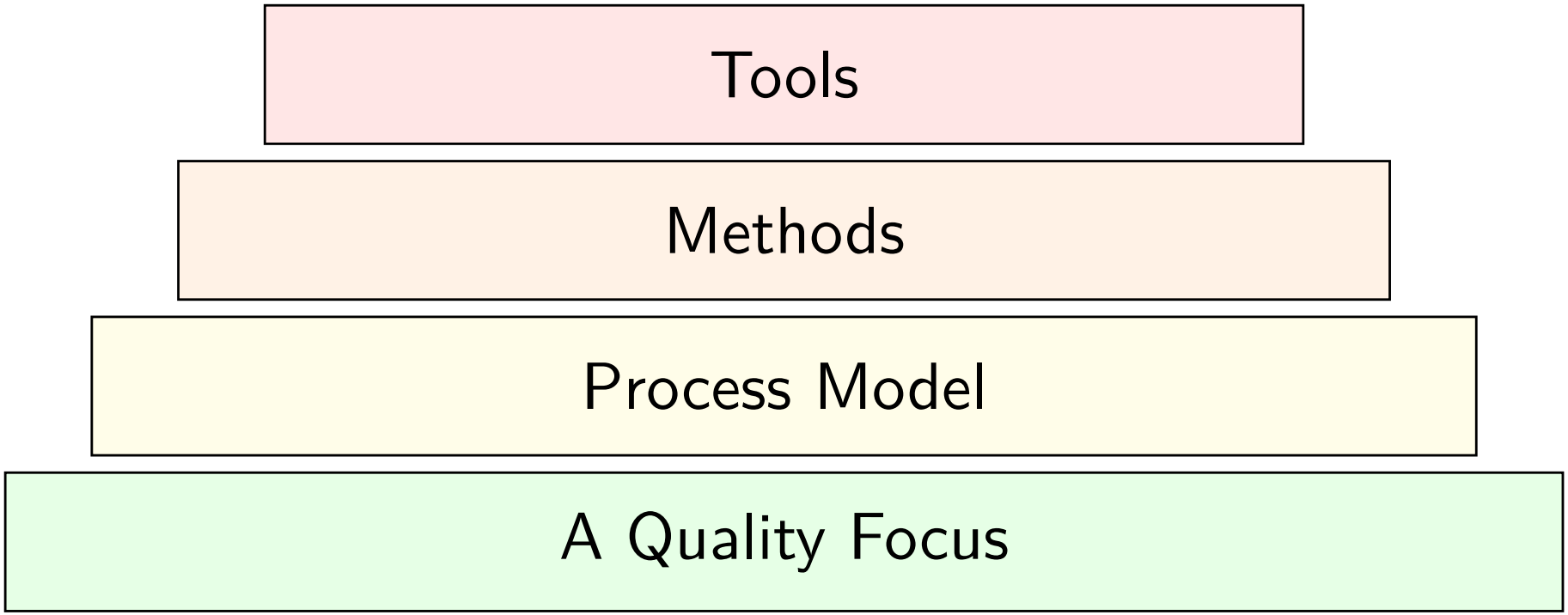
1. The application of a **systematic**, **disciplined**, **quantifiable** approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
2. The study of approaches as in (1).

Key words

Systematic, Disciplined, Quantifiable.

Software Engineering: A Layered Technology

Software engineering is a layered technology.



Layer 1: A Quality Focus

- The bedrock that supports software engineering.
- Fosters a continuous process improvement culture (e.g., Six Sigma, TQM).
- Ensures that the final software meets all functional and non-functional requirements.
- Without organizational commitment to quality, process models devolve into bureaucracy.

Layer 2: Process

- The foundation for software engineering.
- It is the 'glue' that holds the technology layers together.
- Defines a framework for a set of Key Process Areas (KPAs).
- Establishes context in which technical methods are applied, work products (models, documents, data) are produced, milestones are established, and quality is ensured.

Layer 3 & 4: Methods and Tools

Methods:

- Provide the technical 'how-to's' for building software.
- Encompasses tasks: communication, requirements analysis, design modeling, program construction, testing, and support.

Tools:

- Provide automated or semi-automated support for the process and the methods.
- When tools are integrated so that information created by one tool can be used by another, a system for the support of software development (CASE) is established.

Summary

- A Software Product requires exponentially more effort than a simple program due to documentation, generalization, and QA.
- IEEE defines SE as a systematic, disciplined, quantifiable approach.
- SE relies on 4 layers: Quality Focus (bedrock), Process (framework), Methods (technical how-to), and Tools (automation).

Next Lecture Preview

Topic: Software Process and Software Engineering Methods

We will explore:

- What exactly constitutes a 'Process Framework'?
- The 5 fundamental framework activities.
- Umbrella activities that span the entire lifecycle.

Questions?

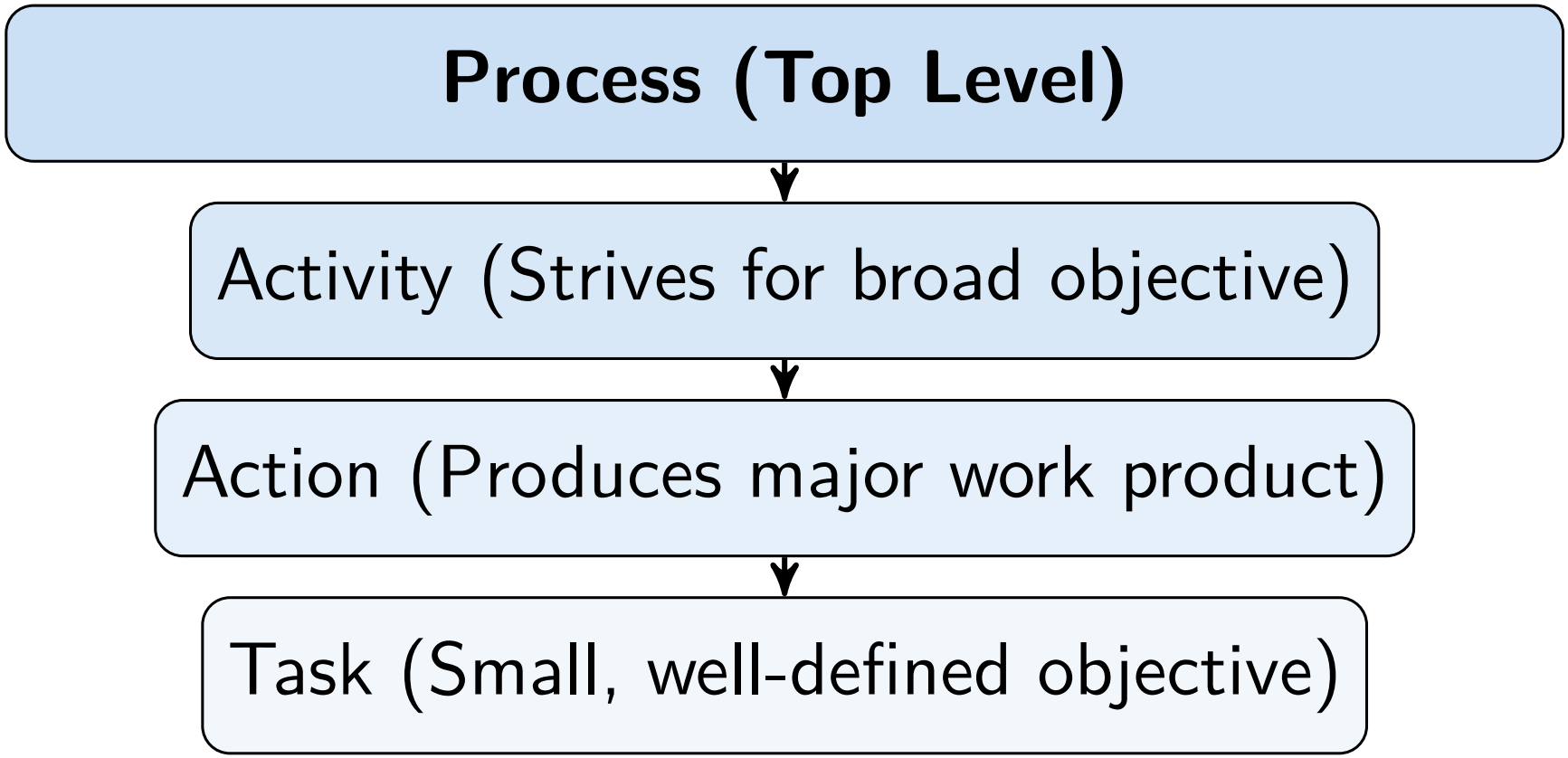
Lecture Agenda

- **Recap:** We explored the IEEE definition of Software Engineering and the 4-layer technology approach (Quality, Process, Methods, Tools).
- 1. What is a Software Process?
- 2. The Generic Process Framework
- 3. The 5 Framework Activities
- 4. Umbrella Activities (Quality, Tracking, SCM)
- 5. SE Methods Applied across the Framework

Defining the Software Process

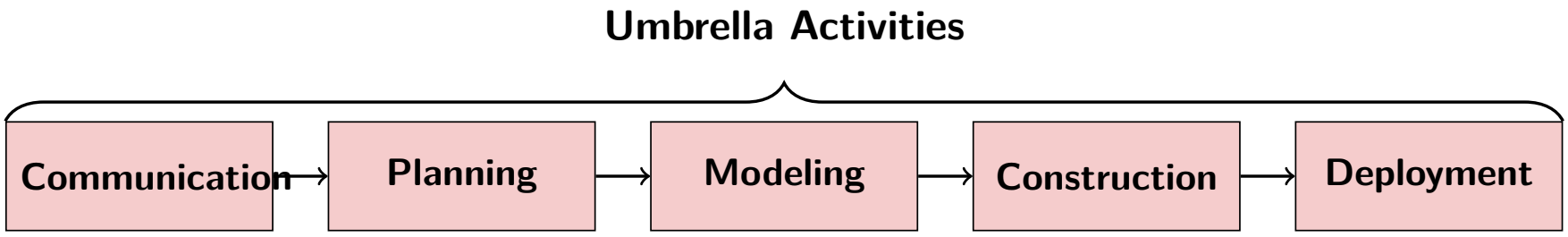
Software Process

A process is a collection of activities, actions, and tasks that are performed when some work product is to be created.



The Generic Process Framework

- Five generic framework activities apply to all software projects:



Framework Activities: Communication & Planning

Communication

- Involves heavy collaboration and discussion with customers and stakeholders.
- **Goal:** Understand objectives, gather requirements, define features.

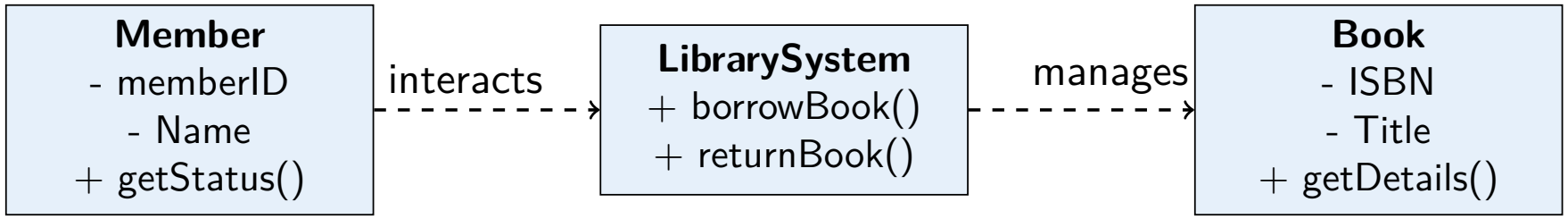
Planning

- Establishes a 'map' for the journey.
- Defines technical tasks, risks, resources, work products, and schedule (e.g., Gantt Charts).

Framework Activities: Modeling

3. Modeling

- Creation of models to better understand requirements and the design that will achieve those requirements.
- Divided into two main actions: **Analysis** (understanding the domain) and **Design** (architecting the solution).
- Common tools: UML diagrams, Data Flow Diagrams (DFDs), ER Diagrams.



Framework Activities: Construction & Deployment

Construction

- Combines code generation (either manual or automated) and the testing that is required to uncover errors.
- Includes Unit Testing and Integration Testing.

Deployment

- Software is delivered to the customer (complete or partially).
- Customer evaluates the delivered product and provides feedback.

Umbrella Activities - Part 1

Umbrella activities occur continuously across the entire software process.

- **1. Software Project Tracking and Control:** Assessing progress against the plan and taking corrective action.
- **2. Risk Management:** Assessing risks (technical, schedule, business) and mitigating them.
- **3. Software Quality Assurance (SQA):** Defining and conducting activities required to ensure software quality.

Umbrella Activities - Part 2

- **4. Technical Reviews:** Assessing SE work products to uncover and remove errors before they propagate.
- **5. Measurement:** Process, project, and product metrics.
- **6. Software Configuration Management (SCM):** Managing the effects of change throughout the software process (e.g., using Git).
- **7. Reusability Management:** Criteria for component reuse.

Activity	**Common Tools**
Software Configuration Management	Git, SVN, Mercurial
Measurement & Quality Assurance	SonarQube, JIRA
Technical Reviews	Gerrit, GitHub Pull Requests

Software Engineering Methods

Methods provide the technical details for building software.

Framework Phase	**Specific Methods**
Communication / Planning	Use Case modeling, requirement elicitation techniques
Modeling (Analysis & Design)	Object-Oriented Design (OOD), Structural Design, UI/UX
Construction	Test-Driven Development (TDD), Pair Programming
Testing (Part of Construction)	Black-box testing, White-box testing

Summary

- **Process Structure:** A process defines the framework. Activities, Actions, and Tasks break down the work.
- **The 5 Activities:** Communication, Planning, Modeling, Construction, Deployment.
- **Umbrella Activities:** Run concurrently throughout all phases (e.g., SCM, SQA, Risk Management).
- **Methods:** Provide the specific technical implementations (e.g., OOD, TDD) to achieve the framework's goals.

Next Lecture Preview

Topic: Software Process Models (Prescriptive Models)

We will dive into specific process models:

- The Classic Waterfall Model.
- Incremental Process Models.
- Evolutionary Models (Prototyping, Spiral).

Questions?

Recap

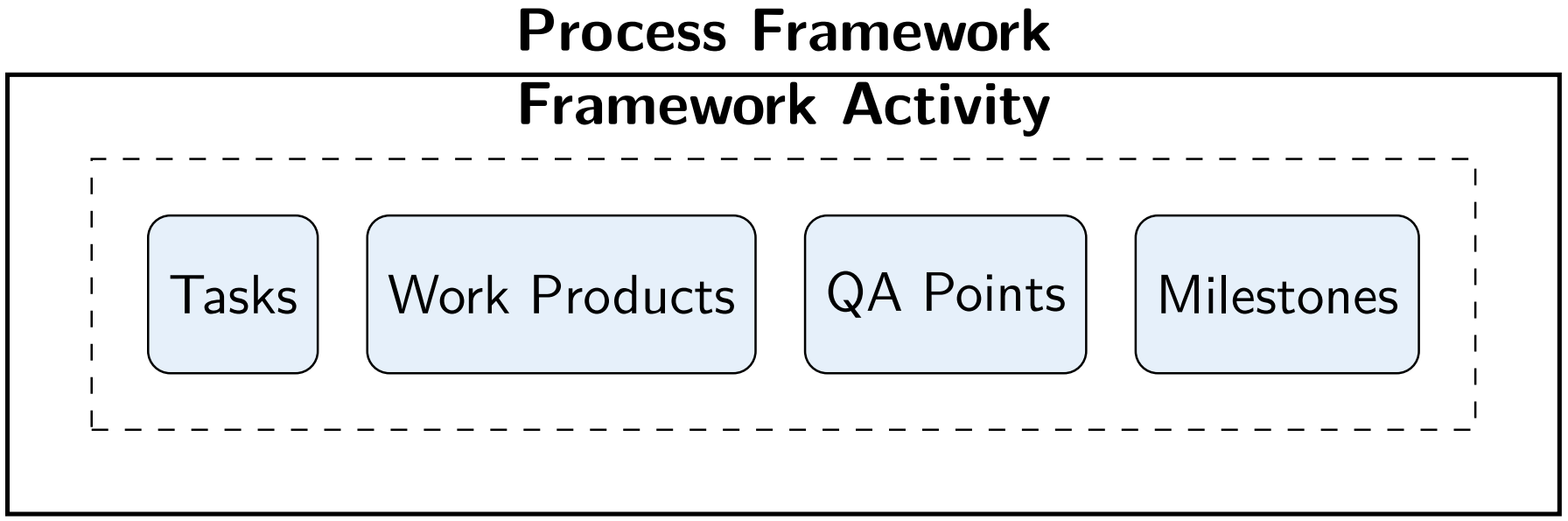
- Review of Software Characteristics
- Changing Nature of Software
- Software Myths

Agenda

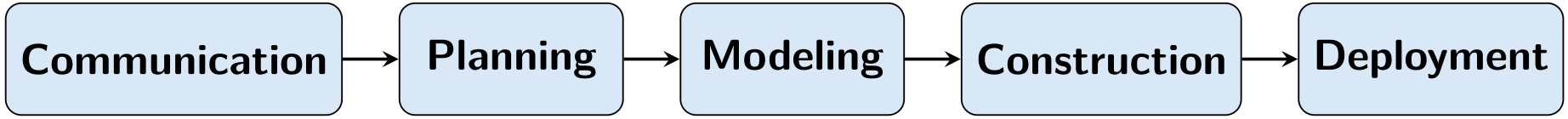
1. Concept of a Software Process Framework
2. Five Generic Framework Activities
3. Deep Dive into Communication, Planning, and Modeling
4. Deep Dive into Construction and Deployment
5. Introduction to Umbrella Activities
6. Key Umbrella Activities in Detail

What is a Process Framework?

- Establishes the foundation for a complete software engineering process.
- Identifies a small number of framework activities applicable to all software projects.
- Independent of the specific process model (e.g., Agile, Waterfall) being used.
- Encompasses framework activities, set of tasks, work products, quality assurance points, and project milestones.



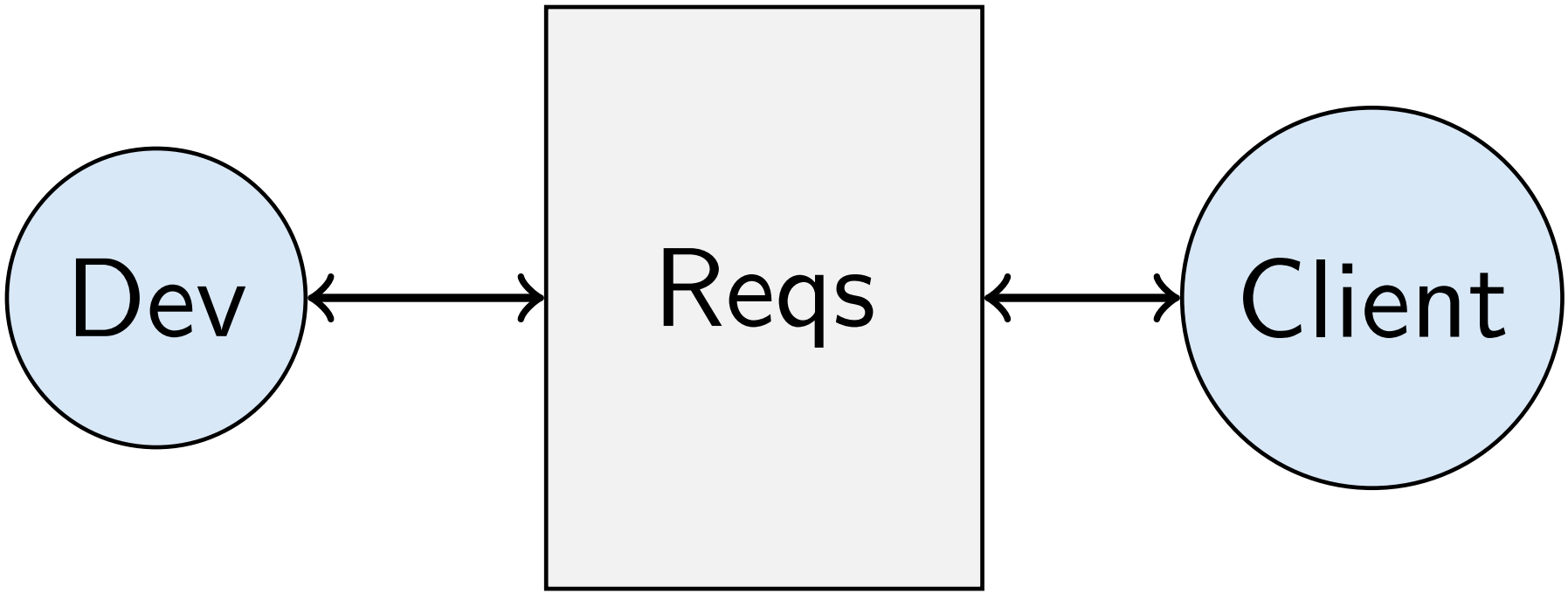
The Five Generic Framework Activities



1. Communication
2. Planning
3. Modeling
4. Construction
5. Deployment

Framework Activity: Communication

- Involves heavy collaboration and communication with the customer.
- **Goal**: Understand stakeholders' objectives for the project.
- **Key output**: Requirement gathering and defining project scope.
- **Techniques used**: Interviews, surveys, use case derivation.



Framework Activity: Planning

- Creates a 'map' for the software engineering journey.
- Defines the software project plan.
- **Key tasks**: Estimating technical tasks, assessing risks, defining resources required.
- **Output**: A schedule (e.g., Gantt chart) tracking work tasks and milestones.

Framework Activity: Modeling

- Creation of models to better understand requirements and design.
- **Analysis modeling**: What is required (Data, Control, Functional).
- **Design modeling**: How it will be achieved (Architecture, User Interface, Component level).
- Translates requirements into a software blueprint.

Framework Activity: Construction

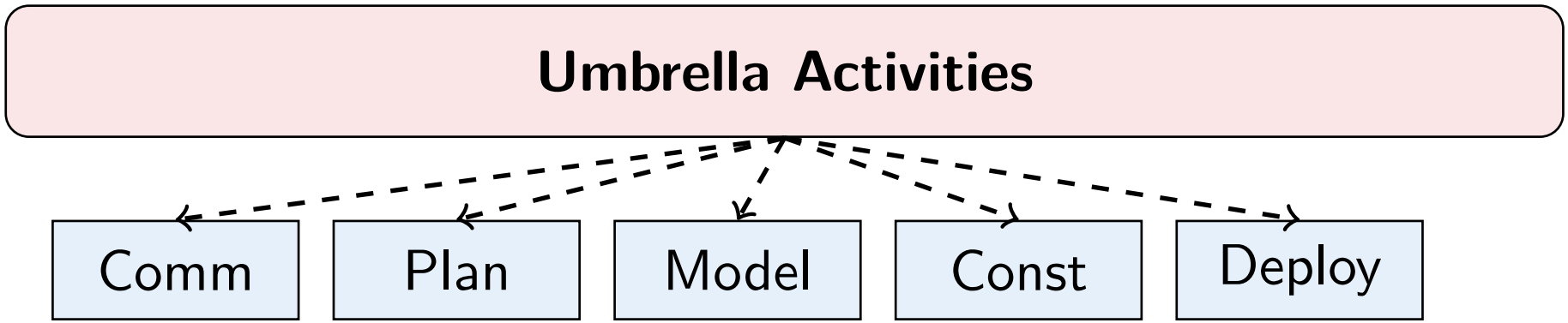
- Combines code generation and testing.
- **Code generation**: Manual coding, automated generation, or using reusable components.
- **Testing**: Uncovering errors in the code. Includes unit testing, integration testing, and system testing.
- Ensures the code matches the design and requirements.

Framework Activity: Deployment

- Delivering the software (as a complete entity or a partially complete increment) to the customer.
- Customer evaluates the delivered product.
- Provides crucial feedback for iterative refinement.
- Includes installation, user training, and initial support.

What are Umbrella Activities?

- Activities that occur throughout the software process.
- They do not belong to one specific phase; they span across all framework activities.
- **Purpose**: Manage, control, and ensure the quality of the software project.
- Operate in the background to support the primary engineering tasks.



Key Umbrella Activities (Part 1)

1. **Software Project Tracking and Control**: Assessing progress against the plan and taking corrective action.
2. **Risk Management**: Assessing risks that may affect outcomes and mitigating them.
3. **Software Quality Assurance (SQA)**: Defining and conducting activities to ensure software quality.

Key Umbrella Activities (Part 2)

4. **Technical Reviews**: Assessing engineering work products to uncover and remove errors before they propagate.
5. **Measurement**: Defining and collecting process, project, and product metrics.
6. **Software Configuration Management (SCM)**: Managing the effects of change throughout the software process.

Key Umbrella Activities (Part 3)

7. **Reusability Management**: Defining criteria for component reuse and establishing reusable component libraries.
8. **Work Product Preparation and Production**: Encompassing activities required to create models, documents, logs, forms, and lists.

Comparing Framework vs. Umbrella Activities

- **Framework Activities**: Sequential or iterative phases (e.g., First communicate, then plan, then model). Direct creation of the product.
- **Umbrella Activities**: Continuous, parallel activities running concurrently with framework activities.
- **Example**: Testing is Construction (Framework), but SQA auditing the testing process is Umbrella.
- **Example**: Writing code is Construction (Framework), but committing code to Git is Configuration Management (Umbrella).

Summary

- A process framework provides five generic activities: Communication, Planning, Modeling, Construction, and Deployment.
- These generic activities are applicable regardless of the specific process model.
- Umbrella activities span the entire process to manage, control, and ensure quality.
- Key umbrella activities include SCM, SQA, Risk Management, and Technical Reviews.

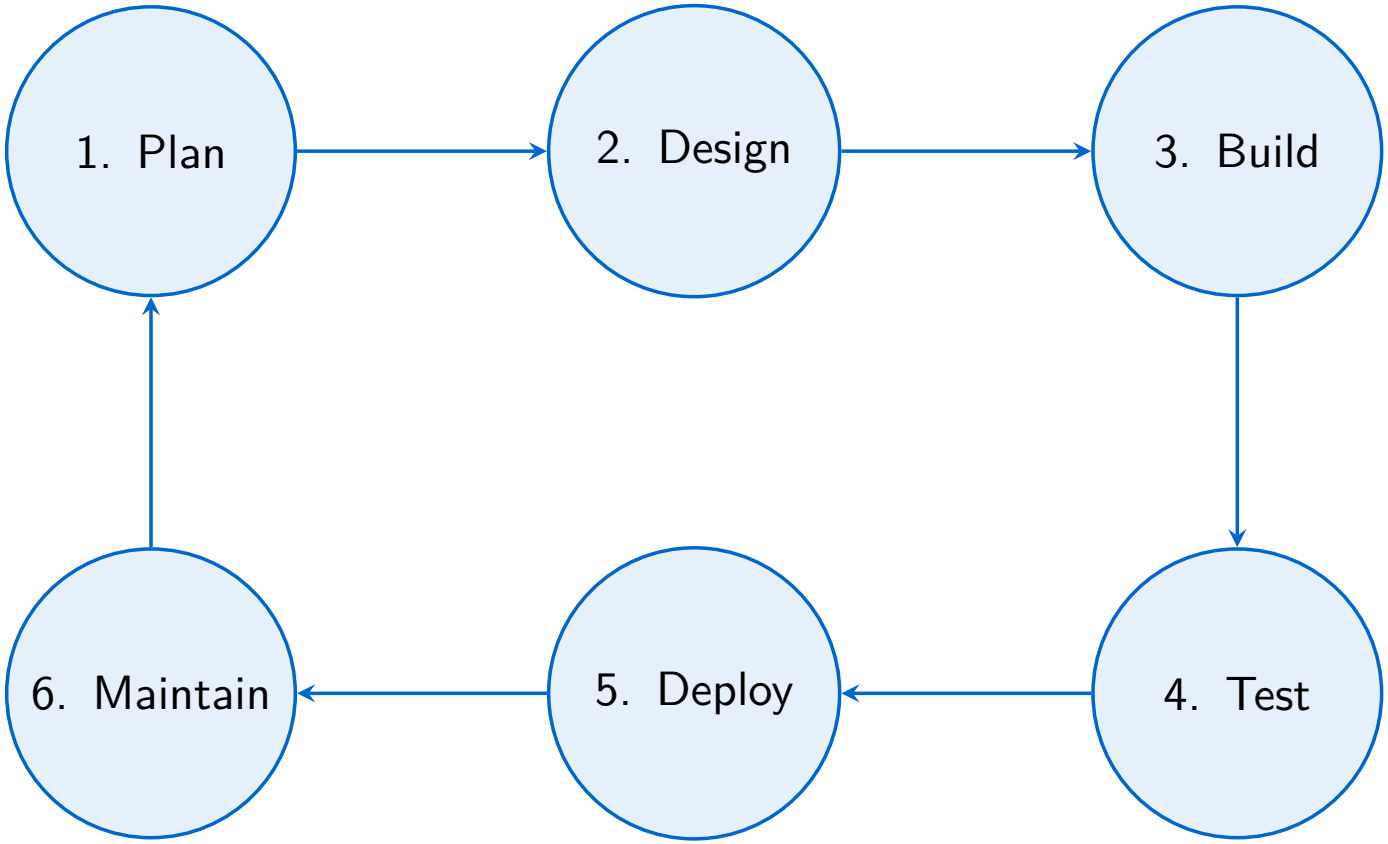
Next Lecture Preview

- Unit 2: Software Development Life Cycle (SDLC).
- Introduction to SDLC.
- Phases of SDLC.
- Why SDLC is necessary for large projects.

Questions?

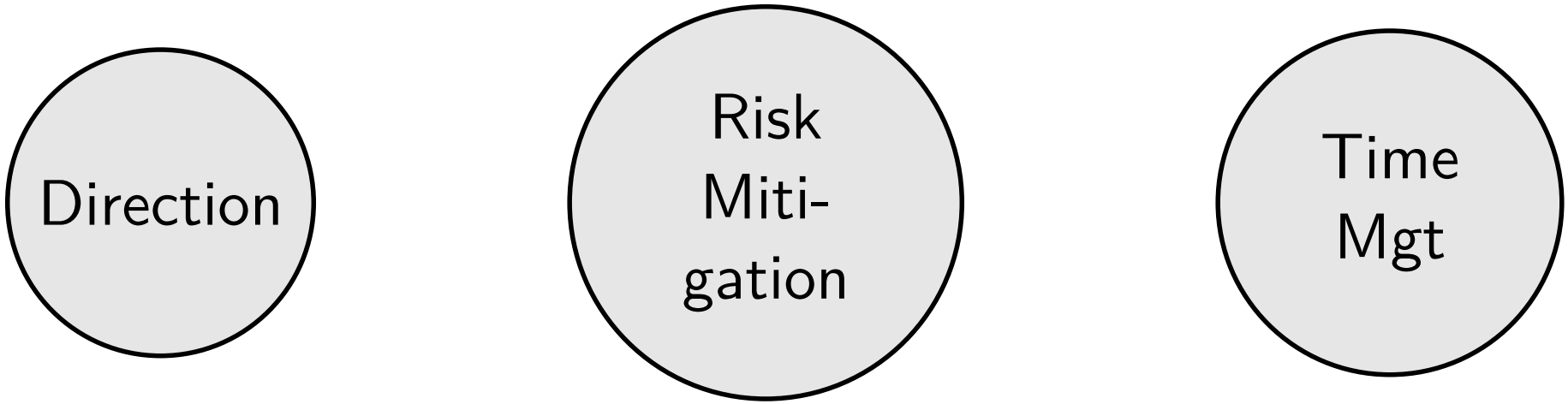
What is SDLC?

- Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality softwares.
- **Aim:** Produce high-quality software that meets or exceeds customer expectations.
- **Goal:** Reach completion within times and cost estimates.
- It provides a structured sequence of phases to develop software.



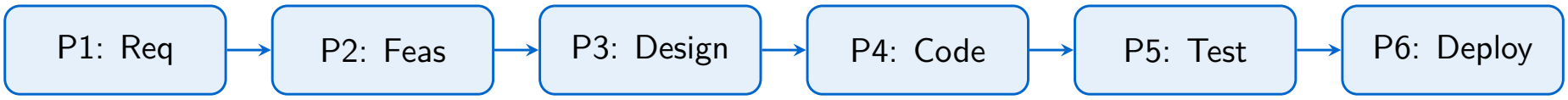
Why do we need SDLC?

- **Structure and Control:** Provides a framework for planning, executing, and controlling the project.
- **Clarity of Goals:** Ensures everyone understands the project objectives and current phase.
- **Risk Mitigation:** Early identification of feasibility issues and risks.
- **Quality Assurance:** Structured testing phases guarantee standard compliance.
- **Cost and Time Management:** Detailed planning prevents budget and schedule overruns.



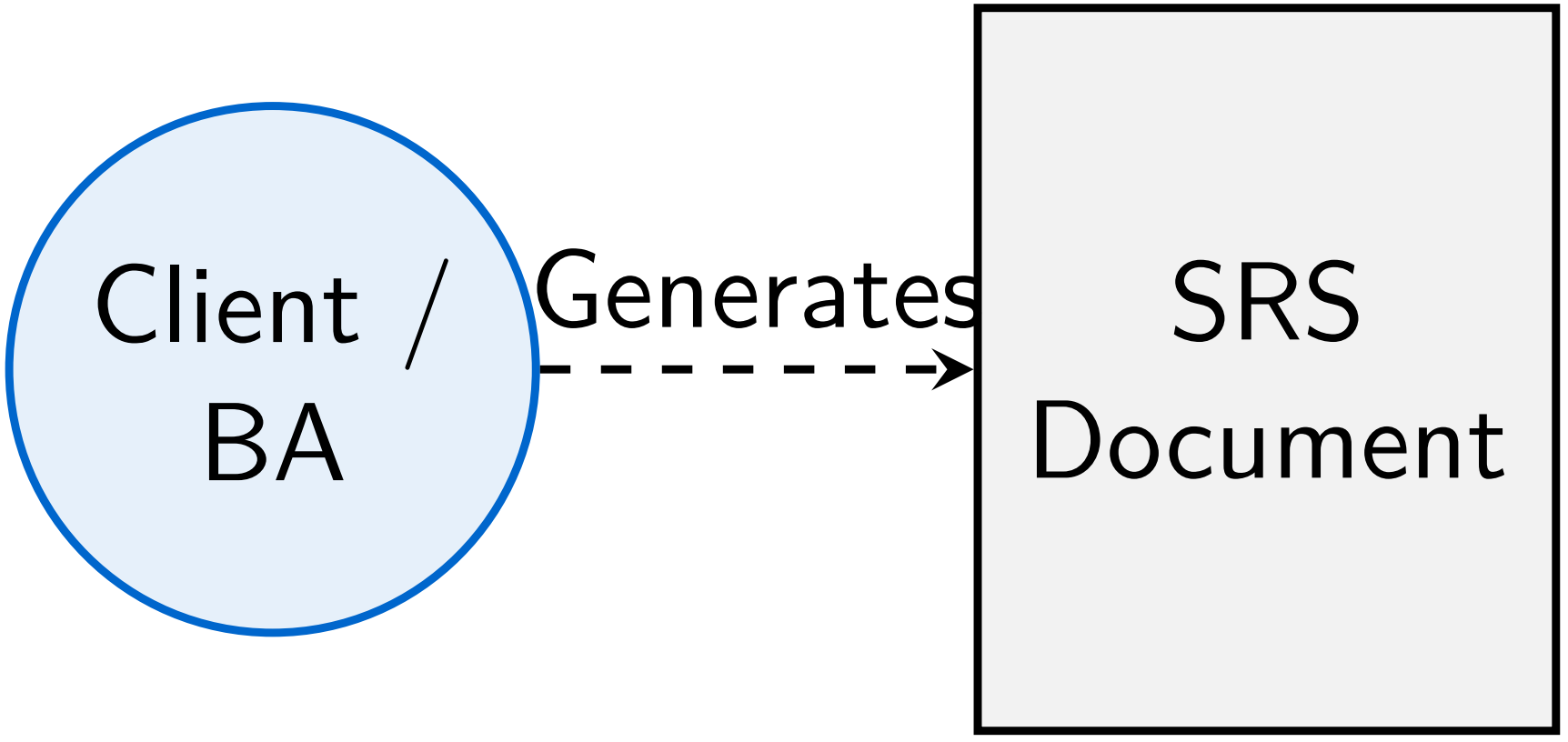
Standard Phases of SDLC

- Phase 1: Requirement Collection and Analysis
- Phase 2: Feasibility Study
- Phase 3: System Design
- Phase 4: Coding / Implementation
- Phase 5: Testing
- Phase 6: Deployment and Maintenance



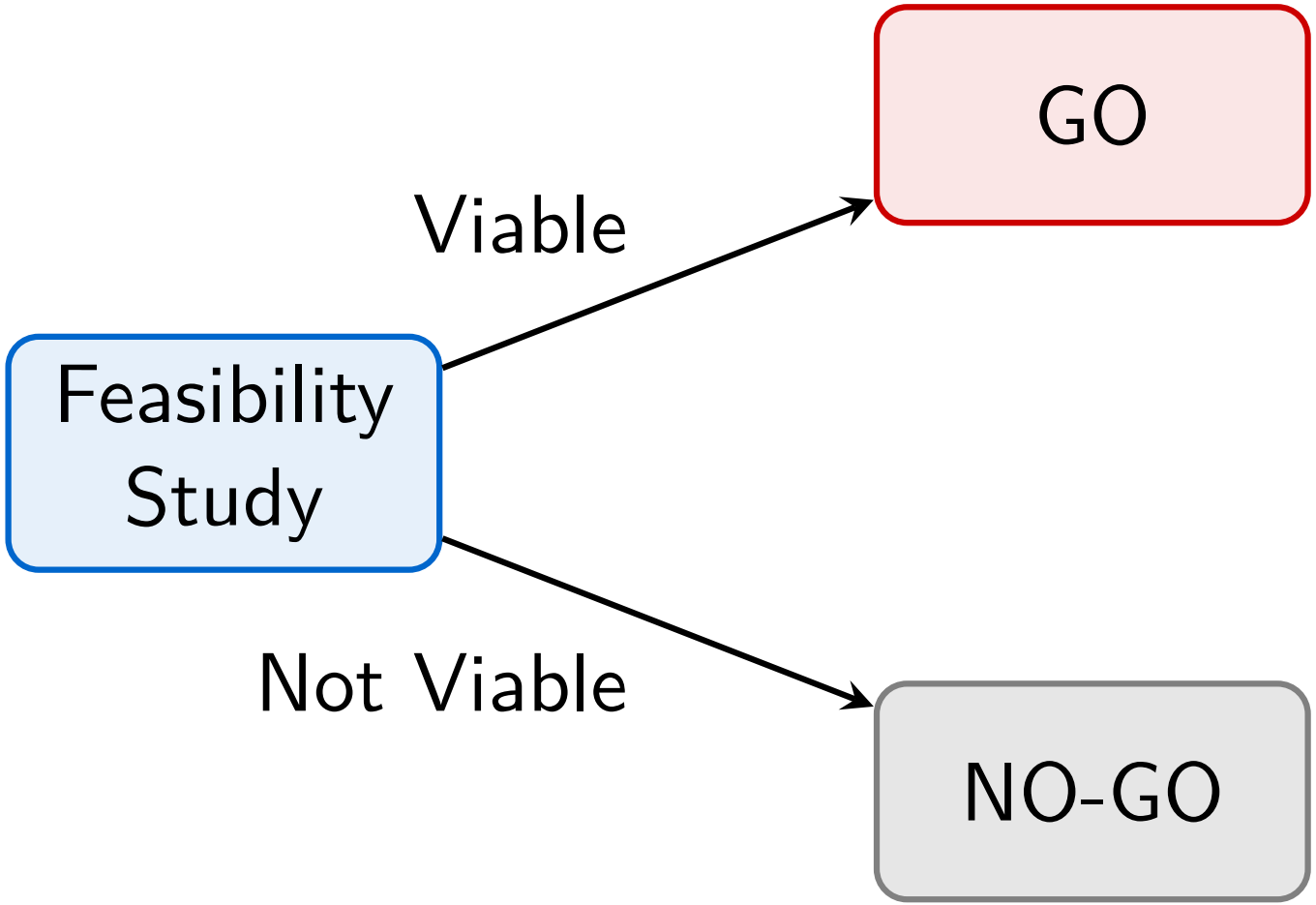
Phase 1: Requirement Collection and Analysis

- Most crucial phase performed by senior members and Business Analysts.
- Inputs from customers, domain experts, and sales department.
- **Output:** Software Requirement Specification (SRS) document.
- Defines 'WHAT' the system should do, not 'HOW'.



Phase 2: Feasibility Study

- Assesses if the project is viable practically and financially.
- Types of Feasibility: Economic (Cost vs Benefit), Technical (Technology available?), Operational (Will it be used?), Legal, Schedule.
- Decision point: Go / No-Go for the project.



Phase 3: System Design

- Translates SRS into a system architecture.
- **High-Level Design (HLD):** Architecture, modules, database design.
- **Low-Level Design (LLD):** Detailed logic, algorithms, UI design for each module.
- **Output:** Design Document Specification (DDS).

Phase 4: Coding / Implementation

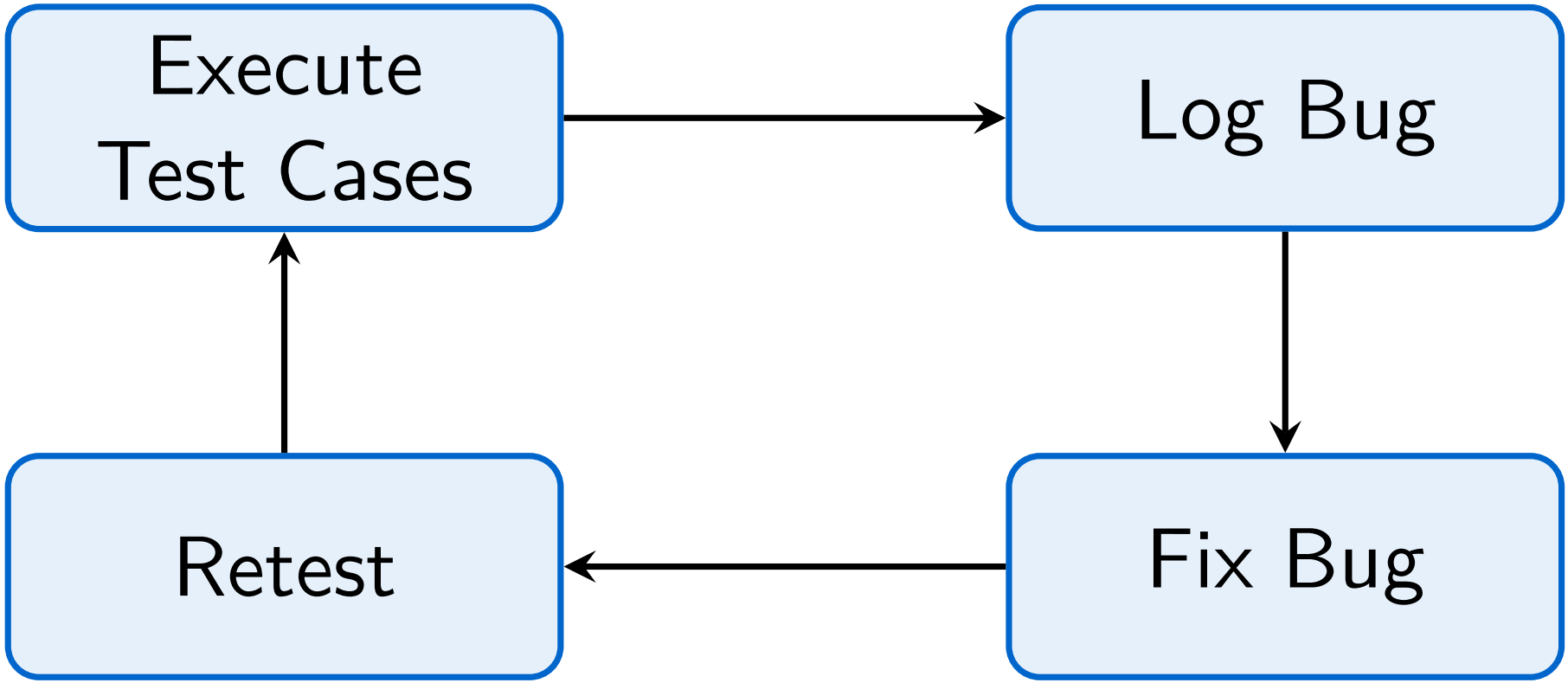
- Translating the design documents into executable source code.
- Longest phase of the SDLC.
- Tasks: Writing code, adhering to coding standards, using version control.
- Developers use tools like compilers, debuggers, and IDEs.

IDE

```
#include <stdio.h>
int main() {
  printf("Coding");
  return 0;
}
```

Phase 5: Testing

- Verifying the software meets the requirements specified in the SRS.
- Types: Unit Testing (developers), Integration Testing, System Testing, Acceptance Testing (UAT).
- Defect tracking: Bugs are logged, reported, and fixed.
- Goal: Deliver a defect-free, stable product.



Phase 6: Deployment and Maintenance

- **Deployment:** Releasing the software to the production environment for user consumption.
- **Maintenance:** Post-release activities.
- Types of Maintenance:
  - Corrective (fixing post-release bugs)
  - Adaptive (environment changes)
  - Perfective (new features)
  - Preventive

Introduction to SDLC Models

- While the phases remain similar, the flow and execution vary based on the chosen model.
- **Predictive Models:** Waterfall Model.
- **Iterative/Incremental Models:** Iterative Enhancement, Spiral Model.
- **Adaptive Models:** Agile Frameworks.

Summary

- SDLC is a structured process to build high-quality software.
- It provides control, risk management, and clarity.
- Core phases include Requirement Analysis, Feasibility, Design, Coding, Testing, and Maintenance.
- The output of one phase often acts as the input for the next.

Next Lecture Preview

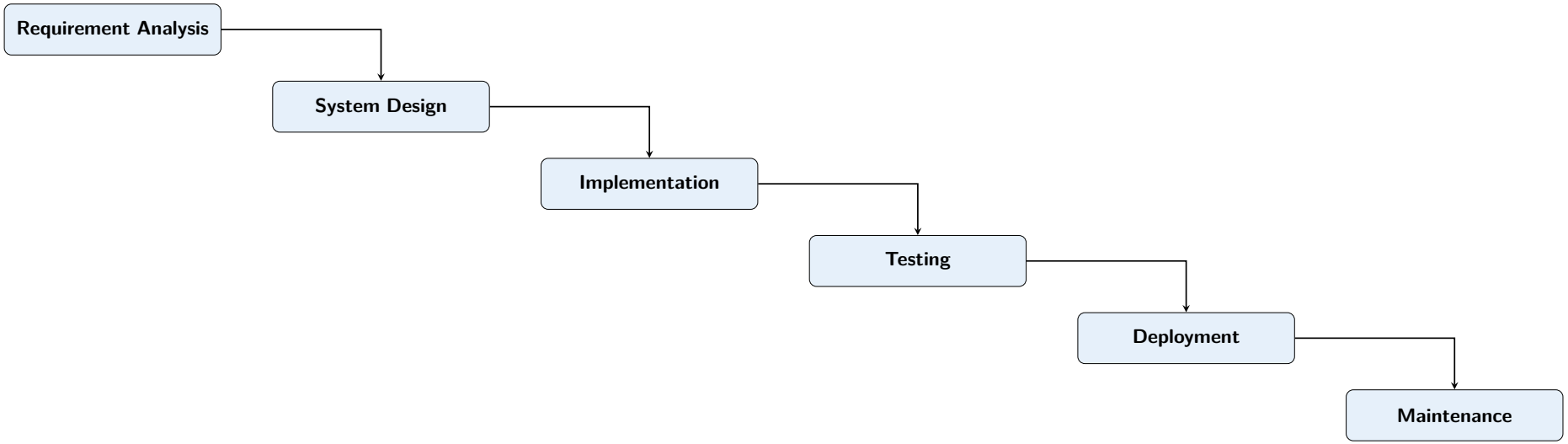
- The Waterfall Model.
- Phases of the Waterfall Model in detail.
- Advantages and Disadvantages of the Waterfall Model.
- When to use the Waterfall Model.

Questions?

What is the Waterfall Model?

- Also known as the Linear Sequential Life Cycle Model or Classic Life Cycle.
- First introduced by Winston W. Royce in 1970.
- It is a strict, sequential approach: you must complete one phase before moving to the next.
- No overlapping of phases is allowed.

Architecture of the Waterfall Model



Phases: Requirements & Design

- **Requirement Analysis**: All possible requirements are captured and documented in an SRS document. Customer involvement is high here.
- **System Design**: The SRS is studied to define system architecture (hardware and system requirements). Output is the Design Document.

Phases: Implementation & Testing

- **Implementation (Coding)**: System is first developed in small programs called units. Units are implemented and tested (Unit Testing).
- **Testing**: All units are integrated into a system. The entire system is tested for faults and failures against the SRS.

Phases: Deployment & Maintenance

- **Deployment**: Once functional and non-functional testing is done, the product is deployed in the customer environment or released.
- **Maintenance**: Fixing issues found by users post-deployment, or enhancing the software for newer environment changes.

Advantages of the Waterfall Model

- Simple and easy to understand and use.
- Phases do not overlap, making project management straightforward.
- Clear milestones and deliverables for each phase.
- Works well for smaller projects where requirements are very well understood.
- Extensive documentation is produced.

Disadvantages of the Waterfall Model (Part 1)

- Inflexible to changes: Once a phase is completed, it's very difficult to go back and change requirements.
- Late Testing: Testing is done very late in the lifecycle. If requirements were misunderstood in phase 1, it is only discovered in phase 4.
- High amount of risk and uncertainty.

Disadvantages of the Waterfall Model (Part 2)

- No working software is produced until late during the life cycle.
- Not suitable for complex, object-oriented, or long-term projects.
- Poor model for long and ongoing projects.
- Customer sees the product only at the end.

When to Use the Waterfall Model?

- Requirements are very well documented, clear, and fixed.
- Product definition is stable.
- Technology is understood and is not dynamic.
- There are no ambiguous requirements.
- Ample resources with required expertise are available freely.
- The project is short.

Waterfall vs. Reality

- **Assumption**: Requirements don't change. **Reality**: Requirements change constantly.
- **Assumption**: Perfect understanding in phase 1. **Reality**: Understanding deepens during coding.
- **Assumption**: Linear flow is natural. **Reality**: Software development is inherently iterative.

Summary

- Waterfall is a linear sequential SDLC model.
- Phases: Requirements $\rightarrow$ Design $\rightarrow$ Implementation $\rightarrow$ Testing $\rightarrow$ Deployment $\rightarrow$ Maintenance.
- Pros: Simple, easy to manage, well-documented.
- Cons: Rigid, late testing, hard to accommodate changes.
- Best used for short projects with fixed, clear requirements.

Next Lecture Preview

- Iterative Enhancement Model.
- How iterative models solve Waterfall's problems.
- The concept of Incremental delivery.

Questions?

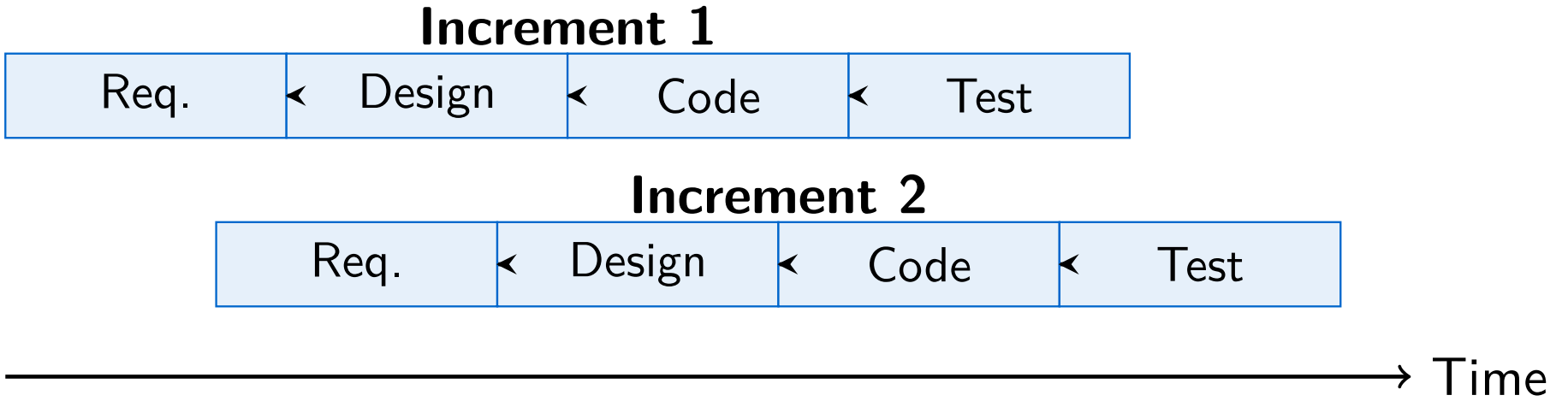
Previous Lecture Recap

Recap

Review of the V-Model and its emphasis on verification and validation.

Introduction to Incremental Model

- The Incremental Model combines elements of linear and parallel process flows.
- It applies linear sequences in a staggered fashion as time progresses.
- Each linear sequence produces deliverable 'increments' of the software.
- The first increment is often a core product focusing on basic requirements.



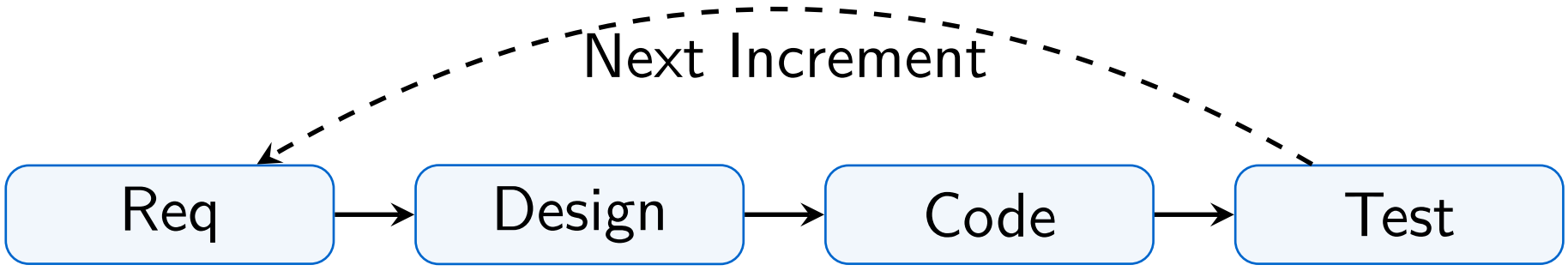
Core Philosophy

- **Divide and Conquer:** Break the software into smaller modules.
- **Iterative Delivery:** Each module passes through the requirements, design, implementation, and testing phases.
- **Integration:** New increments are integrated with existing ones.
- **Focus:** on delivering a functional product early to gather feedback.

Phases within an Increment

Each increment goes through standard SDLC phases:

1. **Requirement Analysis:** specific to the increment.
2. **Design:** Architecture for the increment, keeping overall system in mind.
3. **Code:** Implementation of the specific features.
4. **Test:** Unit testing, integration testing with previous increments, and validation.



The First Increment: Core Product

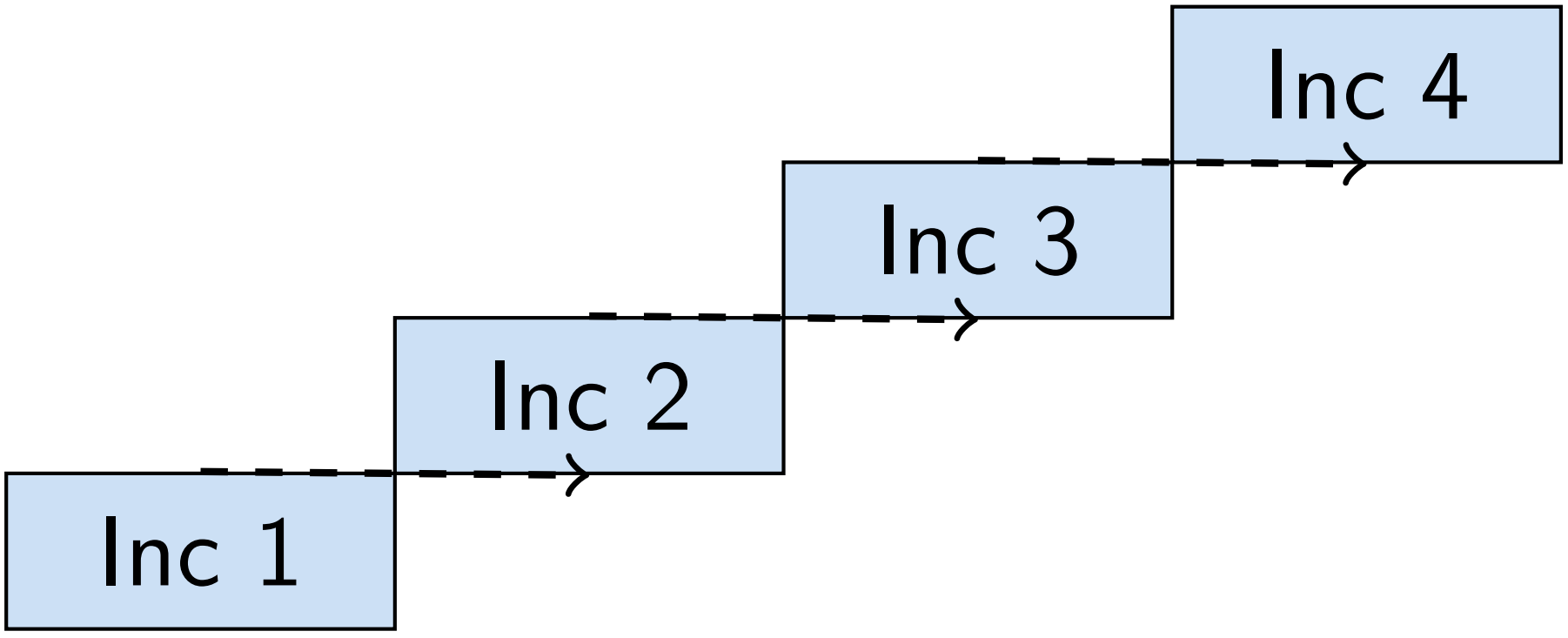
- Usually addresses the basic requirements.
- Many supplementary features remain undelivered.
- Used by the customer for evaluation.
- Feedback is used to plan subsequent increments.

Example: Word Processor

Core Increment: Text entry, basic formatting.
Later Increments: Spell check, advanced page layout, graphics insertion.

Characteristics of Increments

- Each increment is a usable, operational product.
- Later increments add features and capabilities.
- System functionality grows over time.
- Requires a robust underlying architecture to support continuous additions.



Advantages: Risk Management

- Generates working software quickly and early.
- More flexible - less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Lowers initial delivery cost.

Advantages: Customer Engagement

- Customers can respond to each build.
- Early increments can serve as prototypes for later requirements.
- Delivers business value early.
- Reduces the 'wait time' anxiety for clients.

Disadvantages: Architectural Challenges

- Needs good planning and design.
- Requires a clear and complete definition of the whole system before it can be broken down.
- Total cost is higher than waterfall.
- System architecture can degrade as new increments are added ('patchwork' effect).

When to Use the Incremental Model

- Requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some details can evolve with time.
- There is a need to get a product to the market early.
- A new technology is being used.
- Resources with needed skill sets are not available immediately.

Comparison: Waterfall vs. Incremental

Waterfall Model	**Incremental Model**
Delivers product at the end.	Delivers product in stages.
Rigid against change.	More accommodating to change.
High risk of failure late in process.	Risks managed by early deliverables.

Summary

- The Incremental Model builds software in overlapping phases.
- Focuses on delivering a core product first.
- Reduces risk and provides early value to the customer.
- Requires excellent upfront architectural planning to succeed.

Next Lecture Preview

Prototype Model

- How do we handle poorly understood requirements?
- Building throw-away models vs. evolutionary prototypes.
- User evaluation as a core driver.

Questions?

Lecture 2.4: Prototype Model

Complete Lecture Deck – All 45 Lectures

Milav Dabgar

Lecturer, GP Palanpur

July 22, 2026

Recap & Agenda

- **Recap:** The Incremental Model delivers software in functional parts but requires a strong initial architecture.
- **Agenda:**
  - What is Prototyping?
  - Why Build a Prototype?
  - Phases of the Prototype Model
  - Throwaway vs. Evolutionary Prototyping
  - Advantages & Risks
  - Use Cases

What is Prototyping?

- A working model of software with some limited functionality.
- Does not necessarily hold the exact logic used in actual software.
- Built to understand requirements and validate design.
- Evaluated by the end-user.

Why Build a Prototype?

- The “I’ll know it when I see it” phenomenon.
- Customers often struggle to articulate their needs technically.
- Developers might misinterpret written requirements.
- Reduces the cost of changes later in the SDLC.

Phases of Prototype Model

- **1. Requirements Gathering:** Understand basic needs.
- **2. Quick Design:** Focus on visible aspects (UI/UX).
- **3. Build Prototype:** Develop the model rapidly.
- **4. User Evaluation:** Customer tests the prototype.
- **5. Refining Prototype:** Iterate based on feedback.
- **6. Engineer Final Product:** Build the actual system.

Types: Throwaway Prototyping

- Also known as Rapid or Close-ended Prototyping.
- Built quickly to understand requirements.
- Discarded once requirements are finalized.
- The final system is built from scratch.
- Focuses on speed and low cost, ignoring code quality.

Types: Evolutionary Prototyping

- The prototype is steadily refined into the final product.
- Not discarded; forms the heart of the new system.
- Requires better initial architecture than throwaway.
- Useful when core technologies are well understood.

Advantages of Prototyping

- Users are actively involved in development.
- Errors in requirements are detected early.
- Quicker user feedback improves final product quality.
- Reduces overall maintenance costs.
- Provides a working model early in the process.

Disadvantages and Risks

- Can increase complexity of the system.
- Clients may confuse prototype with finished product.
- Developers might make implementation compromises to build prototype quickly.
- Scope creep: Users constantly request new features.
- Throwaway prototypes waste effort if not managed well.

The “Quick and Dirty” Dilemma

- Developers use “quick and dirty” techniques to show progress.
- Inefficient algorithms or dummy databases are used.
- If a throwaway prototype is accidentally pushed to production, it causes severe maintenance issues.
- Discipline is required to throw the throwaway away.

When to Use the Prototype Model

- When requirements are unstable or unknown.
- For UI-heavy applications where user interaction is complex.
- When developing new, untested business concepts.
- To validate feasibility of a complex technical solution.

Summary

- Prototyping mitigates requirement uncertainty.
- Can be Throwaway (discarded) or Evolutionary (refined).
- Massively improves customer satisfaction and system usability.
- Requires strict management of customer expectations.

Next Lecture Preview

- Next time: The Spiral Model.
- Combining prototyping with the systematic aspects of Waterfall.
- Focus on Risk Analysis.
- The four quadrants of the Spiral.

Questions?

Introduction to Spiral Model

- Proposed by Barry Boehm in 1986.
- Combines the iterative nature of prototyping with the controlled and systematic aspects of the Waterfall model.
- Provides potential for rapid development of increasingly complete versions.
- Primary focus is on Risk Assessment and Mitigation.

The Four Quadrants

- 1. Determine Objectives, Alternatives, Constraints.
- 2. Evaluate Alternatives, Identify and Resolve Risks.
- 3. Develop and Verify Next-Level Product.
- 4. Plan the Next Phases.

Quadrant 1: Objective Setting

- Identify objectives of the current phase (e.g., performance, functionality).
- Identify alternative ways to implement this portion.
- Identify constraints (cost, schedule, environment).
- Continuous communication between analyst and customer.

Quadrant 2: Risk Assessment & Reduction

- Evaluate alternatives based on objectives and constraints.
- Identify potential risks (technical, schedule, budget).
- Resolve risks using techniques like prototyping, simulation, or benchmarking.
- If risks are too high, the project may be terminated.

Quadrant 3: Development and Validation

- Develop the software based on the resolved risks.
- If UI risk was dominant, use Evolutionary Prototyping.
- If integration risk was dominant, use Waterfall-like phases.
- Verify that the phase objectives are met (Testing).

Quadrant 4: Review and Planning

- Review the results of the current phase with the customer.
- Evaluate the project status against objectives.
- Plan the next phase of the spiral.
- Make 'Go/No-Go' decisions for the next iteration.

Spiral as a Meta-Model

- It can encompass other models within it.
- First loop might be just a feasibility study.
- Second loop might be defining requirements.
- Third loop might involve creating a prototype.
- Final loop might use a Waterfall approach for implementation.

Advantages of Spiral Model

- High amount of risk analysis ensures early detection of critical issues.
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.
- Strong approval and documentation control.

Disadvantages of Spiral Model

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Not suitable for small or low-risk projects.
- Complex to manage.

When to Use Spiral Model

- When costs and risk evaluation is important.
- For medium to high-risk projects (e.g., aerospace, healthcare systems).
- Long-term project commitment because of potential changes.
- When requirements are complex and need evaluation to get clarity.

Summary

- Spiral Model integrates Waterfall and Prototyping.
- Driven by continuous Risk Analysis.
- Consists of four quadrants repeated in loops.
- Ideal for large, complex, and high-risk systems.

Next Lecture Preview

- Next time: The RAD Model.
- Rapid Application Development.
- Focus on speed and component reusability.
- Timeboxing development phases.

Questions?

Recap & Agenda

Recap:

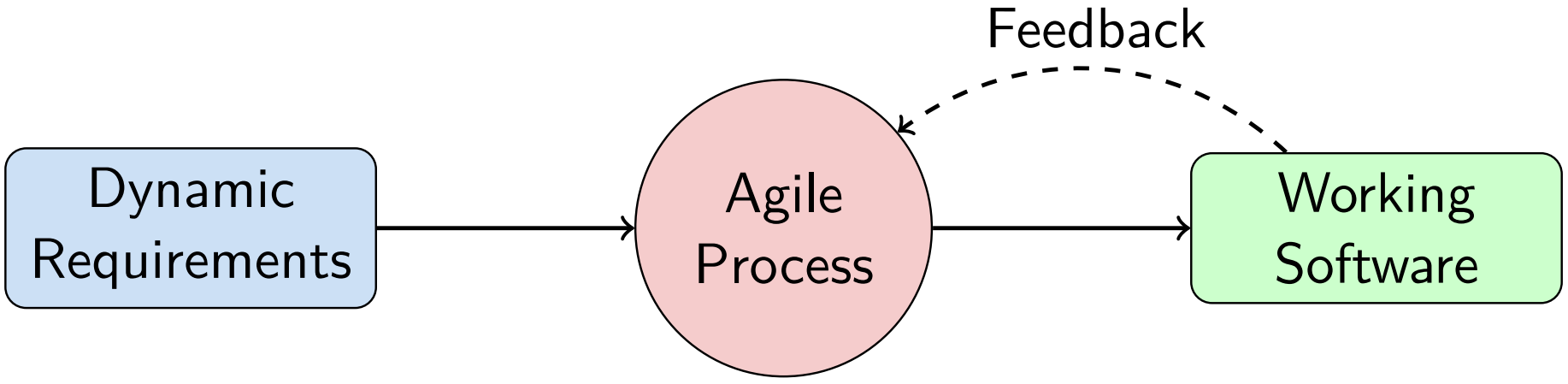
- We previously discussed iterative and incremental models like the Spiral and RAD models, noting their focus on risk analysis and rapid prototyping.

Agenda:

- 1. Introduction to Agile Paradigm
- 2. The Agile Manifesto: 4 Core Values
- 3. The 12 Principles of Agility
- 4. Agile vs. Traditional Models
- 5. Key Takeaways

What is Agility?

- Agility in software engineering means responding effectively to change.
- Focuses on rapid, incremental delivery of working software.
- Emphasizes active customer involvement over rigid contract negotiation.
- Acknowledges that requirements will inevitably change during the SDLC.



The Agile Manifesto (2001)

- Authored by 17 software developers in Snowbird, Utah.
- A formal proclamation of 4 core values and 12 principles.
- Designed to counteract the documentation-heavy, plan-driven processes (e.g., Waterfall).
- It establishes a mindset, not a specific methodology.

Core Value 1: Individuals and Interactions

Value 1

Individuals and interactions **OVER** processes and tools.

- Tools (like JIRA or Git) are useful, but human communication is paramount.
- Cross-functional teams must collaborate directly to solve complex problems.
- Process shouldn't dictate workflow if it hinders productivity.

Core Value 2: Working Software

Value 2

Working software **OVER** comprehensive documentation.

- Documentation (SRS, SDD) is still necessary but shouldn't delay development.
- The primary measure of progress is a functional, tested increment of software.
- Code speaks louder than specifications.

Core Value 3: Customer Collaboration

Value 3

Customer collaboration **OVER** contract negotiation.

- Contracts create an adversarial 'us vs. them' dynamic.
- Customers must be integrated into the development loop (e.g., sprint reviews).
- Continuous feedback ensures the product meets actual business needs.

Core Value 4: Responding to Change

Value 4

Responding to change **OVER** following a plan.

- Agile embraces requirement churn, even late in the SDLC.
- Plans are adaptive; they pivot based on market changes or new insights.
- Rigidity leads to obsolete products upon delivery.

Agility Principles: Delivery & Change

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

Agility Principles: Team & Environment

5. Build projects around motivated individuals; give them trust and support.
6. Face-to-face conversation is the most efficient and effective method of conveying information.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development.

Agility Principles: Technical Excellence

9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, and adjusts behavior.

Summary

- Agile is a mindset defined by the Agile Manifesto.
- It prioritizes individuals, working software, collaboration, and adaptability.
- 12 principles guide the execution, emphasizing continuous delivery, sustainable pace, and self-organization.
- Agile explicitly rejects the rigid, upfront planning of traditional SDLC models.

Next Lecture Preview

- Next, we will look at concrete implementations of the Agile philosophy.
- **Extreme Programming (XP):** Focus on technical practices like Pair Programming and TDD.
- **Scrum:** Focus on project management, roles (Scrum Master, Product Owner), and Sprints.

Questions?

Recap & Agenda

Recap: Lecture 10

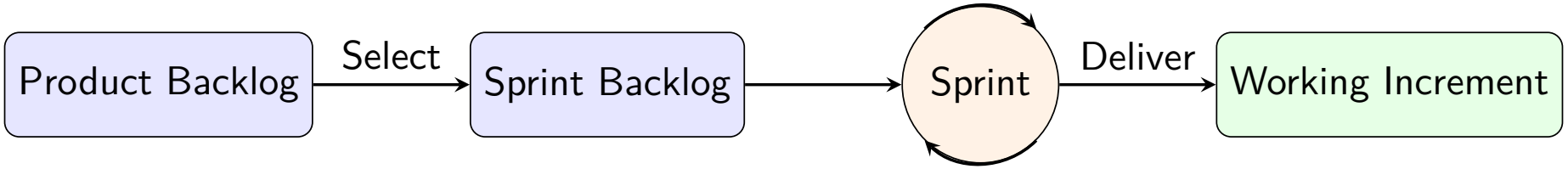
In Lecture 10, we explored the Agile Manifesto, its 4 core values, and 12 principles, establishing the philosophical foundation for Agile development.

Today's Agenda:

- Introduction to Scrum Framework
- Scrum Roles, Artifacts, and Ceremonies
- Introduction to Extreme Programming (XP)
- XP Core Values & Engineering Practices
- Scrum vs. XP Comparison

Scrum Framework Overview

- Scrum is a lightweight framework for developing and sustaining complex products.
- Based on empiricism (knowledge comes from experience) and lean thinking.
- Work is executed in fixed-length iterations called **Sprints** (usually 1-4 weeks).
- Focuses on delivering a 'Potentially Releasable Increment' at the end of each Sprint.



Scrum Roles

- **Product Owner**: Maximizes the product's value. Manages the Product Backlog. Represents stakeholders.
- **Scrum Master**: Ensures Scrum is understood and enacted. Removes impediments. Servant-leader for the team.
- **Development Team**: Cross-functional, self-organizing professionals who do the actual work.
- No traditional 'Project Manager' role in pure Scrum.

Scrum Artifacts

- **Product Backlog**: An ordered, dynamic list of everything needed in the product (Features, bug fixes).
- **Sprint Backlog**: The set of Product Backlog items selected for the Sprint, plus a plan for delivering them.
- **Increment**: The sum of all Product Backlog items completed during a Sprint, integrated with past increments.

Scrum Ceremonies (Events)

- **Sprint Planning**: Defining the Sprint Goal and selecting items for the Sprint Backlog.
- **Daily Scrum (Stand-up)**: 15-minute daily sync. What did I do? What will I do? Any blockers?
- **Sprint Review**: Inspecting the Increment and adapting the Product Backlog with stakeholders.
- **Sprint Retrospective**: Inspecting how the team worked and identifying process improvements.

Extreme Programming (XP) Overview

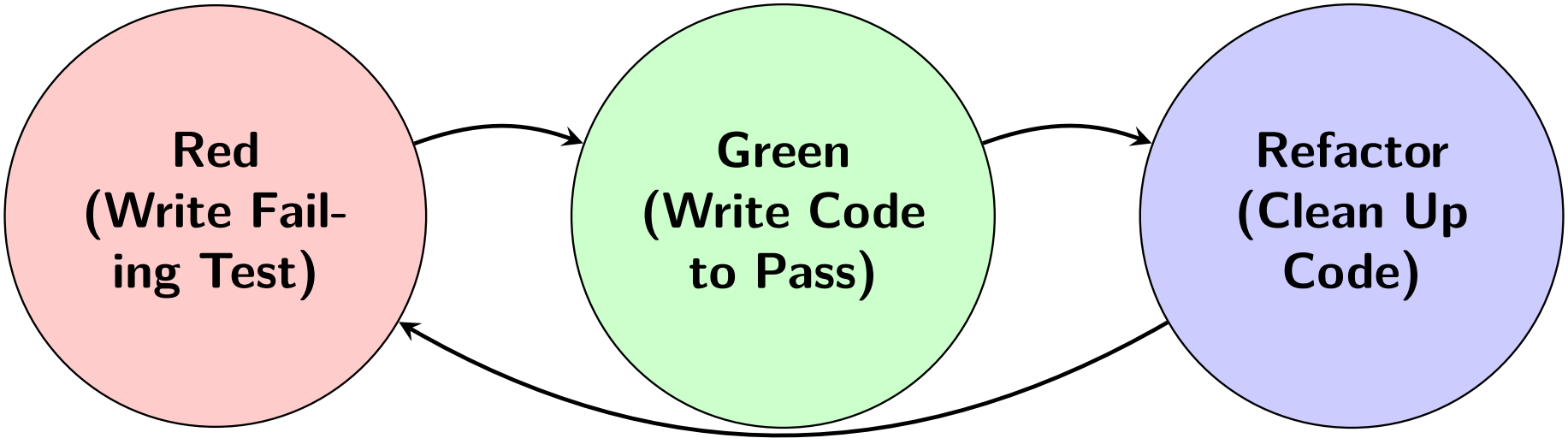
- Created by Kent Beck in the 1990s.
- An Agile framework specifically focused on software engineering practices to ensure high quality.
- Takes ‘best practices’ of software development to extreme levels.
- Highly responsive to changing customer requirements.

XP Core Values

- **Communication**: Everyone is part of the team and communicates face-to-face daily.
- **Simplicity**: Do the simplest thing that could possibly work. Avoid premature optimization.
- **Feedback**: Demonstrate software early and often. Unit tests provide immediate code feedback.
- **Courage**: Tell the truth about progress. Don't fear refactoring.
- **Respect**: Respect team members, the code base, and the customer.

XP Engineering Practices: Coding

- **Pair Programming**: Two developers work at one workstation (Driver and Navigator).
- **Test-Driven Development (TDD)**: Write failing tests BEFORE writing the production code.
- **Continuous Integration (CI)**: Integrate and build the system multiple times a day.
- **Refactoring**: Continuously restructure code without changing external behavior to improve maintainability.



XP Engineering Practices: Planning & Delivery

- **The Planning Game**: Release planning and iteration planning using 'User Stories'.
- **Small Releases**: Release working software into production frequently (e.g., daily or weekly).
- **On-Site Customer**: A real customer must sit with the team full-time to answer questions immediately.
- **Collective Code Ownership**: Anyone can change any line of code at any time.

Comparison: Scrum vs. XP

Aspect	Scrum	Extreme Programming (XP)
Focus	Project Management & Productivity	Engineering Practices & Code Quality
Iteration Length	1 to 4 weeks	1 to 2 weeks
Flexibility	Iteration backlog is locked once Sprint starts	Items can be swapped if work hasn't started
Practices	Doesn't mandate engineering practices	Mandates TDD, Pair Programming, CI, etc.

Summary

- Scrum utilizes Roles, Artifacts, and Ceremonies to manage iterative development.
- XP utilizes strict engineering practices to maximize code quality and responsiveness.
- Both frameworks heavily rely on empirical feedback and customer collaboration.
- Scrum manages the **process**, while XP dictates the **engineering execution**.

Next Lecture Preview

- We will transition to Unit 3: Requirement Engineering.
- Focus on Requirement Gathering and Analysis.
- Understanding what a requirement actually is and why it's the hardest part of software engineering.

Questions?

What is a Software Requirement?

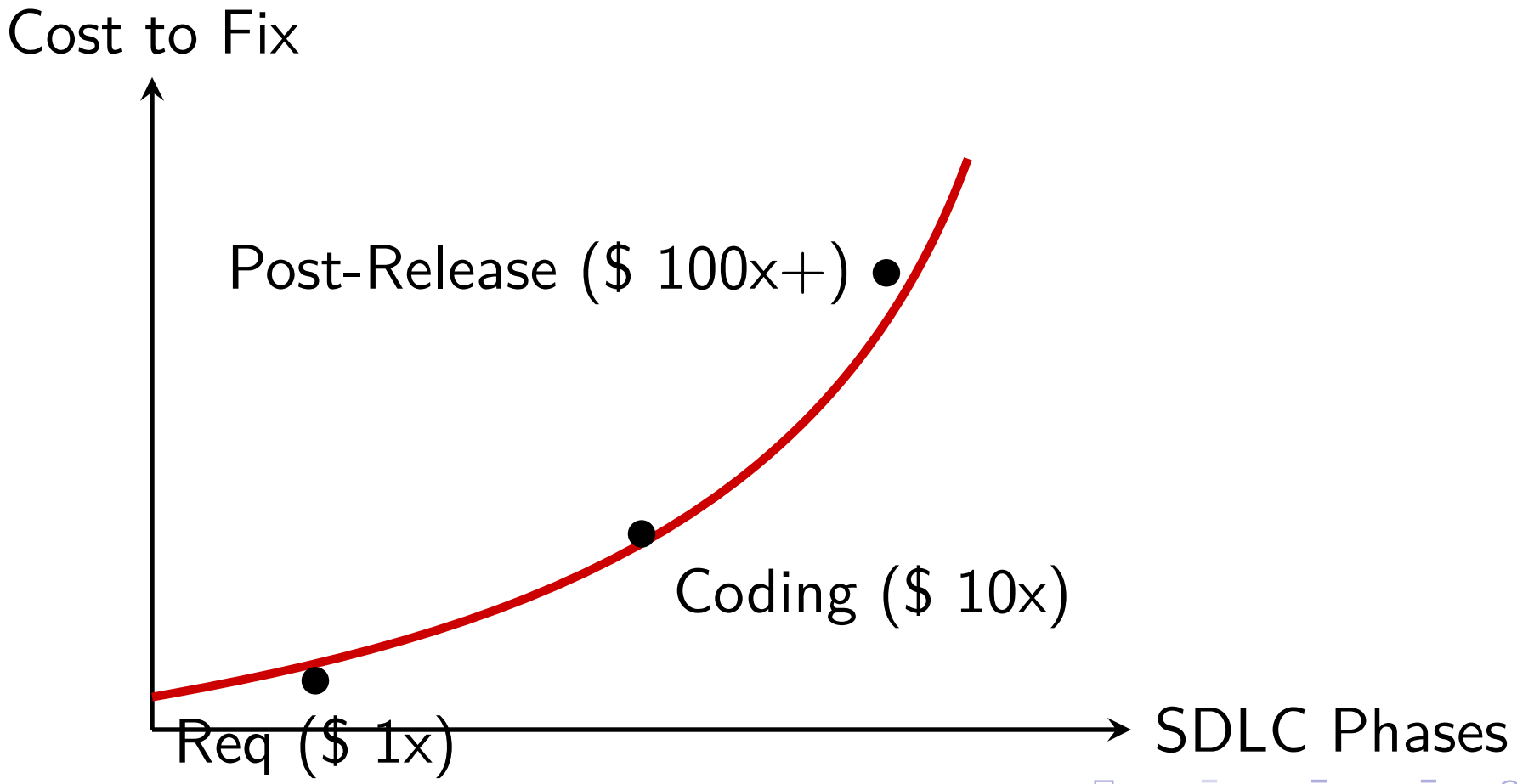
- A condition or capability needed by a user to solve a problem or achieve an objective.
- A description of the services the system should provide and the constraints under which it must operate.
- Requirements range from high-level abstract statements to detailed mathematical functional specifications.

IEEE Definition

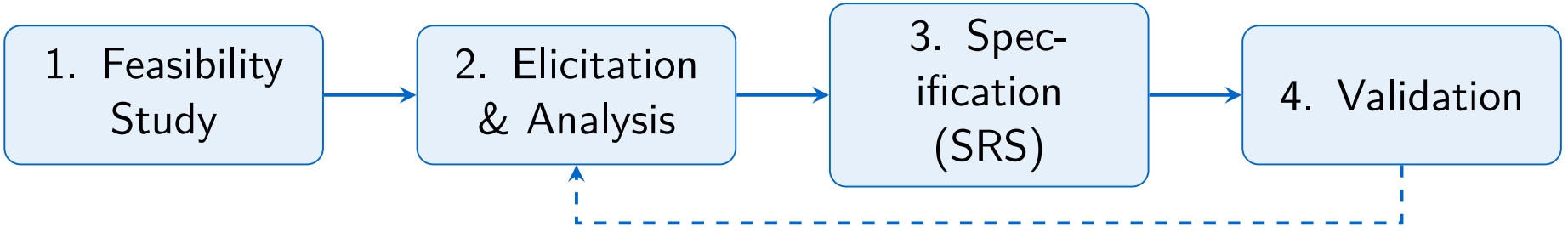
A condition or capability that must be met or possessed by a system to satisfy a contract or standard.

The Cost of Requirement Errors

- Requirement errors are the most common source of software project failure.
- Boehm's Curve: The cost of fixing a requirement error increases exponentially as the project progresses.



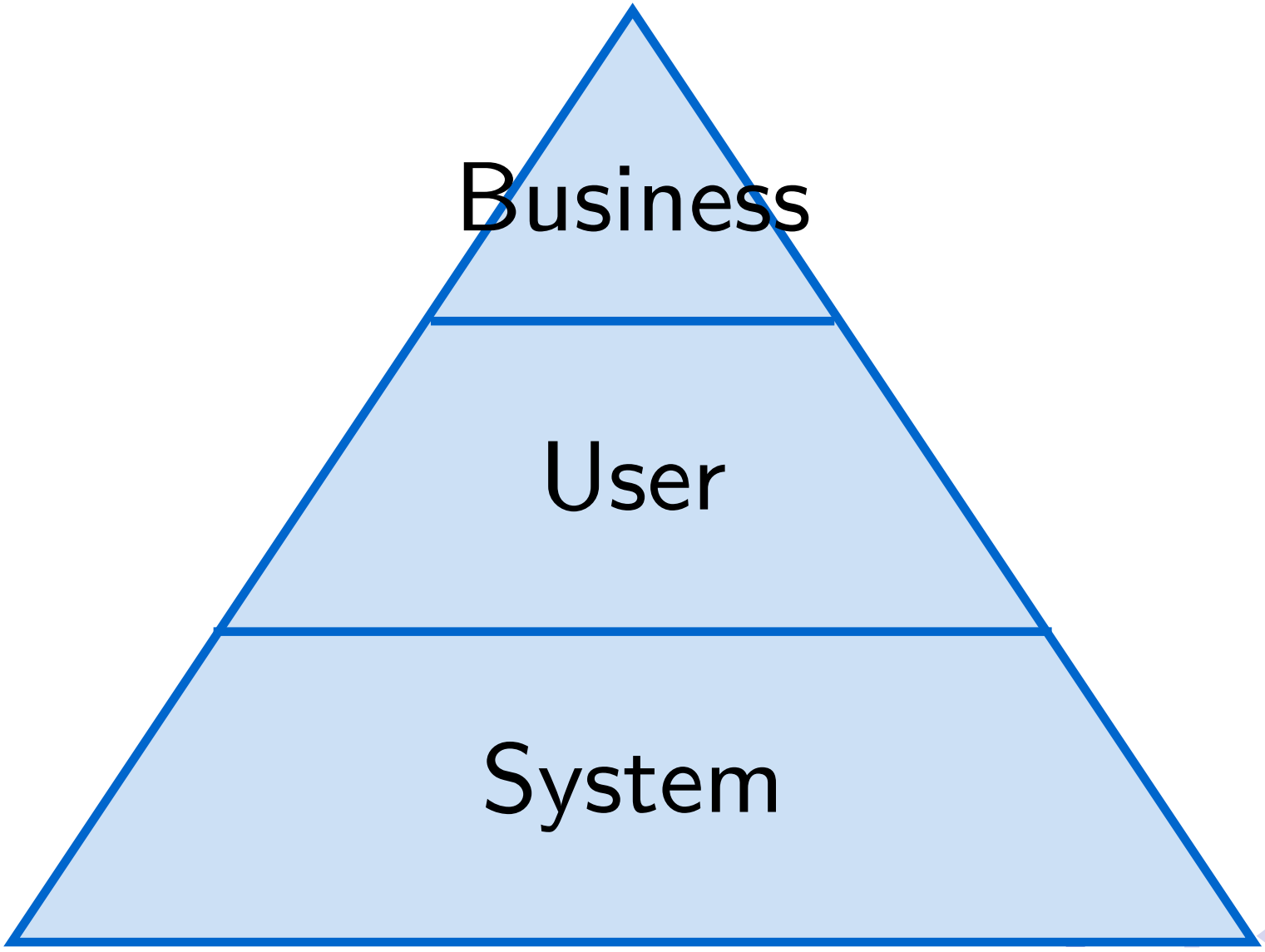
Requirement Engineering (RE) Process



- **Feasibility Study:** Is the system technically and financially viable?
- **Elicitation and Analysis:** Gathering needs from stakeholders and resolving conflicts.
- **Specification:** Documenting the requirements formally.
- **Validation:** Checking that documented requirements define the system the customer actually wants.

Levels of Requirements

- **Business Requirements:** High-level objectives of the organization.
- **User Requirements:** Tasks users must be able to perform.
- **System Requirements:** Detailed descriptions of software functions.



User vs. System Requirements (Library Example)

Context: Library Management System.

- **User Requirement (Natural Language):** 'The librarian shall be able to see a list of all overdue books and issue fines.'
- **System Requirement (Structured/Technical):** 'System must execute daily cron job at 00:00 to query checkout_records. If due_date $<$ current_date, change status to OVERDUE and apply a base penalty of Rs. 10/day to member_account.'

Requirement Elicitation

- Elicitation is the process of **'drawing out'** requirements from stakeholders.
- It is not merely 'gathering'; users often don't know what they want.
- Involves identifying stakeholders: End-users, managers, domain experts, external regulators.
- Requires active communication, observation, and domain analysis.

Challenges in Elicitation

- **The 'I know it when I see it' syndrome:** Users struggle to articulate needs until they see a prototype.
- **Conflicting Requirements:** Management wants high security (complex passwords); Users want ease of use (no passwords).
- **Tacit Knowledge:** Domain experts omit critical details assuming they are 'obvious'.
- **Volatility:** Requirements change as the business environment changes during the project.

Requirement Analysis

- Once elicited, requirements must be analyzed for consistency and completeness.
- **Ambiguity Resolution:** 'System should be fast' $\rightarrow$ 'System must load homepage in under 2 seconds'.
- **Prioritization:** MoSCoW method (Must have, Should have, Could have, Won't have).
- **Modeling:** Creating initial DFDs, Use Case diagrams, or state charts to visualize flow.

Characteristics of Good Requirements

- **Unambiguous:** Only one possible interpretation.
- **Testable:** You can design a specific test to verify it.
- **Traceable:** You can track it from origin (stakeholder) to implementation (code/test).
- **Consistent:** Does not contradict other requirements.

Summary

- Requirements define what the system must do and its operational constraints.
- Errors here are the most expensive to fix later in the SDLC.
- The RE process includes Feasibility, Elicitation, Specification, and Validation.
- Requirements must be transformed from vague user desires into testable, unambiguous system specifications.

Next Lecture Preview

- Next, we will explore specific techniques for Requirement Elicitation.
- Interviews, Questionnaires, and Brainstorming.
- Use Case Modeling and Prototyping.
- How to handle Functional vs. Non-Functional Requirements.

Questions?

Recap & Agenda

Recap: Wrap-up of Agile Development (Unit 2).

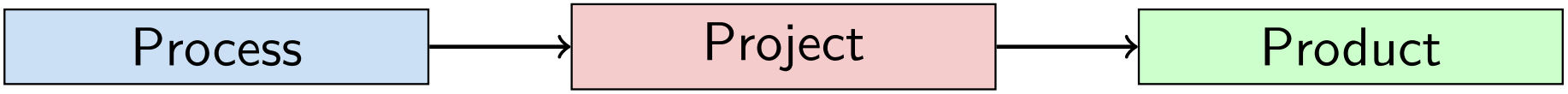
Agenda:

- Introduction to SE Core Principles
- Communication Principles: Listening and Preparation
- Communication Principles: Facilitation and Documentation
- Planning Principles: Scope and Estimation
- Planning Principles: Risk and Schedule Management

What are Core SE Principles?

- Principles are fundamental truths that guide software development across all lifecycle models.
- They operate at the process, project, and product levels.
- Hooker's General Principles: Provide value to the user, Keep it simple (KISS), Maintain the vision, What you produce others will consume, Be open to the future, Plan ahead for reuse, Think!

Hooker's General Principles Foundation



Communication Principles (1/3): Listen and Prepare

- Principle 1: Listen. Focus on the speaker's words, avoid interrupting, and clarify context.
- Principle 2: Prepare before you communicate. Research the business domain (e.g., Library Management rules before meeting a librarian).
- Define an agenda and understand the stakeholders' constraints.

Communication Principles (2/3): Facilitation

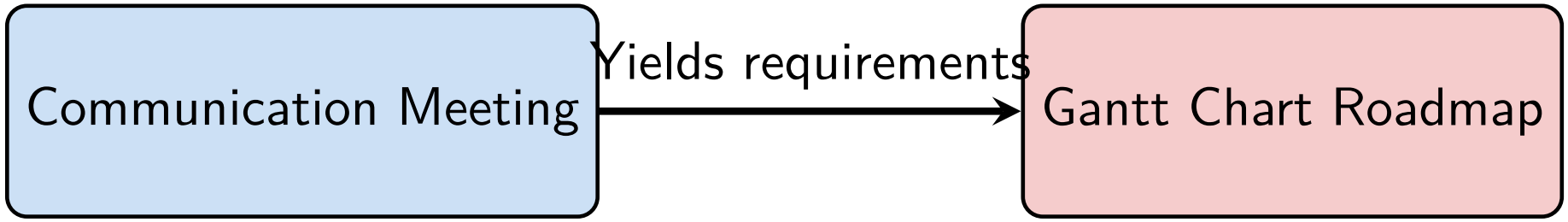
- Principle 3: Someone should facilitate the activity. A designated leader keeps the meeting on track.
- Principle 4: Face-to-face communication is best (or high-fidelity video).
- Principle 5: Take notes and document decisions immediately. Use shared documents to avoid ambiguity.

Communication Principles (3/3): Collaboration & Consensus

- Principle 6: Strive for collaboration. Build consensus on requirements.
- Principle 7: Stay focused, modularize your discussion. E.g., discuss 'Book Issue' separate from 'Member Registration'.
- Principle 8: If something is unclear, draw a picture (UML sketches on a whiteboard).

Moving to Planning Principles

- Once communication yields initial requirements, planning begins.
- Planning defines the roadmap: What needs to be done, by whom, and when.



Planning Principles (1/4): Scope and Estimation

- Principle 1: Understand the scope of the project. Define boundaries (what is IN and OUT of the system).
- Principle 2: Involve stakeholders in the planning activity to ensure realistic expectations.
- Principle 3: Recognize that planning is iterative. Plans must adapt as requirements change.

Planning Principles (2/4): Estimation Techniques

- Principle 4: Estimate based on what you know.
- Use historical data, expert judgment, or algorithmic models (e.g., COCOMO).
- Decompose the problem to estimate accurately (e.g., break down 'Library System' into 'Catalog', 'Circulation', 'Admin').

Planning Principles (3/4): Risk Management

- Principle 5: Consider risk as you define the plan.
- Identify technical risks, business risks, and scheduling risks.
- Establish contingency plans (e.g., what if the lead database developer leaves?).

Impact / Prob	**Low**	**Medium**	**High**
High	Watch	Mitigate	Contingency
Medium	Accept	Watch	Mitigate
Low	Accept	Accept	Watch

Planning Principles (4/4): Schedule and Track

- Principle 6: Be realistic. 100% resource utilization is a myth (factor in meetings, sick leave).
- Principle 7: Adjust granularity as you plan (high-level for distant tasks, detailed for imminent tasks).
- Principle 8: Define how you intend to ensure quality and track progress.

Example: Applying Principles to Library System

- **Communication:** Facilitated meeting with Head Librarian. Sketched UI for 'Search Book' on whiteboard.
- **Planning:** Scope defined (Excludes inter-library loans). WBS created for modules. Risk identified: Barcode scanner integration delay.

Summary

- Communication principles emphasize listening, preparation, facilitation, and visual documentation.
- Planning principles focus on defining scope, realistic estimation, risk management, and iterative scheduling.
- Both phases are foundational for the technical work that follows.

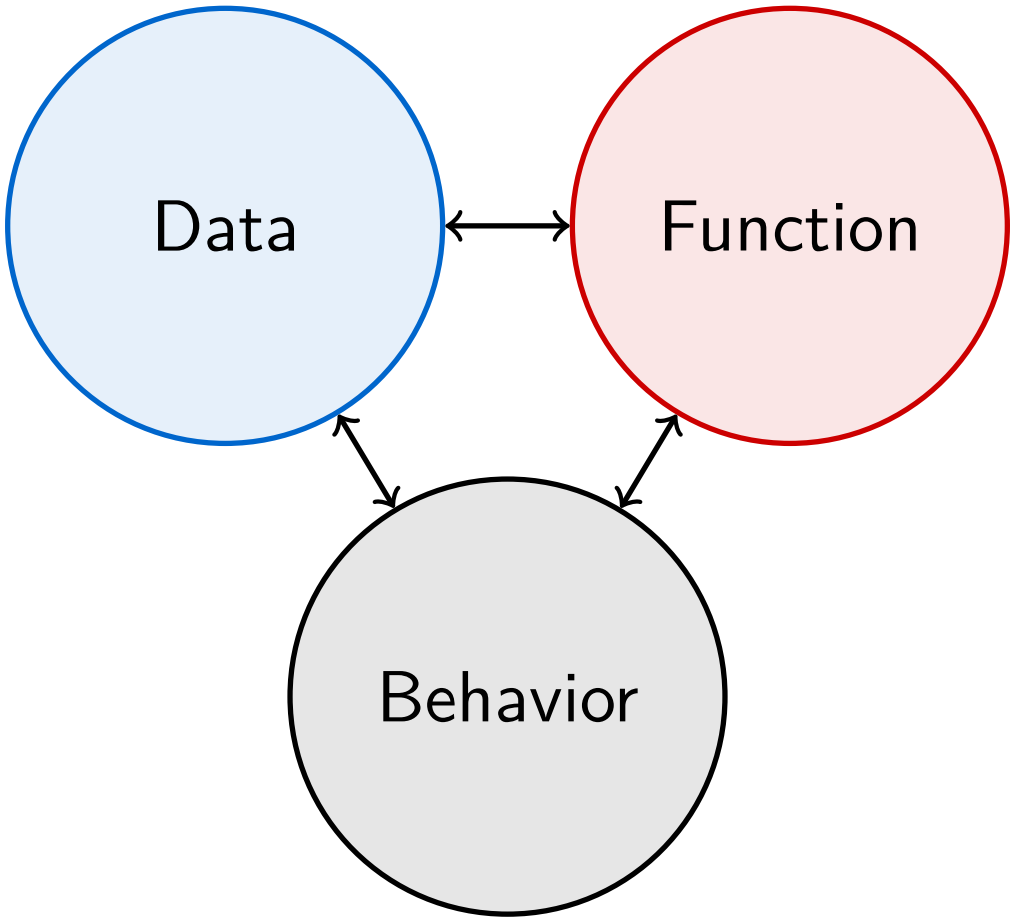
Next Lecture Preview

- Next time: SE Core Principles for Modelling, Construction, and Deployment.
- We will learn how to transition from a plan to a software model and finally to running code.

Questions?

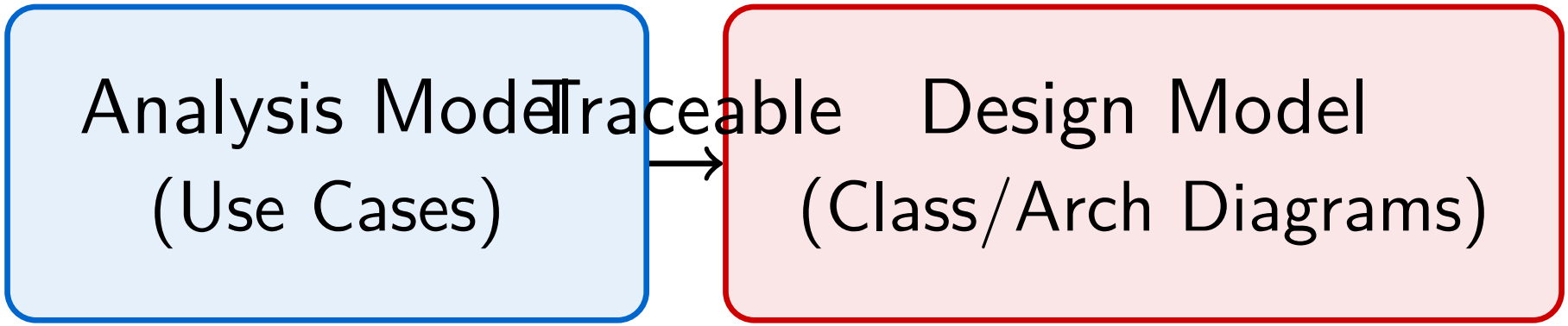
Modelling Principles (1/2): Analysis Modeling

- Analysis models represent customer requirements (information, function, behavior).
- **Principle 1:** The information domain must be represented and understood (e.g., Data models for Books, Members).
- **Principle 2:** The functions that the software performs must be defined (e.g., Calculate Fine).
- **Principle 3:** The behavior of the software must be represented (e.g., State transition when a book is issued).



Modelling Principles (2/2): Design Modeling

- Design models provide details about architecture, UI, and components.
- **Principle 4:** Design should be traceable to the analysis model.
- **Principle 5:** Always consider the architecture of the system to be built.
- **Principle 6:** Design of data is as important as design of processing functions.



Construction Principles (1/3): Preparation

- Construction encompasses coding and testing.
- **Preparation Principle 1:** Understand the problem before you write code.
- **Preparation Principle 2:** Understand the basic design principles and concepts.
- **Preparation Principle 3:** Pick a programming language that meets the needs of the software.

Construction Principles (2/3): Coding Practices

- **Coding Principle 1:** Constrain your algorithms by following structured programming.
- **Coding Principle 2:** Create unit tests before you begin coding (TDD - Test Driven Development).
- **Coding Principle 3:** Keep source code simple and readable (meaningful variable names, proper indentation).
- **Coding Principle 4:** Select data structures that meet the needs of the design.

Construction Principles (3/3): Testing Principles

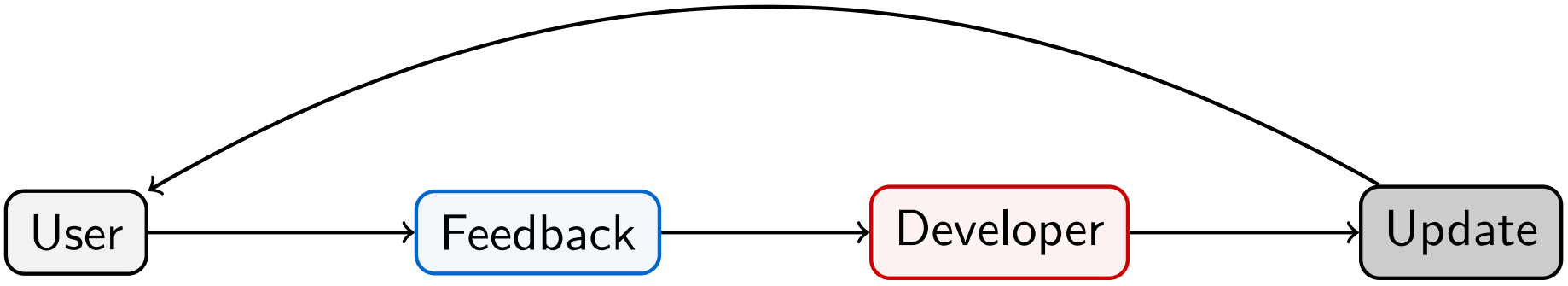
- **Testing Principle 1:** All tests should be traceable to customer requirements.
- **Testing Principle 2:** Tests should be planned long before testing begins.
- **Testing Principle 3:** The Pareto principle applies to software testing (80% of errors are in 20% of modules).
- **Testing Principle 4:** Exhaustive testing is not possible.

Deployment Principles (1/2): Delivery Strategy

- Deployment involves delivery, support, and feedback.
- **Principle 1:** Customer expectations for the software must be managed.
- **Principle 2:** A complete delivery package should be assembled and tested (Installers, Configs).
- **Principle 3:** A support regime must be established before the software is delivered.

Deployment Principles (2/2): Documentation and Feedback

- **Principle 4:** Appropriate instructional materials must be provided to end users (Manuals, Tooltips).
- **Principle 5:** Buggy software should be fixed first, delivered later (Avoid rushing to meet deadlines at the cost of quality).
- **Principle 6:** Elicit feedback from users to improve the next iteration.

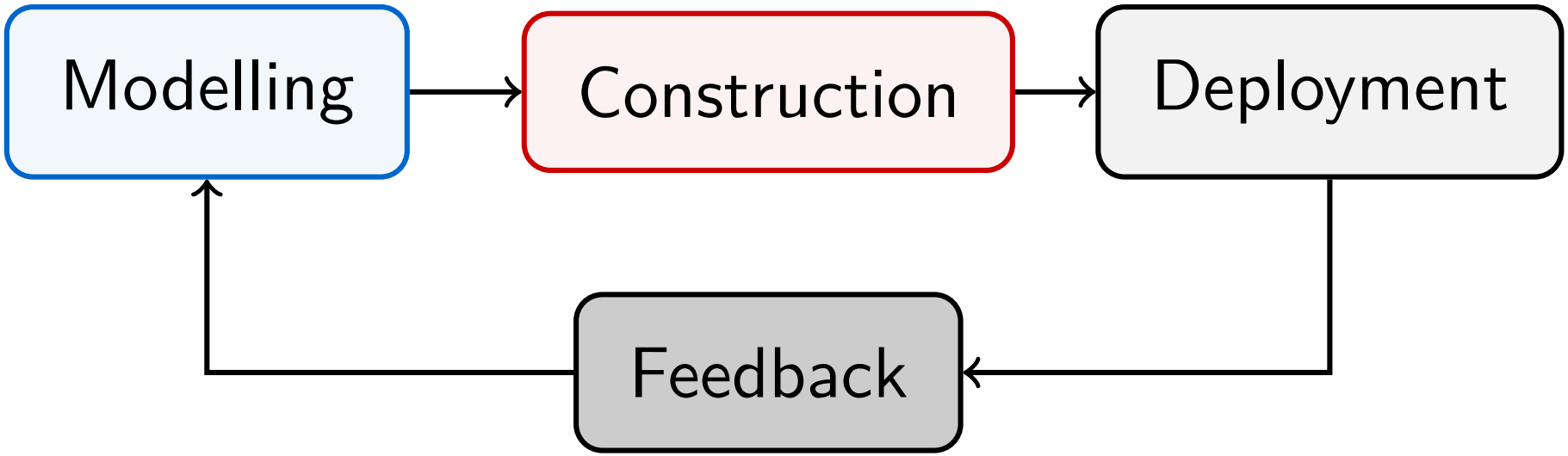


Example: Library System Construction & Deployment

- **Modelling:** Created an ER diagram showing the 1-to-many relationship between Member and Book Issue.
- **Construction:** Developed unit tests for the 'calculate_fine' function before implementing it.
- **Deployment:** Delivered a web interface, provided a 2-page user guide for Librarians, and set up a feedback email.

Summary

- Modelling principles bridge the gap between requirements and technical design.
- Construction principles emphasize preparation, clean coding, and focused testing.
- Deployment principles focus on managing expectations, smooth delivery, and establishing support loops.



Next Lecture Preview

- Next time: Deep Dive into Requirement Engineering.
- We will specifically look at Requirement Gathering and Analysis techniques.

Questions?

Recap & Agenda

Recap: SE Core Principles

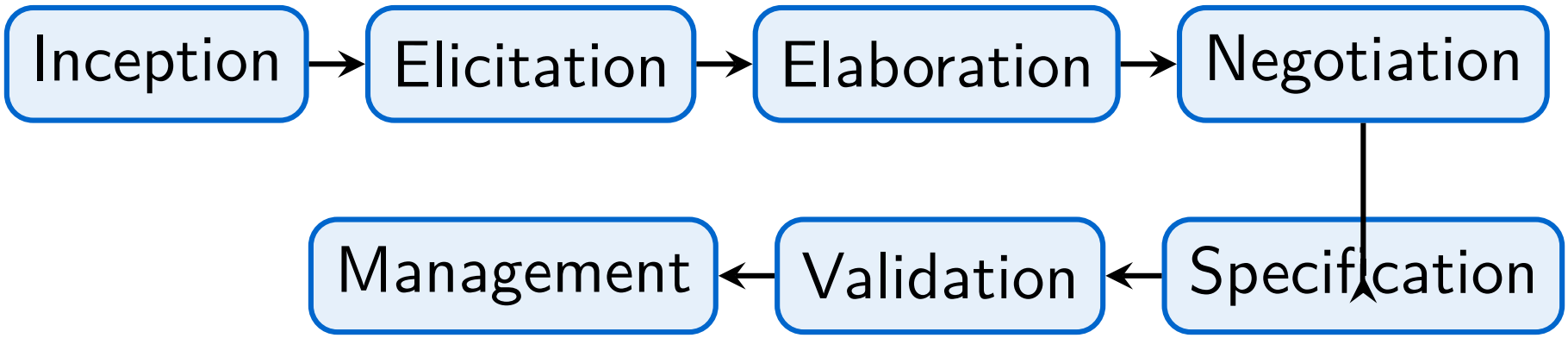
- Modelling, Construction, Software deployment.

Agenda

- What is Requirement Engineering?
- Types of Requirements: Functional vs Non-Functional
- Requirement Elicitation (Gathering) Techniques
- Requirement Analysis and Negotiation
- Analysis Modeling Basics

What is Requirement Engineering?

- Requirement Engineering (RE) is the process of defining, documenting, and maintaining software requirements.
- It provides the appropriate mechanism for understanding what the customer wants.
- Key tasks: Inception, Elicitation, Elaboration, Negotiation, Specification, Validation, and Management.



Functional vs. Non-Functional Requirements

- **Functional Requirements (FRs):** What the system MUST do. (e.g., The system shall calculate fines for overdue books).
- **Non-Functional Requirements (NFRs):** How the system performs. Qualities like security, performance, usability.
- Examples of NFRs: The search query must return results in $<$ 2 seconds; Passwords must be hashed.

Functional Requirements (FRs)	**Non-Functional Requirements (NFRs)**
Features	Speed
Actions	Security
e.g. Issue Book	e.g. Response $<$ 2s

Requirement Elicitation Techniques (1/2)

Elicitation is the act of drawing out requirements from stakeholders.

- **1. Interviews:** One-on-one sessions. Good for deep insights. (e.g., Interviewing the Head Librarian).
- **2. Questionnaires/Surveys:** Broad reach. Good for getting data from hundreds of library members.
- **3. Observation:** Watch users do their job. Helps uncover 'unspoken' rules.

Requirement Elicitation Techniques (2/2)

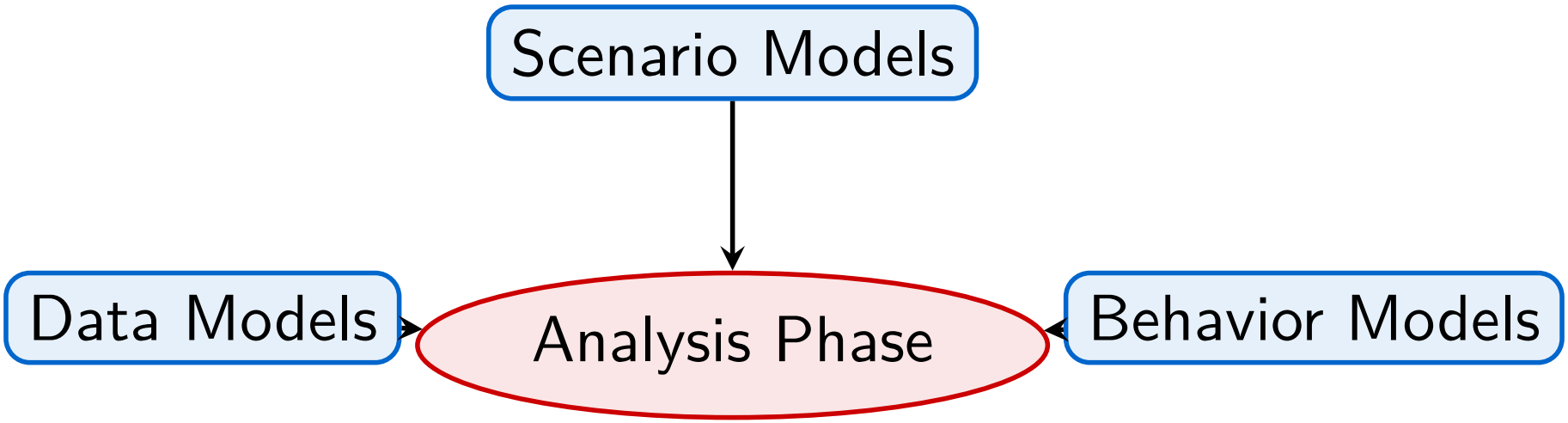
- **4. Document Analysis:** Reviewing existing manuals, forms, and regulations.
- **5. Brainstorming:** Unstructured group thinking to generate new ideas.
- **6. JAD (Joint Application Development):** Structured workshops with stakeholders and IT teams working together.

Requirement Analysis

- Once gathered, requirements are a mess. Analysis organizes them.
- Goal: Identify missing requirements, resolve conflicts, and categorize them.
- Conflict example: Librarian wants strict 3-step verification for borrowing (Security). Members want 1-click borrowing (Usability).
- Analysis leads to Negotiation: Balancing FRs against NFRs and budget/time constraints.

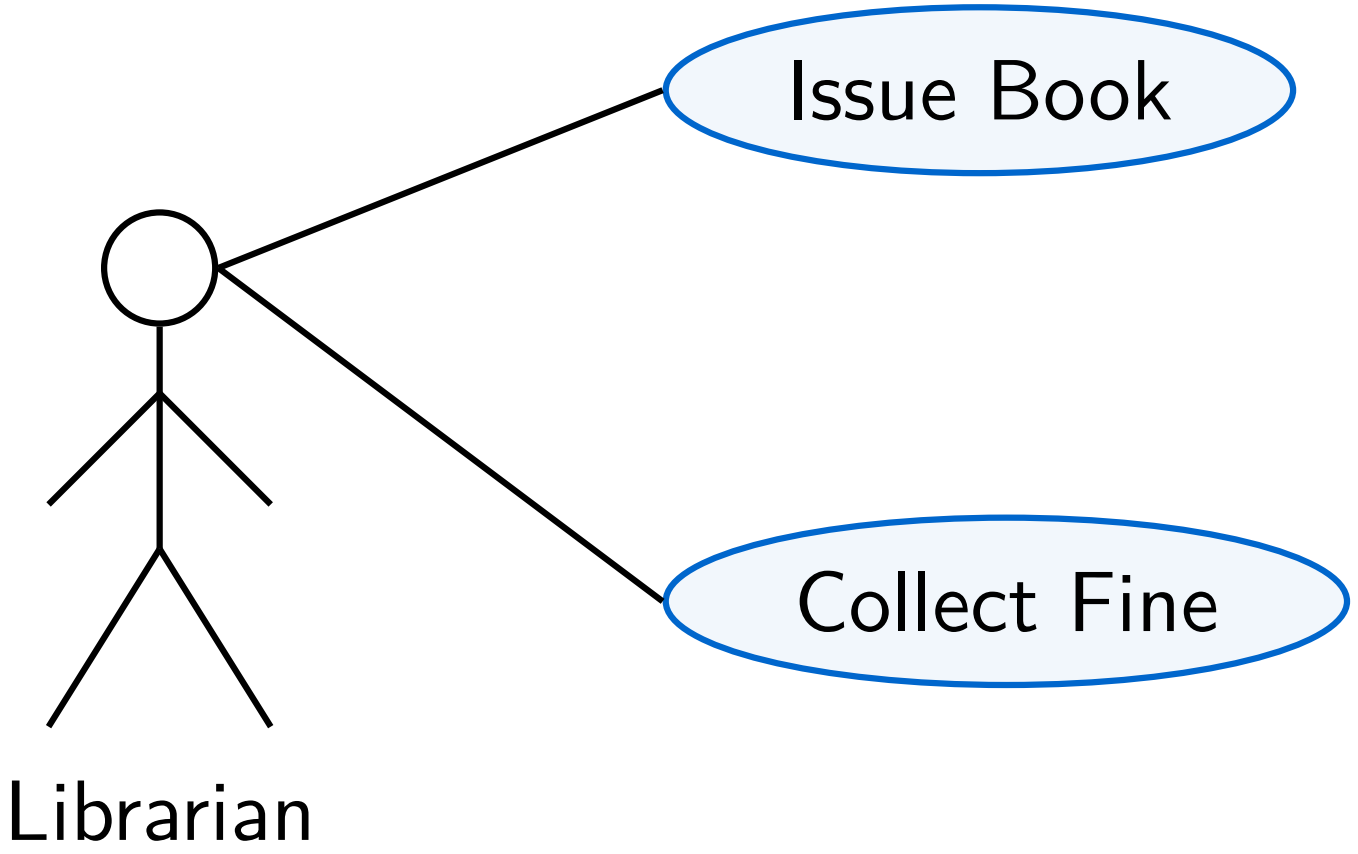
Analysis Modeling

- We create models to better analyze the requirements visually.
- **Scenario-based models:** Use Cases (How users interact with the system).
- **Data models:** Entity-Relationship Diagrams (ERD) (What data is stored).
- **Behavioral models:** State Diagrams (How the system responds to events).



Deep Dive: Use Case Modeling

- Use cases describe a specific usage scenario from the user's point of view.
- Components: Actors (Member, Librarian) and Use Cases (Search Book, Issue Book).
- Helps ensure all functional requirements are mapped to an actual user goal.



Requirement Specification (SRS)

- The final output of this phase is the Software Requirements Specification (SRS) document.
- It serves as a contract between the client and the development team.
- Must be unambiguous, complete, verifiable, and traceable.

Example: Library System RE in Action

- **Elicitation:** Interviewed Librarian. Found they need a way to track damaged books.
- **Analysis:** Categorized as a Functional Requirement. Modeled a 'Report Damage' Use Case.
- **NFR Elicited:** Must be able to scan barcodes (Performance/Hardware constraint).
- **Specification:** Added to Section 3 of the SRS document.

Summary

- Requirement Engineering involves Inception, Elicitation, Analysis, and Specification.
- Functional requirements define 'what' it does; Non-Functional define 'how well' it does it.
- Techniques like interviews, observation, and JAD help gather raw needs.
- Modeling and negotiation refine these into a final SRS document.

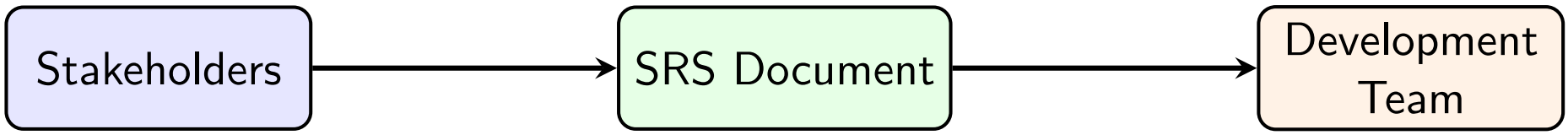
Next Lecture Preview

- **Next time:** System Modeling using UML.
- We will learn how to draw detailed Class Diagrams, Activity Diagrams, and Sequence Diagrams based on our SRS.

Questions?

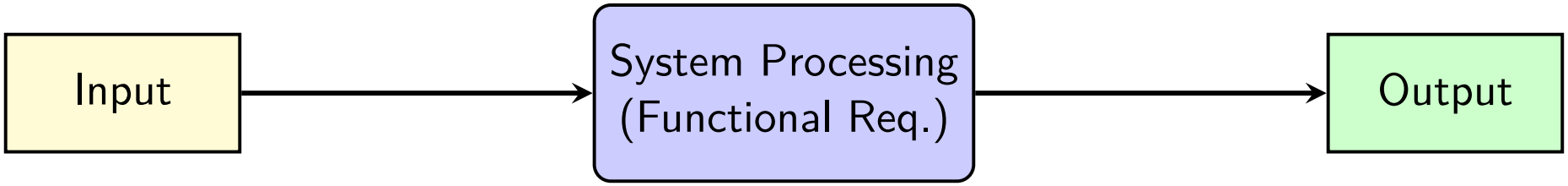
The SRS Document Context

- SRS is a formal document representing the software's intended behavior.
- It bridges the gap between stakeholders (clients) and developers.
- Acts as a baseline for project planning, design, coding, and testing.
- Typically follows standard templates like IEEE 830-1998.



What are Functional Requirements?

- Define WHAT the system must do.
- Specify the behavior between the system and its environment.
- Include calculations, technical details, data manipulation and processing.
- Describe system reactions to specific inputs.



Characteristics of Good Functional Requirements

- **Unambiguous:** Only one possible interpretation.
- **Testable:** Can be verified through software testing.
- **Complete:** All required functions are defined.
- **Consistent:** No two requirements contradict each other.
- **Traceable:** Can be tracked from origin to implementation.

Syntax of a Functional Requirement

- Use the word 'SHALL' to indicate a mandatory requirement.
- Standard Format: The [System/Role] shall [Action] [Object] [Condition].
- Example: The System shall generate a late fee of Rs. 10 per day if the return date is exceeded.
- Assign unique IDs (e.g., FR-01, FR-02) for traceability.

Syntax Breakdown

[System] + shall + [Action] + [Condition]

Example: Library Management System (User Management)

- **FR-101:** The system shall allow the Librarian to register a new Member using Name, Email, and Phone Number.
- **FR-102:** The system shall assign a unique 6-digit alphanumeric Member ID upon successful registration.
- **FR-103:** The system shall send an automated email with default login credentials to the newly registered Member.

Example: Library Management System (Book Operations)

- **FR-201:** The system shall allow a Member to search for books by Title, Author, or ISBN.
- **FR-202:** The system shall display the availability status (Available / Checked Out) in search results.
- **FR-203:** The system shall prevent a Member from issuing more than 3 books simultaneously.

Data Flow and Functional Requirements

- Functional requirements often translate directly to Data Flow Diagrams (DFDs).
- Inputs specified in FRs become incoming data flows.
- Actions become DFD processes.
- Outputs become outgoing data flows or data store updates.

Differentiating Functional vs Business Requirements

Business Requirements	**Functional Requirements**
High-level objectives of the organization.	Specific system behaviors to achieve business objectives.
Example: 'Reduce book loss by 20%.'	*Example:* 'The system shall flag any unreturned book after 30 days and alert the Librarian.'

Validating Functional Requirements

- **Requirements Reviews:** Peer reviews and stakeholder walkthroughs.
- **Prototyping:** Building mockups to clarify poorly understood requirements.
- **Test-Case Generation:** Writing conceptual test cases for each FR to prove testability.
- **Traceability Matrix:** Ensuring every FR maps to a specific business need.

Req ID	Business Need	Test Case ID
FR-101	BN-02: Manage Users	TC-U-01
FR-201	BN-05: Access Catalog	TC-C-04

Common Pitfalls

- **Vagueness:** Using terms like 'user-friendly' or 'fast'.
- **Implementation Bias:** Dictating HOW the system should do it (e.g., 'Use an Oracle DB') instead of WHAT.
- **Missing Edge Cases:** Forgetting to specify error handling behavior.
- **Contradictions:** FR-1 says 'Admin can delete' but FR-20 says 'Records must be immutable'.

Summary

- Functional requirements detail the specific behaviors and functions of a system.
- They must be unambiguous, complete, and testable.
- A formal syntax (e.g., using 'shall') and unique identifiers are essential.
- They serve as the foundation for design and testing phases.

Next Lecture Preview

- **Topic:** Non-Functional Requirements (NFRs).
- We will explore the 'how' of the system: Performance, Security, Reliability.
- Understanding the difference between FRs and NFRs.
- Metrics for measuring NFRs.

Questions?

Recap & Agenda

Recap: Detailed study of Functional Requirements, their characteristics, and syntax using the Library Management System example.

Today's Agenda:

- What are Non-Functional Requirements (NFRs)?
- Classification of NFRs
- Key NFR Types (Performance, Security, Reliability)
- Measuring NFRs (Metrics)
- Functional vs. Non-Functional Requirements
- Documenting NFRs in SRS

What are Non-Functional Requirements?

- Define system properties and constraints (the 'HOW').
- Focus on quality attributes like performance, usability, and security.
- Apply to the system as a whole, rather than individual features.
- Failure to meet NFRs can make the entire system unusable.

Classification of NFRs

- **1. Product Requirements:** Relate directly to software behavior (e.g., speed, space).
- **2. Organizational Requirements:** Derived from policies (e.g., specific IDE, coding standards).
- **3. External Requirements:** Derived from outside factors (e.g., legislative, ethical, interoperability).

Performance Requirements

- Specify timing, processing speed, and resource consumption.
- **Response Time:** How fast the system responds to input.
- **Throughput:** Number of transactions processed per second.
- **Capacity:** Maximum number of concurrent users.

Example: LMS Performance NFRs

- **NFR-P1:** The system shall process a book search query within 2 seconds for a database of 100,000 records.
- **NFR-P2:** The system shall support at least 500 concurrent user sessions without performance degradation.
- **NFR-P3:** Book issue transactions must complete within 3 seconds.

Security Requirements

- Protect data and system resources from unauthorized access.
- **Authentication:** Verifying user identity.
- **Authorization:** Restricting access to specific features based on roles.
- **Data Encryption:** Protecting data at rest and in transit (e.g., AES-256).

Reliability and Availability Requirements

- **Reliability:** Probability that the system operates without failure under given conditions.
- Measured by Mean Time Between Failures (MTBF).
- **Availability:** The proportion of time the system is operational.
- Usually expressed as 'Nines' (e.g., 99.9% uptime = $\sim$8.7 hours downtime/year).

Usability Requirements

- Focus on user experience and ease of use.
- **Learnability:** Time required for a novice to learn the system.
- **Efficiency:** Number of clicks required to perform a standard task.
- **Accessibility:** Compliance with standards like WCAG for users with disabilities.

Functional vs Non-Functional Requirements

Functional (FR)	**Non-Functional (NFR)**
Defines a specific behavior	Defines a quality attribute
(e.g., System sends an email)	(e.g., Email is sent within 5 seconds)
Captured in Use Cases	Act as constraints on the system architecture

Measuring NFRs (Metrics)

NFRs MUST be verifiable. Use clear metrics.

- **Speed Metric:** Processed transactions/second.
- **Size Metric:** MBytes/Number of ROM chips.
- **Ease of use Metric:** Training time.
- **Reliability Metric:** Rate of failure occurrence.

Documenting NFRs in SRS

- Often grouped together in a dedicated section (e.g., Section 3 of IEEE 830).
- Use a consistent naming convention (e.g., SEC-01 for Security, PERF-01 for Performance).
- Must be prioritized, as NFRs often conflict (e.g., highest security often degrades performance).

Summary

- NFRs define the quality attributes and constraints of a system.
- They include performance, security, reliability, and usability.
- They must be quantified with specific metrics to be testable.
- They are just as critical to project success as functional requirements.

Next Lecture Preview

Topic: Introduction to Software Design.

- Transitioning from 'What' to 'How'.
- Characteristics of Good Software Design.
- Concepts of Coupling and Cohesion.

Questions?

Recap & Agenda

Recap:

- Understanding Non-Functional Requirements (Performance, Security) and documenting them in the SRS.

Agenda:

- Introduction to Software Design
- The Transition from Requirements to Design
- Key Characteristics of Good Design
- Modularity and Abstraction
- Deep Dive: Cohesion
- Deep Dive: Coupling
- Cohesion vs. Coupling

What is Software Design?

- The process of defining the architecture, components, interfaces, and other characteristics of a system.
- Translates the SRS (the 'WHAT') into a blueprint for implementation (the 'HOW').
- Produces Design Documents (High-Level Design and Low-Level Design).
- Crucial for minimizing errors before coding begins.

Characteristics of Good Software Design

- **Correctness:** Satisfies all requirements in the SRS.
- **Understandability:** Easy for developers to comprehend.
- **Efficiency:** Optimal use of processing and memory resources.
- **Maintainability:** Easy to fix bugs and add new features.
- **Modularity:** Divided into manageable, independent components.

Modularity

- Dividing a complex system into smaller, manageable, and independent modules.
- Benefits:
  - Easier to develop in parallel.
  - Easier to test and debug.
  - Increases reusability of components.
- Example: A Library System separated into 'User Management', 'Catalog', and 'Billing' modules.

Abstraction

- Hiding unnecessary implementation details while exposing essential features.
- **Data Abstraction:** Hiding internal data representations (e.g., private variables in classes).
- **Control Abstraction:** Hiding the exact sequence of execution behind a simple interface.
- Reduces cognitive load on developers using the module.

Analogy

Driving a car: The driver uses the steering wheel (abstraction) without needing to understand the engine mechanics (implementation).

Core Design Principles: Cohesion and Coupling

- The two most important metrics for evaluating modularity.
- **Cohesion:** Measures the strength of the relationship between elements WITHIN a single module.
- **Coupling:** Measures the degree of interdependence BETWEEN different modules.
- **The Golden Rule:** Aim for HIGH Cohesion and LOW Coupling.

Deep Dive: Cohesion

- High Cohesion means a module performs one well-defined task.
- All elements in the module are focused on the same single purpose.
- Makes the module robust, reusable, and easy to understand.
- **Example of Low Cohesion:** A 'Utils' class that formats dates, connects to DBs, and parses XML.
- **Example of High Cohesion:** A 'DateFormatter' class that only handles date conversions.

Types of Cohesion (Best to Worst)

1. **Functional (Best):** Every part contributes to a single, specific task.
2. **Sequential:** Output of one part is input to the next.
3. **Communicational:** Parts operate on the same input data.
4. **Temporal:** Parts are executed at the same time (e.g., Initialization module).
5. **Logical:** Parts perform similar logical tasks (e.g., all I/O operations).
6. **Coincidental (Worst):** No meaningful relationship.

Deep Dive: Coupling

- Low Coupling means modules are largely independent of each other.
- Changes in one module should have minimal impact on other modules.
- High Coupling leads to the 'ripple effect'—changing one thing breaks another.
- Modules should communicate through well-defined, simple interfaces.

Types of Coupling (Best to Worst)

1. **Data (Best):** Modules communicate by passing only necessary data as parameters.
2. **Stamp:** Modules share composite data structures (e.g., passing a whole Record when only one field is needed).
3. **Control:** One module passes control flags to dictate logic in another.
4. **Common:** Modules share global data.
5. **Content (Worst):** One module directly modifies the internal workings of another.

Cohesion vs. Coupling

Feature	Cohesion	Coupling
Scope	INTRA-module (inside)	INTER-module (between)
Goal	Maximize Cohesion	Minimize Coupling
Effect (High)	Better readability and reusability	Ripple effect, hard to maintain
Effect (Low)	Hard to understand (arbitrary)	Better maintainability and easier testing

Summary

- Good software design bridges the gap between requirements and implementation.
- Key characteristics include modularity, understandability, and maintainability.
- Modularity is achieved through Abstraction.
- The golden rule of design is HIGH Cohesion and LOW Coupling.

Next Lecture Preview

- **Topic:** Introduction to UML (Unified Modeling Language).
- Visualizing software architecture.
- Different types of UML diagrams.
- Use Case Diagrams in detail.

Questions?

What is Software Analysis?

- Focuses on 'WHAT' the system needs to do.
- Outcome: Requirements Specification (SRS).
- Describes the problem domain.
- Identifies user needs, system features, and constraints.
- Models: Use cases, DFDs (Logical view).

What is Software Design?

- Focuses on 'HOW' the system will meet its requirements.
- Outcome: Software Design Document (SDD).
- Describes the solution domain.
- Defines architecture, components, interfaces, and data structures.
- Models: Class diagrams, ER Diagrams, Architecture diagrams.

Analysis vs. Design: Core Focus

Software Analysis	**Software Design**
Problem Domain (Customer-centric)	Solution Domain (Developer-centric)
Models the real-world environment	Models the software internals

Analysis vs. Design: Output and Artifacts

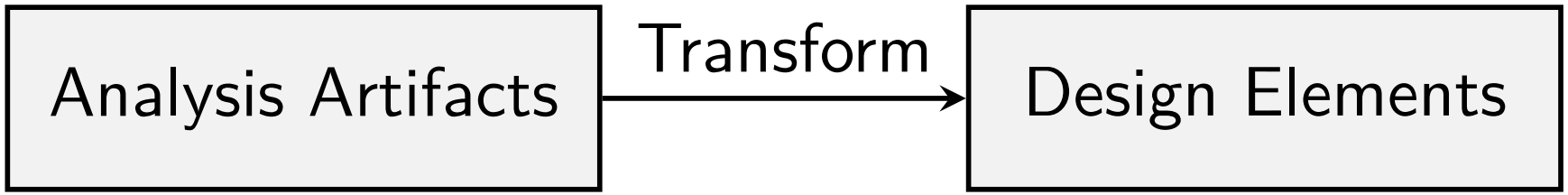
- Analysis Output: Software Requirements Specification (SRS), logical models.
- Design Output: Software Design Document (SDD), physical models.
- Analysis involves logical DFDs; Design involves physical DFDs and architecture.
- Analysis defines interfaces functionally; Design defines them technically.

Analysis vs. Design: Perspective and Abstraction

- Analysis: High level of abstraction. Hides implementation details.
- Design: Lower level of abstraction. Introduces technical specifics (DB, UI, Network).
- Analysis is independent of technology (mostly).
- Design is heavily dependent on chosen technologies (e.g., Java, SQL).

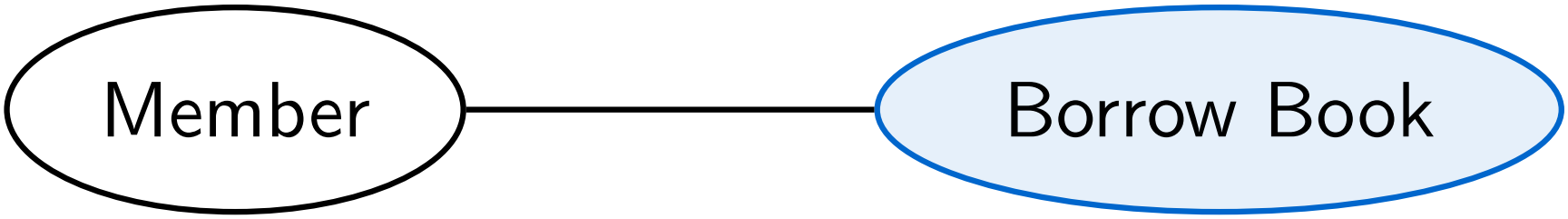
Transformation: Analysis Model to Design Model

- Data Dictionary $\rightarrow$ Data Design (Database Schema).
- DFDs / Use Cases $\rightarrow$ Architectural Design.
- Process Specification $\rightarrow$ Component-level Design.
- State Transition Diagram $\rightarrow$ Interface Design.



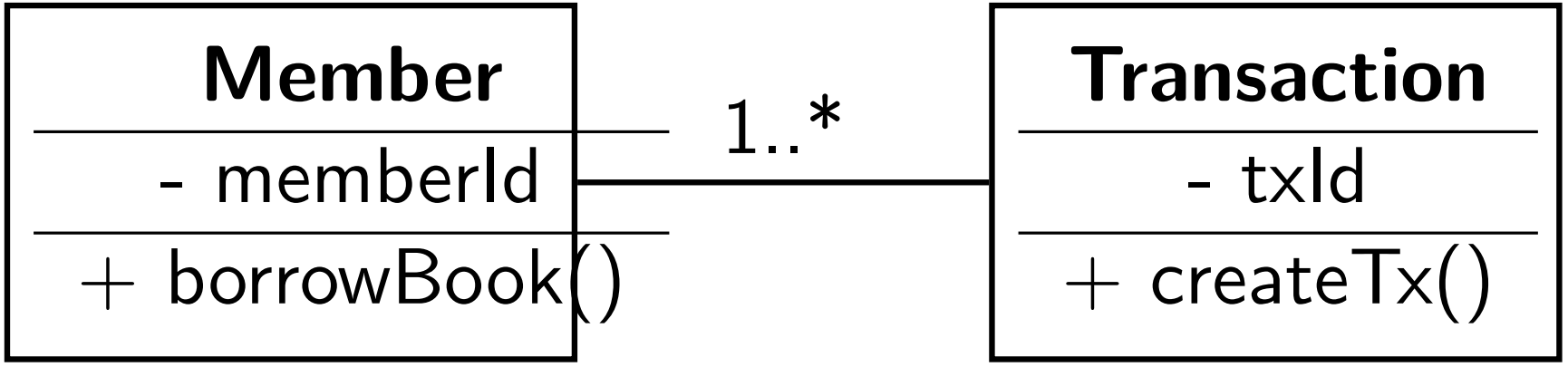
Example: Library Management System - Analysis

- Requirement: A user can borrow a book.
- Analysis Model: Use Case 'Borrow Book'. Actor: Member.
- Focus: What happens when a book is borrowed? (Check availability, update limit).



Example: Library Management System - Design

- Design Model: Class Diagram with ‘Book‘, ‘Member‘, and ‘Transaction‘ classes.
- Focus: Methods like `checkAvailability()`, attributes like `issueDate`.
- Database Schema: Tables for Books (PK: BookID) and Transactions.



Objectives of Good Design

- Firm foundation for the coding phase.
- Accommodate changes easily (Maintainability).
- Maximize module independence (High Cohesion, Low Coupling).
- Ensure performance and security constraints are met.

Summary

- Analysis = WHAT (Problem Domain, SRS).
- Design = HOW (Solution Domain, SDD).
- Design bridges the gap between requirements and implementation.
- Transformation maps logical models to physical/technical blueprints.

Next Lecture Preview

- Topic: Classification of Cohesion.
- Understanding module strength.
- Different types of cohesion (Functional, Sequential, etc.).

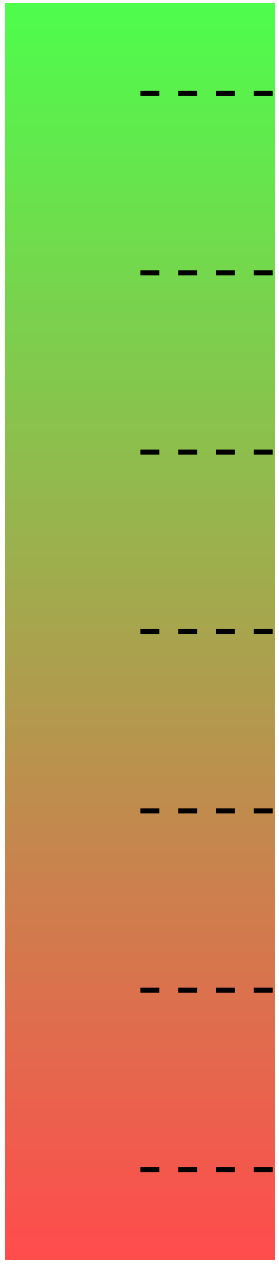
Questions?

What is Cohesion?

- Cohesion is a measure of the functional strength of a module.
- It measures how closely related and focused the responsibilities of a single module are.
- Intra-module interaction: Focuses on elements WITHIN a module.
- Rule of Thumb: A module should do exactly ONE thing, and do it well.

Strong Cohesion (Module)

The Cohesion Spectrum



- - - - - - - - - - - - -> **Functional (Best)**

- - - - - - - - - - - - -> Sequential

- - - - - - - - - - - - -> Communicational

- - - - - - - - - - - - -> Procedural

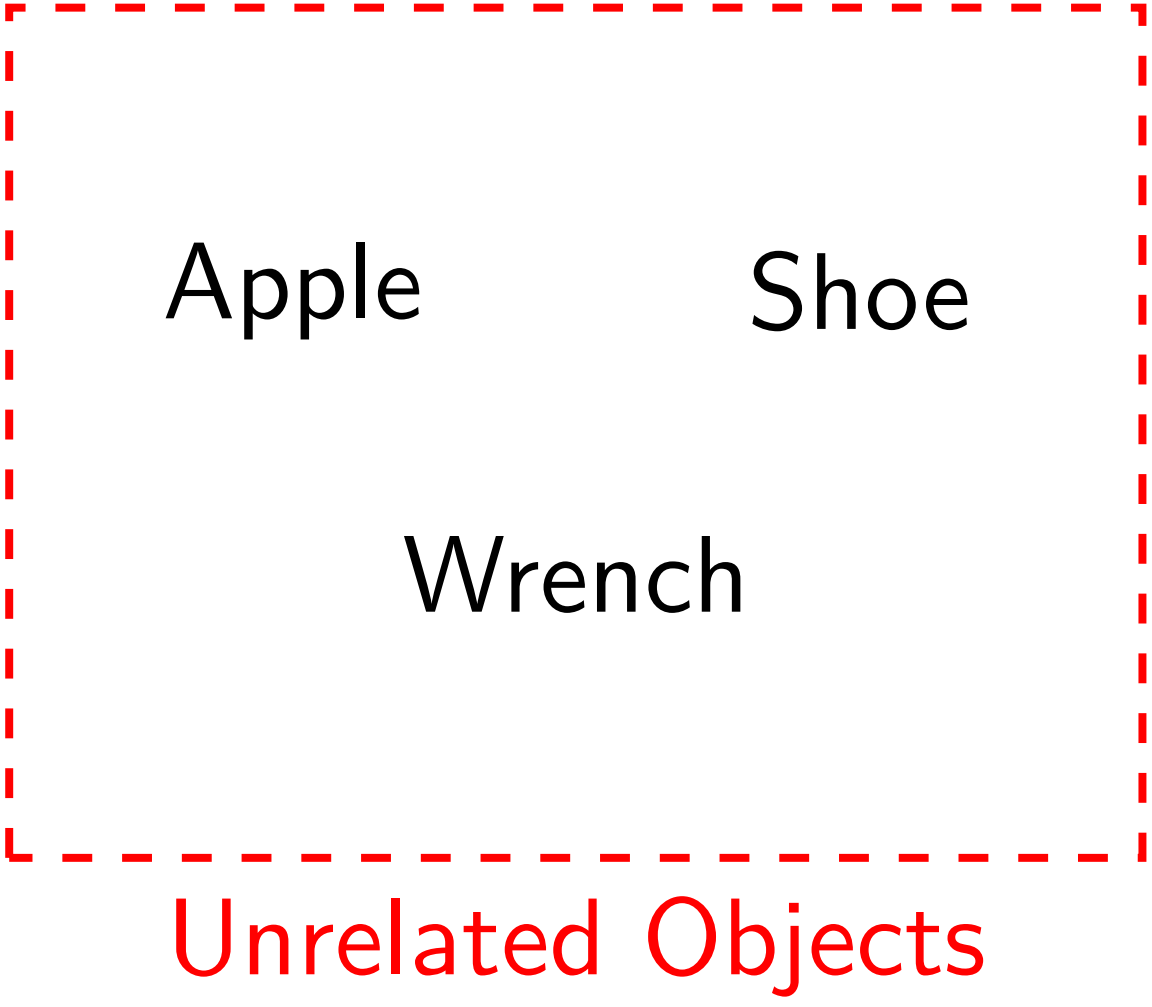
- - - - - - - - - - - - -> Temporal

- - - - - - - - - - - - -> Logical

- - - - - - - - - - - - -> **Coincidental (Worst)**

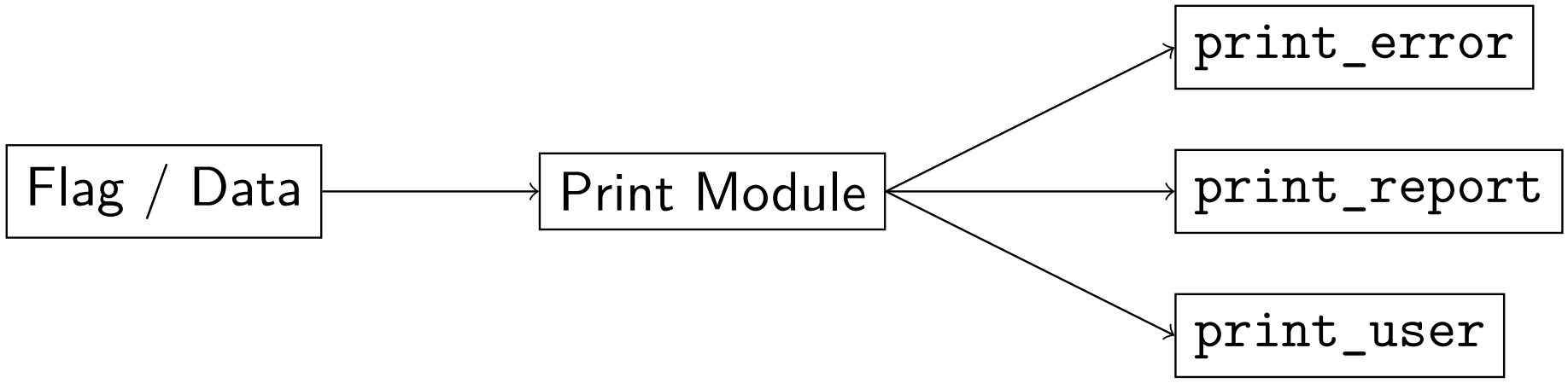
1. Coincidental Cohesion (Worst)

- Elements are grouped together arbitrarily.
- No meaningful relationship between the functions in the module.
- Like a 'utility' module where random unrelated functions are dumped.
- Extremely hard to maintain and reuse.



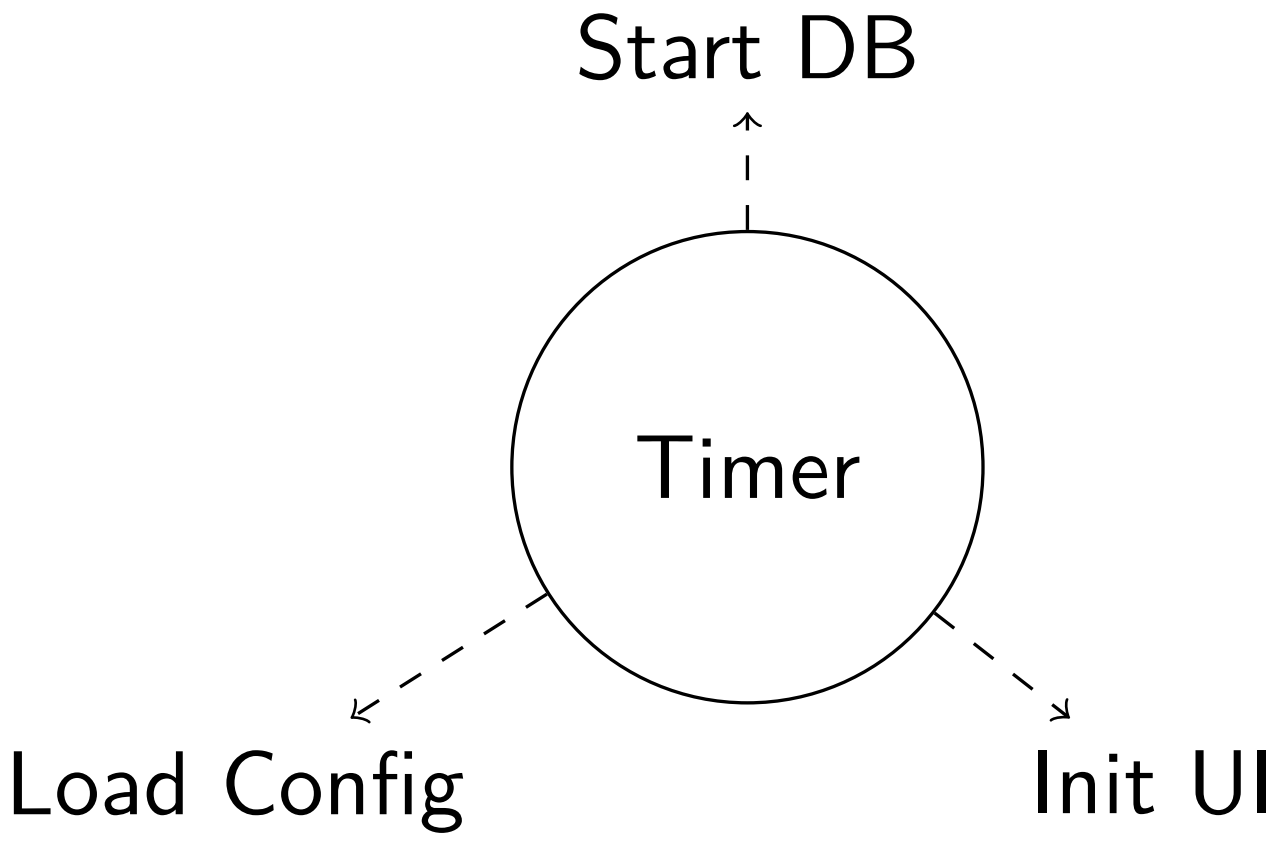
2. Logical Cohesion

- Elements perform similar logical operations, but on different data.
- Example: A module containing all 'print' routines (`print_error`, `print_report`, `print_user_details`).
- A flag is often passed to decide which logic to execute.
- Slightly better than coincidental, but still poor.



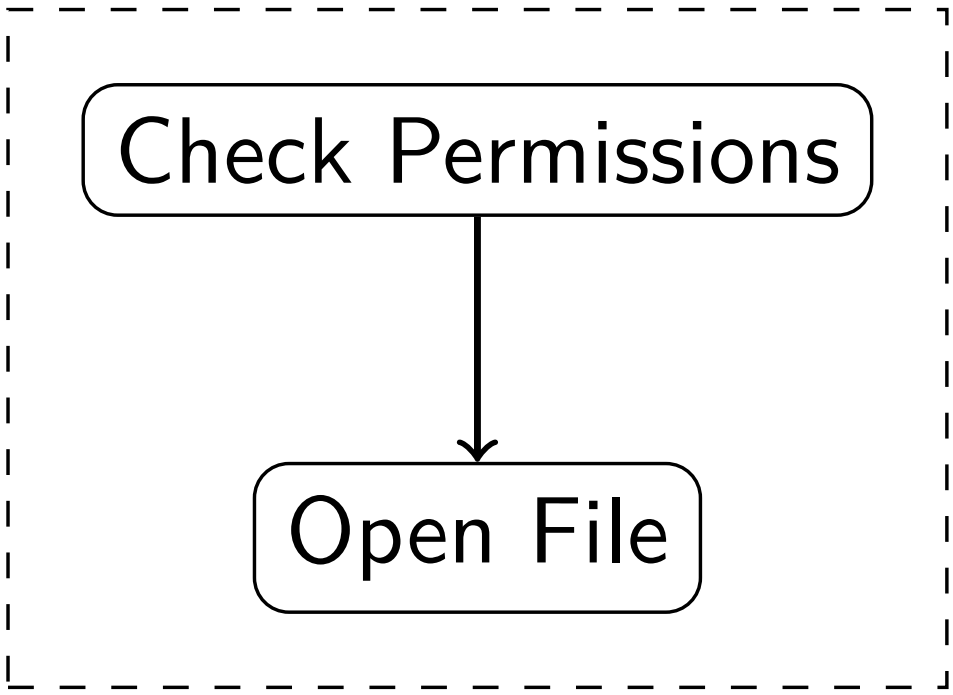
3. Temporal Cohesion

- Elements are related by execution timing.
- Tasks must be executed in the same time span.
- Example: An 'Initialization' module that opens files, connects to DB, and clears arrays.
- Tasks are grouped by WHEN they happen, not WHAT they achieve together.



4. Procedural Cohesion

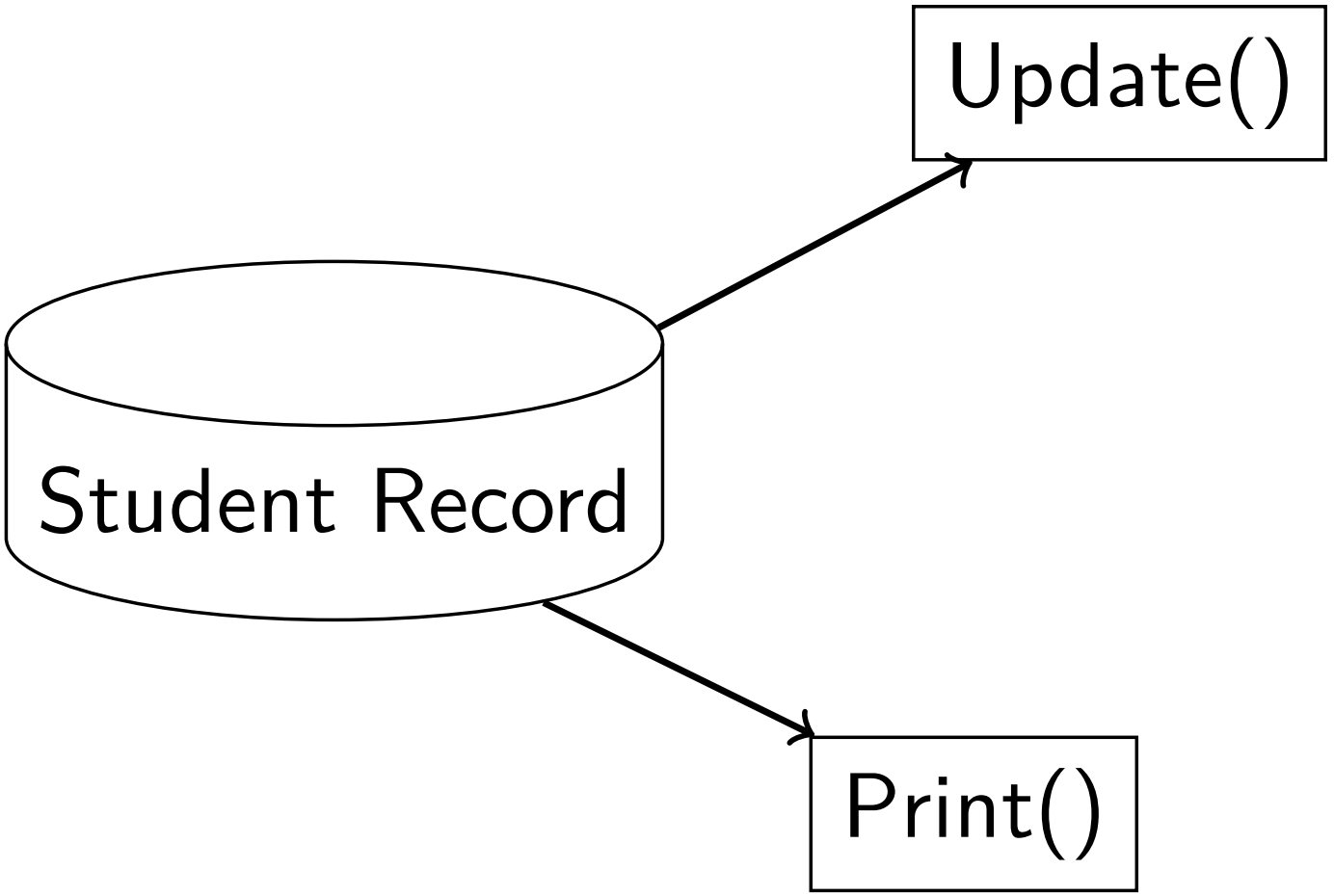
- Elements are grouped because they always follow a specific sequence of execution.
- Example: 'Check file permissions', then 'Open file'.
- They share an execution order but don't necessarily share the same data.
- Often results from flowcharts directly translated into code.



Procedural Module

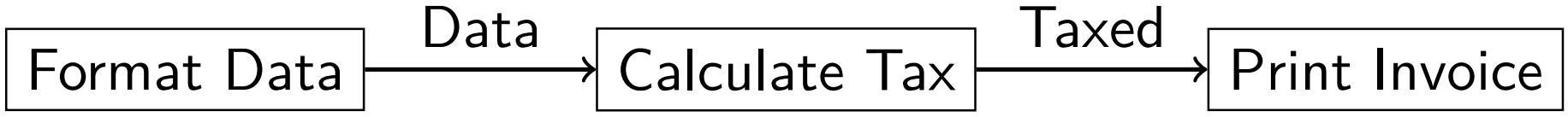
5. Communicational Cohesion

- Elements operate on the same core data structure or dataset.
- Example: A module that 'Updates Student Record' and 'Prints Student Record'.
- Good level of cohesion because data ties the functions together.
- Better reusability than procedural.



6. Sequential Cohesion

- The output of one element serves as the input to the next element.
- Data flows in an assembly line fashion.
- Example: 'Format Data' $\rightarrow$ 'Calculate Tax' $\rightarrow$ 'Print Invoice'.
- Highly related and easy to maintain.



7. Functional Cohesion (Best)

- Every element in the module contributes exactly to ONE single well-defined task.
- Maximum module strength, highest reusability.
- Library Management System Example: `CalculateFine()` module.
- It takes overdue days, applies the formula, and returns the fine amount. Nothing else.

CalculateFine

Input: OverdueDays

Output: Amount

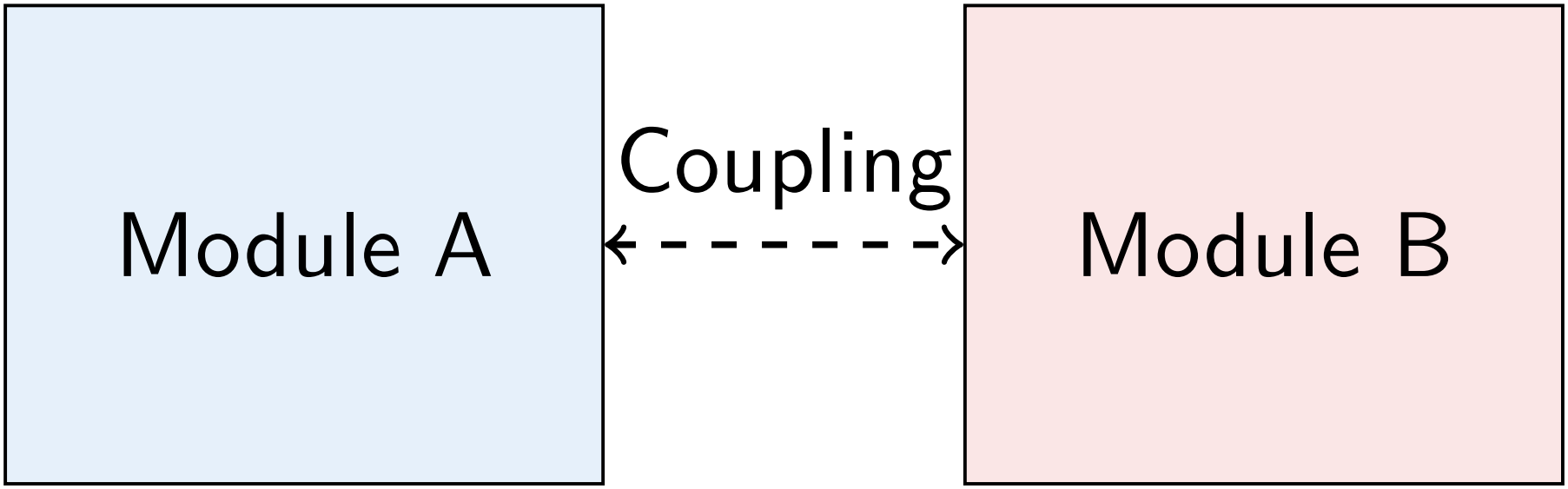
Summary

- Cohesion measures functional strength WITHIN a module.
- High cohesion implies single-mindedness of a module.
- **Goal:** Strive for Functional or Sequential Cohesion.
- **Avoid:** Coincidental and Logical Cohesion.

Cohesion Type	**Desirability**
Functional	**Goal (Best)**
Sequential	Highly Desirable
Communicational	Good
Procedural	Fair
Temporal	Poor
Logical	Very Poor
Coincidental	**Avoid (Worst)**

Next Lecture Preview

- Topic: Classification of Coupling.
- Understanding inter-module dependencies.
- Balancing High Cohesion with Low Coupling.



Questions?

Recap & Agenda

Previous Lecture Recap:

- Classification of Cohesion and intra-module strength.

Today's Agenda:

- What is Coupling?
- Cohesion vs. Coupling
- Why is Low Coupling Desirable?
- Levels of Coupling (Worst to Best)
- Detailed Classification of Coupling

What is Coupling?

Coupling

Coupling is a measure of the degree of interdependence between software modules.

- It measures how strongly one module is connected to, or relies on, other modules.
- Inter-module interaction: Focuses on connections BETWEEN modules.
- **Goal:** Modules should be as independent as possible (Low Coupling).

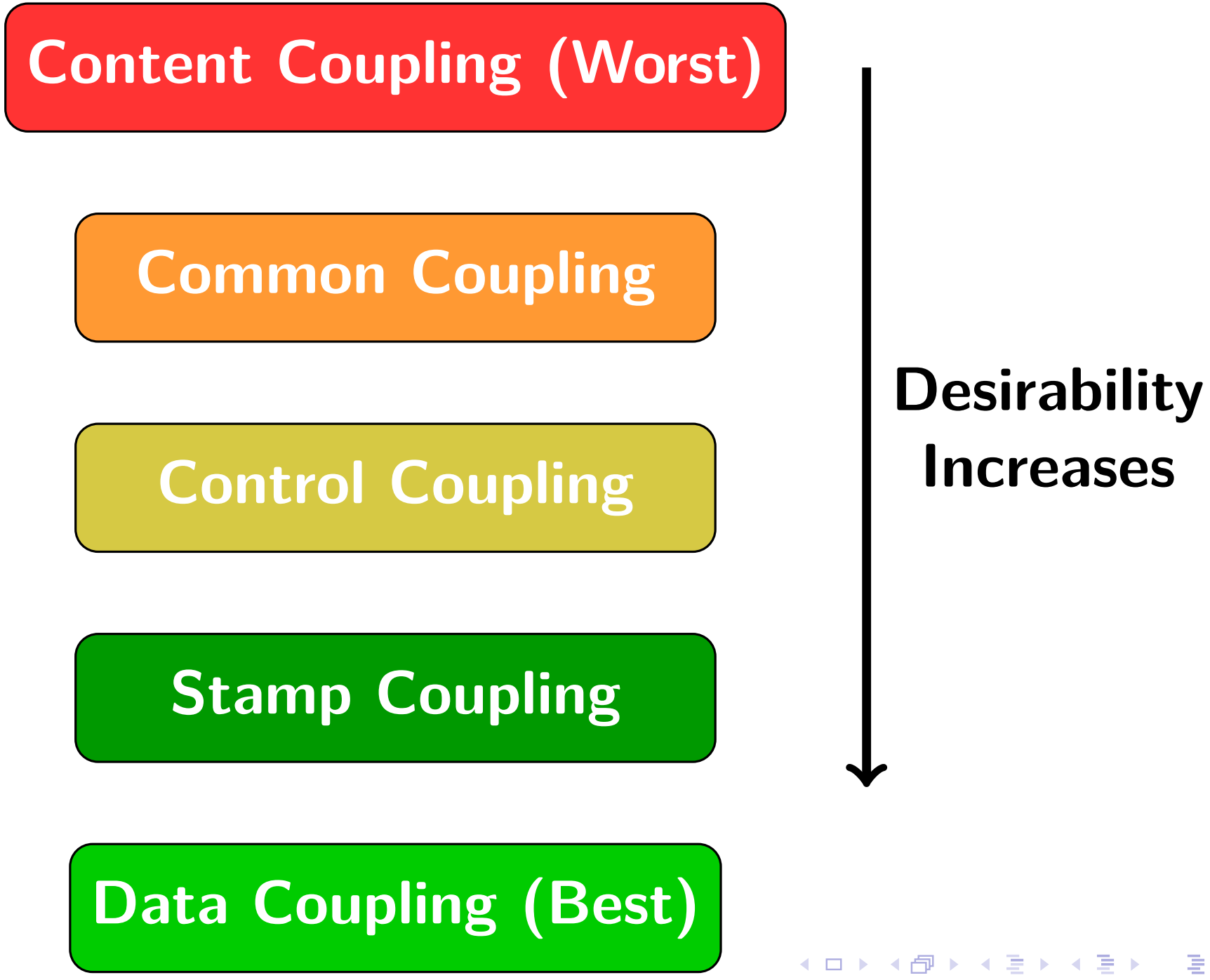
The Golden Rule of Software Design

The Formula for Success

HIGH Cohesion + LOW Coupling = Excellent Design

- **High Cohesion:** Modules are highly focused on a single task.
- **Low Coupling:** Modules have minimal dependency on one another.
- **Benefits:** Easier to maintain, test, and reuse.

The Coupling Spectrum



1. Content Coupling (Worst)

- One module directly modifies or relies on the internal working/data of another module.
- Also known as *Pathological Coupling*.
- **Example:** Module A directly alters local variables or memory offsets inside Module B.
- A change in Module B's internal structure instantly breaks Module A.

2. Common Coupling

- Multiple modules share the same global data space or global variables.
- **Example:** A global configuration array accessed by UI, Database, and Logic modules directly.
- **Problems:** Difficult to track who changed the data. Reduces reusability.
- Violates the principle of data hiding.

3. Control Coupling

- One module controls the flow of execution in another module.
- Happens by passing control flags or boolean variables.
- **Example:** Module A calls Module B with a flag `isStudent`. Module B has an `if-else` based on this flag.
- Implies Module A knows about the internal logic of Module B.

4. Stamp Coupling

- Modules share a composite data structure, but only use a part of it.
- **Example:** Passing an entire `StudentObject` to a module that only needs the `StudentID` to print an ID card.
- Creates unnecessary dependencies on the entire data structure.
- If the `StudentObject` structure changes, the ID card module might need recompiling even if `StudentID` didn't change.

5. Data Coupling (Best)

- Modules communicate by passing only essential, primitive data items via parameters.
- **Example (Library System):** `CalculateFine(memberID, overdueDays)`.
- The module only receives exactly what it needs.
- Independent, highly reusable, and easy to maintain.

No Coupling (Uncoupled)

- Modules have zero communication with each other.
- While they are perfectly independent, they cannot form a system.
- A system requires at least some interaction.
- Therefore, **Data Coupling** is the practical "best" achievable level.

Summary

- Coupling measures inter-module dependency.
- **Goal:** Minimize coupling (aim for Data Coupling).
- Avoid Content and Common (Global) coupling.
- Low coupling ensures a modular, resilient, and maintainable system architecture.

Takeaway 1	Coupling measures the degree of interaction between modules.
Takeaway 2	Data coupling is most desirable, content coupling the least.
Takeaway 3	Ultimate goal: High Cohesion and Low Coupling.

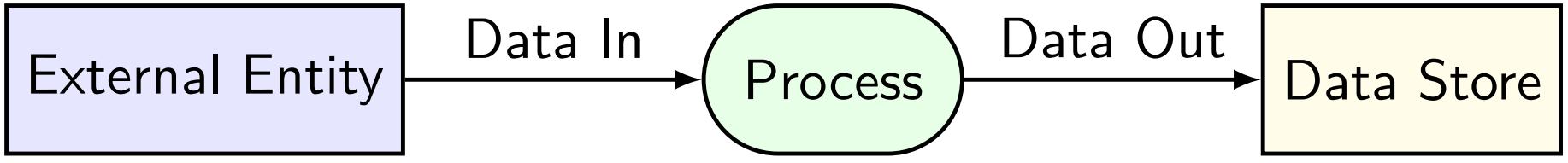
Next Lecture Preview

- **Topic:** Functional Oriented Design vs Object Oriented Design methodologies.
- Different paradigms for approaching software architecture.
- Understanding Top-Down vs. Bottom-Up approaches.

Questions?

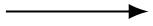
What is a Data Flow Diagram?

- A graphical representation of the 'flow' of data through an information system.
- Focuses on processes, data stores, external entities, and data flows.
- Does not specify control logic (no loops, no decision branches).
- Used during the analysis phase to model system requirements logically.



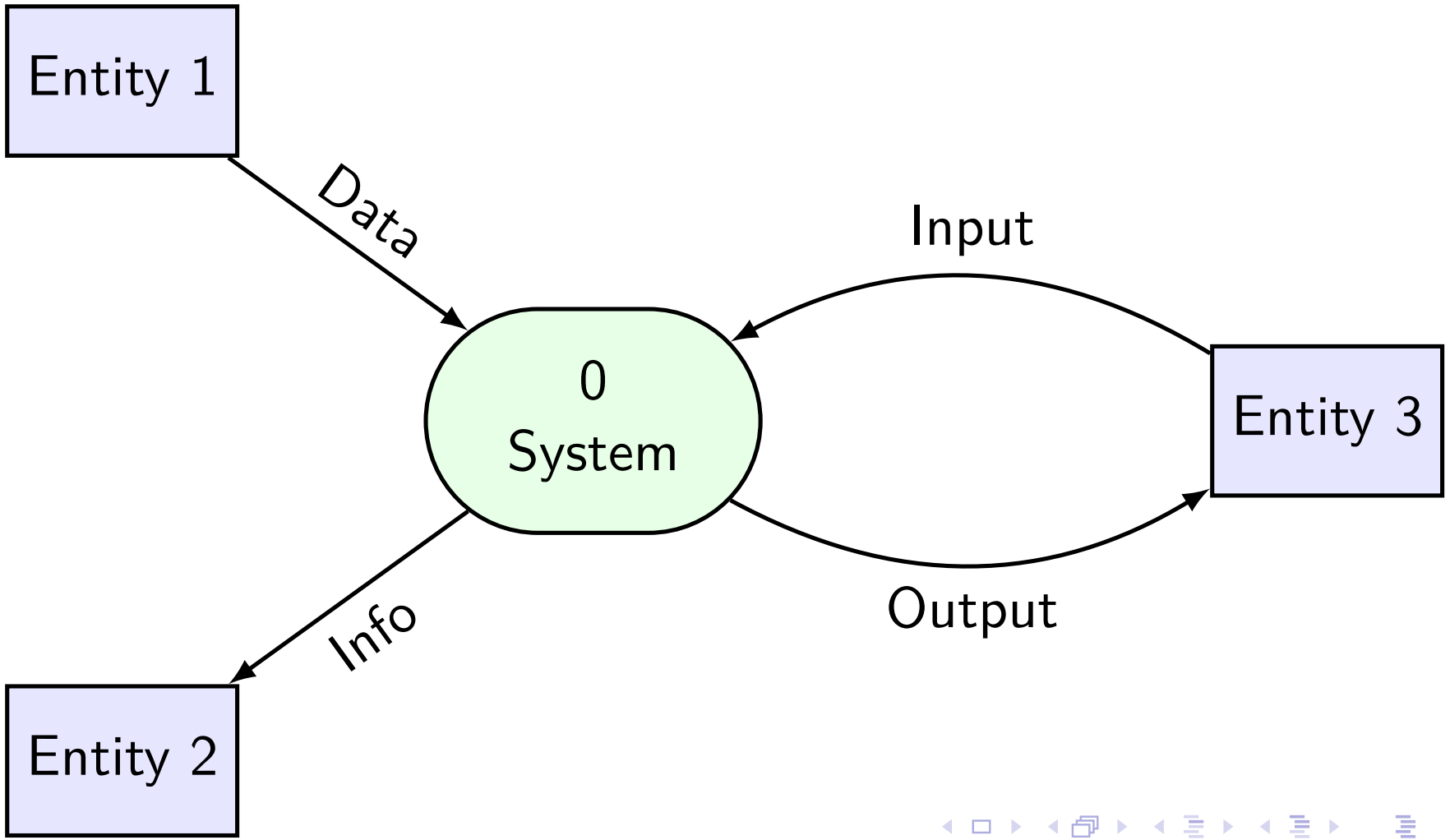
DFD Notations (Gane & Sarson)

- **External Entity:** Represented by a square or double square. A source or destination of data.
- **Process:** Represented by a rounded rectangle. Transforms incoming data flows into outgoing data flows.
- **Data Store:** Represented by an open-ended rectangle. Resting place for data.
- **Data Flow:** Represented by directed arrows. Shows data movement between components.

Component	Symbol	Description
External Entity	Entity	A source or destination of data outside the system boundary.
Process	Process	Transforms incoming data into outgoing data.
Data Store	D1 Store	A resting place for data.
Data Flow		Directed arrows showing data movement.

Context Diagram (Level 0 DFD)

- Highest level view of the system.
- Represents the entire software system as a single process (Process 0).
- Shows interaction with external entities (users, other systems).
- **Contains ZERO data stores inside the context diagram.**



Library Management System: Scope

- **Goal:** Automate library operations like issuing, returning, and managing books.
- **Key Actors (External Entities):** Student (Member), Librarian, Book Vendor.
- **Key Inputs:** Book requests, Member details, Book details.
- **Key Outputs:** Issue receipts, Fine notices, Book orders.

Identifying Entities: Member (Student)

Inputs to System

- Book availability inquiry
- Book issue/return request
- Fine payment

Outputs from System

- Search results
- Issue/Return confirmation
- Fine receipt

Identifying Entities: Librarian

Inputs to System

- Add/Update/Remove Member details
- Add/Update/Remove Book details

Outputs from System

- Inventory reports
- Overdue book lists
- Member status reports

Identifying Entities: Book Vendor

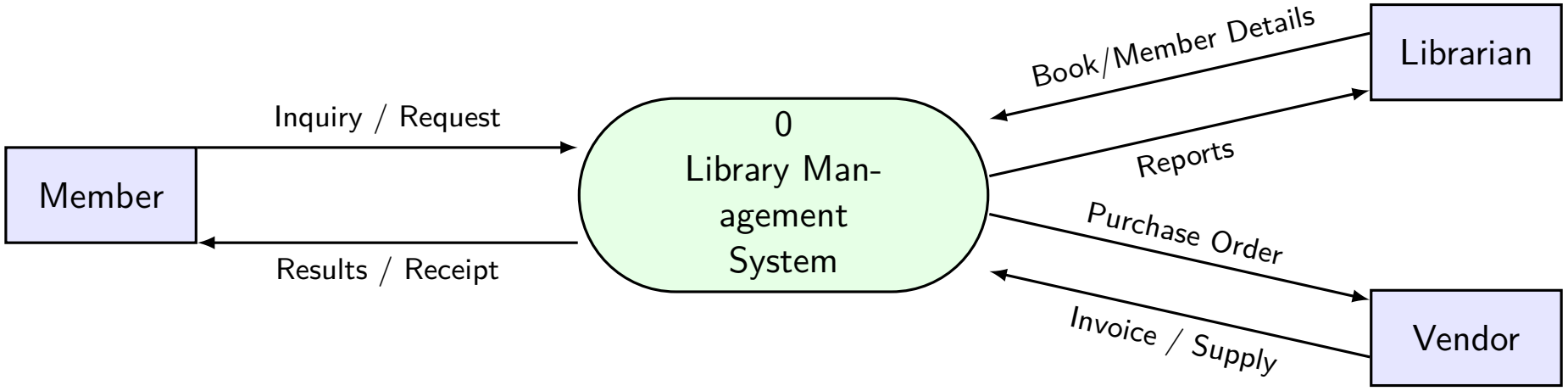
Inputs to System

- Invoice details
- Book supply confirmation

Outputs from System

- Purchase Orders (POs)
- Payment details

Constructing the Context Diagram



Validating the Context Diagram

- **Check 1:** Are there any data stores? (Should be **NO**)
- **Check 2:** Do all data flows have a meaningful name?
- **Check 3:** Is there any external entity communicating directly with another external entity? (Should be **NO**, must go through the system)
- **Check 4:** Does the central process have both inputs and outputs?

Summary

- DFDs show the logical flow of data through a system.
- The Context Diagram (Level 0) defines system boundaries.
- It uses a single process to represent the entire system.
- External entities interact with the system via data flows, with no data stores shown.

Next Lecture Preview

- Exploding the Context Diagram
- Creating Level 1 DFDs
- Introducing Data Stores
- Continuing the Library Management System Case Study

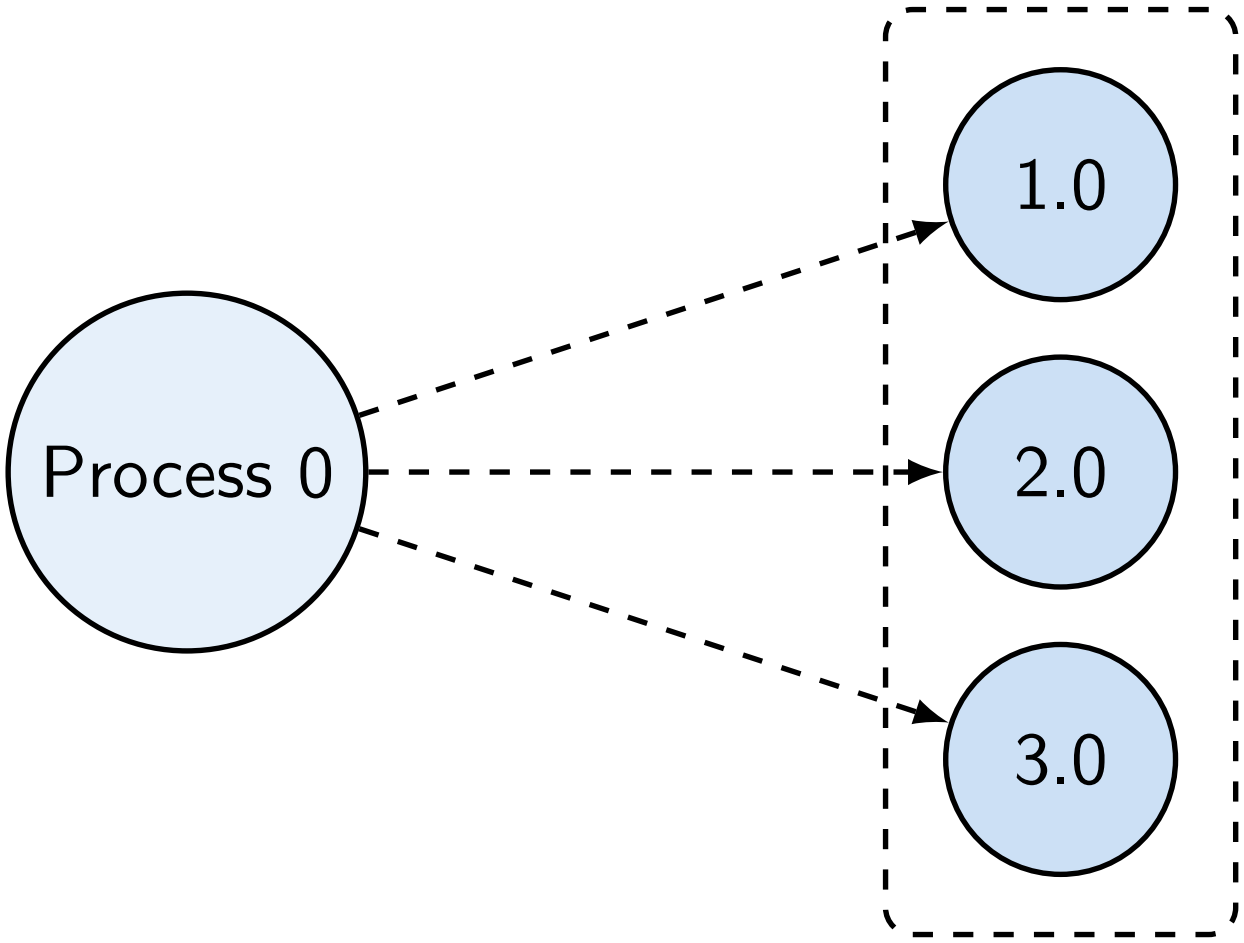
Questions?

Agenda

- Concept of Process Explosion / Decomposition
- Introducing Data Stores
- Rule of Balancing between Levels
- Identifying Internal Processes for Library Management
- Constructing the Level 1 DFD

Process Explosion (Decomposition)

- The act of breaking a high-level process into smaller, detailed sub-processes.
- Process 0 (Level 0) is exploded into Processes 1.0, 2.0, 3.0, etc., in Level 1.
- Provides a closer look at the logical operations of the system.
- Reveals where data is stored internally.

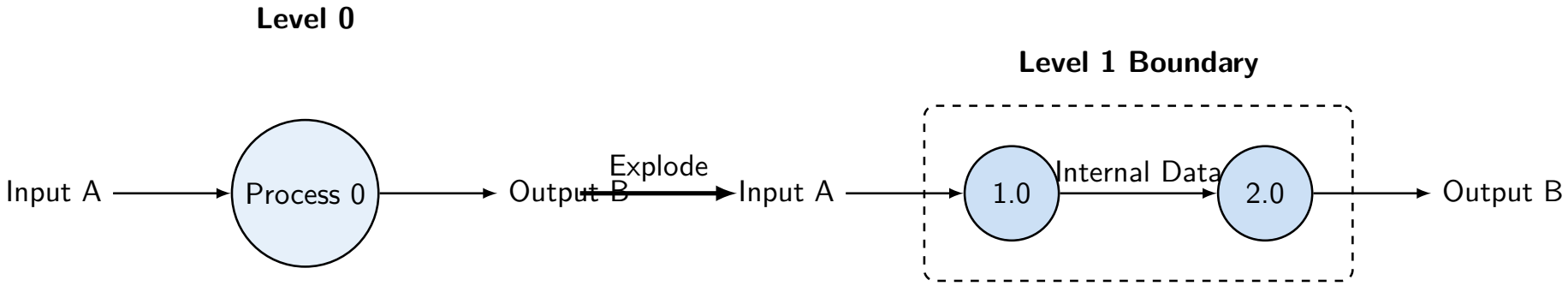


Introducing Data Stores

- Data stores appear for the first time in Level 1 DFDs.
- Represent data at rest (files, database tables).
- Data flow INTO a store = Writing/Updating data.
- Data flow OUT of a store = Reading/Retrieving data.

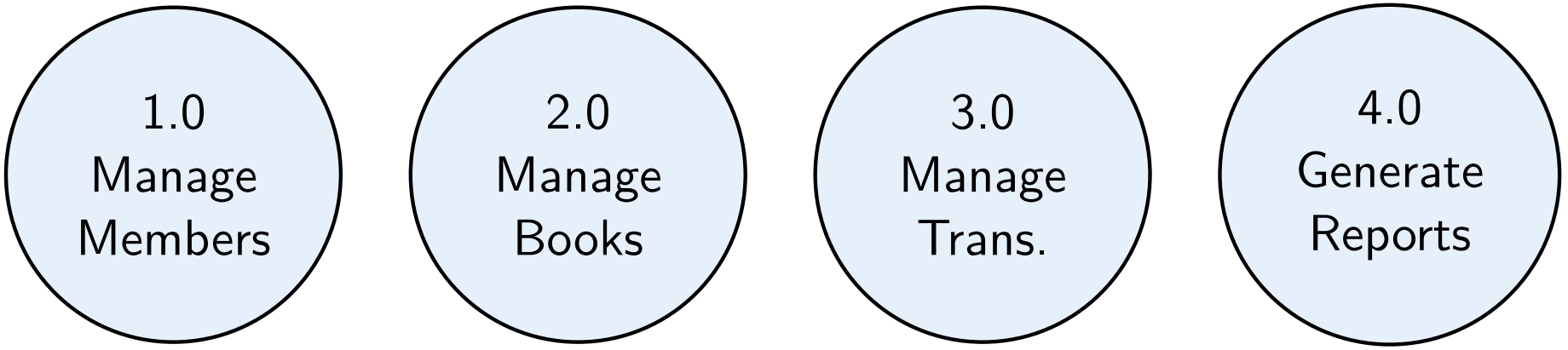
The Rule of Balancing

- Inputs and outputs of the parent process must perfectly match the inputs and outputs of the child DFD.
- Prevents data loss or spontaneous data generation during decomposition.



Library System: Core Processes

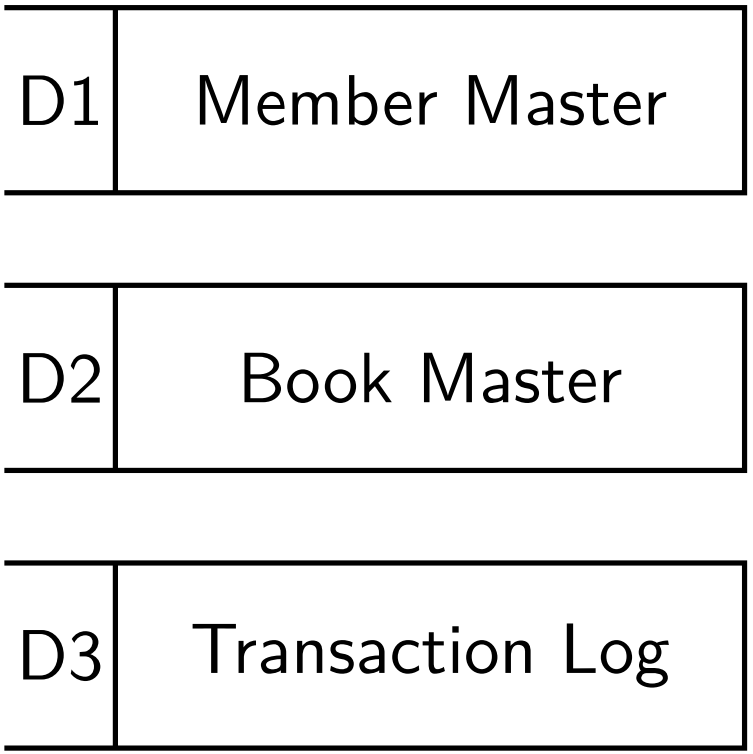
Decomposing the Library System into major functional areas:



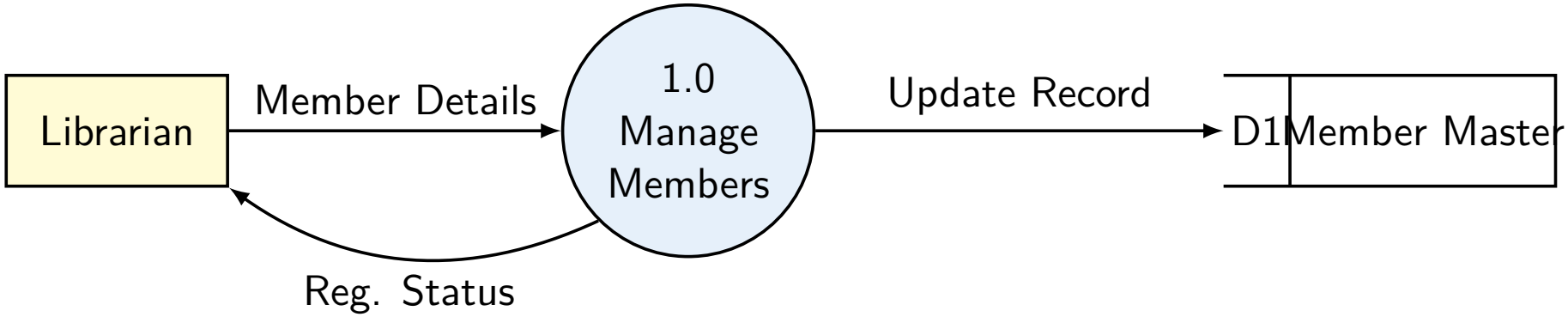
- **1.0**: Registration, updates, deletions.
- **2.0**: Cataloging, inventory updates.
- **3.0**: Issue, return, fines.
- **4.0**: Administrative reporting.

Library System: Data Stores

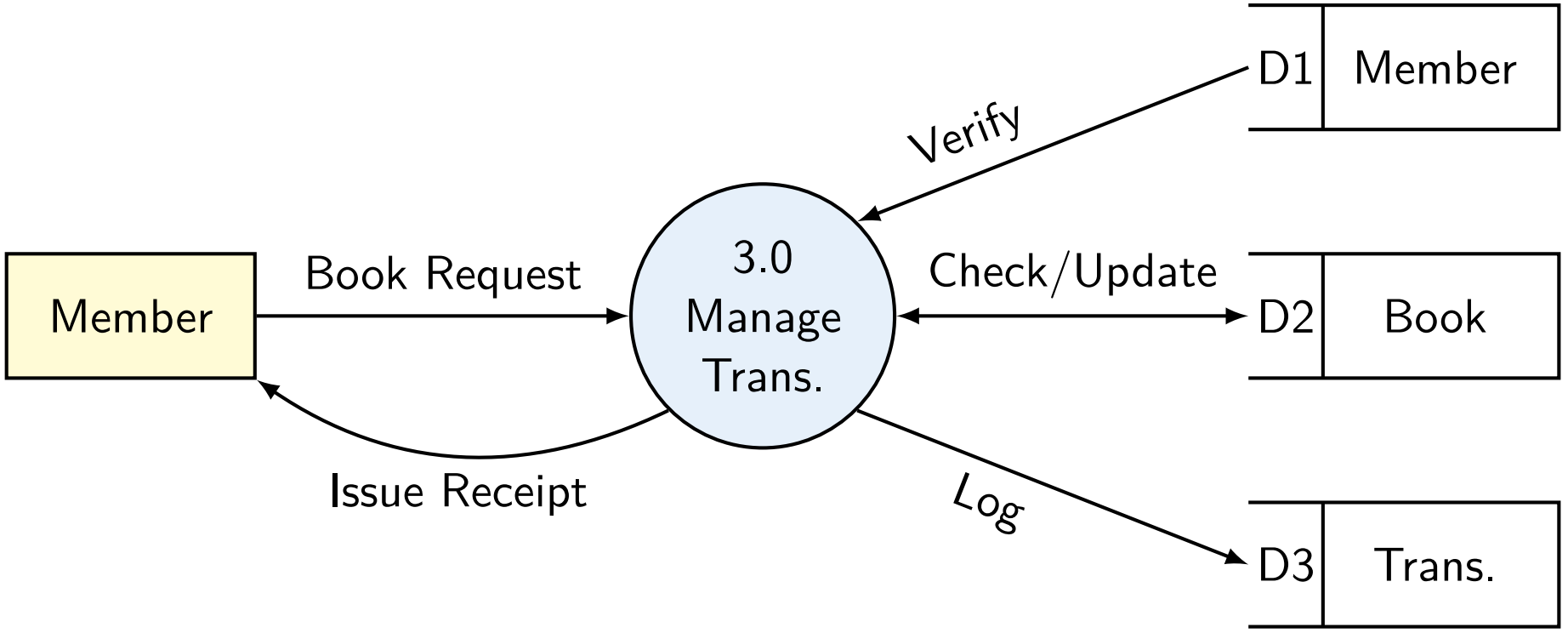
What data needs to be stored permanently?



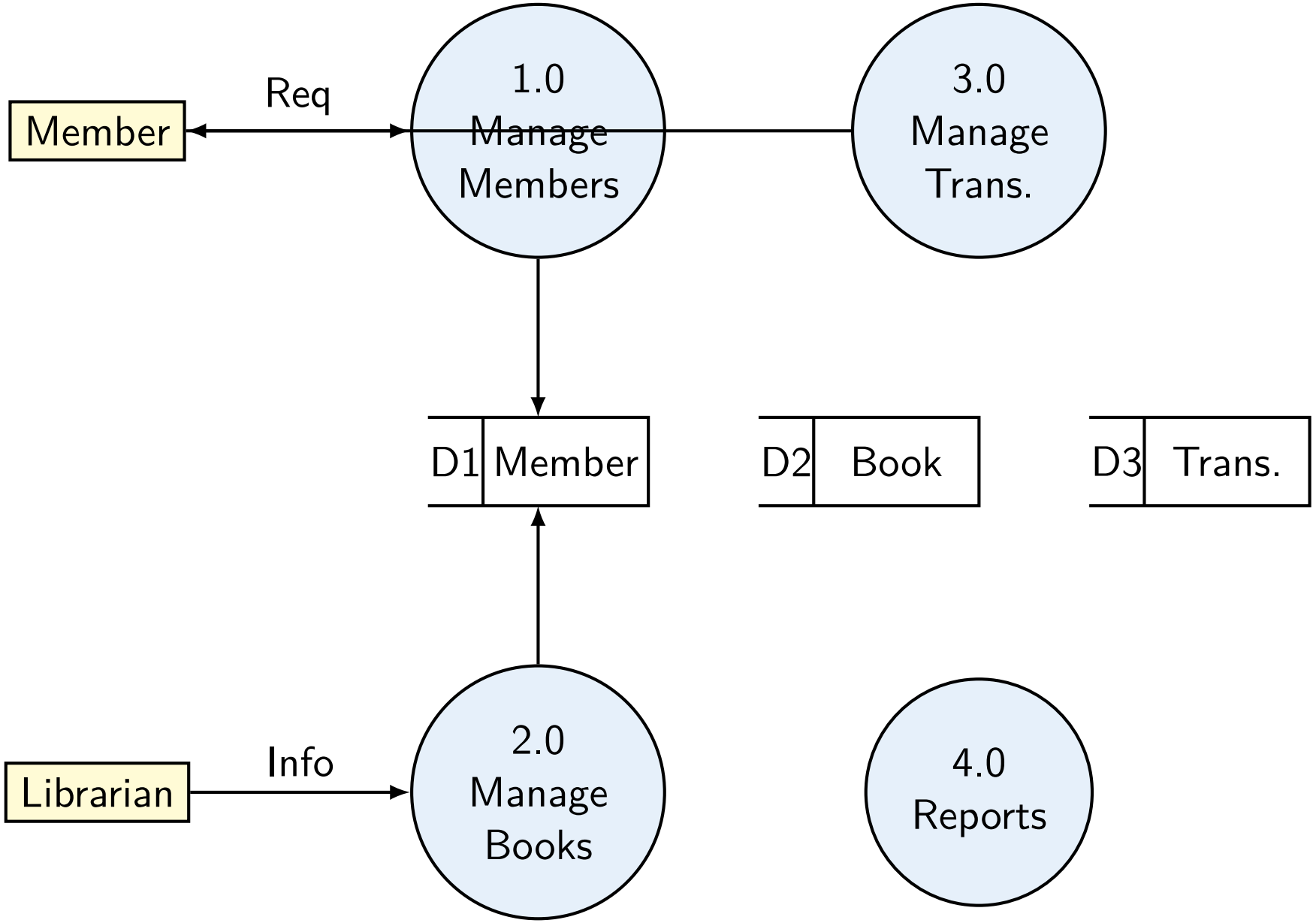
Drafting Process 1.0: Manage Members



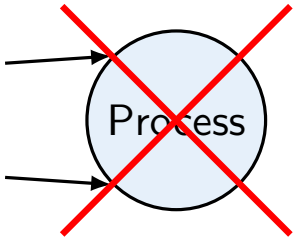
Drafting Process 3.0: Manage Transactions



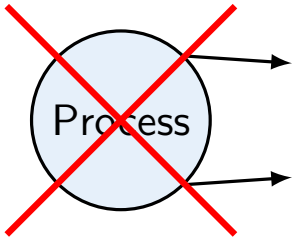
Constructing Level 1 DFD: Putting it Together



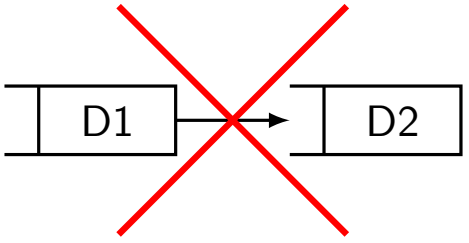
Checking for Common DFD Errors



Black Hole (No Output)



Miracle (No Input)



Store to Store

Summary

- Level 1 DFD explodes the system into major functional processes.
- Data stores are introduced to represent data at rest.
- The balancing rule ensures consistency with the Level 0 Context Diagram.
- All data must flow through a process; no direct entity-to-store communication.

Next Lecture Preview

- Transitioning from Structured Analysis to Object-Oriented Analysis
- Introduction to UML (Unified Modeling Language)
- Use Case Diagrams
- Class Diagrams for the Library Management System

Questions?

Recap & Agenda

Recap

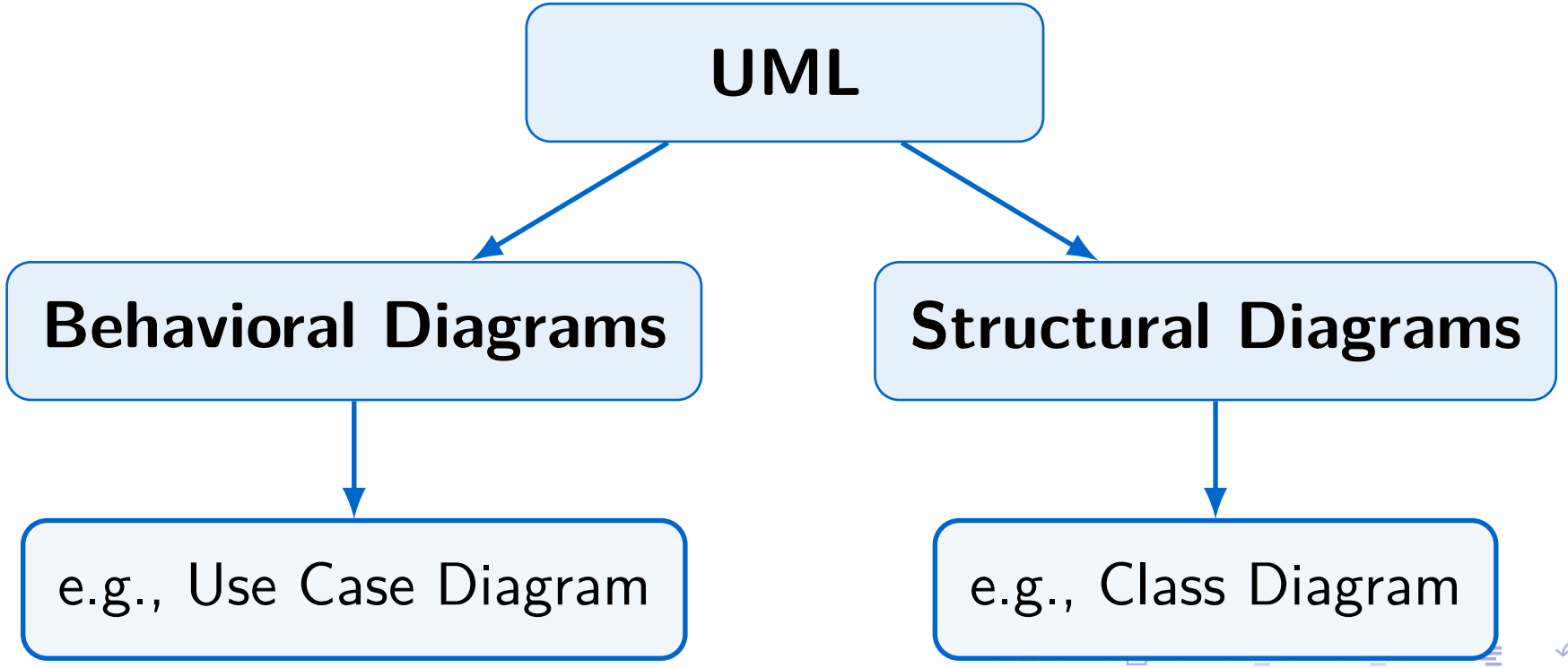
In the previous lecture, we built a Level 1 DFD, detailing the functional processes and data stores of a Library System.

Agenda:

- Introduction to UML
- Use Case Diagrams: Actors, Use Cases, and System Boundary
- Building a Use Case Diagram for the Library System
- Class Diagrams: Classes, Attributes, Methods
- Relationships: Association, Aggregation, Multiplicity
- Building a Class Diagram for the Library System

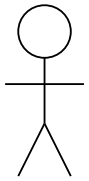
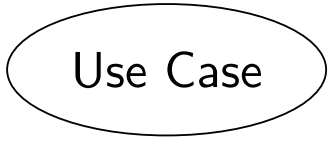
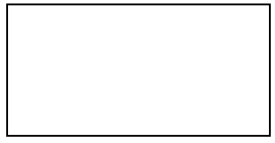
What is UML?

- **Unified Modeling Language (UML)** is the standard visual modeling language for software.
- It provides a set of graphical notation techniques to create visual models of object-oriented systems.
- **Behavioral Diagrams:** Show what the system DOES (e.g., Use Case Diagram).
- **Structural Diagrams:** Show what the system IS made of (e.g., Class Diagram).



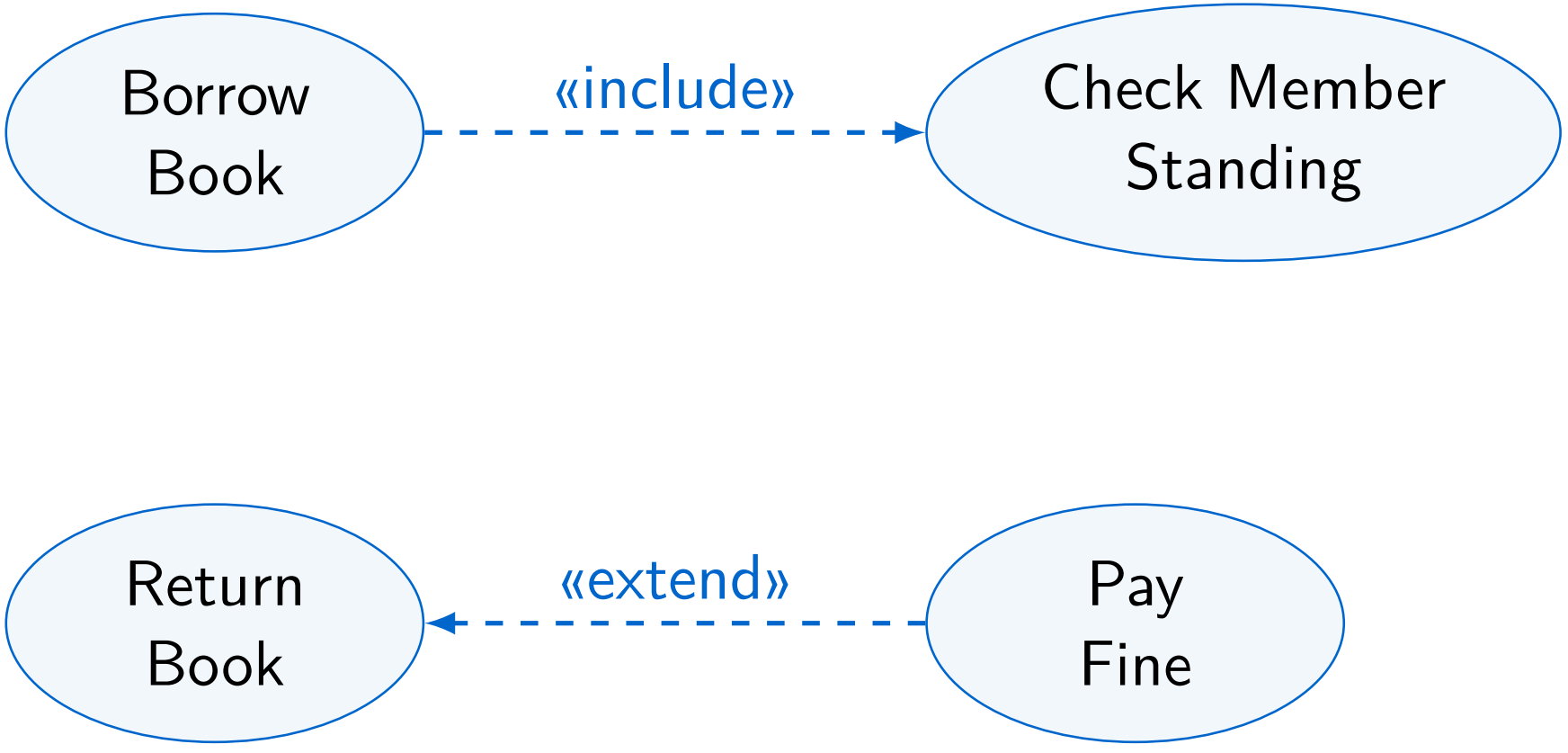
Use Case Diagram: Core Concepts

- **Actor:** Represents a user or external system interacting with the software.
- **Use Case:** Represents a distinct functionality or goal the system provides.
- **System Boundary:** Defines the scope of the system.
- **Association:** Connects an Actor to a Use Case they participate in.

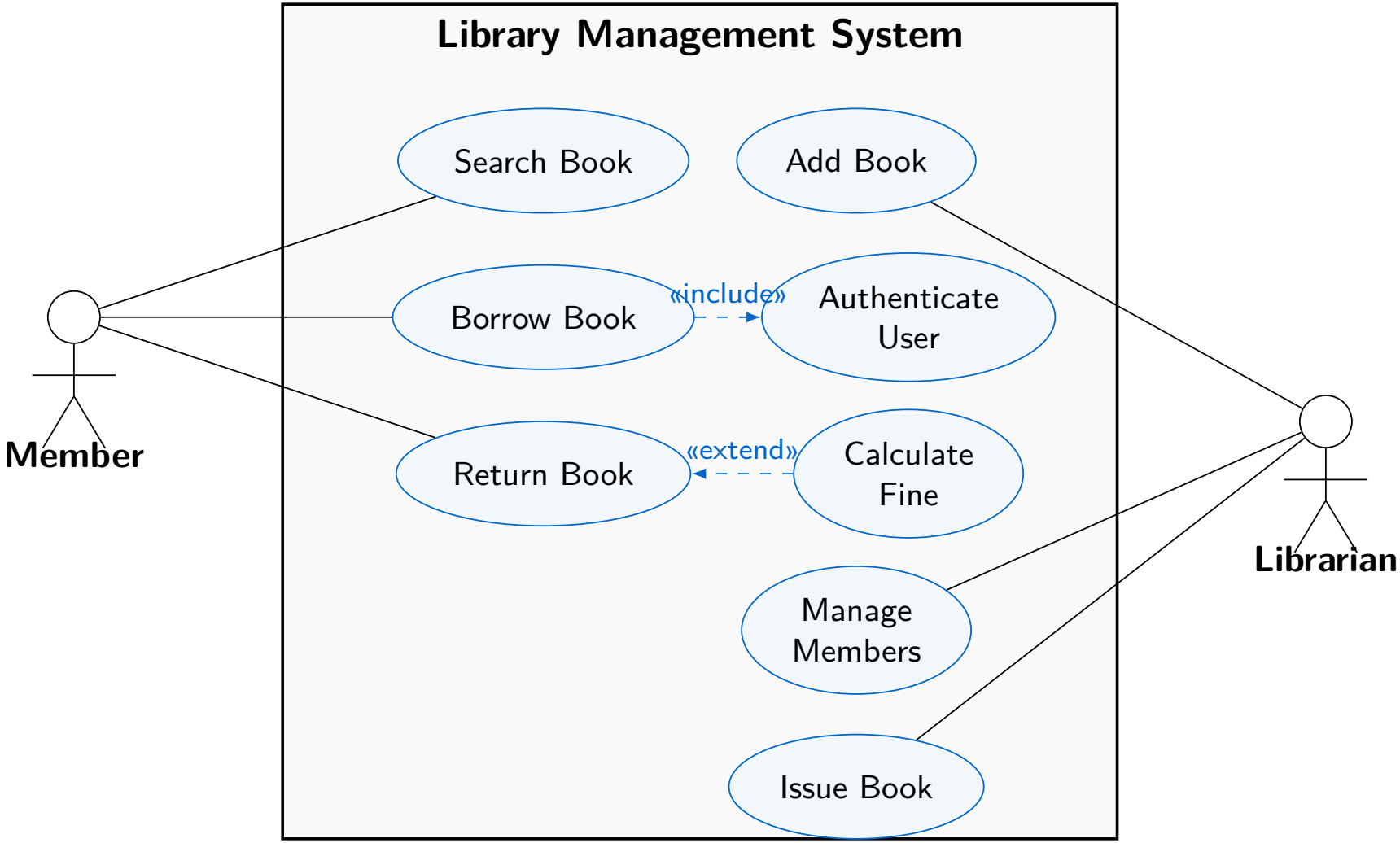
Notation	Description
	**Actor (Stick Figure):** Represents a user or external system.
	**Use Case (Oval):** Distinct functionality or goal.
	**System Boundary (Rectangle):** Defines scope of system.
	Association (Solid Line): Connects Actor to Use Case.

Advanced Use Case Relationships

- **«include»:** The base use case explicitly incorporates the behavior of another. (e.g., 'Borrow Book' «include» 'Check Member Standing').
- **«extend»:** The base use case implicitly incorporates optional behavior. (e.g., 'Return Book' «extend» 'Pay Fine').

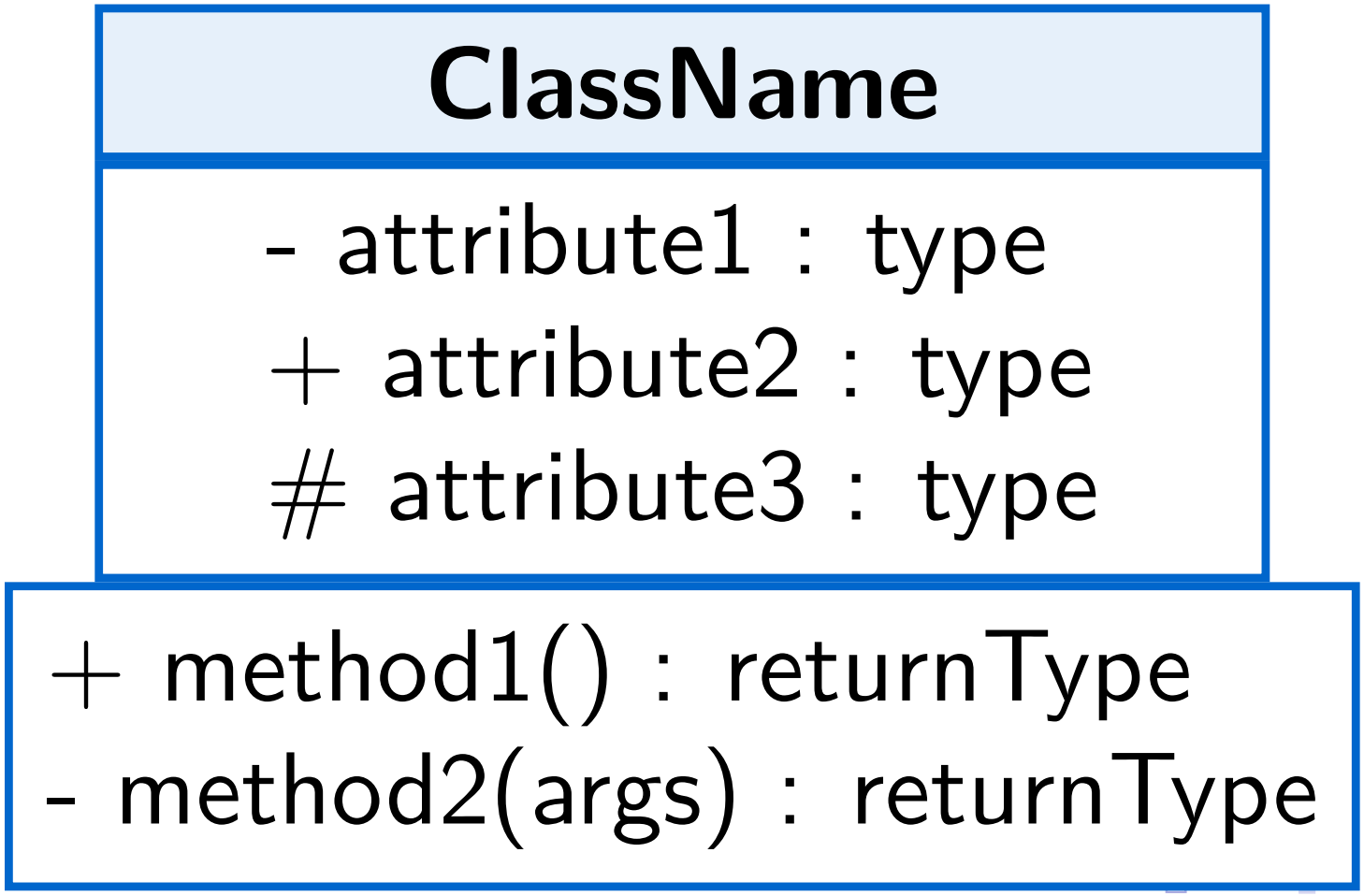


Use Case Diagram: Library Management System



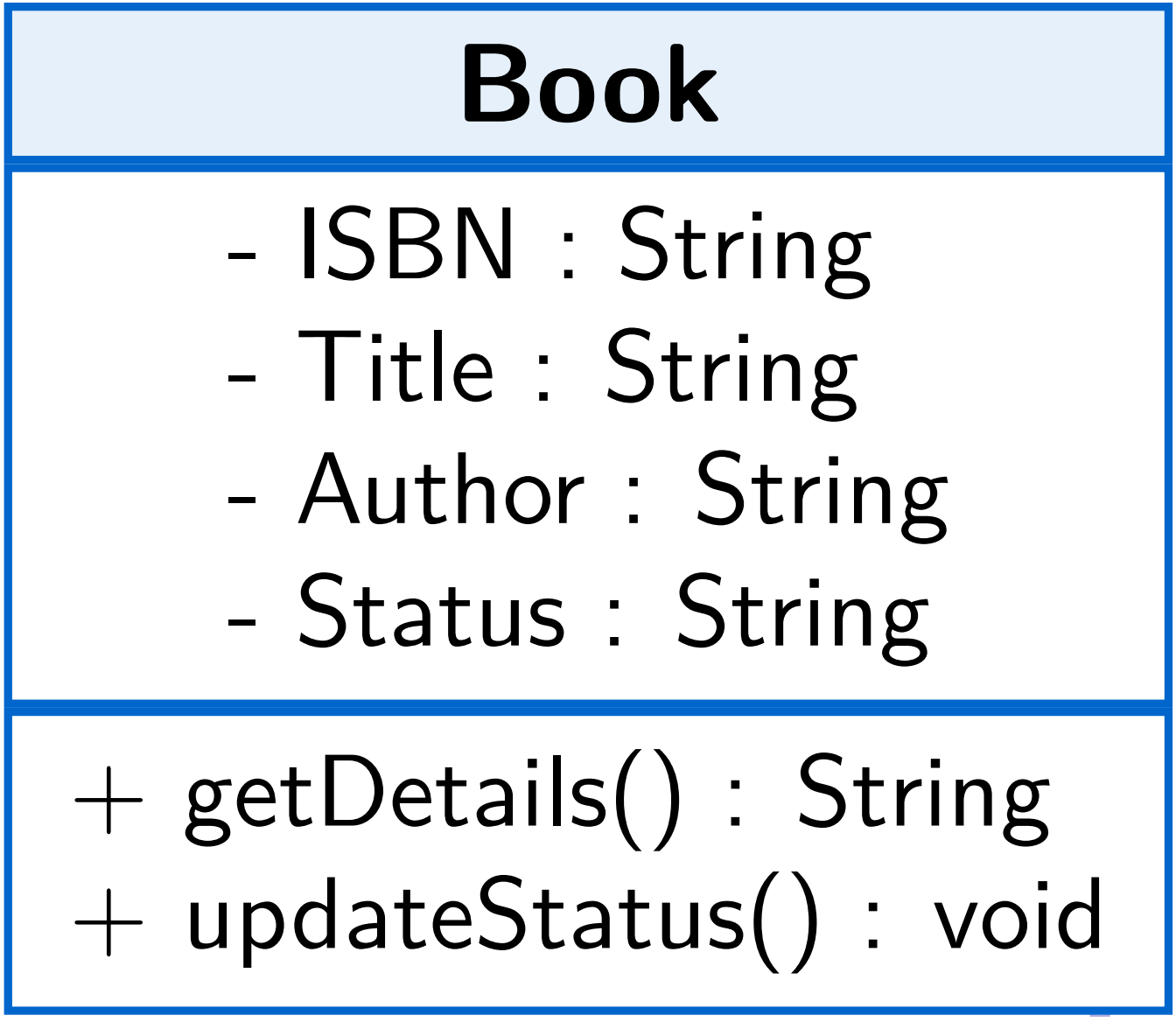
Class Diagram: Core Concepts

- The backbone of object-oriented modeling.
- **Class:** Represented as a box with 3 compartments: Name, Attributes, Methods.
- **Attributes:** Data variables (e.g., title, author).
- **Methods/Operations:** Functions (e.g., checkAvailability(), issueBook()).
- **Visibility:** + (Public), - (Private), # (Protected).



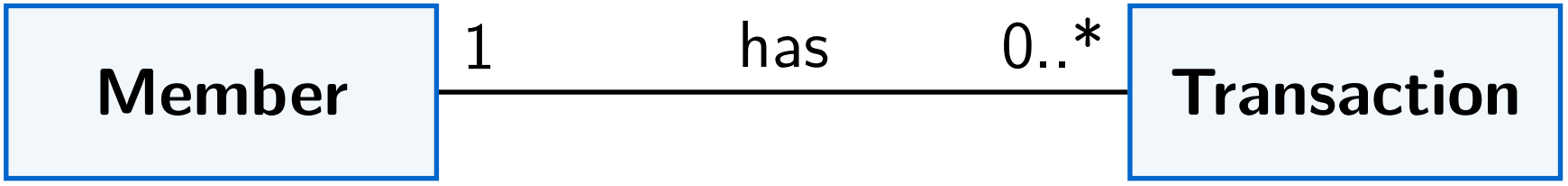
Identifying Classes for the Library System

- From our requirements, we extract nouns to identify potential classes.
- Candidate Classes: Book, Member, Librarian, Transaction, Account.
- Let's define **Book**:



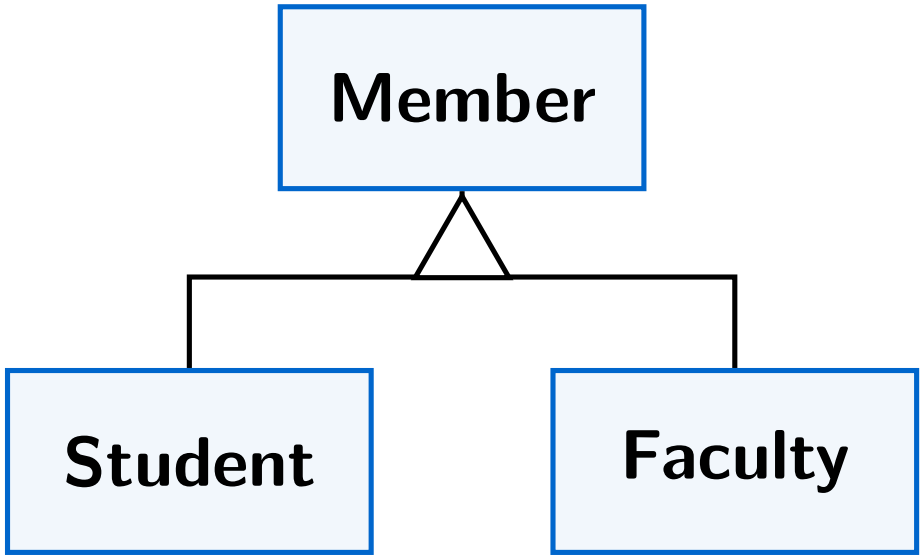
Class Relationships (Multiplicity & Associations)

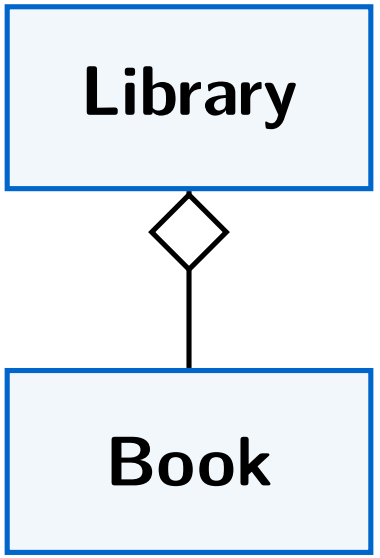
- **Association:** A standard link between classes (e.g., Member 'borrows' Book).
- **Multiplicity:** Indicates how many objects are involved.
  - **1** : Exactly one
  - **0..*** : Zero or more
  - **1..*** : One or more
- **Example:** 1 Member can have 0..* Transactions.



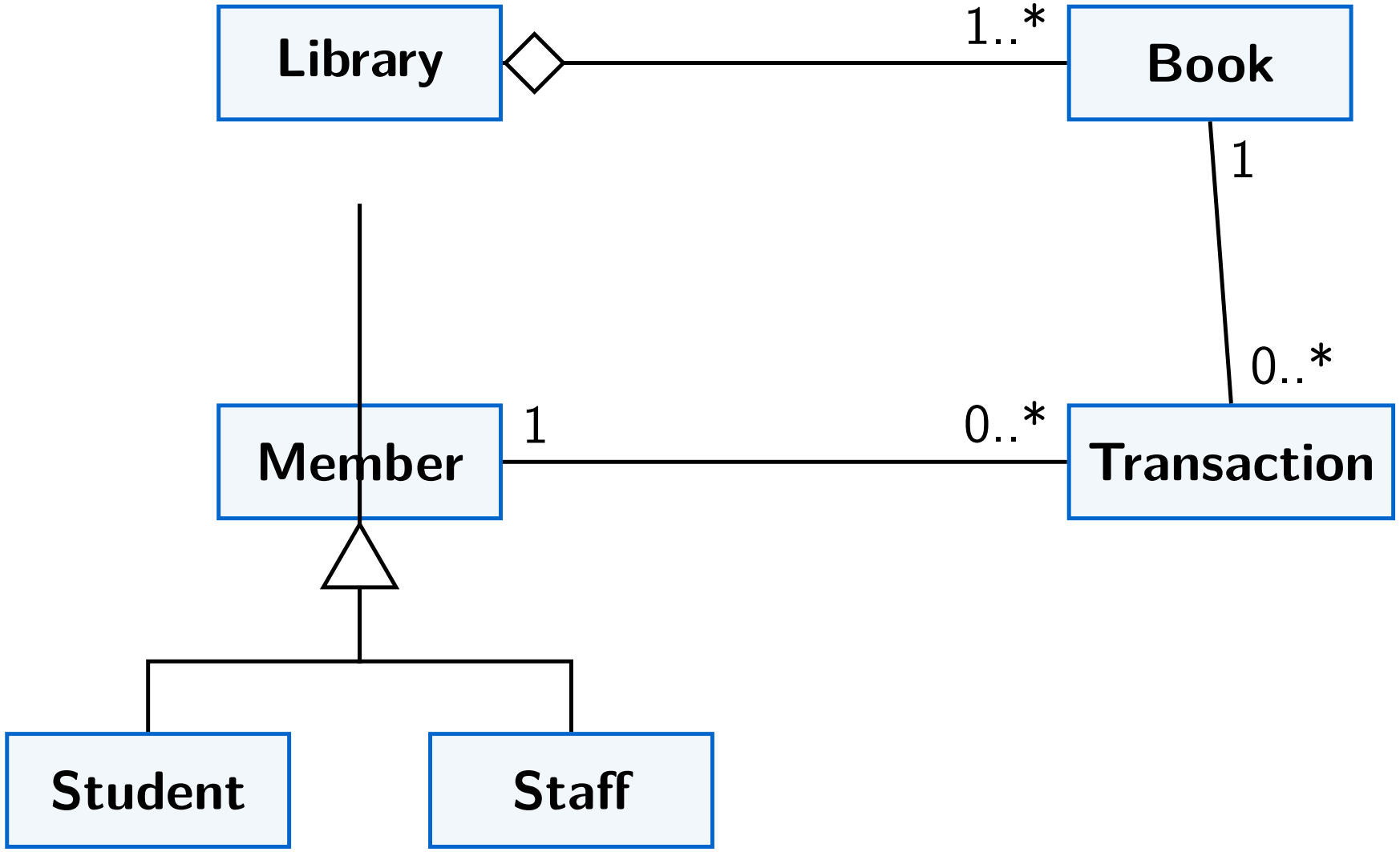
Advanced Relationships (Inheritance & Aggregation)

- **Inheritance (Generalization):** 'IS-A' relationship. Solid line with hollow triangle.
  - Example: 'Student' and 'Faculty' inherit from 'Member'.
- **Aggregation/Composition:** 'HAS-A' relationship. Line with a diamond.
  - Example: 'Library' aggregates 'Book'.



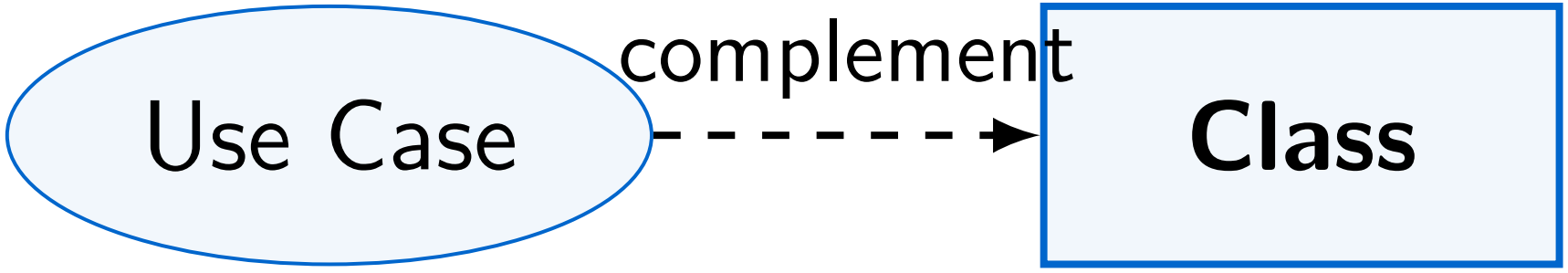


Class Diagram: Library Management System



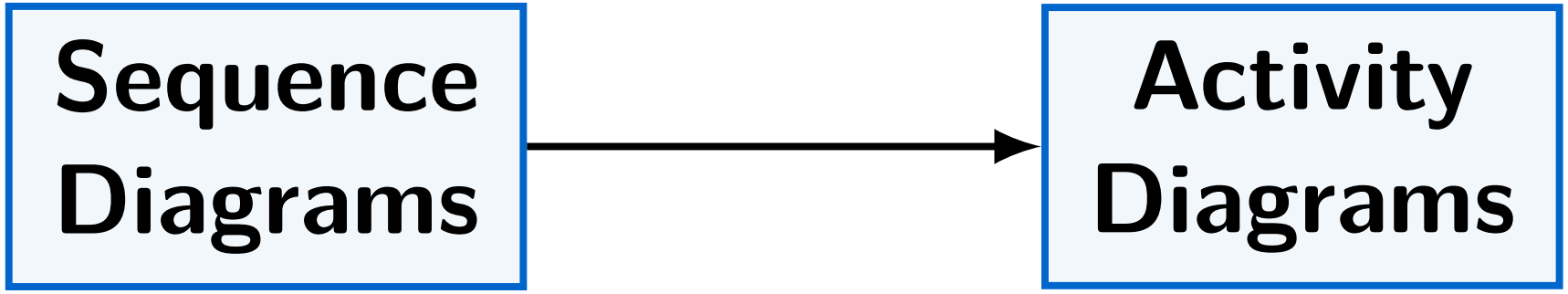
Summary

- **Use Case diagrams** model system behavior from the user's perspective (Actors & Use Cases).
- **Class diagrams** model the static structure (Classes, Attributes, Methods, Relationships).
- **UML** provides a standardized way to visualize object-oriented system design.
- Both diagrams are essential for translating requirements into a software architecture.



Next Lecture Preview

- Dynamic Modeling with UML
- **Sequence Diagrams:** Modeling time-ordered interactions
- **Activity Diagrams:** Modeling workflows
- Completing the UML perspective for the Library System



Questions?

Introduction to Dynamic Modeling

- Static models (Class Diagrams) describe the structural elements of a system.
- Dynamic models capture the behavior of the system over time.
- They show how objects collaborate to realize use cases.
- Key dynamic diagrams in UML: Sequence, Communication, Statechart, and Activity diagrams.

UML Sequence Diagrams

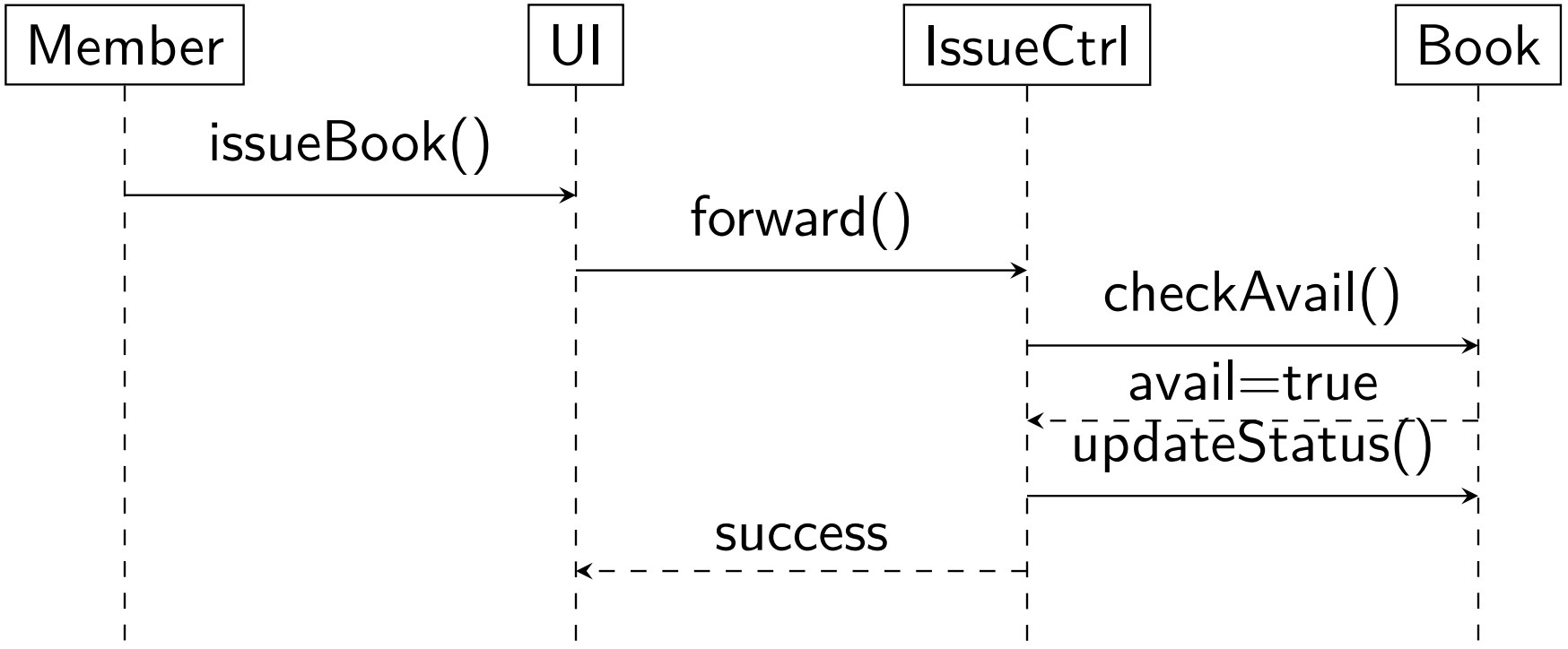
- An interaction diagram that emphasizes the time-ordering of messages.
- Two dimensions: Vertical axis represents time (top to bottom), Horizontal axis represents different objects.
- **Lifeline**: Represents the existence of an object over a period of time.
- **Activation**: Shows the time period during which an object is performing an operation.

Sequence Diagram Notations

- **Synchronous Message**: Solid line with a solid arrowhead (Caller waits for response).
- **Asynchronous Message**: Solid line with an open arrowhead (Caller does not wait).
- **Return Message**: Dashed line with an open arrowhead.
- **Self Message**: A message an object sends to itself.
- **Creation/Destruction**: Creating an object (message to «create») and destroying an object (denoted by 'X' at end of lifeline).

Sequence Diagram Example: Issue Book

- Scenario: A library member requests to issue a book.
- Actors/Objects: Member (Actor), UI (Boundary), IssueController (Control), Book (Entity), MemberAccount (Entity).
- Messages: `verifyMember()`, `checkAvailability()`, `updateStatus()`, `generateReceipt()`.



Sequence Diagram: Advanced Fragments

- Combined Fragments allow modeling of complex control flow within a sequence diagram.
- **alt (Alternative)**: Models IF-THEN-ELSE logic (e.g., Book available vs. Book not available).
- **opt (Option)**: Models IF logic without an ELSE block.
- **loop**: Models a repetitive sequence (e.g., iterating through a list of books).
- **par (Parallel)**: Models concurrent operations.

UML Activity Diagrams

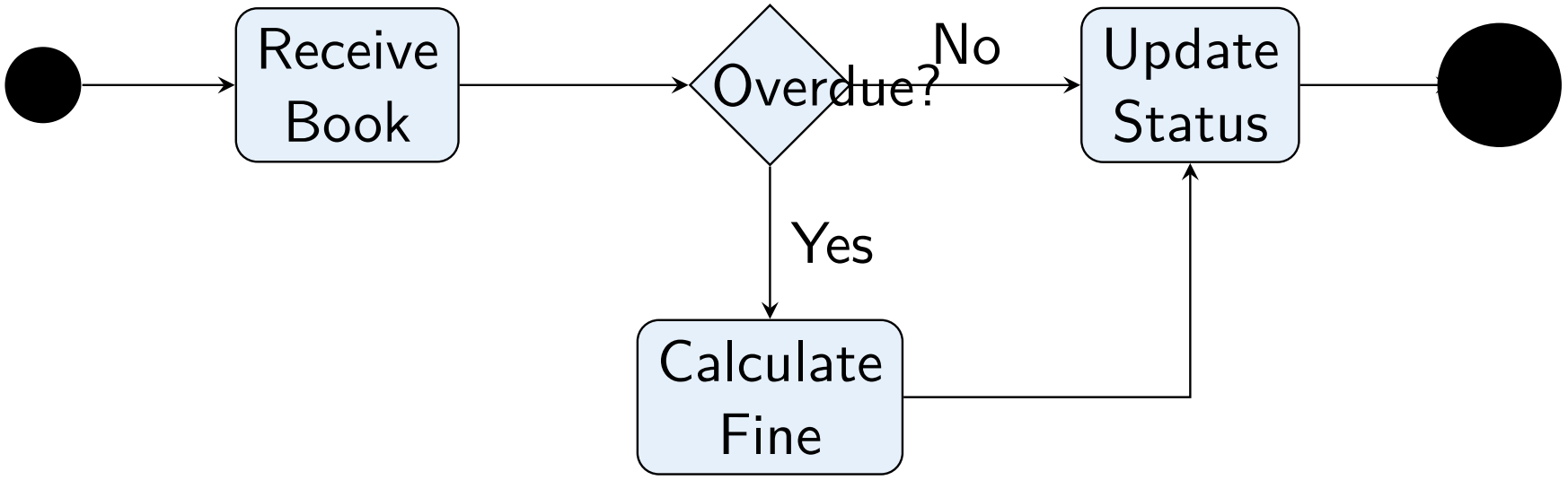
- Activity diagrams represent the workflow or stepwise activities of a process.
- Focus on the flow of control from one activity to another.
- Excellent for modeling business processes and complex algorithms.
- Similar to traditional flowcharts but support parallel execution.

Activity Diagram Notations

- **Initial Node (Start)**: Solid black circle.
- **Activity/Action Node**: Rounded rectangle.
- **Decision Node**: Diamond shape (conditional branching).
- **Merge Node**: Diamond shape (merging conditional paths).
- **Fork/Join Nodes**: Heavy solid line (synchronization bars for parallel threads).
- **Final Node (End)**: Solid black circle inside a hollow circle.

Activity Diagram Example: Book Return Workflow

- Scenario: Processing a returned library book.
- Start → Receive Book → Check Due Date.
- Decision: Is it overdue?
- If Yes → Calculate Fine → Receive Payment → Merge.
- If No → Merge.
- Update Book Status to Available → End.



Activity Diagram: Swimlanes

- **Swimlanes (Partitions)** divide an activity diagram into columns or rows.
- Each partition represents the responsibility of a particular class, organizational unit, or role.
- Clarifies WHO or WHAT is performing the action.
- Transitions can cross swimlane boundaries.

Comparing Sequence and Activity Diagrams

Feature	Sequence Diagrams	Activity Diagrams
Focus	Time-ordered sequence of messages	Overall flow of control / workflow
Primary Elements	Lifelines, Activations, Messages	Actions, Control flows, Decisions, Forks
Best Used For	Modeling interactions within a single use case	Modeling business processes and algorithms

Summary

- Dynamic models capture system behavior over time.
- Sequence diagrams visualize message passing and object lifetimes chronologically.
- Activity diagrams model workflows, decision branching, and parallel processing.
- Both are essential for translating functional requirements into system design.

Next Lecture Preview

- Transitioning to Unit 4: Software Project Management.
- The Management Spectrum: The 4 P's (People, Product, Process, Project).
- Understanding the critical elements required to manage a software project successfully.

Questions?

Recap & Agenda

Recap:

- In the previous lecture, we concluded Unit 3 by exploring dynamic modeling using UML Sequence and Activity Diagrams.

Agenda:

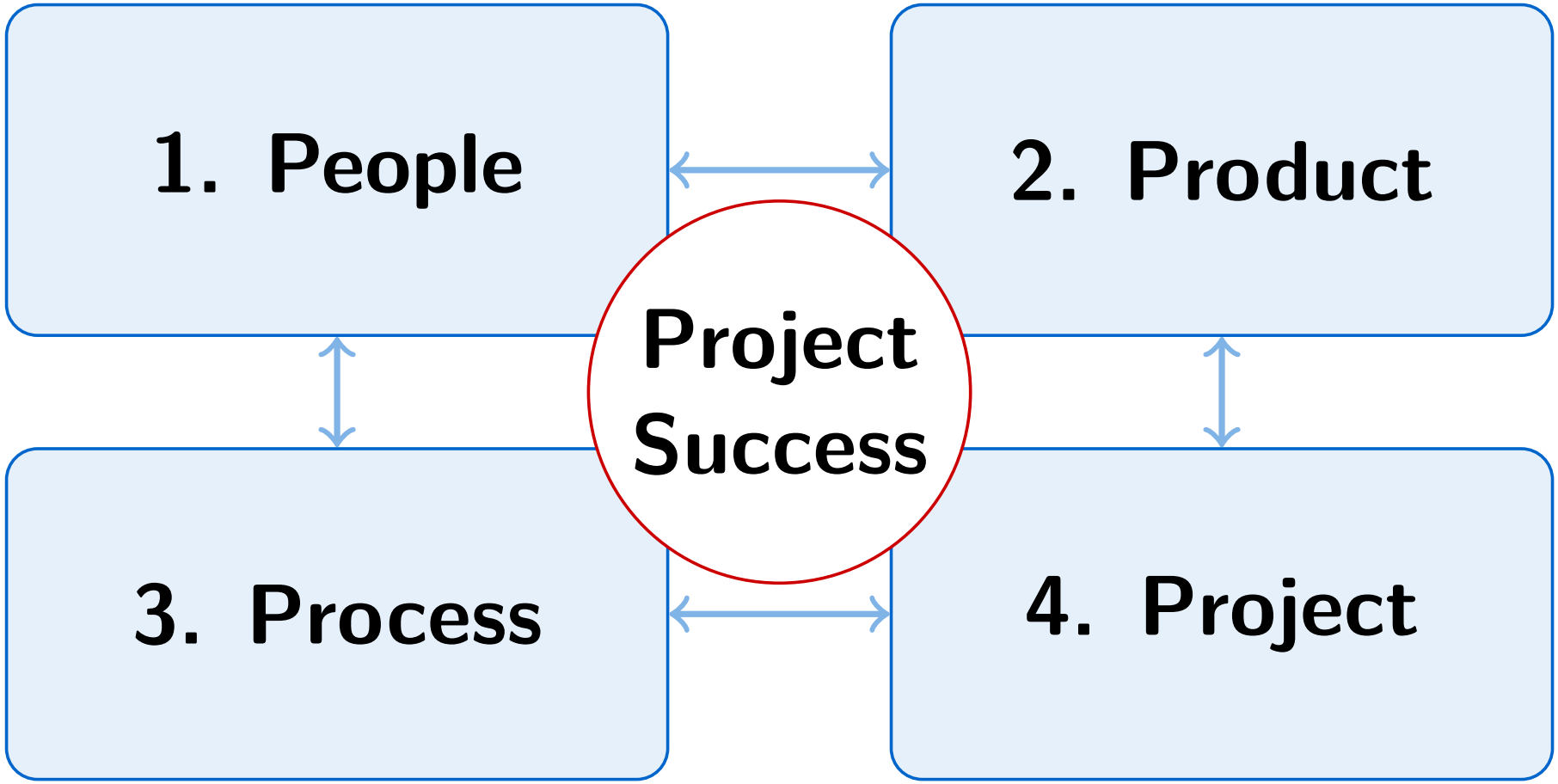
- What is Software Project Management?
- Overview of the Management Spectrum
- The First P: People
- The Second P: Product
- The Third P: Process
- The Fourth P: Project
- Integration of the 4 P's

What is Software Project Management?

- Software project management is the umbrella activity within software engineering.
- It begins before any technical activity is initiated and continues throughout the project lifecycle.
- **Goal:** Deliver a software product that meets customer requirements, on time, and within budget.
- **Challenges:** Intangibility of software, changing requirements, and rapidly evolving technology.

The Management Spectrum: The 4 P's

- Effective software project management focuses on the four P's:



The First P: People

- Software engineering is a highly human-intensive activity.
- **Key aspects:** Recruiting, selection, performance management, training, compensation, career development.
- **Players in the process:**
  - *Senior Managers:* Define business issues.
  - *Project (Technical) Managers:* Plan, motivate, organize, and control practitioners.
  - *Practitioners:* Deliver the technical skills.
  - *Customers/End Users:* Specify requirements and use the system.

People: Team Organization

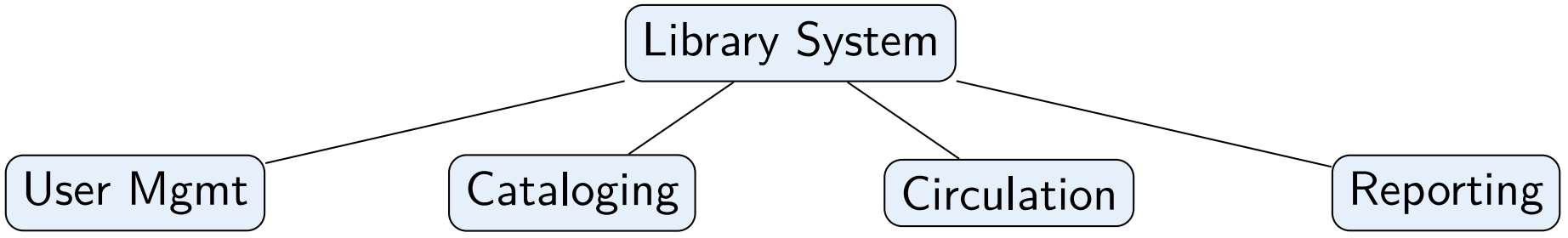
- How a team is organized affects communication and productivity.
- **Democratic Decentralized (DD):** No permanent leader, decisions by consensus. Good for complex, R&D problems.
- **Controlled Decentralized (CD):** Defined leader, horizontal communication. Good for large, straightforward projects.
- **Controlled Centralized (CC):** Top-level problem solving, rigid vertical communication. Best for strict deadlines.

The Second P: Product

- Before a project can be planned, the product objectives and scope must be established.
- **Scope Definition:** What is to be built? What are the boundaries? Must be unambiguous and understandable.
- **Context:** How does the software fit into a larger system or business environment?
- **Information Objectives:** What data goes in (input) and what data comes out (output)?

Product: Problem Decomposition

- Also known as partitioning or 'divide and conquer'.
- A complex product is broken down into manageable functions or components.
- **Example (Library System):** Decompose into User Management, Cataloging, Circulation (Issue/Return), and Reporting.
- Decomposition makes estimation and task assignment possible.



The Third P: Process

- The process provides the framework from which a comprehensive plan for software development can be established.
- **Choosing the right paradigm:** Waterfall, Spiral, Agile, Prototyping.
- **Framework Activities:** Communication, Planning, Modeling, Construction, Deployment.
- **Umbrella Activities:** Software Quality Assurance (SQA), Configuration Management (SCM), Risk Management.

The Fourth P: Project

- The project is the execution of the process to build the product using the people.
- Focus on planning, monitoring, and controlling.
- **Key Project Activities:**
  - Cost and Schedule Estimation
  - Risk Analysis
  - Progress Tracking (Metrics and Milestones)

Why Do Software Projects Fail?

- Unrealistic or unarticulated project goals (Product).
- Inaccurate estimates of needed resources (Project).
- Badly defined system requirements (Product).
- Poor reporting of the project's status (Project/Process).
- Unmanaged risks (Project).
- Poor communication among customers, developers, and users (People).

Integrating the 4 P's

- The 4 P's are interdependent.
- A highly skilled Team (People) cannot succeed with ambiguous Requirements (Product) or a chaotic Workflow (Process).
- A rigid Process can stifle talented People.
- A large Product scope requires a rigorous Process and a closely monitored Project schedule.

Summary

- Software project management is an essential umbrella activity.
- The Management Spectrum consists of the 4 P's:
  - **People:** The stakeholders and team dynamics.
  - **Product:** The scope and decomposed requirements.
  - **Process:** The SDLC model and framework activities.
  - **Project:** Planning, estimation, and control.

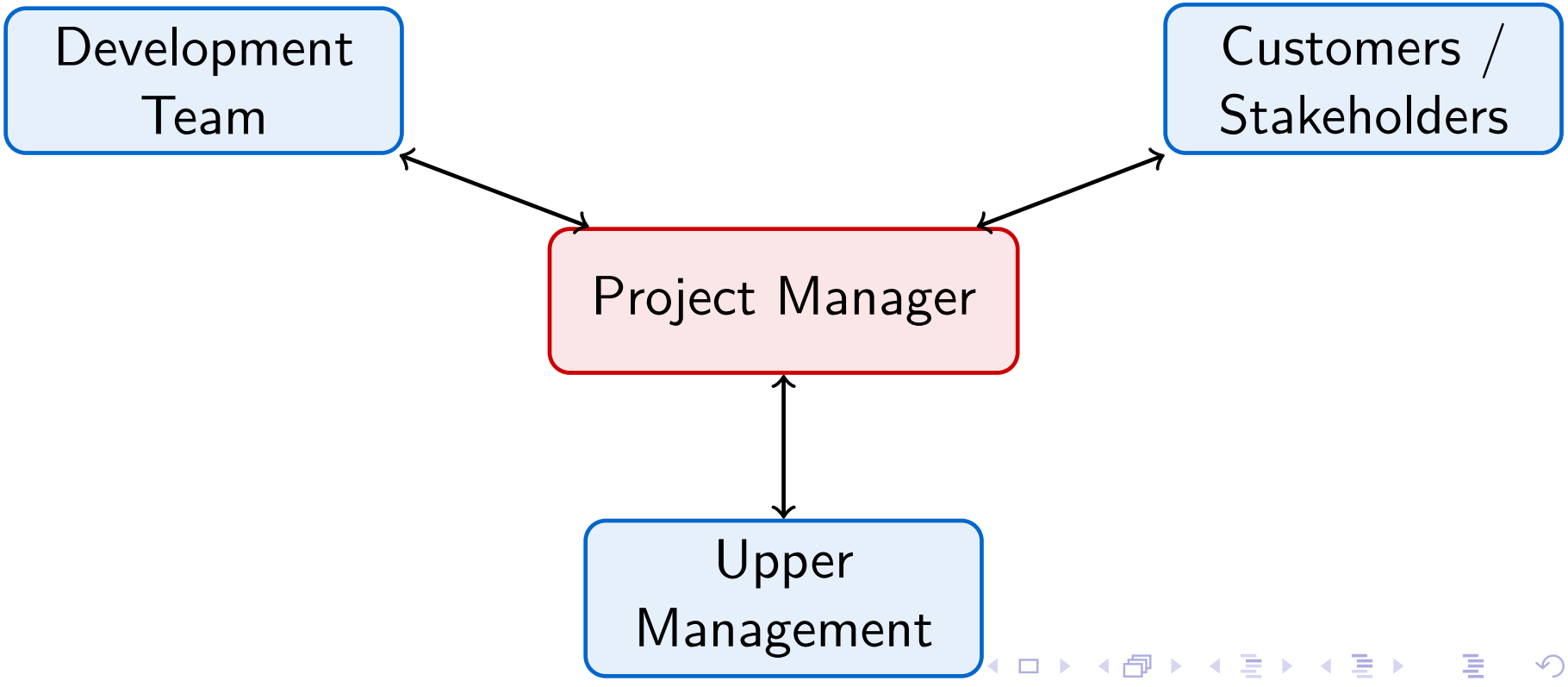
Next Lecture Preview

- Responsibility of the Software Project Manager.
- Roles and duties in planning, organizing, and controlling.
- Key skills required for effective project management.

Questions?

Who is a Software Project Manager?

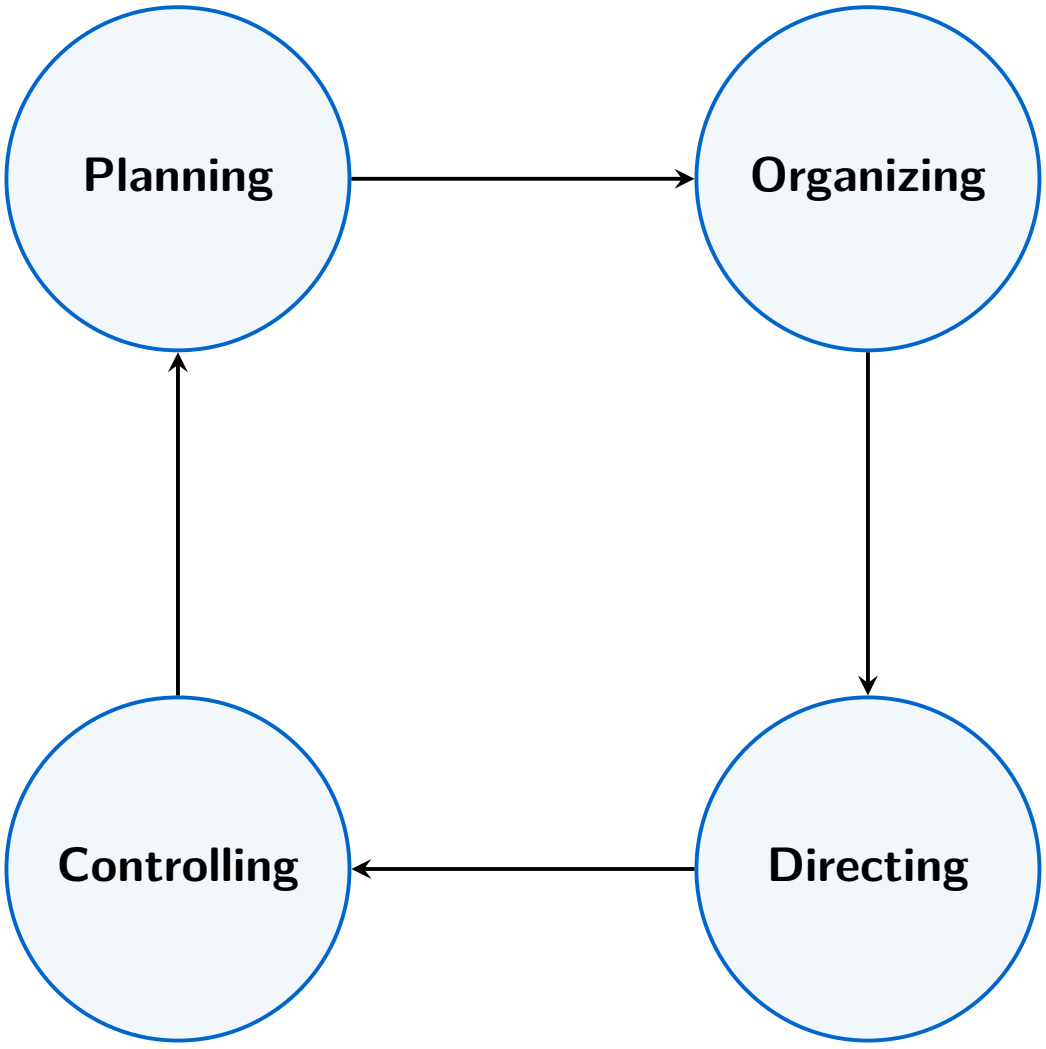
- The individual responsible for managing a project from inception to execution.
- Acts as a bridge between the development team and the stakeholders/customers.
- Focuses on managerial tasks rather than purely technical tasks.
- Primary goal: Ensure the project is delivered on time, within budget, and meets quality standards.



Core Phases of Managerial Responsibility

The responsibilities can be categorized into four primary phases:

1. **Project Planning:** Defining scope, estimating effort, creating schedules.
2. **Organizing (Staffing):** Building the team, assigning roles.
3. **Directing (Executing):** Leading the team, facilitating communication.
4. **Controlling:** Tracking progress, managing risks, ensuring quality.

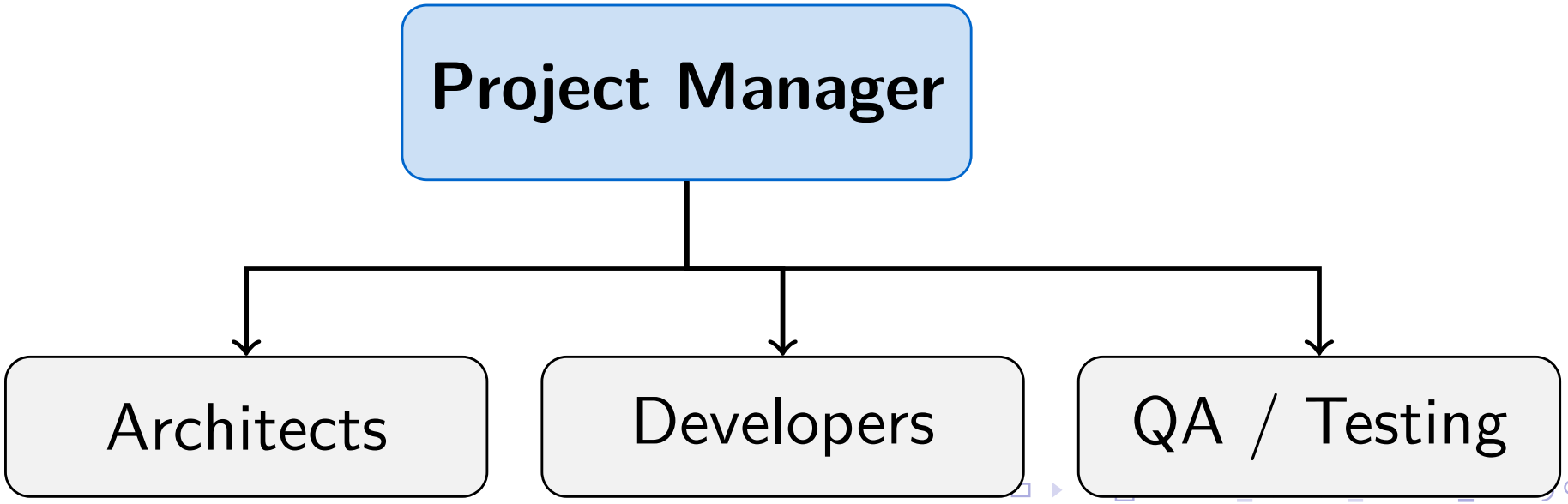


1. Project Planning & Estimation

- **Scope Definition:** Ensuring clear boundaries for the project.
- **Effort and Cost Estimation:** Using models like COCOMO to estimate Person-Months and budget.
- **Scheduling:** Defining tasks, dependencies, and milestones using tools like Gantt charts and PERT/CPM.
- **Resource Allocation:** Planning hardware, software, and human resource needs.

2. Organizing and Staffing

- **Team Selection:** Identifying the right mix of skills (Architects, Developers, QA, UI/UX).
- **Role Assignment:** Defining clear responsibilities for each team member.
- **Team Structure:** Deciding on Democratic, Chief Programmer, or mixed team structures.
- **Environment Setup:** Ensuring the team has the necessary tools, software licenses, and infrastructure.



3. Directing and Leading

- **Motivation:** Keeping team morale high, recognizing achievements.
- **Conflict Resolution:** Managing interpersonal and technical disagreements.
- **Communication Facilitation:** Conducting meetings (e.g., Daily Stand-ups), acting as the primary point of contact for stakeholders.
- **Removing Impediments:** Unblocking team members so they can focus on technical work.

4. Monitoring and Controlling

- **Tracking Progress:** Comparing actual progress against the planned schedule (Earned Value Analysis).
- **Quality Assurance:** Ensuring SQA processes and testing phases are executed properly.
- **Configuration Management:** Overseeing version control and managing changes to requirements.
- **Reporting:** Providing regular status reports to senior management and customers.

5. Risk Management

Proactive Problem Solving

Proactive identification of potential problems before they occur.

- **Risk Identification:** What can go wrong? (e.g., key developer leaves, technology fails).
- **Risk Analysis:** Assessing probability and impact.
- **Risk Mitigation:** Developing contingency plans (Plan B).
- **Risk Monitoring:** Continuously watching for risk triggers during the project.

Essential Skills of a Project Manager

- **Leadership & Motivation:** Ability to inspire the team.
- **Communication Skills:** Must be able to speak 'technical' to developers and 'business' to stakeholders.
- **Problem Solving:** Quick decision-making under pressure.
- **Negotiation Skills:** Managing scope creep and budget constraints with clients.
- **Technical Awareness:** Understanding the SDLC and underlying technologies, even if not writing code.

The Consequences of Poor Management

- **Without active monitoring:** Schedule slips become exponential.
- **Without scope control:** Feature creep drains budget and resources.
- **Without risk management:** Unforeseen technical issues halt progress.
- **Without leadership:** High team turnover and low morale.

Summary

- The Software Project Manager bridges technical execution and business goals.
- Core responsibilities span Planning, Organizing, Directing, and Controlling.
- Effective estimation, scheduling, and risk management are critical.
- Success requires a blend of soft skills (leadership/communication) and technical awareness.

Key Takeaways

- Project managers focus on people, planning, and control, not just coding.
- Risk management is a proactive, continuous responsibility.
- Communication and negotiation are as critical as technical awareness.

Next Lecture Preview

Topic: Software Metrics and Estimation (LOC and FP)

- Project Metrics and Estimation Techniques.
- Software Measurement: Size-oriented metrics (LOC) and Function-oriented metrics (FP).
- Introduction to empirical estimation models.

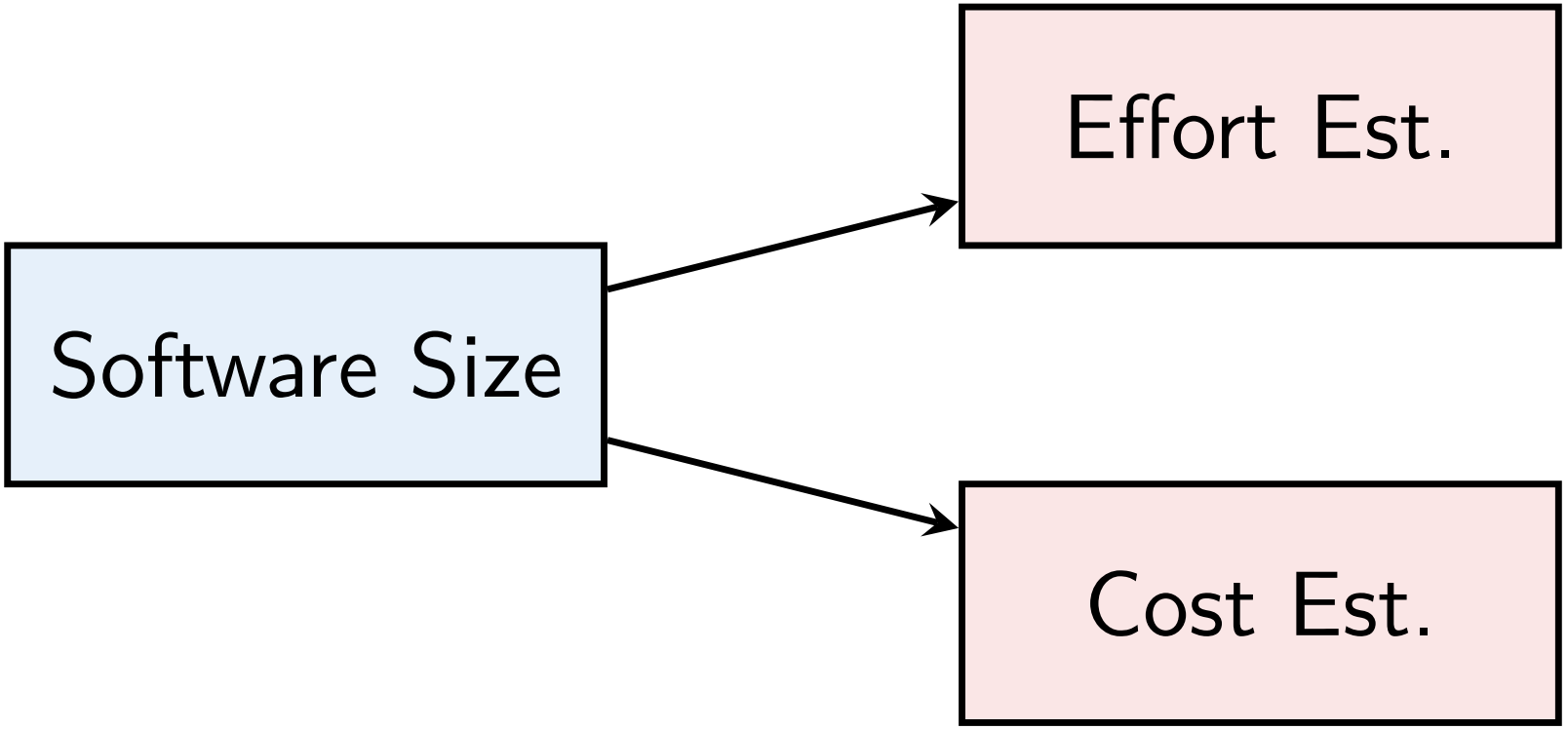
Questions?

Agenda

- Need for Software Size Estimation
- Definition of Lines of Code (LoC)
- Counting Rules and Variants (KLOC, NCLOC)
- Example: Estimating LoC
- Advantages of LoC
- Limitations and Language Dependency
- Summary and Next Steps

The Need for Size Estimation

- Project cost and schedule are heavily dependent on the size of the software.
- Size affects effort (person-months) and cost (budget).
- Direct metrics: Measure size directly (e.g., Lines of Code).
- Indirect metrics: Measure size based on functionality (e.g., Function Points).



What is Lines of Code (LoC)?

- LoC is a direct measure of the physical size of a software program.
- Typically measured in KLOC (Kilo Lines of Code), where 1 KLOC = 1,000 lines.
- Used as a primary input for empirical cost estimation models like COCOMO.
- Provides a quantitative basis for productivity metrics (e.g., KLOC per person-month).

KLOC Conversion

$$\text{KLOC} = \frac{\text{Total Lines of Code}}{1000}$$

Variants of LoC

- **Physical SLOC**: Total number of physical lines in the source code files, including blank lines and comments.
- **Logical SLOC**: Number of executable statements (e.g., counting semicolons in C/Java).
- **NCLOC** (Non-Comment Lines of Code): Excludes blank lines and comment lines; only executable and declaration lines count.

Variant	Description
Physical SLOC	Counts absolutely everything, including whitespace.
Logical SLOC	Focuses only on logical commands (e.g., statements).
NCLOC	Excludes comments/blanks, measures real code written.

Counting Rules for LoC

- No universal standard exists, but conventions must be established per project.
- **Common inclusions**: Declarations, executable statements, control structures.
- **Common exclusions**: Blank lines, comments, auto-generated code, library headers.
- Consistency is key: The same counting tool and rules must be used across the organization.

Example: Calculating LoC

Consider a simple C program:

```
1. /* Calculate Sum */
2. int a=5, b=10;
3.
4. int sum = a + b;
5. printf("%d", sum);
```

- Line 1: Comment (Excluded)
- Line 2: Declaration (Included)
- Line 3: Blank (Excluded)
- Line 4: Statement (Included)
- Line 5: Statement (Included)

Physical LoC = 5 **NCLOC = 3 (Lines 2, 4, 5)**

Estimating LoC Before Coding

- How do we estimate LoC before the software is written?
- Use historical data and expert judgment.
- **Three-point estimation** (PERT technique):

Expected LoC (E)

$$E = \frac{\text{Optimistic (O)} + 4 \times \text{Most Likely (M)} + \text{Pessimistic (P)}}{6}$$

Example: $O = 2000,\ M = 3000,\ P = 5000$

$$E = \frac{2000+4(3000)+5000}{6} = \frac{19000}{6} \approx 3166 \text{ LOC}$$

Advantages of Using LoC

- **Simplicity**: Easy to understand and compute.
- **Automation**: Many automated tools exist to count LoC (e.g., cloc).
- **Historical Baseline**: Huge databases of historical projects exist based on LoC.
- **Direct integration** into algorithmic cost models (COCOMO).

Disadvantages & Limitations of LoC

- **Language Dependent**: 1 line of Python $\neq$ 1 line of Assembly.
- Penalizes well-designed, concise code.
- Difficult to estimate accurately early in the Software Development Life Cycle (SDLC).
- Cannot accommodate non-coding tasks (e.g., requirements, design, testing) directly.

Summary

- LoC is a direct measure of software size, typically expressed in KLOC.
- Clear counting rules (e.g., NCLOC) are necessary for consistency.
- While easy to automate and use in models like COCOMO, it is highly language-dependent.
- It struggles to accurately reflect productivity or functionality.

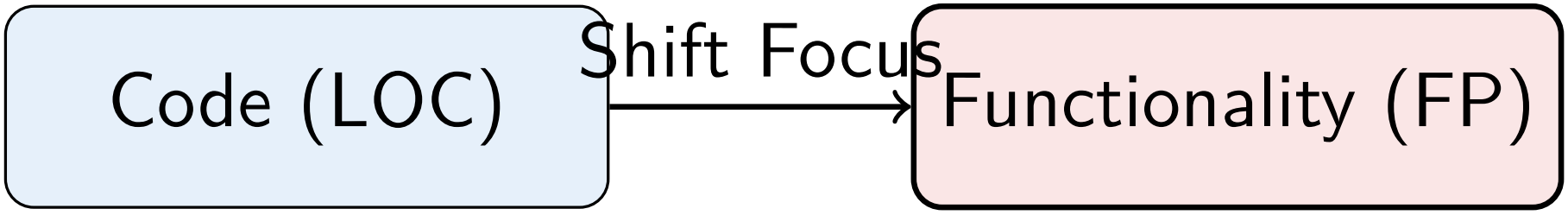
Next Lecture Preview

- **Topic**: Metrics for Size Estimation: Function Points (FP)
- Moving from direct measurement to indirect measurement.
- How to measure software size based on user functionality rather than physical code lines.

Questions?

Introduction to Function Points (FP)

- Proposed by Allan Albrecht at IBM in 1979.
- Measures software size based on the functionality requested by and provided to the user.
- Independent of the programming language, technology, or development methodology.
- Focuses on the 'information domain' rather than the lines of code.



Why Function Points over LOC?

Lines of Code (LOC)	Function Points (FP)
• Highly dependent on the programming language (e.g., Assembly vs. Python). • Difficult to estimate accurately early in the project. • Penalizes high-level languages.	• Independent of programming language. • Can be estimated directly from requirements or design specifications. • Focuses on user-perceivable value.

The 5 Information Domain Parameters

FPA identifies five parameters that represent system functionality:

- **1. External Inputs (EI):** User inputs that change system data (e.g., forms).
- **2. External Outputs (EO):** Outputs generated for the user (e.g., reports, screens).
- **3. External Inquiries (EQ):** User queries that do not change data (e.g., search).
- **4. Internal Logical Files (ILF):** Logical groupings of data maintained within the system (e.g., databases).
- **5. External Interface Files (EIF):** Data grouped by another system but read by the current system.

Weighting Factors for Parameters

Each parameter is assigned a complexity weight: Simple, Average, or Complex.

Parameter	**Simple**	**Average**	**Complex**
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

Calculating Unadjusted Function Points (UFP)

Unadjusted Function Points (UFP) Formula

$$UFP = \sum(\text{Count of parameter} \times \text{Weighting factor})$$

For each of the 5 domains, we multiply the count by the respective weight and sum them up.

$$UFP = (EI \times W_{ei}) + (EO \times W_{eo}) + (EQ \times W_{eq}) + (ILF \times W_{ilf}) + (EIF \times W_{eif})$$

This gives the raw size based strictly on the data and transaction counts.

General System Characteristics (GSC)

UFP does not account for technical or environmental factors.
We assess 14 General System Characteristics (GSCs), rating each on a scale of 0 to 5:

- 0: No influence, 1: Incidental, 2: Moderate, 3: Average, 4: Significant, 5: Essential.

Examples of GSCs:

1. Data Communications
2. Distributed Data Processing
3. Performance
4. Heavily Used Configuration
5. Transaction Rate... (and 9 others)

Calculating Value Adjustment Factor (VAF)

The **Total Degree of Influence (TDI)** is the sum of the 14 GSC scores.

$$TDI = \sum_{i=1}^{14} GSC_i \quad (\text{Max } TDI = 14 \times 5 = 70)$$

Value Adjustment Factor (VAF)

$$VAF = 0.65 + (0.01 \times TDI)$$

- Minimum VAF (TDI $= 0$): $0.65 + (0.01 \times 0) = 0.65$
- Maximum VAF (TDI $= 70$): $0.65 + (0.01 \times 70) = 1.35$

Final Function Point Calculation

The final Function Point (FP) count combines UFP and VAF.

Function Points (FP)

$$FP = UFP \times VAF$$

Once FP is calculated, it can be used to estimate:

- Effort (Person-Months)
- Duration (Months)
- Cost
- Defect Density (Defects / FP)

Example: Calculating FP (Part 1)

Suppose a system has the following counts (all Average complexity):

- EI = 50 (Weight: 4)
- EO = 40 (Weight: 5)
- EQ = 35 (Weight: 4)
- ILF = 6 (Weight: 10)
- EIF = 4 (Weight: 7)

Step 1: Calculate UFP

$$UFP = (50 \times 4) + (40 \times 5) + (35 \times 4) + (6 \times 10) + (4 \times 7)$$

Example: Calculating FP (Part 2)

$$UFP = 200 + 200 + 140 + 60 + 28 = 628$$

Step 2: Determine VAF Assume the system is highly complex, and the TDI across the 14 factors is 50.

$$VAF = 0.65 + (0.01 \times 50) = 0.65 + 0.50 = 1.15$$

Step 3: Calculate Final FP

$$FP = UFP \times VAF = 628 \times 1.15 = 722.2 \text{ Function Points}$$

Converting FP to LOC

Once FP is known, it can be converted to LOC using language-specific gearing factors (or 'backfiring').

Examples of LOC per FP (QSM data):

- C: 128 LOC/FP
- Java: 53 LOC/FP
- Python: 24 LOC/FP

For our example (722.2 FP):

- In Java: $722.2 \times 53 \approx 38,276$ LOC
- In Python: $722.2 \times 24 \approx 17,333$ LOC

Summary

- Function Points measure the functional size of software from the user's perspective.
- It relies on 5 information domains: EI, EO, EQ, ILF, EIF.
- UFP calculates raw size using predefined weights.
- VAF adjusts the size based on 14 technical characteristics.
- **Formula:** $FP = UFP \times VAF$.
- FP is an excellent metric for early project estimation.

Next Lecture Preview

Project Cost Estimation Approaches: Overview

- Heuristic Techniques (e.g., Expert Judgment, Delphi).
- Analytical Techniques (e.g., Halstead's Software Science).
- Empirical Models (Introduction to algorithmic models like COCOMO).

Questions?

The Need for Cost Estimation

- Software estimation predicts the effort (person-months), duration (months), and cost (dollars/rupees) required.
- **Why is it difficult?**
  - Software is intangible and unique.
  - Requirements frequently change.
  - Human productivity varies widely.
- **Consequences of poor estimation:**
  - **Underestimation:** Project delays, budget overruns, team burnout.
  - **Overestimation:** Loss of bids, inefficient resource allocation.

Three Categories of Estimation

1. **Heuristic Estimation:**
   - Based on human intuition, experience, and subjective reasoning.
2. **Analytical Estimation:**
   - Derived mathematically from the fundamental structural properties of the software (e.g., operators and operands).
3. **Empirical (Algorithmic) Estimation:**
   - Based on historical data, statistics, and regression analysis of past projects.

Heuristic Approach: Expert Judgment

- One or more domain experts estimate the effort based on their past experience with similar projects.
- **Advantages:**
  - Very fast and cheap.
  - Experts can account for unique, hard-to-quantify project characteristics.
- **Disadvantages:**
  - Highly subjective; relies heavily on the expert's competence.
  - Prone to cognitive biases (e.g., optimism bias).
  - Difficult to document and justify the reasoning.

Heuristic Approach: Wideband Delphi

- A consensus-based estimation technique to reduce individual bias.
- **The Process:**
  1. A coordinator provides project specs to a panel of experts.
  2. Experts anonymously provide their estimates.
  3. The coordinator distributes the summary of estimates.
  4. Experts discuss the variations (especially the extreme highs and lows).
  5. The process is repeated until consensus is reached.

Analytical Approach

- Derives formulas mathematically rather than relying strictly on empirical data or human guessing.
- Focuses on the micro-level properties of the software.
- Halstead's Software Science is the primary example.
- **Assumption:** Any program is a collection of Operators (e.g., +, -, if, while) and Operands (variables, constants).

Analytical: Halstead's Software Science

- **Parameters:**
  - n_1 = number of distinct operators, n_2 = number of distinct operands
  - N_1 = total occurrences of operators, N_2 = total occurrences of operands
- **Metrics Derived:**
  - **Vocabulary (n)** = $n_1 + n_2$
  - **Length (N)** = $N_1 + N_2$
  - **Volume (V)** = $N \times \log_2(n)$ (Represents information content)
  - **Effort (E)** = Volume / Level (Represents mental effort required)

Limitations of Analytical Approaches

- **Why isn't Halstead's model widely used for cost estimation today?**
  - It calculates effort based on the code AFTER it is written.
  - Estimation is needed BEFORE the project starts.
  - It ignores external factors like team experience, tools, and schedule constraints.
- **Conclusion:**
  - Analytical approaches are better used for complexity measurement rather than early cost estimation.

Empirical (Algorithmic) Approach

- Uses statistical models built from historical project data.
- **General form of empirical models:**

$$Effort = A \times (Size)^B \times M$$

- **Variables:**
  - **Size:** Usually in KLOC or Function Points.
  - **A & B:** Constants derived from regression analysis of past projects.
  - **M:** Cost multiplier based on project attributes (e.g., team skill, complexity).

Empirical Approach: Advantages

- **Why is the empirical approach the industry standard?**
  - **Objective and Repeatable:** The formula yields the same result for the same inputs.
  - **Calibratable:** Organizations can tweak the constants A and B based on their own internal past projects.
  - **Incorporates Drivers:** Accounts for non-functional requirements and team capabilities.
- **Famous Example:** The COCOMO Model (Constructive Cost Model).

Comparison of Estimation Approaches

Approach	Speed	Accuracy	Basis
Heuristic (e.g., Delphi)	Fast	Depends on expert	Human Intuition
Analytical (e.g., Halstead)	N/A (Needs code)	High for complexity	Math/Structure
Empirical (e.g., COCOMO)	Moderate	High if calibrated	Historical Data

Summary

- Accurate cost estimation prevents project failure.
- **Heuristic techniques** (Expert Judgment, Delphi) rely on experience and consensus.
- **Analytical techniques** (Halstead) derive mathematical relationships from code structure.
- **Empirical techniques** build algorithmic formulas ($Effort = A \times Size^B$) based on historical data.
- Empirical models like COCOMO are most widely used for practical estimation.

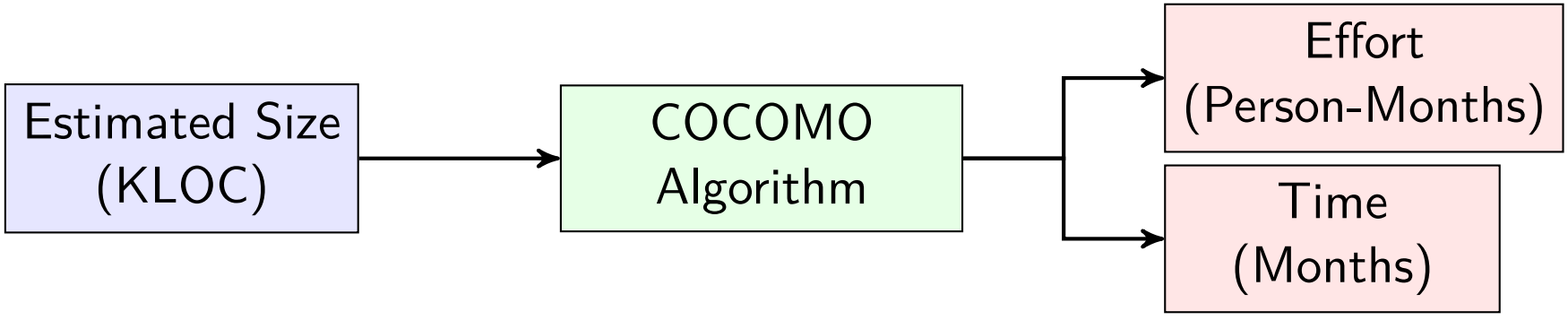
Next Lecture Preview

- In the next lecture, we will dive deeply into the most famous empirical model:
- **The COCOMO Model (Constructive Cost Model) by Barry Boehm.**
- We will learn to calculate effort for:
  - Basic COCOMO
  - Intermediate COCOMO
  - The three modes of projects: Organic, Semi-detached, and Embedded.

Questions?

Introduction to COCOMO

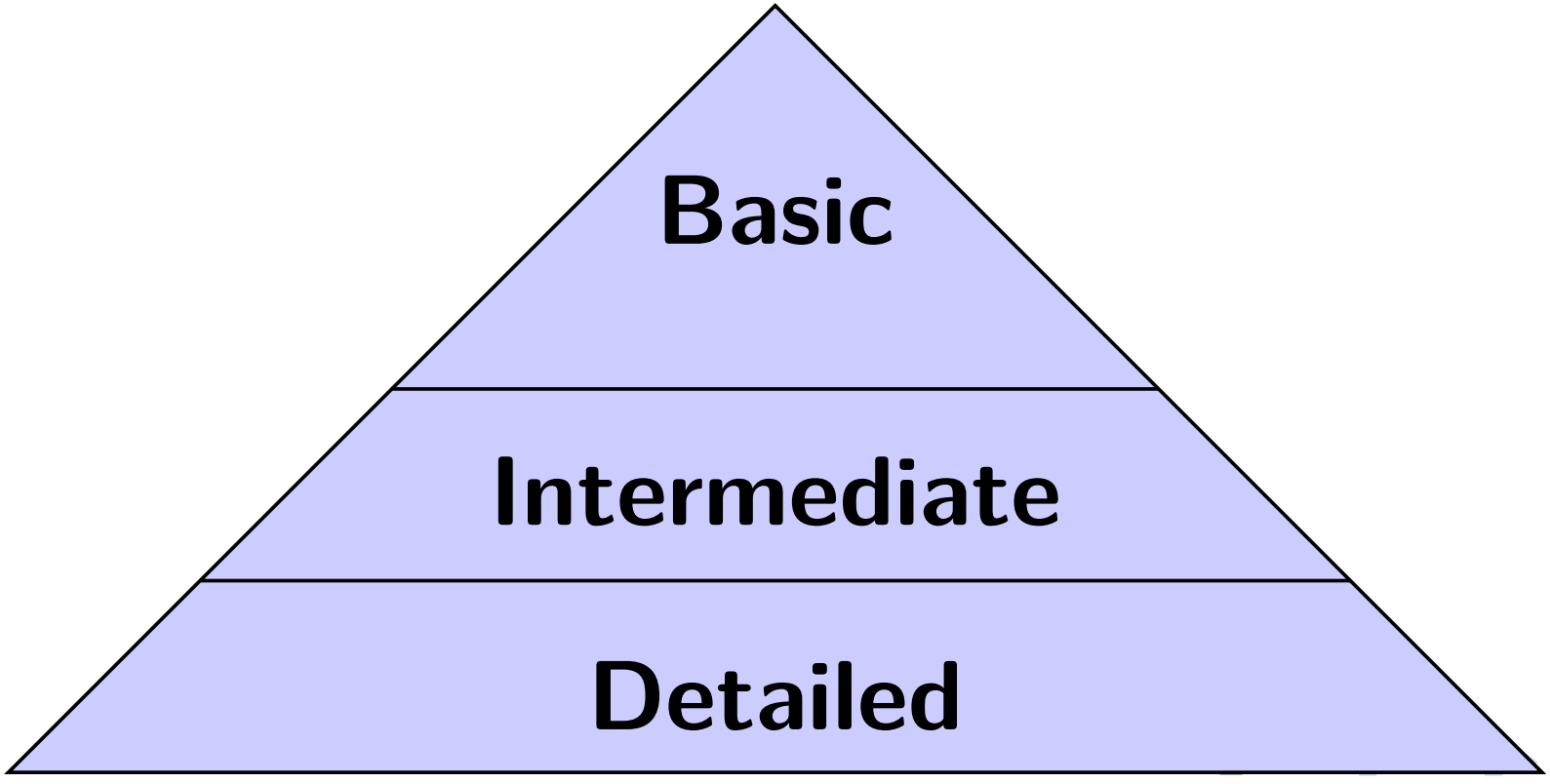
- COCOMO stands for Constructive Cost Model.
- Developed by Barry W. Boehm in 1981.
- It is an algorithmic model that derives software estimates (effort and duration) mathematically.
- The primary input metric is the estimated size of the software in KLOC (Kilo Lines of Code).



The Hierarchy of COCOMO Models

Boehm proposed three levels of the model to allow for increasingly detailed estimates:

1. **Basic COCOMO**: Good for quick, early, rough order of magnitude estimates.
2. **Intermediate COCOMO**: Adds 15 cost drivers (subjective assessments of product, hardware, personnel, and project attributes).
3. **Detailed COCOMO**: Evaluates cost drivers at each specific phase of the software engineering process.



Three Modes of Software Projects

COCOMO categorizes software projects into three distinct modes based on complexity:

- **Organic**: Small, simple projects handled by a small team with prior experience (e.g., payroll system).
- **Semi-detached**: Medium-sized projects with mixed team experience and moderate constraints (e.g., database management system).
- **Embedded**: Complex, tightly constrained systems with rigid requirements (e.g., flight control software, real-time systems).

Comparison of Development Modes

Mode	Size (KLOC)	Innovation & Constraints	Team Experience
Organic	< 50	Low innovation, minimal constraints	High prior experience, small team
Semi-Detached	$50 - 300$	Medium constraints, mixed HW/SW	Mixed experience
Embedded	> 300	High innovation, severe constraints	High requirement for specialized skills

- **Organic**: Size typically < 50 KLOC. High innovation is not required.
- **Semi-Detached**: Size typically 50 - 300 KLOC. Medium constraints, mixed hardware/software interaction.
- **Embedded**: Size often > 300 KLOC. High innovation, severe constraints, real-time processing.

Basic COCOMO Equations

The model uses two fundamental equations:

Formulas

1. **Effort (E)** $= a \times (\text{KLOC})^b$
   - E is measured in Person-Months (PM).
2. **Development Time (D)** $= c \times (E)^d$
   - D is measured in Months.

- **Average Staffing (S)** $= E/D$
  - S is measured in Persons.

Basic COCOMO Constants Table

The constants (a, b, c, d) vary depending on the project mode:

Mode	a	b	c	d
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

- Organic: a=2.4, b=1.05, c=2.5, d=0.38
- Semi-detached: a=3.0, b=1.12, c=2.5, d=0.35
- Embedded: a=3.6, b=1.20, c=2.5, d=0.32

Mathematical Example: Setup

Scenario:

- A team is developing a simple student record system.
- Estimated Size: 32,000 lines of code (32 KLOC).
- Project Type: Familiar team, relaxed constraints $\rightarrow$ Organic Mode.
- Constants to use: a=2.4, b=1.05, c=2.5, d=0.38.

Calculation Setup

$$E = a \times (\text{KLOC})^b$$

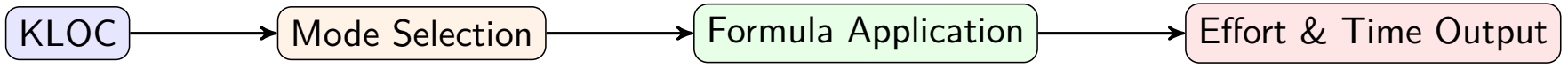
$$E = 2.4 \times (32)^{1.05}$$

Limitations of Basic COCOMO

- Assumes lines of code (KLOC) can be accurately estimated early on (which is extremely difficult).
- Does not account for differences in hardware constraints, personnel skills, or modern tools.
- Ignores software reuse and assumes everything is developed from scratch.
- Limited to procedural languages originally; harder to adapt to Object-Oriented or Agile paradigms.

Summary

- COCOMO is an algorithmic model based on KLOC.
- Projects are categorized into Organic, Semi-detached, and Embedded modes.
- Basic equations calculate Effort in Person-Months and Duration in Months.
- The model is a baseline but lacks consideration for environmental and personnel factors in its basic form.



Next Lecture Preview

- In the next lecture, we will dive into Intermediate COCOMO.
- We will introduce the 15 Cost Drivers (Product, Computer, Personnel, Project).
- We will learn how to calculate the Effort Adjustment Factor (EAF).
- We will perform deep mathematical calculations with real project scenarios.

$$E = a \times (\text{KLOC})^b \times \text{EAF}$$

Questions?

Introduction to Intermediate COCOMO

- Basic COCOMO assumes a standard environment, which is rarely realistic.
- Intermediate COCOMO introduces 15 Cost Drivers to account for the working environment.
- The formula changes slightly:

Intermediate COCOMO Equations

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

$$\text{Time (D)} = c_i \times (\text{E})^{d_i}$$

The 15 Cost Drivers

Cost drivers are grouped into four categories:

- **1. Product Attributes:** Reliability required, Database size, Product complexity.
- **2. Computer Attributes:** Execution time constraints, Storage constraints, Virtual machine volatility, Turnaround time.
- **3. Personnel Attributes:** Analyst capability, Programmer capability, Applications experience, Virtual machine experience, Language experience.
- **4. Project Attributes:** Modern programming practices, Use of software tools, Required development schedule.

Rating Cost Drivers

- Each cost driver is rated on a scale:
- Very Low, Low, Nominal, High, Very High, Extra High.
- **Nominal** always has a multiplier value of 1.00.

Example - Analyst Capability (ACAP)

- **Very Low:** 1.46 (Increases effort)
- **Nominal:** 1.00 (No change)
- **Very High:** 0.71 (Decreases effort)

Intermediate COCOMO Constants

- Note: The 'a' coefficients for Intermediate COCOMO are slightly different from Basic COCOMO:

Project Mode	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

- The Time equations (c and d constants) remain the same as Basic COCOMO.

Scenario: Library Management System

System Specifications

- **System:** Library Management System
- **Estimated Size:** 50 KLOC
- **Mode:** Semi-Detached
  $(a = 3.0, b = 1.12, c = 2.5, d = 0.35)$

Cost Drivers Evaluated:

1. High Reliability Required: 1.15
2. High Database Size: 1.08
3. Very High Programmer Capability: 0.70
4. High Use of Tools: 0.91

All other 11 drivers are Nominal (1.00).

Step 1: Calculating the EAF

- EAF is the product of all 15 cost driver multipliers.

$$\begin{aligned}\text{EAF} &= (\text{Reliability}) \times (\text{DB Size}) \times (\text{Programmer Cap.}) \\ &\quad \times (\text{Tools}) \times (1.00)^{11} \\ \text{EAF} &= 1.15 \times 1.08 \times 0.70 \times 0.91 \times 1 \\ \mathbf{EAF} &= \mathbf{0.791}\end{aligned}$$

Step 2: Calculating Effort (E)

- **Formula:** $E = a_i \times (\text{KLOC})^{b_i} \times \text{EAF}$
- **Constants (Semi-detached):** $a = 3.0, b = 1.12$

Calculation:

$$E = 3.0 \times (50)^{1.12} \times 0.791$$
$$E = 3.0 \times (80.12) \times 0.791$$
$$E = 240.36 \times 0.791$$
$$\mathbf{E = 190.12 \text{ Person-Months (PM)}}$$

Step 3: Calculating Time (D)

- **Formula:** $D = c \times (E)^d$
- **Constants (Semi-detached):** $c = 2.5, d = 0.35$

Calculation:

$$D = 2.5 \times (190.12)^{0.35}$$

$$D = 2.5 \times (6.33)$$

$$\mathbf{D = 15.82\ Months}$$

Step 4: Calculating Average Staffing

- **Formula:** Staffing (S) = Effort/Development Time

Calculation:

$$S = \frac{190.12 \text{ PM}}{15.82 \text{ Months}}$$

$$\mathbf{S = 12.01 \text{ Persons}}$$

Interpretation

We need an average team size of 12 full-time developers to complete this 50 KLOC project in 15.8 months.

Summary

- Intermediate COCOMO uses an Effort Adjustment Factor (EAF).
- EAF is calculated by multiplying 15 environmental cost drivers.
- The formula ($E = a \times \text{KLOC}^b \times \text{EAF}$) yields highly customized Effort estimates.
- Effort and Time dictate the required average staffing.

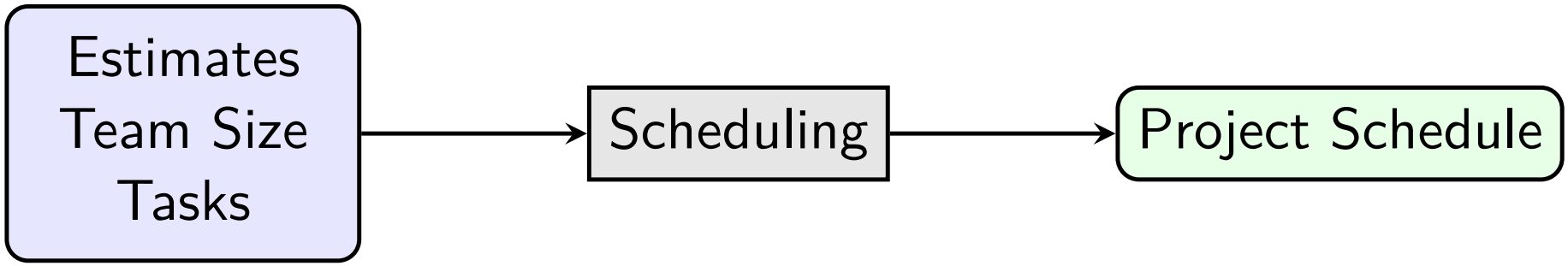
Next Lecture Preview

- In the next lecture, we transition to Project Scheduling.
- We will learn how to map our calculated Time estimates to a real calendar.
- Introduction to the Work Breakdown Structure (WBS).
- Creating and analyzing Gantt Charts for scheduling.

Questions?

What is Project Scheduling?

- Scheduling is the process of mapping estimated tasks to a timeline.
- It answers: **Who is doing what, and when?**
- Key Inputs:
  - Total estimated time (from COCOMO).
  - Available resources (Staffing).
  - Task breakdown.



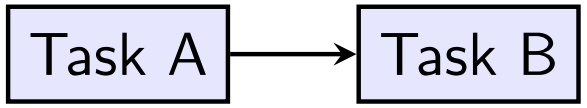
Work Breakdown Structure (WBS)

- Before scheduling, you must know WHAT to schedule.
- **WBS** is a hierarchical decomposition of the total scope of work.
- **Levels of WBS**:
  1. Project (e.g., Library Management System)
  2. Phases (e.g., Requirements, Design, Coding)
  3. Tasks (e.g., Database Design, UI Design)
  4. Sub-tasks (e.g., Design Member Table)

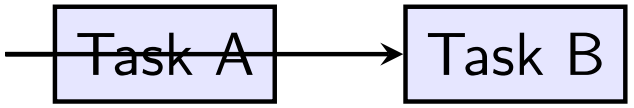
Task Dependencies

Tasks rarely happen in isolation; they depend on one another.

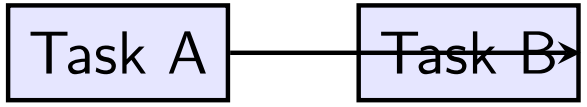
1. **Finish-to-Start (FS)**



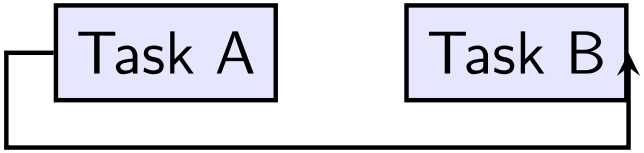
2. **Start-to-Start (SS)**



3. **Finish-to-Finish (FF)**



4. **Start-to-Finish (SF)**



What is a Gantt Chart?

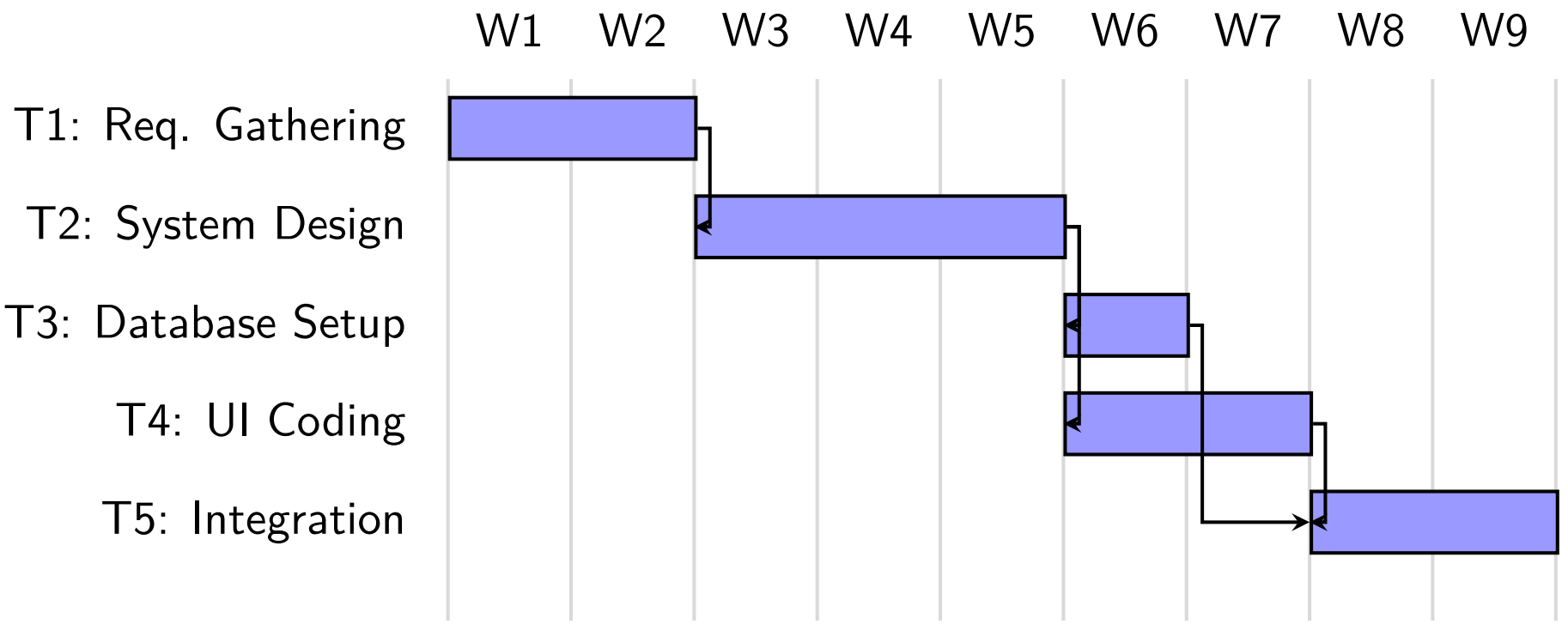
- Invented by Henry Gantt in the 1910s.
- A Gantt chart is a bar chart that visually represents a project schedule.
- **Structure**:
  - **Y-Axis**: List of tasks (from WBS).
  - **X-Axis**: Time (Days, Weeks, Months).
  - **Horizontal Bars**: Represent task duration.
  - **Arrows**: Represent dependencies.

LMS Example: Task List & Dependencies

Project: Library Management System

ID	Task Name	Duration (Weeks)	Predecessors
T1	Requirement Gathering	2	-
T2	System Design	3	T1 (FS)
T3	Database Setup	1	T2 (FS)
T4	UI Coding	2	T2 (FS)
T5	Integration	2	T3, T4 (FS)

LMS Example: Gantt Chart Visualization



Milestones and Tracking

- **Milestones** are zero-duration events marking significant achievements.
- Examples: 'Design Document Approved', 'Beta Release'.
- **Tracking Progress**:
  - Gantt charts can be shaded to show % complete.
  - A vertical 'Today' line helps identify if tasks are behind or ahead of schedule.

Advantages of Gantt Charts

1. **Visual Clarity**: Extremely easy to understand at a glance.
2. **Communication**: Great for showing progress to stakeholders.
3. **Time Management**: Clearly shows start and end dates.
4. **Overlap Visibility**: Easy to spot parallel tasks.

Disadvantages of Gantt Charts

1. **Complexity**: Can become unreadable for very large projects with thousands of tasks.
2. **Updating**: Hard to update manually if a single early task is delayed (though modern software fixes this).
3. **Critical Path**: Unlike PERT charts, Gantt charts don't always easily show the critical path of a project.

Summary

- Project scheduling maps time estimates to a real calendar.
- **WBS** breaks projects down into manageable tasks.
- Tasks have dependencies (FS, SS, FF, SF).
- **Gantt Charts** visually represent the task schedule over a timeline using horizontal bars.

Next Lecture Preview

- In the next lecture, we will explore advanced scheduling techniques.
- Introduction to Network Diagrams.
- PERT (Program Evaluation and Review Technique).
- CPM (Critical Path Method) and calculating slack time.

Questions?

Recap & Agenda

Recap

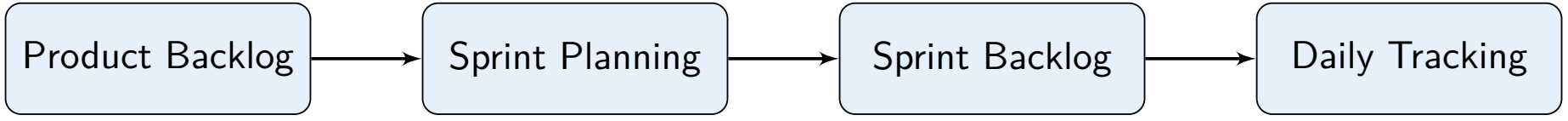
In the previous lecture, we covered Gantt charts and PERT/CPM charts for traditional project scheduling.

Today's Agenda:

- Introduction to Agile Tracking
- What is a Sprint Burn Down Chart?
- Anatomy of the Chart (Axes and Lines)
- Interpreting Chart Patterns
- Updating the Chart

Introduction to Agile Tracking

- Agile emphasizes short, time-boxed iterations called **Sprints** (typically 1-4 weeks).
- At the start of a Sprint, the team commits to a set of User Stories from the Sprint Backlog.
- Tasks are estimated in Story Points or Hours.
- Tracking daily progress is essential to ensure the team can meet the Sprint goal.



What is a Sprint Burn Down Chart?

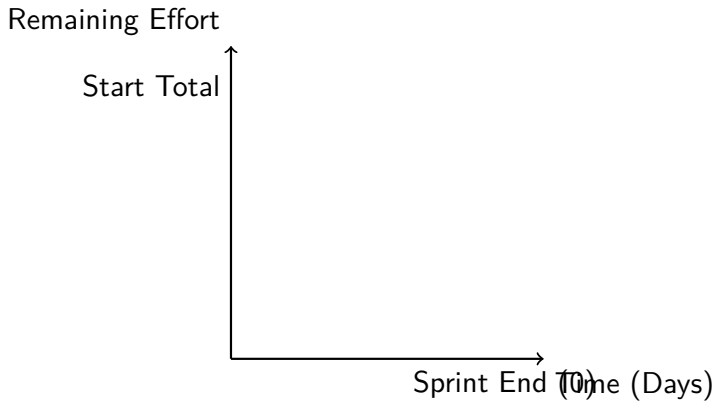
Sprint Burn Down Chart

A visual measurement tool that shows the completed work per day against the projected rate of completion for the current sprint.

- Its primary purpose is to **track the remaining work**.
- It provides a clear, daily status report of the sprint's progress.
- Helps the Scrum Master and team identify scope creep or schedule slips early.

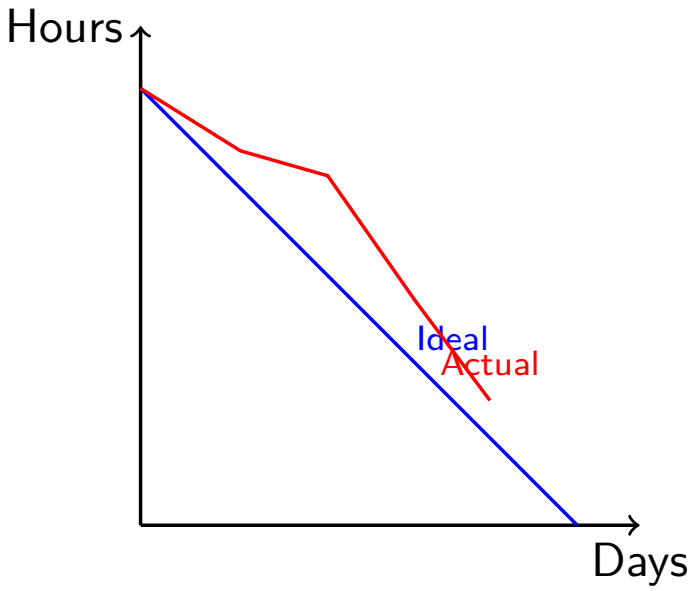
Anatomy of the Chart: The Axes

- **X-Axis (Horizontal):** Represents Time. Usually displayed in days of the sprint (e.g., Day 1 to Day 10).
- **Y-Axis (Vertical):** Represents Remaining Effort. Usually measured in Story Points or Task Hours.
- **Starting Point:** The total estimated effort for all tasks in the Sprint Backlog on Day 0.
- **Ending Goal:** Zero remaining effort on the last day of the sprint.



Anatomy of the Chart: The Lines

- **Ideal Burn Down Line (Guideline):** A straight line drawn from the starting total effort to zero on the last day. Represents a steady, uniform pace of work.
- **Actual Burn Down Line:** A line plotted daily based on the actual remaining work.
- If Actual Line is **BELOW** Ideal Line: Team is ahead of schedule.
- If Actual Line is **ABOVE** Ideal Line: Team is behind schedule.



Calculating the Ideal Line

- **Example:** A 2-week Sprint (10 working days).
- Sprint Backlog total estimate = 200 hours.
- **Ideal Burn Rate** = Total Effort / Number of Days.
- Ideal Burn Rate = 200 / 10 = **20 hours per day**.
- Day 1 Ideal Remaining: 180 hours. Day 2: 160 hours... Day 10: 0 hours.

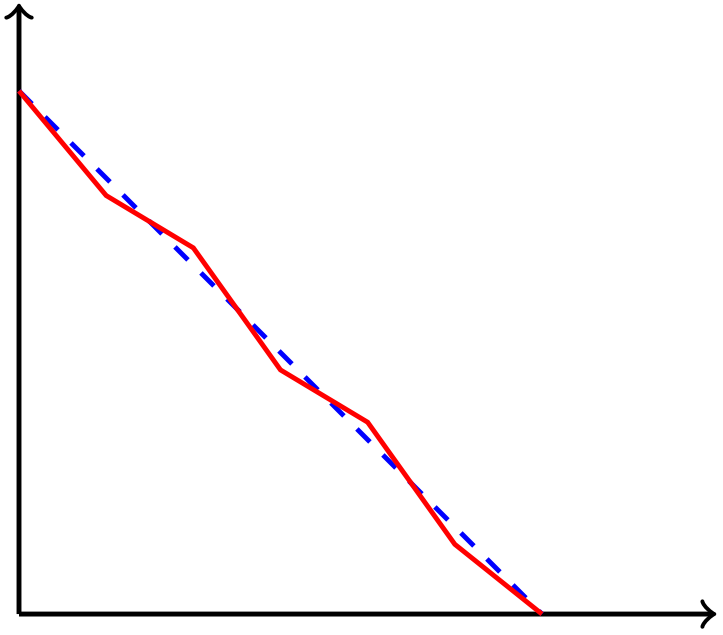
Day	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Ideal Hours	200	180	160	140	120	100	80	60	40	20	0

Plotting the Actual Line

- During the **Daily Stand-up**, team members update the remaining hours on their assigned tasks.
- The Scrum Master sums up the remaining hours for **ALL** tasks.
- This total is plotted on the chart for that specific day.
- **Note:** We track REMAINING work, not COMPLETED work. If a task estimate increases, the actual line goes UP.

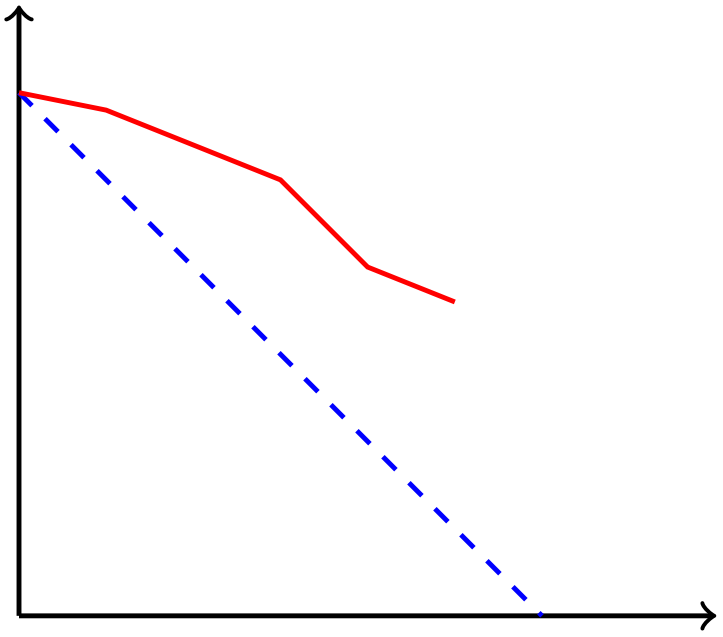
Interpreting Chart Patterns: The "Good" Scenario

- **Pattern:** Actual line closely hugs the Ideal line and reaches zero on the last day.
- **Meaning:** The team's estimates were accurate.
- Pace is consistent.
- No major blockers or scope creep occurred.



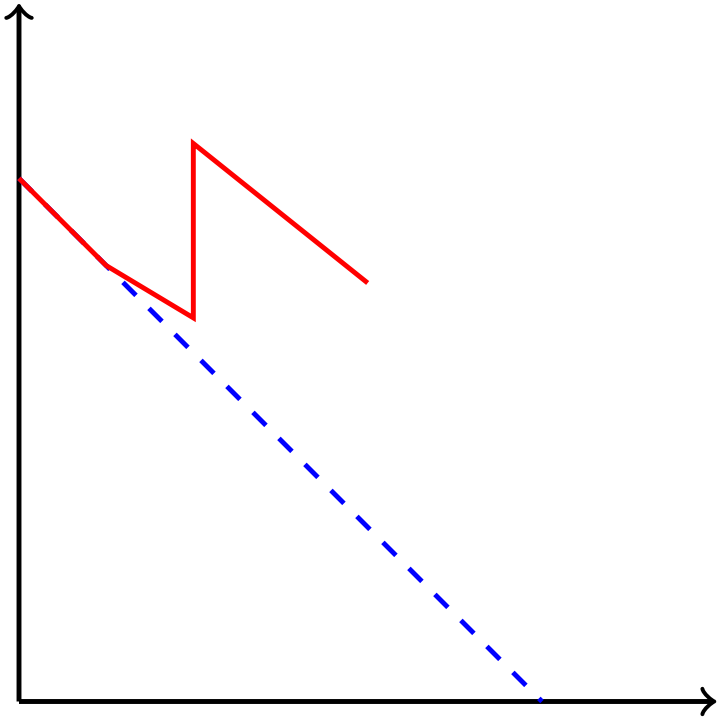
Interpreting Chart Patterns: Behind Schedule

- **Pattern:** Actual line stays consistently **ABOVE** the ideal line.
- **Causes:** Tasks were under-estimated, lower team velocity, or blockers preventing task completion.
- **Action:** Scrum Master must clear blockers; team may need to negotiate removing a User Story from the sprint.



Interpreting Chart Patterns: Scope Creep

- **Pattern:** The actual line abruptly **jumps upwards** in the middle of the sprint.
- **Causes:** New tasks were added to the sprint backlog, or hidden complexity was discovered.
- **Action:** Agile principles discourage adding scope mid-sprint. The Product Owner and team must discuss why this happened.



Limitations of Burn Down Charts

- Shows remaining work, but doesn't show **WHICH** tasks are delayed.
- Doesn't explicitly show scope changes (unlike a Burn UP chart).
- Relies heavily on accurate daily reporting by developers.
- Can lead to a false sense of security if developers "game" the chart by dropping estimates without finishing work.

Burn Down Chart	**Burn Up Chart**
Focuses on remaining work	Focuses on completed work
Hard to see scope changes	Scope changes clearly visible
Goal is to reach zero	Goal is to reach target scope

Summary

- **Sprint Burn Down Charts** visually track remaining work against time.
- **Ideal Line** represents a steady pace; **Actual Line** represents real progress.
- Daily updates are crucial for early detection of schedule issues.
- The chart is a diagnostic tool, prompting team discussion and corrective action.

Next Lecture Preview

- We will transition to a new critical area of Project Management: **Risk Management**.
- **Topic:** Risk Identification and Risk Assessment.
- We will learn how to identify potential project failures before they occur and how to quantify their impact.

Questions?

Recap & Agenda

Previous Lecture

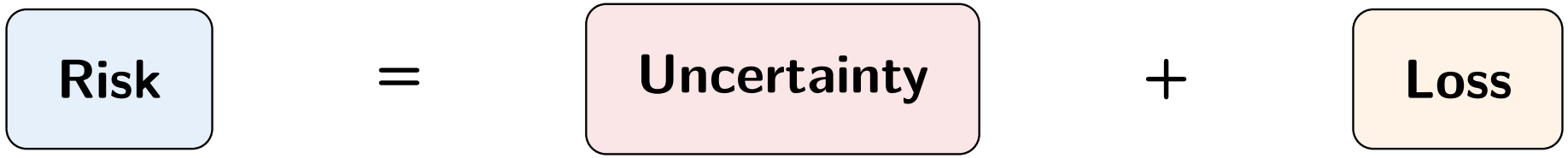
In the previous lecture, we discussed Agile project scheduling using the Sprint Burn Down Chart.

Today's Agenda:

- What is a Software Risk?
- Categories of Risks
- Risk Identification Techniques
- Risk Assessment Criteria
- Calculating Risk Exposure (RE)
- Risk Prioritization

What is a Software Risk?

- A risk is a potential problem that may or may not happen in the future.
- It involves two characteristics: **Uncertainty** (the event may or may not occur) and **Loss** (if it occurs, unwanted consequences arise).
- Risk Management is the process of identifying, assessing, and controlling these risks.
- **Goal:** Minimize the impact of negative events on the project.



Categories of Software Risks

- **Project Risks:** Threaten the project plan. If they occur, schedule slips and costs increase. (e.g., staff turnover, budget cuts).
- **Technical Risks:** Threaten the quality and timeliness of the software. (e.g., unproven technology, ambiguous requirements, integration failures).
- **Business Risks:** Threaten the viability of the software to be built. (e.g., building an excellent product that nobody wants, losing market share to a competitor).

Known vs. Predictable vs. Unpredictable Risks

- **Known Risks:** Uncovered after careful evaluation of the project plan and environment. (e.g., unrealistic delivery date).
- **Predictable Risks:** Extrapolated from past project experience. (e.g., staff turnover, poor communication with the customer).
- **Unpredictable Risks:** The 'unknown unknowns'. They occur without warning. (e.g., a global pandemic, sudden bankruptcy of a major vendor).

Step 1: Risk Identification

- Risk Identification is a systematic attempt to specify threats to the project plan.
- **Common Techniques:**
  1. **Brainstorming:** Team gathers to list all possible risks without filtering.
  2. **Checklists:** Using a standard list of common software risks (e.g., Boehm's Top 10 Software Risks).
  3. **Assumption Analysis:** Reviewing project assumptions to see what happens if they are false.

Boehm's Top 10 Software Risks (Example)

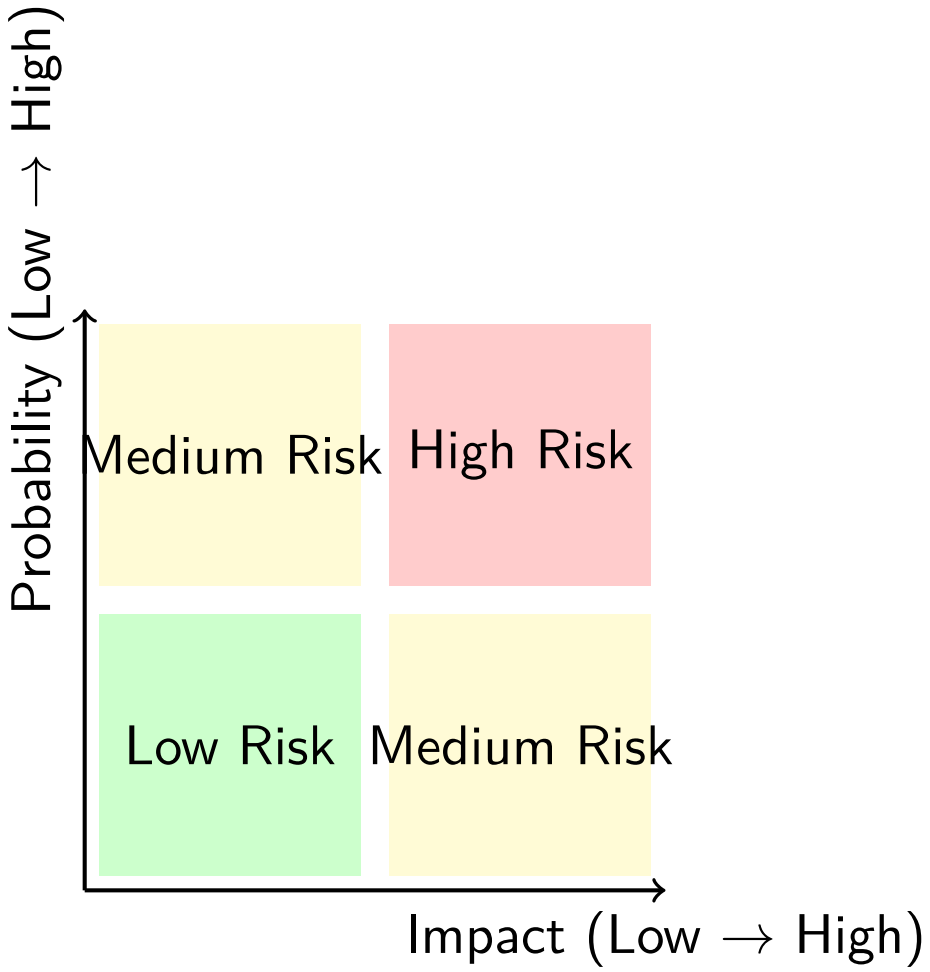
Barry Boehm identified common risks in software engineering:

1. Personnel shortfalls.
2. Unrealistic schedules and budgets.
3. Developing the wrong software functions.
4. Developing the wrong user interface.
5. Gold plating (adding unnecessary features).
6. Continuous stream of requirement changes.

Step 2: Risk Assessment

Once identified, risks must be assessed to prioritize them. We evaluate two main factors for each risk:

- **Probability (P):** The likelihood that the risk will occur (typically expressed as a percentage or a fraction from 0.0 to 1.0).
- **Impact (I) or Loss (C):** The cost to the project if the risk does occur (expressed in dollars, days lost, or a scale like 1-10).



Calculating Risk Exposure (RE)

- Risk Exposure quantifies the overall threat of a risk.
- **Formula:**

Risk Exposure Formula

$$RE = P \times C$$

Where:

- $P =$ Probability of occurrence $(0 < P < 1.0)$
- $C =$ Cost or impact of the loss

RE gives us an expected value of the loss, allowing objective comparison between different risks.

Risk Exposure Example

Risk 1

Key developer leaves the project.
Probability (P) = 30% (0.30)
Impact (C) = Delay of 20 days
(Cost of training replacement).

RE1 = 0.30×20 days

= **6 days expected loss.**

Risk 2

Server failure during testing.
Probability (P) = 10% (0.10)
Impact (C) = Delay of 5 days.

RE2 = 0.10×5 days
= **0.5 days expected loss.**

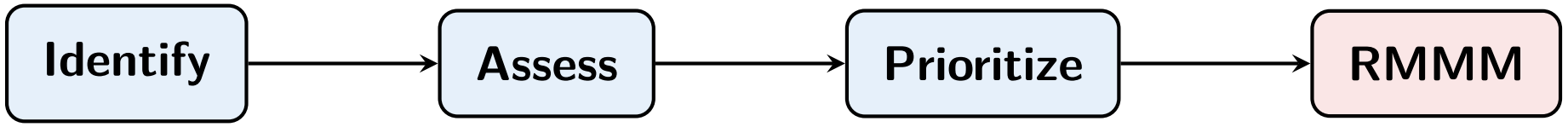
Risk Prioritization (Risk Table)

- Risks are sorted by their Risk Exposure (highest to lowest).
- A cutoff line is drawn.
- Risks above the line are 'Primary Risks' and require a detailed mitigation plan.
- Risks below the line are kept on a watch list but do not receive immediate resources.

Risk Description	**Category**	**Probability**	**Impact (Days)**	**RE (Days)**
Key developer leaves	Project	0.30	20	6.0
Unrealistic schedule	Project	0.50	10	5.0
		— Cutoff Line —		
Server failure	Technical	0.10	5	0.5

Introduction to RMMM

- Identification and Assessment are only the first half of Risk Management.
- The next step is developing an RMMM Plan.
- **RMMM = Risk Mitigation, Monitoring, and Management.**
- This defines exactly what we will do about the prioritized risks.



Summary

- Risk involves uncertainty and potential loss.
- Risks fall into Project, Technical, and Business categories.
- Identification uses brainstorming and checklists.
- Assessment uses the Risk Exposure formula:
  $RE = \text{Probability} \times \text{Impact}$.
- Prioritization ensures focus on the most critical threats.

Next Lecture Preview

- **Topic:** Risk Management: Risk Control.
- We will dive deep into the RMMM plan (Risk Mitigation, Monitoring, and Management).
- We will discuss Proactive vs. Reactive risk strategies.

Questions?

Reactive vs. Proactive Risk Strategies

Reactive Strategy

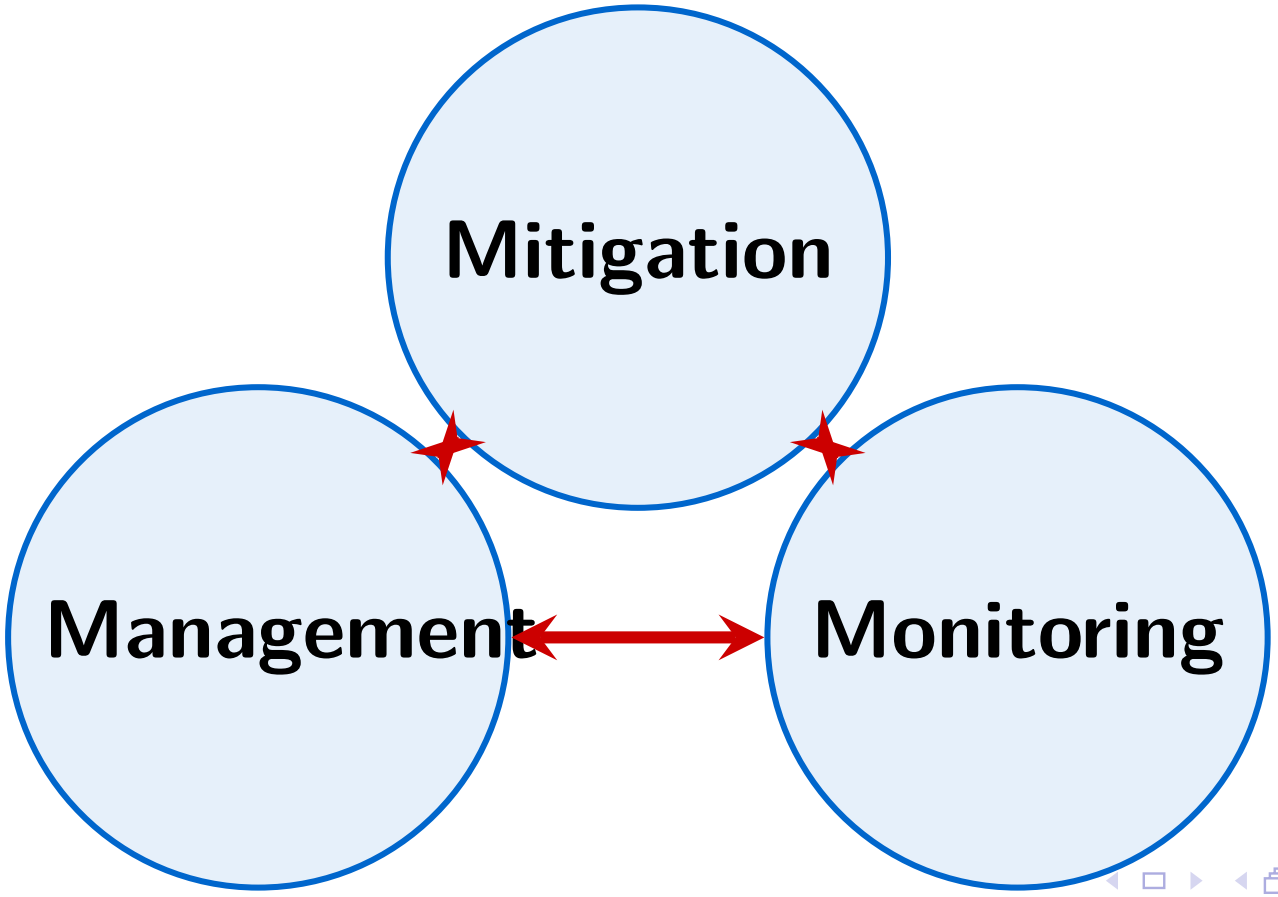
- 'Indiana Jones' approach.
- Team does nothing until a risk actually becomes a problem.
- Response is 'fire-fighting'.
- Often leads to project failure or massive delays.

Proactive Strategy

- 'Chess player' approach.
- Identifies risks early, assesses them, and acts to prevent them.
- Develops a contingency plan in case prevention fails.

The RMMM Plan

- For all risks above the cutoff line in our Risk Table, we create an RMMM plan.
- RMMM stands for:
  1. **Risk Mitigation** (How can we avoid it?)
  2. **Risk Monitoring** (How do we know it's happening?)
  3. **Risk Management** (What do we do if it happens?)
- RMMM is a formal document that becomes part of the Software Project Plan.



1. Risk Mitigation

- Mitigation is synonymous with **Risk Avoidance**.
- **Goal:** Reduce the Probability (P) of the risk occurring.
- Action is taken BEFORE the risk becomes a problem.

Example: High staff turnover

- **Mitigation Actions:** Improve working conditions, offer bonuses, peer reviews to share knowledge.

2. Risk Monitoring

- Even with mitigation, we cannot guarantee the risk won't occur.
- Monitoring involves tracking project indicators to see if a risk is becoming more likely.
- **Goal:** Detect the early warning signs of a risk.

Example: High staff turnover

- **Monitoring Actions:** Track general attitude, monitor extra hours worked (burnout indicator), track job market trends.

3. Risk Management (Contingency Plan)

- Management comes into play when Mitigation fails and the risk becomes an actual Issue.
- Also known as **Contingency Planning**.
- **Goal:** Minimize the Impact (C) of the issue.

Example: High staff turnover actually happens

- **Management Actions:** Re-assign staff, hire contractors, lower project scope, trigger penalty clauses in customer contracts.

Example: RMMM for a Technical Risk

- **Risk:** A third-party API we depend on is unreliable.

Phase	Action
Mitigation	Implement aggressive caching; use timeouts and circuit breaker design patterns in our code.
Monitoring	Track the API's daily uptime percentage and our system's error logs.
Management	Switch to a pre-identified backup API vendor, even if it costs more temporarily.

Risk Information Sheet (RIS)

- Rather than a massive document, RMMM is often recorded on a single page per risk called a **Risk Information Sheet (RIS)**.
- Contains:
  - Risk ID, Date, Probability, Impact.
  - Description of the risk.
  - Mitigation steps, Monitoring indicators, Management plan.
  - Assigned Owner (who is responsible for this risk).

The Cost of Risk Management

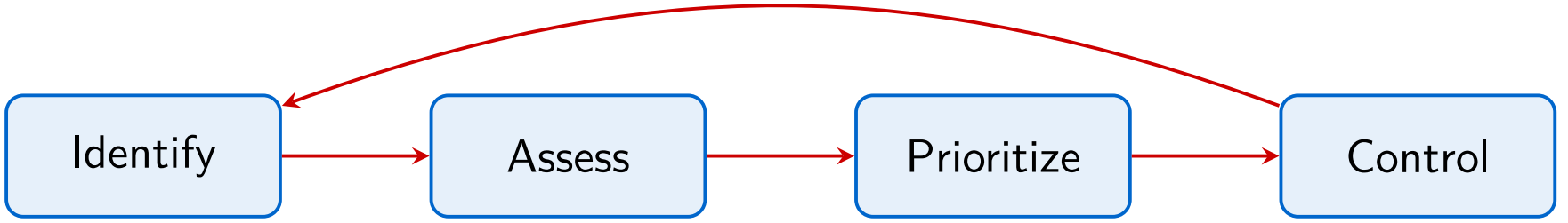
- Risk management is not free. It costs time and money.
- **Cost-Benefit Analysis:** We must ensure that the cost of mitigating a risk is NOT greater than the Risk Exposure (RE).
- If RE = \$10,000, we should not spend \$15,000 to mitigate it.
- Sometimes, the best strategy for low RE risks is simply to **accept them**.

Safety Risks and Hazard Analysis

- In safety-critical software (e.g., medical devices, avionics), risk takes on a different meaning.
- **Software Safety** is a quality assurance activity that focuses on identifying hazards that could cause project failure or human injury.
- Techniques like **Fault Tree Analysis (FTA)** are used to map out exactly how software failures could lead to catastrophic events.

Summary of Risk Management

- Identify risks using checklists and brainstorming.
- Assess risks using Probability x Impact (Risk Exposure).
- Prioritize the top risks in a Risk Table.
- Control risks using the RMMM plan: Mitigate (avoid), Monitor (watch), Manage (contingency).
- Always use a proactive strategy.



Next Lecture Preview

- We are concluding Unit 4 on Software Project Management.
- Next, we will move to a new unit focusing on Software Design.
- **Topic:** Introduction to Software Design concepts and principles.

Questions?

Recap & Agenda

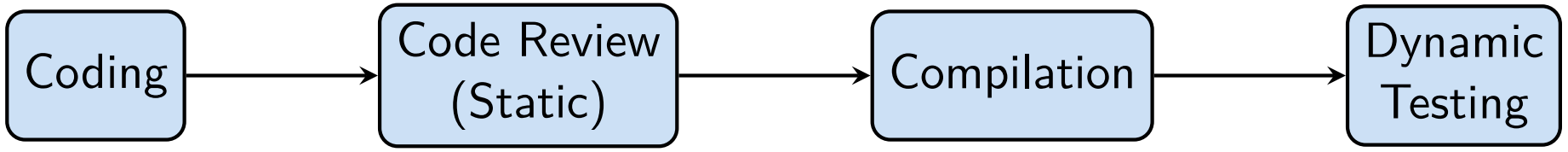
Recap:
Introduction to Software Coding standards and guidelines.

Agenda:

- Introduction to Code Review
- What is a Code Walkthrough?
- The Walkthrough Process
- Roles in a Walkthrough Session
- Advantages and Limitations
- Guidelines for Effective Walkthroughs

Introduction to Code Review

- Systematic examination of computer source code.
- Intended to find mistakes overlooked in the initial development phase.
- Improves overall software quality and developers' skills.
- Types of Static Testing: Walkthroughs, Inspections, and Static Analysis.



What is a Code Walkthrough?

- An informal or semi-formal peer review technique.
- The author of the code leads the review process.
- The author reads through the code logic step-by-step with peers.
- Primary goal: Discover defects, check coding standards, and share knowledge.

The Walkthrough Process

1. **Preparation:** Author prepares the code and distributes it to reviewers in advance (optional but recommended).
2. **Meeting:** Author presents the code, explaining the logic and flow.
3. **Examination:** Reviewers ask questions, point out potential bugs, or suggest improvements.
4. **Note-taking:** A designated scribe records all identified defects and action items.
5. **Follow-up:** Author fixes the defects and updates the code.

Roles in a Walkthrough

Role	Description
Author	The developer who wrote the code. Leads the session and explains the logic.
Reviewers/Peers	Other developers who analyze the code, ask questions, and identify defects.
Scribe/Recorder	Documents all issues, bugs, and suggestions raised during the meeting.
Moderator (Optional)	Ensures the meeting stays on track and within the time limit.

Characteristics of Walkthroughs

- No formal checklist is strictly mandated.
- Focuses heavily on logic verification and algorithmic correctness.
- Open-ended structure allowing for brainstorming alternative solutions.
- Serves as an excellent training tool for junior developers.

Advantages of Code Walkthroughs

- **Early Defect Detection:** Finds bugs before dynamic testing, reducing fixing costs.
- **Knowledge Sharing:** Spreads domain and technical knowledge across the team.
- **Improves Code Readability:** Feedback ensures code is understandable to others.
- **Team Building:** Collaborative problem-solving improves team cohesion.

Disadvantages & Limitations

- **Time-Consuming:** Requires synchronizing schedules of multiple developers.
- **Author Bias:** The author leading the review might unintentionally gloss over their own blind spots.
- **Lack of Formal Metrics:** Difficult to track defect rates precisely due to informal nature.
- **Can go off-topic:** Without a moderator, discussions may drift into architectural debates.

Guidelines for Effective Walkthroughs

- Keep sessions short (under 90 minutes) to maintain focus.
- Review the code, not the coder (maintain a positive, blame-free environment).
- Limit the scope: Review manageable chunks of code (e.g., 200-400 lines).
- Prepare beforehand: Reviewers should glance at the code before the meeting.

Example Scenario

- **Scenario:** A developer wrote a sorting algorithm for a large dataset.
- **Walkthrough:** The developer explains the time complexity and memory usage step-by-step.
- **Outcome:** A peer notices an edge case where the array is already sorted, causing worst-case $O(n^2)$ performance.
- **Fix:** The developer adds a flag to optimize for pre-sorted arrays.

Summary

- Code walkthroughs are informal peer reviews led by the code's author.
- They are effective for finding logic errors and sharing knowledge.
- Key roles include the Author, Reviewers, and Scribe.
- They require a collaborative, blame-free environment to be successful.

Next Lecture Preview

Topic: Code review: Code Inspection

- We will explore the highly formalized Fagan Inspection method.
- Understand the differences between informal walkthroughs and formal inspections.
- Learn about defect tracking and checklists in code inspections.

Questions?

Recap & Agenda

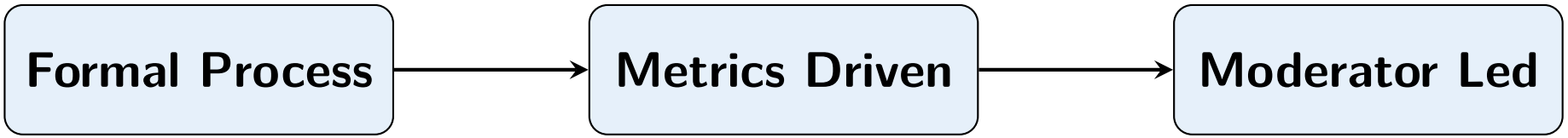
Recap:
Code Walkthroughs: an informal, author-led peer review technique.

Agenda:

- What is Code Inspection?
- The Fagan Inspection Method
- Phases of Formal Inspection
- Roles in Code Inspection
- Use of Checklists
- Inspection vs. Walkthrough
- Advantages and Disadvantages

What is Code Inspection?

- A formal, rigorous, and highly structured code review process.
- Relies heavily on documented procedures and checklists.
- Led by a trained Moderator, NOT the author of the code.
- Primary goal: Detect and document defects, and collect metrics for quality improvement.



The Fagan Inspection Method

- Developed by Michael Fagan at IBM in the 1970s.
- Standardized the formal inspection process across the industry.
- Proven to find a high percentage of defects before testing begins.
- Emphasizes defect data collection to prevent future errors.

Phases of the Inspection Process

- **1. Planning:** Moderator selects team, checks entry criteria, and schedules meeting.
- **2. Overview:** Author provides a brief background of the code module.
- **3. Preparation:** Reviewers study the code individually using checklists.
- **4. Inspection Meeting:** Reader reads the code; defects are identified and logged.
- **5. Rework:** Author resolves the logged defects.
- **6. Follow-up:** Moderator verifies the fixes and approves exit criteria.

Roles in Formal Inspection

- **Moderator:** The leader. Manages the process, schedules, and ensures rules are followed.
- **Author:** Wrote the code. Answers questions and performs rework.
- **Reader:** Reads the code aloud line-by-line or block-by-block during the meeting.
- **Recorder (Scribe):** Formally logs all defects and categorizes them.
- **Inspector:** Examines the code for defects against the checklist.

The Role of Checklists

- Inspections rely on historical defect data.
- Checklists are customized per language (e.g., memory leaks in C++, NullPointer in Java).
- Ensures reviewers look for common, hard-to-spot errors.
- Updated regularly based on metrics collected from past inspections.

Sample Checklist Items

- Are all variables initialized?
- Are array bounds checked?
- Are exceptions handled properly?

Inspection vs. Walkthrough

Feature	**Inspection**	**Walkthrough**
Leader	Moderator	Author
Formality	High, checklist-driven	Low, informal
Preparation	Mandatory	Optional
Goal	Defect logging & metrics	Defect finding & training
Pace	Driven by Reader	Driven by Author

Advantages of Code Inspection

- Highest defect removal efficiency among static testing methods.
- Metrics collection allows for continuous process improvement.
- Eliminates author bias since the author does not lead the review.
- Enforces strict adherence to coding standards.

Disadvantages & Challenges

- High cost in terms of time and resources (preparation and meeting time).
- Can be perceived as overly bureaucratic by developers.
- Requires trained moderators and readers to be effective.
- Logistical difficulty in scheduling multiple key personnel.

Entry and Exit Criteria

- **Entry Criteria:** Code must compile without errors; static analysis tools must be run; all relevant documents (design, requirements) must be available.
- **Exit Criteria:** All logged defects are fixed by the author; Moderator verifies the fixes; metrics are recorded in the database.

Summary

- Code inspection is a highly formal, checklist-driven review process.
- It follows the Fagan methodology with strict phases (Planning to Follow-up).
- Roles are well-defined: Moderator, Reader, Author, Recorder, Inspector.
- It focuses on defect removal and metric collection, avoiding author bias.

Next Lecture Preview

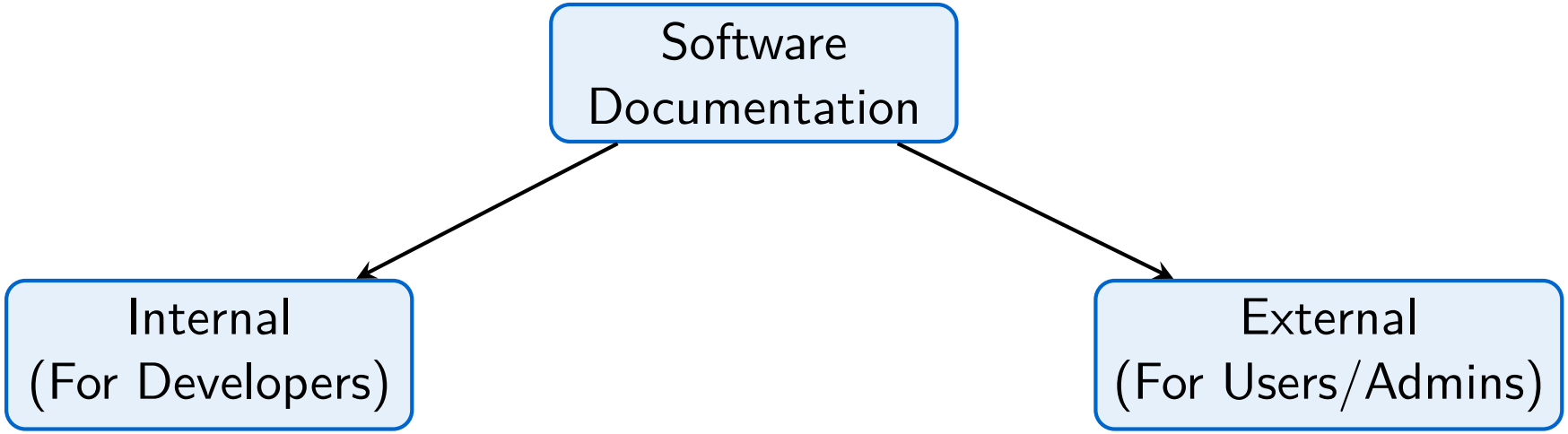
Topic: Software Documentation: Internal Documentation

- We will shift our focus from reviewing code to documenting it.
- Learn about the importance of comments, naming conventions, and standard preambles.
- Discuss tools that automate internal documentation generation.

Questions?

What is Software Documentation?

- Written text or illustration that accompanies computer software.
- Explains how the software operates, how to use it, or how to maintain it.
- Crucial for knowledge transfer, maintenance, and user support.
- Categorized broadly into Internal and External documentation.



Internal vs. External Documentation

Aspect	Internal Documentation	External Documentation
Target Audience	Developers, Maintainers	End-users, System Admins
Location	Embedded in source code	Manuals, PDFs, Web pages
Content	Algorithms, code logic, variables	Installation guides, user manuals, API usage
Purpose	Facilitate code maintenance	Facilitate software usage

What is Internal Documentation?

- Documentation written directly within the source code.
- Does not affect the execution of the program (ignored by compilers).
- Aims to make the code comprehensible to other developers.
- Includes meaningful variable names, formatting, and comments.

Internal Documentation Formula

$$\text{Code} + \text{Formatting} + \text{Comments} = \text{Internal Doc}$$

Meaningful Identifiers

- The first line of defense in internal documentation.
- Names should reveal intent. Avoid cryptic abbreviations.
- Follow language-specific conventions (e.g., camelCase, snake_case).

Poor Naming

```
int d; // elapsed time in days
```

Good Naming

```
int elapsedTimeInDays;
```

Code Formatting and Indentation

- Visual structure aids cognitive parsing of code logic.
- Consistent indentation reveals control flow (loops, conditionals).
- Use blank lines to separate logical blocks of code.
- Adopt a standard style guide (e.g., Google Java Style, PEP 8 for Python).

Types of Comments

1. **Block Comments:** Used at the beginning of a file, class, or complex method to explain overall purpose.
2. **Inline Comments:** Placed on the same line as code to explain tricky or non-obvious logic.
3. **Docstrings/Documentation Comments:** Specially formatted comments intended for extraction by tools (e.g., Javadoc).

Golden Rule

Comment the 'Why', not the 'What'.

Standard Preambles (File Headers)

A block comment at the very top of a source file. Standard details usually include:

- File Name & Project Name
- Author(s) & Date of Creation
- Version History / Modification Log
- Brief description of the file's contents.

Example Preamble

```
/*
 * File: Main.java
 * Author: John Doe
 * Date: 2026-07-22
 * Description: Main entry point for the Applicatio
 */
```

Best Practices for Internal Documentation

- **Keep comments updated:** Outdated comments are worse than no comments.
- **Don't over-comment:** Avoid cluttering the code with redundant explanations.
- **Explain workarounds:** If you use a 'hack' to fix a bug, document exactly why it was necessary.
- Write clear, grammatically correct notes.

Documentation Generators

- Tools that parse internal documentation comments to generate external HTML/PDF references.
- Examples: Javadoc (Java), Doxygen (C++), Sphinx (Python).
- They read specific tags like `@param`, `@return`, and `@throws`.
- Bridges the gap between internal code and API reference manuals.

Summary

- Internal documentation lives inside the source code for developers.
- It includes meaningful names, consistent formatting, and comments.
- Comments should explain the 'Why', not the 'What'.
- Standard preambles and documentation generators streamline code maintenance.

Next Lecture Preview

- **Topic:** Software Documentation: External Documentation
- We will explore documentation aimed at the end-users and administrators.
- Topics will include User Manuals, Installation Guides, and System Administration Docs.
- Understand how external docs impact user satisfaction.

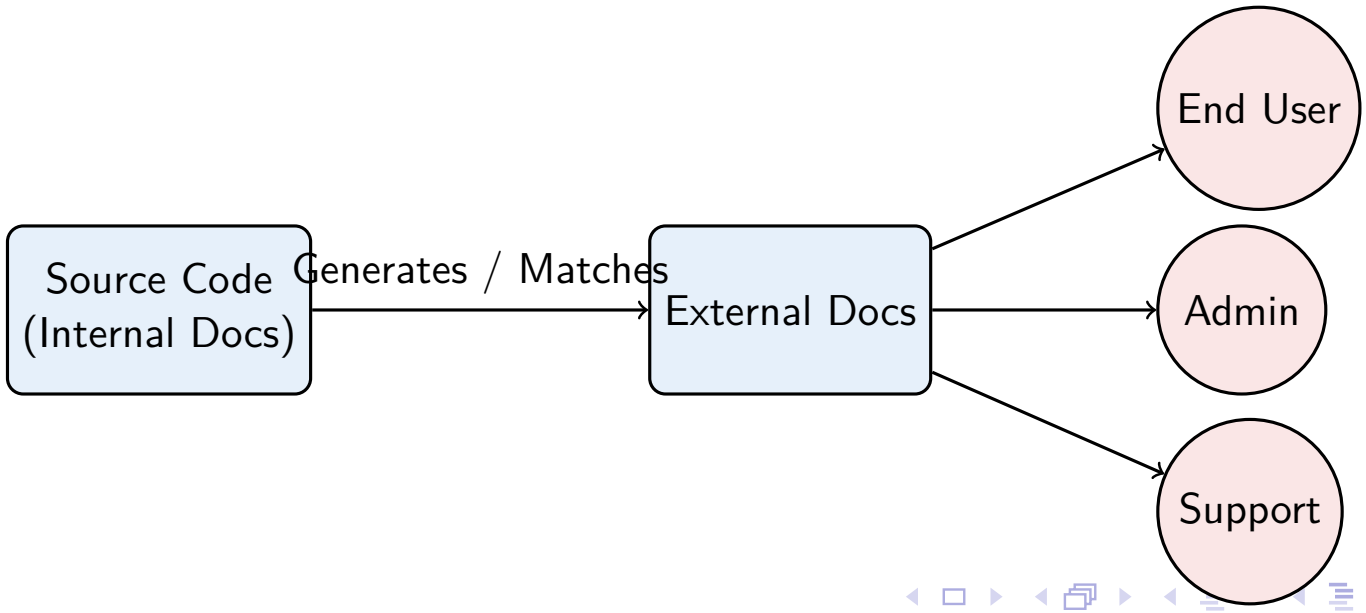
Questions?

Agenda for External Documentation

- Definition and Role of External Documentation
- IEEE 1063: Standard for Software User Documentation
- End-User Manuals: Structure and Contents
- System Administration/Operations Manuals
- Online Help Systems and Knowledge Bases
- Summary and Next Steps

What is External Documentation?

- Artifacts maintained outside the source code repository intended for users, operators, and support staff.
- Unlike internal documentation (comments, AST-based docs), external docs treat the software as a black-box.
- Primary Goals: Usability, Installability, and Operational Support.
- Key Challenges: Synchronization with agile releases, versioning, and localization (L10n).

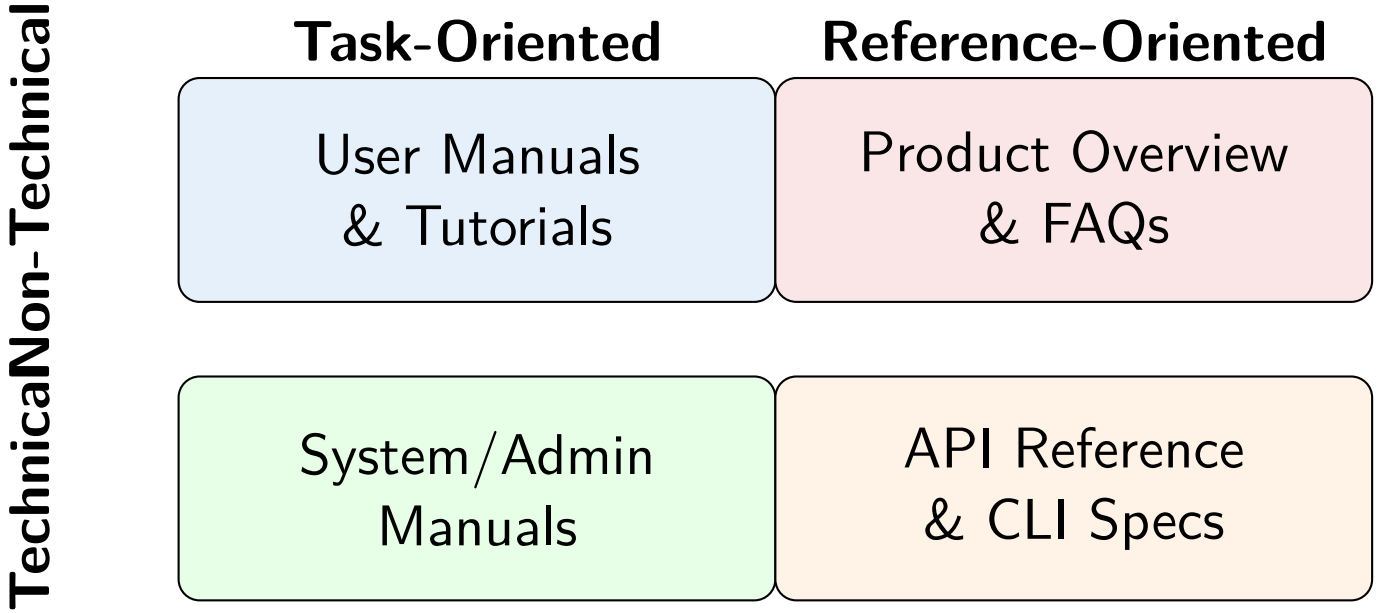


IEEE 1063 Standard Overview

- IEEE 1063-2001 (R2007) is the standard for software user documentation.
- Defines minimum requirements for the structure, information content, and format of user documentation.
- Applies to printed manuals, electronic documents, and online help.
- Mandates clear identification of the intended audience and prerequisites.

Types of External Documentation

- **User Manuals:** Task-oriented guides for end-users to accomplish business goals.
- **System/Admin Manuals:** Technical guides for IT staff covering installation, configuration, and maintenance.
- **Reference Manuals:** Exhaustive lists of features, API endpoints, or command-line arguments.
- **Tutorials/Quick Start Guides:** Pedagogical documents designed for onboarding.



The End-User Manual: Structure

- 1. Introduction: Scope, intended audience, and system overview.
- 2. Getting Started: Installation (if applicable) and initial setup.
- 3. Task-Based Workflows: Step-by-step instructions for core use cases (e.g., 'How to generate a monthly report').
- 4. Troubleshooting/FAQ: Solutions for common user errors and edge cases.
- 5. Glossary & Index: Defining domain-specific terminology.

TOC: Library Management System User Manual

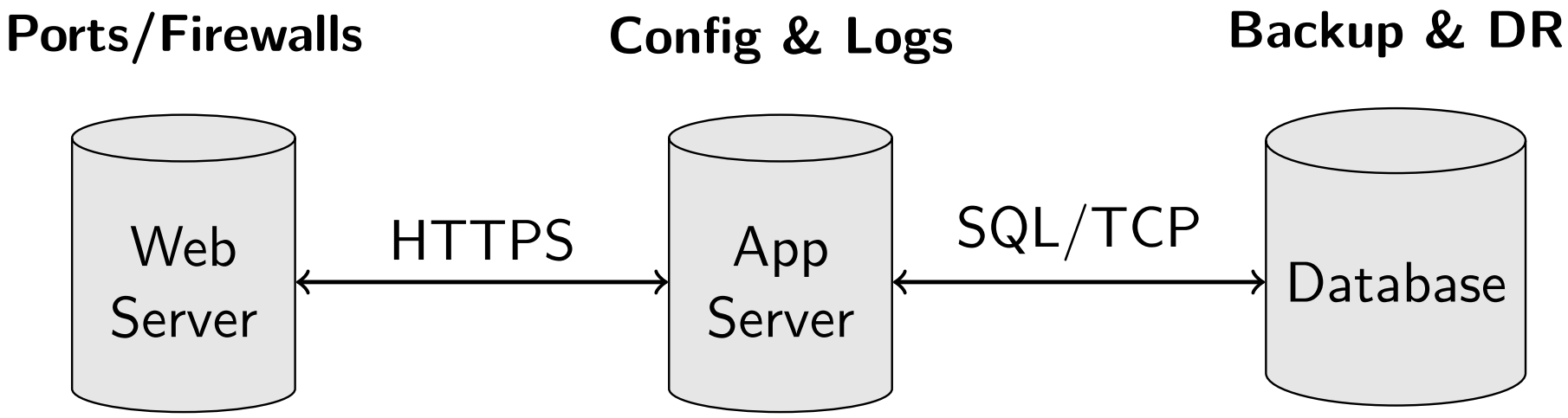
Writing Task-Oriented Procedures

- **Imperative Mood:** Use command verbs (e.g., 'Click the Submit button', not 'The Submit button should be clicked').
- **Granularity:** Break complex tasks into sub-tasks (max 7-9 steps per procedure).
- **Visuals:** Integrate screenshots with callouts (arrows, highlighting) for context.
- **Feedback loop:** Explicitly state the expected result after an action (e.g., 'A confirmation dialog will appear').

Poor Procedure	**Effective Procedure**
The user has to navigate to settings and then maybe change the password.	1. Click **Settings**. 2. Select **Password**. 3. Enter new password. *Result: A confirmation dialog will appear.*

System Administration Manuals (SAM)

- **Target Audience:** IT Ops, DevOps, Database Administrators.
- **Focus:** Deployment, configuration, monitoring, and recovery rather than standard usage.
- **Pre-requisites:** Detailed hardware requirements, OS dependencies, and network configurations (Ports, Firewalls).
- **Critical Component:** Disaster Recovery (DR) and Backup/Restore procedures.



SAM Content: Configuration & Deployment

- **Deployment Topologies:** Single-node vs. High-Availability (HA) cluster setup.
- **Configuration Files:** Exhaustive listing of JSON/YAML/XML parameters, default values, and valid ranges.
- **Security Hardening:** Best practices for SSL/TLS certificates, RBAC setup, and least privilege access.
- **Performance Tuning:** JVM heap sizes, connection pool limits, and caching strategies.

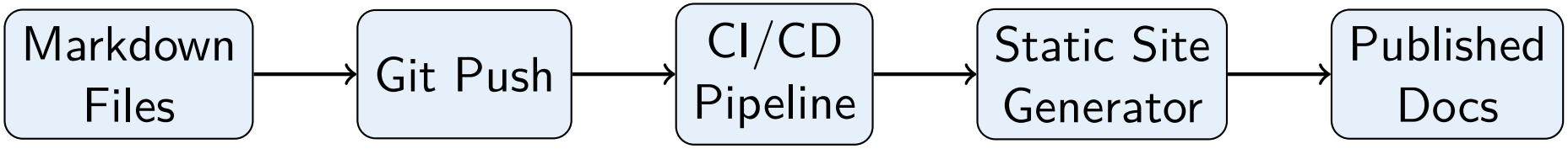
config.yaml

```
server:
  port: 8080
  ssl: true
db:
  pool_size: 50
```

Parameter	Type	Description
`port`	Int	Port to bind (default: 8080)
`ssl`	Bool	Enable TLS encryption
`pool_size`	Int	DB connections (max 100)

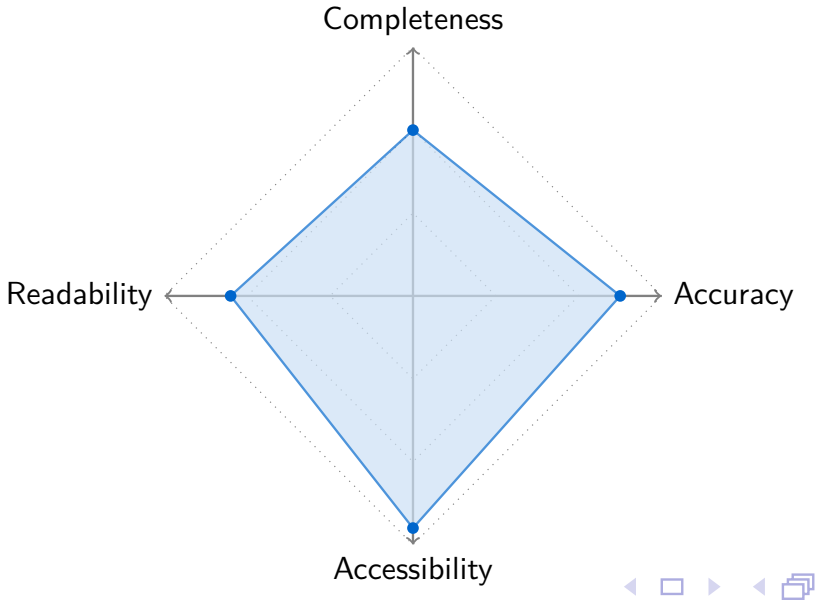
Online Help and Context-Sensitive Docs

- **Context-Sensitive Help:** Tooltips, inline hints, and localized '?' icons providing immediate assistance without leaving the UI.
- **Knowledge Bases (KB):** Searchable repositories of articles, often utilizing taxonomies and tagging.
- **Docs-as-Code:** Managing documentation using Markdown/AsciiDoc in Git, alongside source code (e.g., Sphinx, MkDocs, Docusaurus).
- **CI/CD Integration:** Automatically generating and publishing static doc sites upon release.



Quality Metrics for Documentation

- **Accuracy:** Technical correctness reflecting the current software version.
- **Completeness:** Coverage of all critical use cases and error states.
- **Readability:** Assessed using metrics like Flesch-Kincaid readability tests.
- **Accessibility:** Adherence to WCAG guidelines for online documentation (screen-reader compatibility).



Summary

- ✓ External documentation serves users and administrators, treating the system as a black box.
- ✓ User Manuals are task-oriented and written in the imperative mood.
- ✓ System Admin Manuals focus on deployment, configuration, and disaster recovery.
- ✓ Docs-as-code integrates documentation into the standard CI/CD software lifecycle.

Next Lecture Preview

- **Topic:** Unit Testing
- **Key concepts:** White-box testing fundamentals.
- **Techniques:** Statement coverage, Branch coverage, and Path coverage.
- **Tools:** Introduction to JUnit/NUnit testing frameworks.

Questions?

Recap & Agenda

Recap: Lecture 40

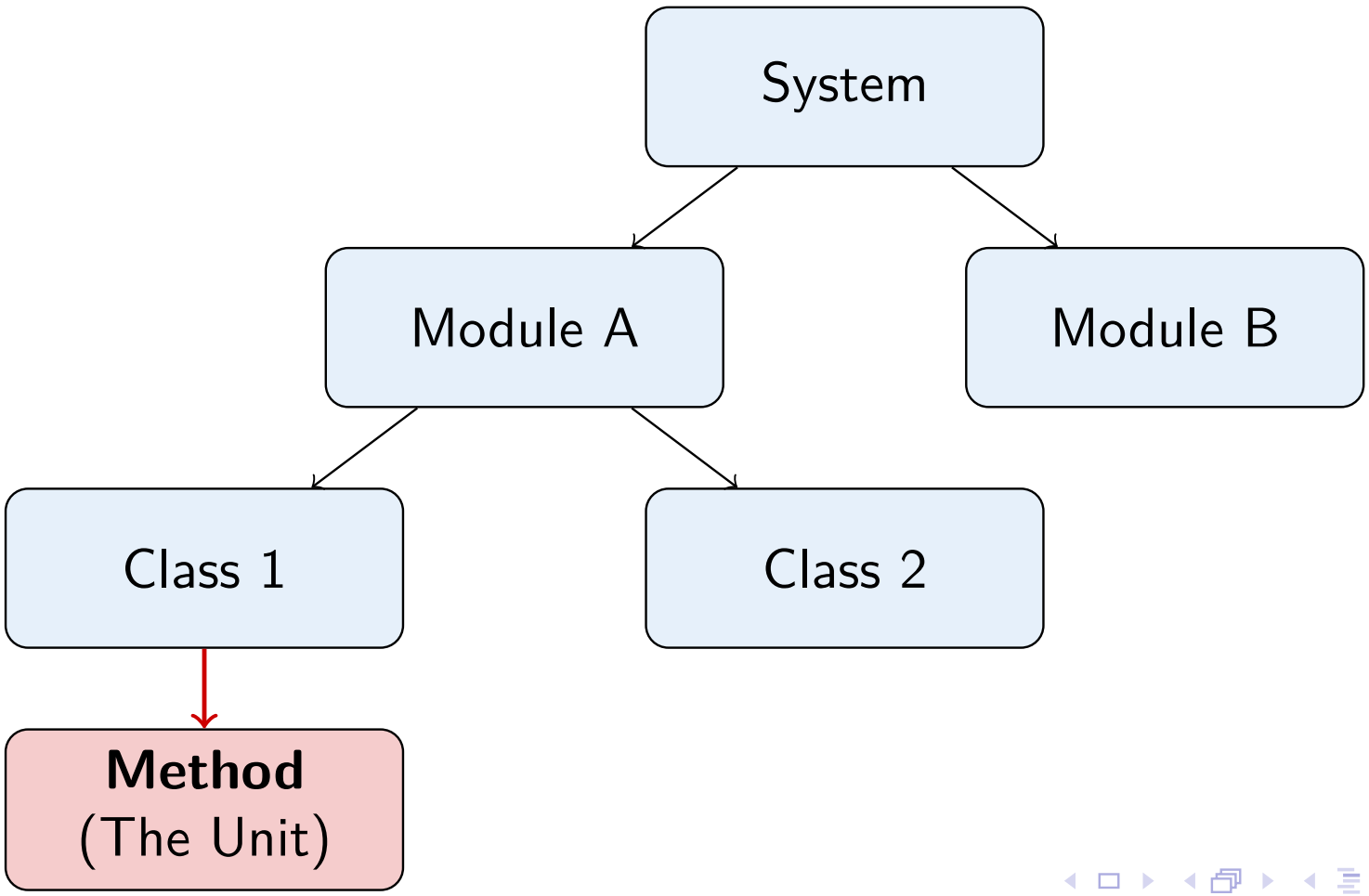
In lecture 40, we discussed External Documentation, emphasizing user manuals, system administration guides, and the docs-as-code methodology.

Today's Agenda:

- Definition and Scope of Unit Testing
- Isolating Components: Stubs and Drivers
- White-Box Testing Fundamentals
- Code Coverage Metrics (Statement, Branch, Condition)
- Control Flow Graphs and Cyclomatic Complexity
- Basis Path Testing

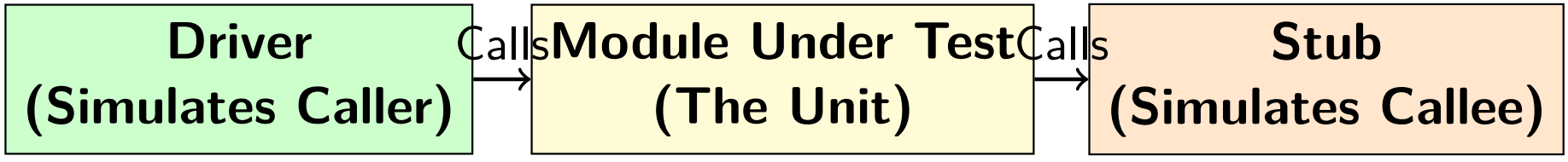
What is a Unit?

- A 'Unit' is the smallest testable part of an application.
- In procedural programming, a unit is a function or procedure.
- In Object-Oriented Programming (OOP), a unit is typically a method within a class.
- Goal: Verify that the internal logic of the unit functions exactly as designed.



Unit Isolation: Drivers and Stubs

- A unit rarely operates in a vacuum; it calls other functions or is called by them.
- **Driver:** A 'main' program that accepts test case data, passes it to the unit, and prints the results.
- **Stub:** A dummy subprogram that replaces a module called by the unit being tested.
- **Mock Objects:** Advanced stubs that verify interactions (e.g., in OOP testing frameworks like Mockito).



Introduction to White-Box Testing

- Also known as Glass-Box, Clear-Box, or Structural Testing.
- Prerequisite: Access to the source code.
- Focus: Internal mechanisms, data structures, and control flow logic.
- Contrast with Black-Box: Black-box tests functionality without knowing the internal code.

White-Box

Tests based on knowledge of the internal logic and structure of the code.

Black-Box

Tests based on inputs and outputs, treating the code as an opaque box.

Code Coverage Metrics: Statement Coverage

- **Formula:** $\frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$
- **Goal:** Ensure every line of code is executed at least once.
- Limitation: Cannot detect missing control flow paths (e.g., an empty 'else' block).
- Often considered the weakest form of coverage metric, but a necessary baseline.

```
1: int result = 0;
2: if (X > 0) {
3:   result = 1;
4: }
5: return result;
```

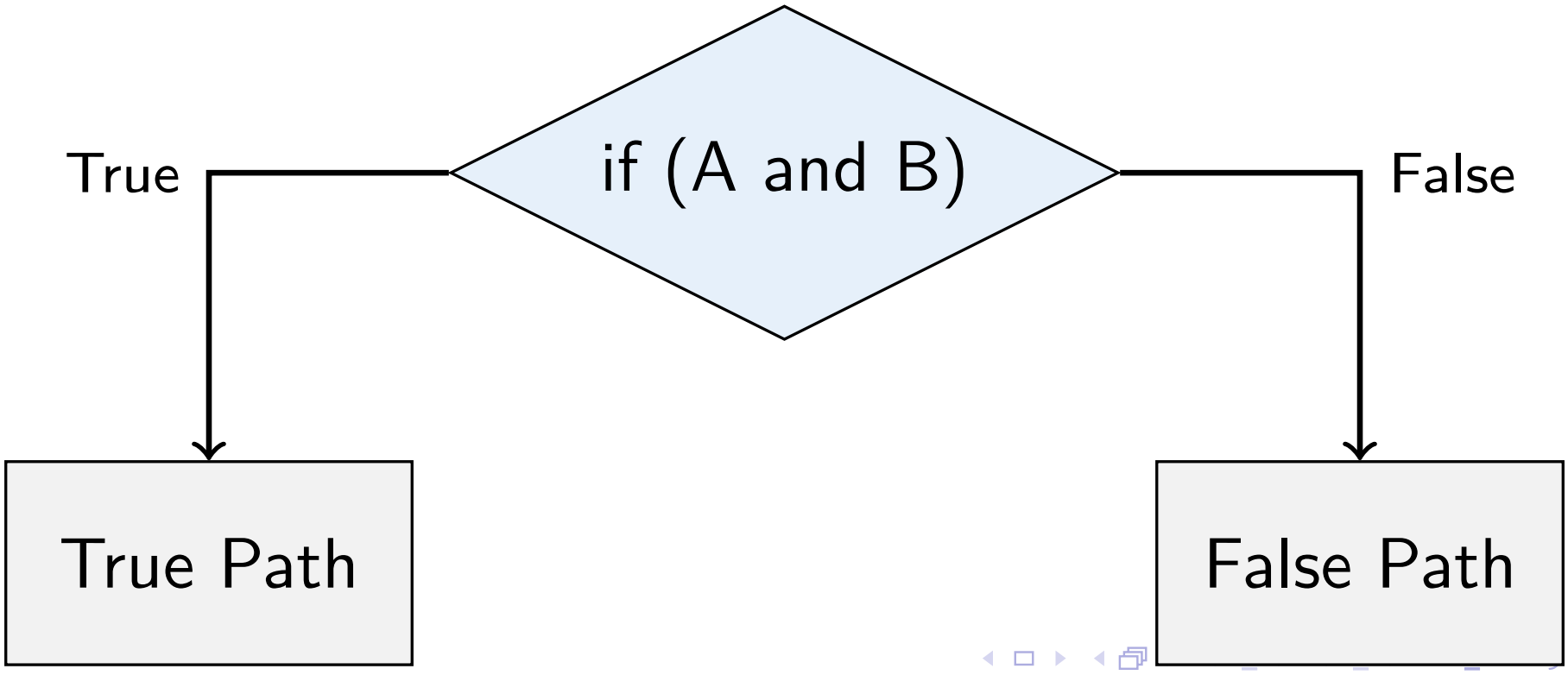
Test Case: $X = 5$

Lines executed: 1, 2, 3, 4, 5

Coverage: 100%

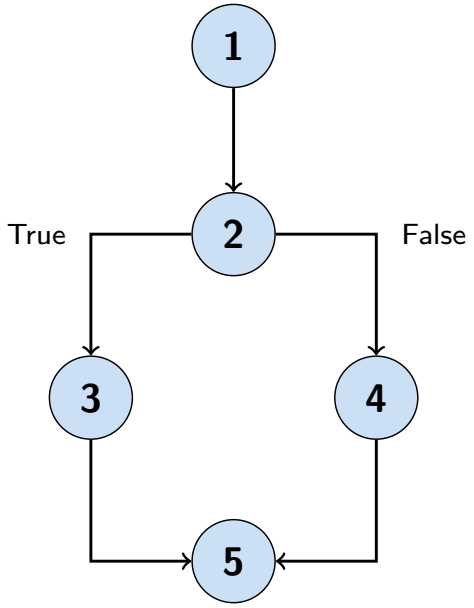
Code Coverage Metrics: Branch & Decision Coverage

- **Branch Coverage:** Ensure every possible branch (True and False) of every control structure is executed.
- **Formula:** $\frac{\text{Number of executed branches}}{\text{Total number of branches}} \times 100$
- Stronger than statement coverage.
- **Condition Coverage:** Evaluates every sub-expression (boolean operand) in a compound decision independently.



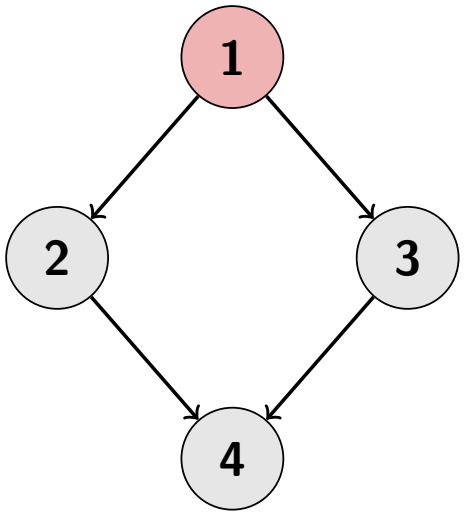
Control Flow Graphs (CFG)

- A directed graph representing all paths that might be traversed through a program during its execution.
- **Nodes:** Represent computational statements or expressions.
- **Edges:** Represent transfer of control (branches).
- **Predicate Nodes:** Nodes that contain a condition (outgoing degree ≥ 2).



Cyclomatic Complexity (V(G))

- A software metric introduced by Thomas McCabe to measure the logical complexity of a program.
- Defines the number of independent paths in the basis set of a program.
- **Formula 1:** $V(G) = E - N + 2$ (Where E = edges, N = nodes)
- **Formula 2:** $V(G) = P + 1$ (Where P = number of predicate nodes)



Nodes (N) = 4
Edges (E) = 4
Predicates (P) = 1 (Node 1)
$V(G) = 4 - 4 + 2 = 2$
$V(G) = 1 + 1 = 2$

Basis Path Testing

1. Draw the Control Flow Graph from the source code.
2. Calculate Cyclomatic Complexity V(G).
3. Determine the basis set of linearly independent paths (exactly V(G) paths).
4. Prepare test cases that force execution of each path in the basis set.

Concept	**Example**
Code	`if (X > 0) { Y = 1; } else { Y = 2; }`
CFG Nodes	1 (Condition), 2 (True Block), 3 (False Block), 4 (End)
V(G)	$E = 4, N = 4 \implies V(G) = 4 - 4 + 2 = 2$
Independent Paths	Path 1: $1 \to 2 \to 4$ (Test with X = 1)
	Path 2: $1 \to 3 \to 4$ (Test with X = -1)

[

fragile]Unit Testing Frameworks (xUnit)

- Modern unit testing is automated using frameworks derived from SUnit.
- Java: JUnit, TestNG
- C++: GoogleTest, Catch2
- Python: unittest, pytest
- Features: Annotations (@Test), Assertions (assertEquals), and Test Runners.

Example: JUnit 5 Snippet

```
@Test
public void testAddition() {
    Calculator calc = new Calculator();
    int result = calc.add(2, 3);
    assertEquals(5, result, "2+3=5");
}
```

Summary

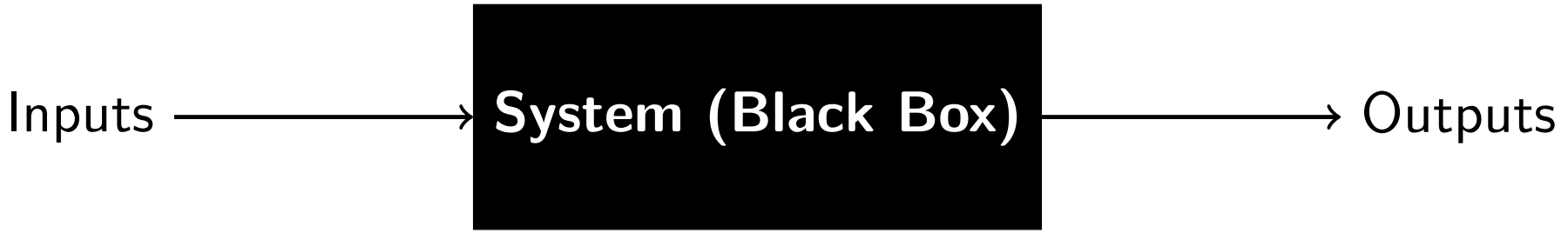
- **Unit testing** isolates and verifies the smallest pieces of code using stubs and drivers.
- **White-box testing** relies on internal structural knowledge of the code.
- **Code coverage metrics** (Statement, Branch) quantify testing thoroughness.
- **Cyclomatic Complexity** provides a mathematical basis for determining test paths.

Key Takeaway

Cyclomatic complexity ($V(G) = E - N + 2$) dictates the number of required independent test paths, ensuring all paths are tested.

Next Lecture Preview

- **Topic:** Black-Box Testing
- **Focus:** Testing functionality without internal code knowledge.
- **Techniques:** Equivalence Partitioning and Boundary Value Analysis.
- **Context:** System and Acceptance testing phases.



Internal logic is hidden from the tester.

Questions?

Recap & Agenda

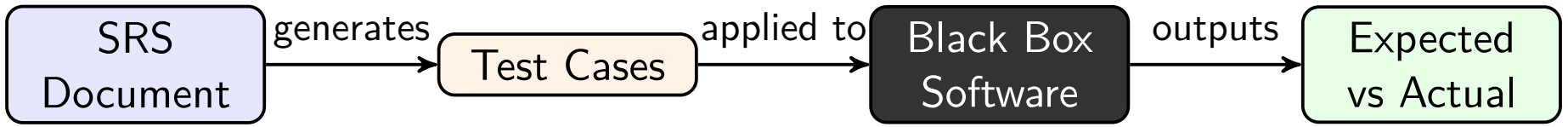
Recap:
In lecture 41, we covered Unit Testing and White-Box techniques, specifically calculating Cyclomatic Complexity to determine independent execution paths.

Agenda:

- Introduction to Black-Box Testing
- Comparison: Black-Box vs. White-Box
- Equivalence Class Partitioning (ECP)
- Boundary Value Analysis (BVA)
- Decision Table Testing
- Summary and Integration

What is Black-Box Testing?

- Also known as Behavioral or Specification-based Testing.
- Tester has no knowledge of internal code structure, control flow, or implementation details.
- Tests are derived entirely from Software Requirements Specifications (SRS).
- Focus: Does the software behave as expected given a specific input?

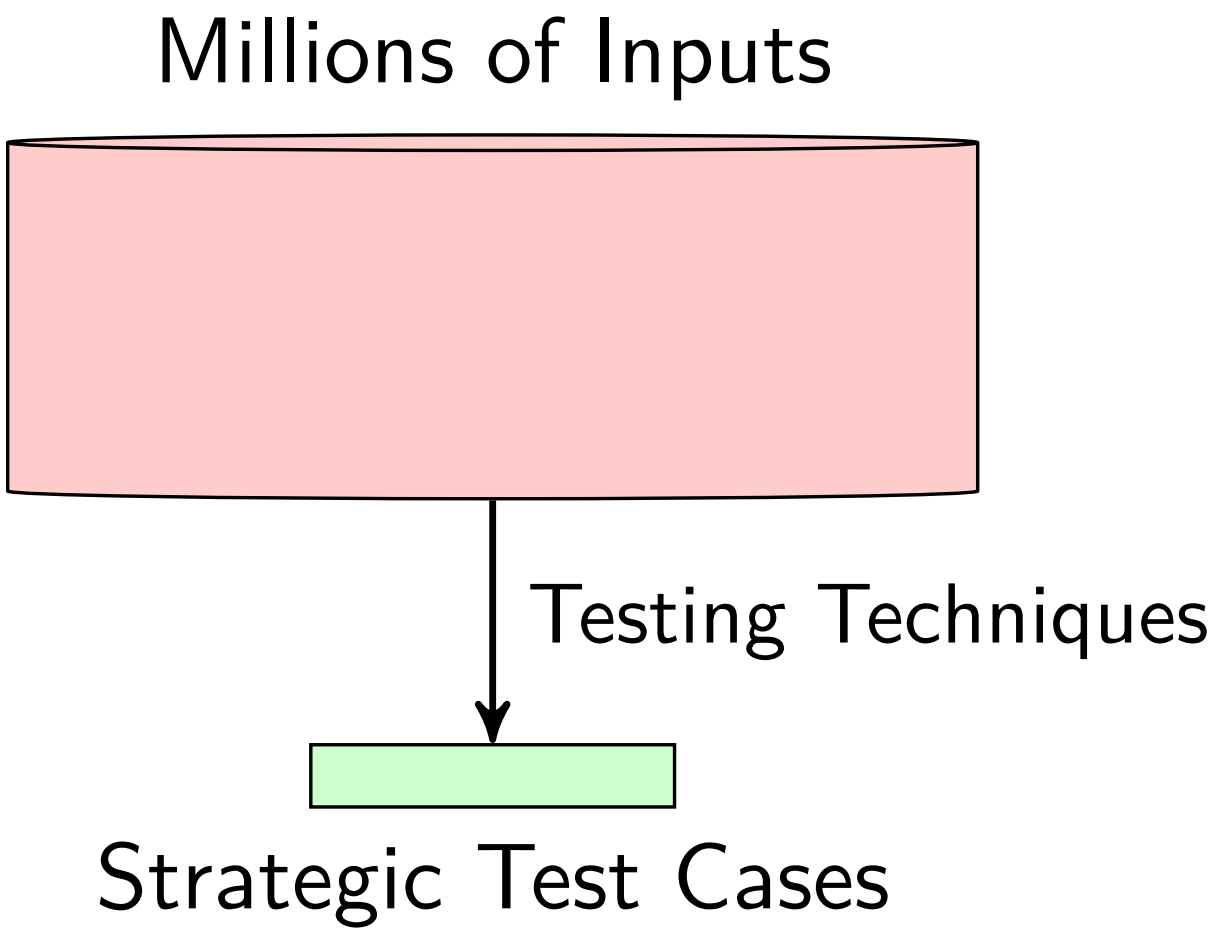


Black-Box vs. White-Box

White-Box Testing	Black-Box Testing
Tests internal logic, paths, conditions.	Tests external behavior, I/O validation.
Performed by Developers (Unit/Integration).	Performed by QA/Testers (System/Acceptance).
Misses missing requirements (unimplemented features).	Misses hidden malicious code or unexecuted dead code.
Focuses on Code Coverage.	Focuses on Requirement Coverage.

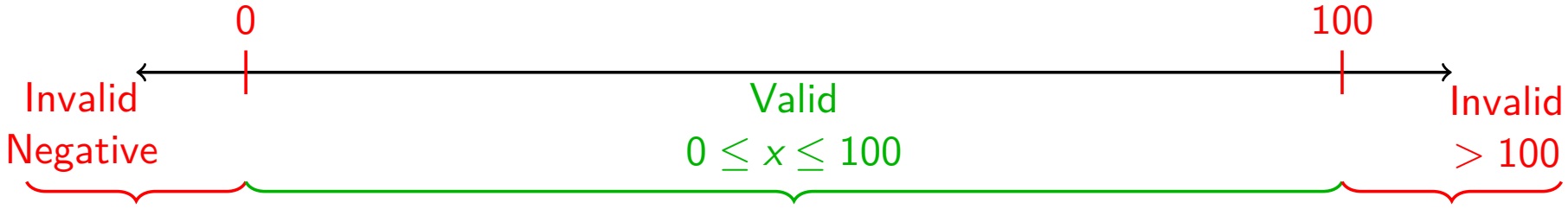
The Exhaustive Testing Problem

- Consider a text field accepting a 1-to-3 digit integer (0-999).
- Exhaustive testing requires 1,000 valid test cases, plus infinite invalid cases.
- Testing every possible input is computationally and economically impossible.
- Solution: We must mathematically select a representative subset of inputs.



Equivalence Class Partitioning (ECP)

- Divides the input domain into classes of data from which test cases can be derived.
- Premise: Testing one value in a partition is equivalent to testing any other value.
- Valid Equivalence Classes: Inputs that should be accepted.
- Invalid Equivalence Classes: Inputs that should be rejected.



ECP Example

Requirement

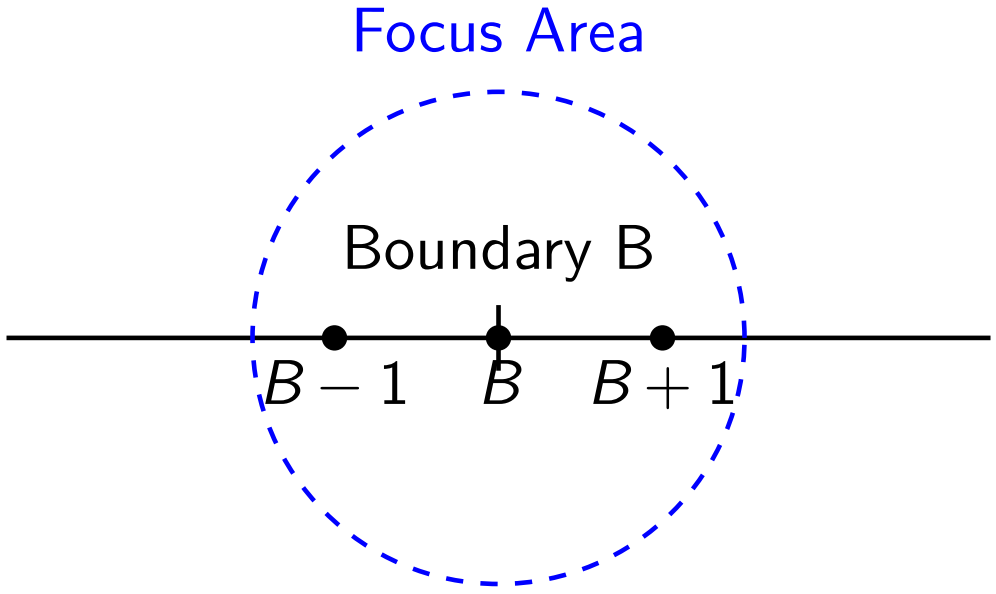
A system accepts numeric passwords between 10 and 50 (inclusive).

- **Valid Partition:** $10 \leq N \leq 50$. Select one test value (e.g., 25).
- **Invalid Partition 1:** $N < 10$. Select one test value (e.g., 5).
- **Invalid Partition 2:** $N > 50$. Select one test value (e.g., 60).
- **Result:** 3 test cases instead of infinite.

Partition	Range	Test Value	Expected Result
Invalid 1	$N < 10$	5	Reject
Valid	$10 \leq N \leq 50$	25	Accept
Invalid 2	$N > 50$	60	Reject

Boundary Value Analysis (BVA)

- A refinement of ECP based on the heuristic that errors tend to occur at the boundaries.
- Focuses on testing the edge cases of equivalence partitions.
- For a boundary B, test: B, $B-1$, and $B+1$.
- Identifies off-by-one errors (e.g., using '$<$' instead of '$<=$').

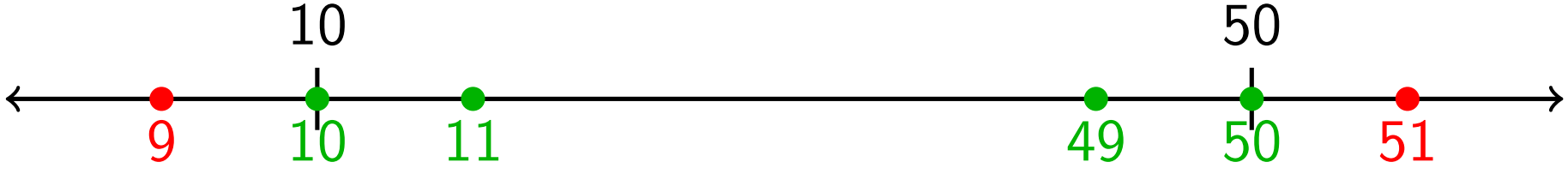


BVA Example

Requirement

System accepts values 10 to 50 (inclusive).

- **Lower Boundary (10):** Test 9 (Invalid), 10 (Valid), 11 (Valid).
- **Upper Boundary (50):** Test 49 (Valid), 50 (Valid), 51 (Invalid).
- Combining with ECP gives a highly robust test suite with 6 edge-case inputs.



Decision Table Testing

- Used when inputs have complex logical relationships or boolean dependencies.
- Also known as Cause-Effect graphing.
- Maps input conditions (Causes) against system actions (Effects).
- Prevents combinatorial explosion by identifying impossible combinations.

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Is Member?	T	T	F	F
Has Overdue Book?	T	F	T	F
Actions				
Allow Checkout	X	✓	X	✓
Deny Checkout	✓	X	✓	X

Decision Table Example: E-Commerce

- **Conditions:** C1=New Customer, C2=Order $>$ \$1000.
- **Actions:** A1=Give 10% Discount, A2=Give Free Shipping.

	Description	**Rule 1**	**Rule 2**	**Rule 3**
C1	New Customer	T	T	F
C2	Order $>$ \$1000	T	F	T
A1	Give 10% Discount	✓	✓	X
A2	Give Free Shipping	✓	X	✓

- Rule 1: T, T $\rightarrow$ A1, A2 (New customer, large order)
- Rule 2: T, F $\rightarrow$ A1 (New customer, small order)
- Rule 3: F, T $\rightarrow$ A2 (Returning customer, large order)

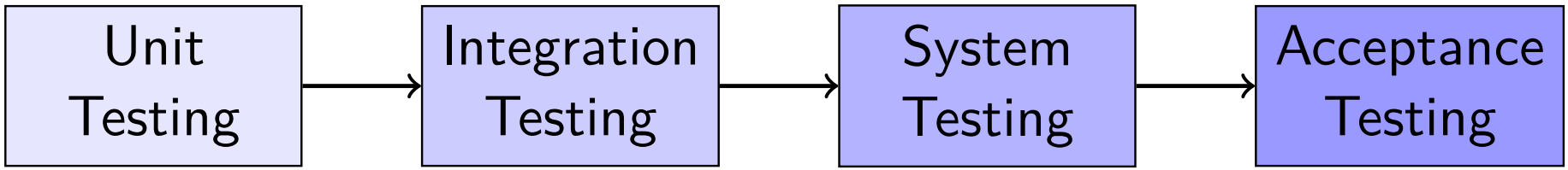
Summary

- **Black-box testing** validates software against SRS without internal code knowledge.
- Exhaustive testing is impossible; strategic sampling is required.
- **Equivalence Partitioning** reduces test cases by grouping identical behaviors.
- **Boundary Value Analysis** targets edge cases where logic errors cluster.
- **Decision Tables** handle complex, multi-variable business rules.

Next Lecture Preview

Topic: Software Testing Levels

- Integration Testing: Top-down vs. Bottom-up.
- System Testing & Regression Testing.
- Acceptance Testing: Alpha and Beta testing.



Questions?

Recap & Agenda

Recap: In the previous lecture, we covered Black Box Testing and Equivalence Class Partitioning.

Agenda:

- What is White Box Testing?
- Control Flow Graphs (CFG)
- Cyclomatic Complexity Formula
- Coverage Types (Statement, Branch, Path)
- Basis Path Testing
- Loop Testing & Mutation Testing
- Tools and Summary

What is White Box Testing?

- Also called Glass Box, Structural, or Clear Box Testing.
- Focuses on the internal structure, design, and coding of software.
- Requires programming knowledge.
- Inputs are tested against the internal execution paths.
- Helps uncover hidden errors in logic and control flows.

Control Flow Graphs (CFG)

- A CFG is a graphical representation of all paths that might be traversed through a program during its execution.
- Nodes: Represent basic blocks of sequential statements.
- Edges: Represent control flow (jumps, loops, branches).
- Predicate Nodes: Nodes containing a condition (e.g., if-else, while).

Cyclomatic Complexity - The Formula

- A software metric used to indicate the logical complexity of a program.
- **Formula 1:** $V(G) = E - N + 2$
  (E = Number of edges, N = Number of nodes)
- **Formula 2:** $V(G) = P + 1$
  (P = Number of predicate (decision) nodes)
- **Formula 3:** $V(G)$ = Number of regions in the planar graph

Statement Coverage

- Ensures that every statement in the code is executed at least once.
- Formula: $\left(\frac{\text{Number of statements executed}}{\text{Total number of statements}}\right) \times 100$
- Weakest form of coverage.
- Cannot detect hidden branches or missing else statements.

Branch Coverage

- Ensures that every branch (True/False) from a decision node is executed at least once.
- Stronger than statement coverage.
- Formula: $\left(\frac{\text{Tested decision outcomes}}{\text{Total decision outcomes}}\right) \times 100$
- Detects anomalies in IF, WHILE, SWITCH blocks.

Path Coverage

- Ensures every possible execution path through the code is tested.
- Provides the highest level of coverage.
- Number of paths = Cyclomatic Complexity $V(G)$.
- Impractical for large programs due to exponential path explosion.

Basis Path Testing Example

1. Draw the CFG from code.
2. Calculate $V(G)$. Let's say E=6, N=5. $V(G) = 6 - 5 + 2 = 3$.
3. Identify the 3 independent paths.
   - Path 1: $1 \to 2 \to 3 \to 5$
   - Path 2: $1 \to 2 \to 4 \to 5$
   - Path 3: $1 \to 2 \to 3 \to 4 \to 5$
4. Design test cases to trigger each path.

Loop Testing

- Focuses on the validity of loop constructs (while, for).
- Simple Loops: Test with 0, 1, 2, n-1, n, and n+1 iterations.
- Nested Loops: Start at innermost loop, hold others constant.
- Concatenated Loops: Treat as simple if independent.
- Unstructured Loops: Refactor code, do not test directly.

Mutation Testing

- A fault-based testing technique.
- Introduce small changes (mutations) to the source code.
- Examples: Changing '$>$' to '$<$', or '$==$' to '$!=$'.
- If a test case fails (kills the mutant), the test suite is good.
- If it passes, the test suite is weak.

Popular White Box Tools

- JUnit / TestNG: For Java unit testing.
- NUnit: For .NET frameworks.
- JaCoCo: For measuring Java code coverage.
- PyTest / Coverage.py: For Python.
- SonarQube: For static code analysis and structural review.

Advantages & Disadvantages

Pros	**Cons**
Finds hidden errors and dead code.	Highly complex and expensive.
Optimizes code structure.	Requires skilled developers, not just QA.
Maximum coverage achievable.	Tests must be rewritten if implementation changes.

Summary

- White Box Testing requires internal code knowledge.
- Cyclomatic Complexity $V(G) = E - N + 2$.
- Coverage progresses from Statement $\rightarrow$ Branch $\rightarrow$ Path.
- Basis Path Testing guarantees all paths are hit without exponential explosion.
- Complements Black Box testing for complete Quality Assurance.

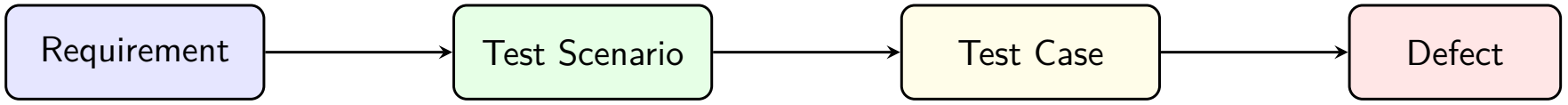
Next Lecture Preview

- Topic: Test Case Templates
- How to write industry-standard test cases
- IEEE 829 Standard
- Practical examples: Login and Checkout modules

Questions?

What is a Test Case?

- A set of conditions or variables under which a tester determines whether a system satisfies requirements.
- A documented sequence of steps, data, and expected results.
- Goal: To uncover defects and ensure requirement traceability.
- Can be Manual or Automated.



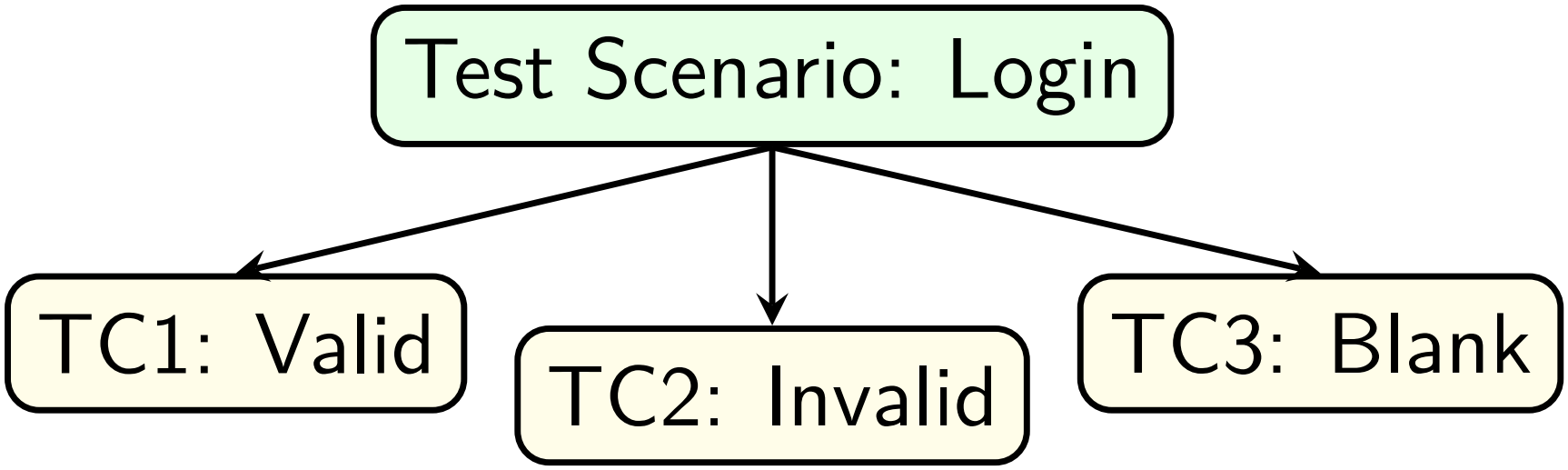
Test Scenario vs. Test Case

Test Scenario

- Broad and high-level.
- **'What'** to be tested.
- Example: Verify the login functionality.

Test Case

- Detailed and specific.
- **'How'** to be tested.
- Example: Enter valid username 'admin', valid password '123', click Login.



Standard Test Case Template (IEEE 829)

- Based on IEEE Standard for Software and System Test Documentation.
- Standardized documentation ensures consistency across QA teams.
- Divided into two main sections: **Header/Metadata** and **Execution Steps**.
- Maintained in spreadsheets (Excel) or tools (Jira/Zephyr).

TC ID	Title	Pre-condition	Steps	Expected Result	Status
TC_01	Login	Active user	Enter ID/Pass	Dashboard loads	Pass

Header Attributes

- **Test Case ID**: Unique identifier (e.g., `TC_LOG_001`).
- **Test Module**: Component being tested (e.g., Authentication).
- **Test Title/Description**: Brief summary of the test's intent.
- **Author/Date**: Who wrote it and when.
- **Priority**: High, Medium, Low (helps in regression test selection).

Pre-conditions and Test Data

- **Pre-conditions**: State the system must be in before the test starts.
  - Example: User must be registered, DB must be active.
- **Test Data**: Exact inputs to be provided during execution.
  - Example: `Username=`user1', Password=`Password@123'`
- Crucial for reproducibility.

Execution Steps and Expected Results

- **Test Steps**: Numbered, sequential actions the tester must perform.
- **Expected Result**: What the system SHOULD do based on the SRS.
- **Actual Result**: What the system ACTUALLY did (filled during execution).
- **Status**: Pass, Fail, Blocked, or Not Run.

Step #	Action	Expected Result	Actual Result
1	Click 'Submit'	Form is submitted	Error 500 displayed

Post-conditions & Defect IDs

- **Post-conditions**: State of the system after test execution.
  - Example: User session is created, log entry is written.
- **Defect ID**: If Status is FAIL, link the test case to a bug tracker ID (e.g., `BUG-404`).
- **Comments**: Tester notes or workarounds found.

Example 1: Login Module (Valid)

TC_LOG_001: Valid Login

Pre-condition: User exists on login page.
Steps:

1. Enter ‘admin@sys.com’
2. Enter ‘pass123’
3. Click Login

Expected: Dashboard loads within 2s, welcome message shown.
Status: **PASS**

Example 2: ATM Withdrawal (Invalid)

TC_ATM_042: Withdraw exceeding balance

Pre: Account balance is $100.
Steps:

1. Insert Card & PIN
2. Select Withdraw
3. Enter $500

Expected: Error 'Insufficient Funds', card ejected.
Status: **FAIL** | **Actual:** Dispensed $500. | **Defect:** BUG-911

Requirements Traceability Matrix (RTM)

- Maps Requirements to Test Cases.
- Ensures 100% test coverage of the SRS document.
- **Forward Traceability**: Req $\rightarrow$ Test Case (Are we building it right?)
- **Backward Traceability**: Test Case $\rightarrow$ Req (Are we adding scope creep?)

Requirement ID	**TC_001**	**TC_002**	**TC_003**
REQ-01 (Login)	X	X	
REQ-02 (Cart)			X

Best Practices for Writing Test Cases

- **Keep them atomic**: One test case = one specific condition.
- **Be deterministic**: Do not use words like 'sometimes' or 'might'.
- **Assume nothing**: State all prerequisites explicitly.
- **Write for others**: A new tester should be able to execute it.
- **Update frequently**: Maintain tests as requirements change.

Test Management Tools

- **Excel/Google Sheets**: Good for startups, hard to scale.
- **Jira + Zephyr/Xray**: Industry standard for Agile teams.
- **TestRail**: Dedicated robust QA management.
- **ALM / Quality Center**: Enterprise-level traceability.

Summary

- Test Cases document exact steps and expected outcomes.
- Standard attributes include Pre-conditions, Steps, and Expected Results.
- Expected $\neq$ Actual implies a Defect.
- RTM ensures alignment with SRS requirements.
- Atomic, clear, and well-maintained tests ensure high software quality.

Next Lecture Preview

- Topic: Course Recap / Final Review
- Review of SDLC, SRS, System Design
- Review of COCOMO and Testing
- Final exam preparation

Any Questions?

Recap & Agenda

Previous Lecture Recap

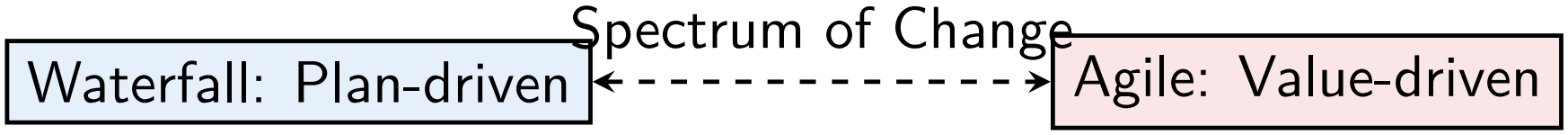
In Lecture 44, we learned how to author standardized test cases and utilize Traceability Matrices.

Today's Agenda:

- Unit 1 to 5 Rapid Tour
- Key Formulas & Diagrams Review
- Exam Strategies & Q&A

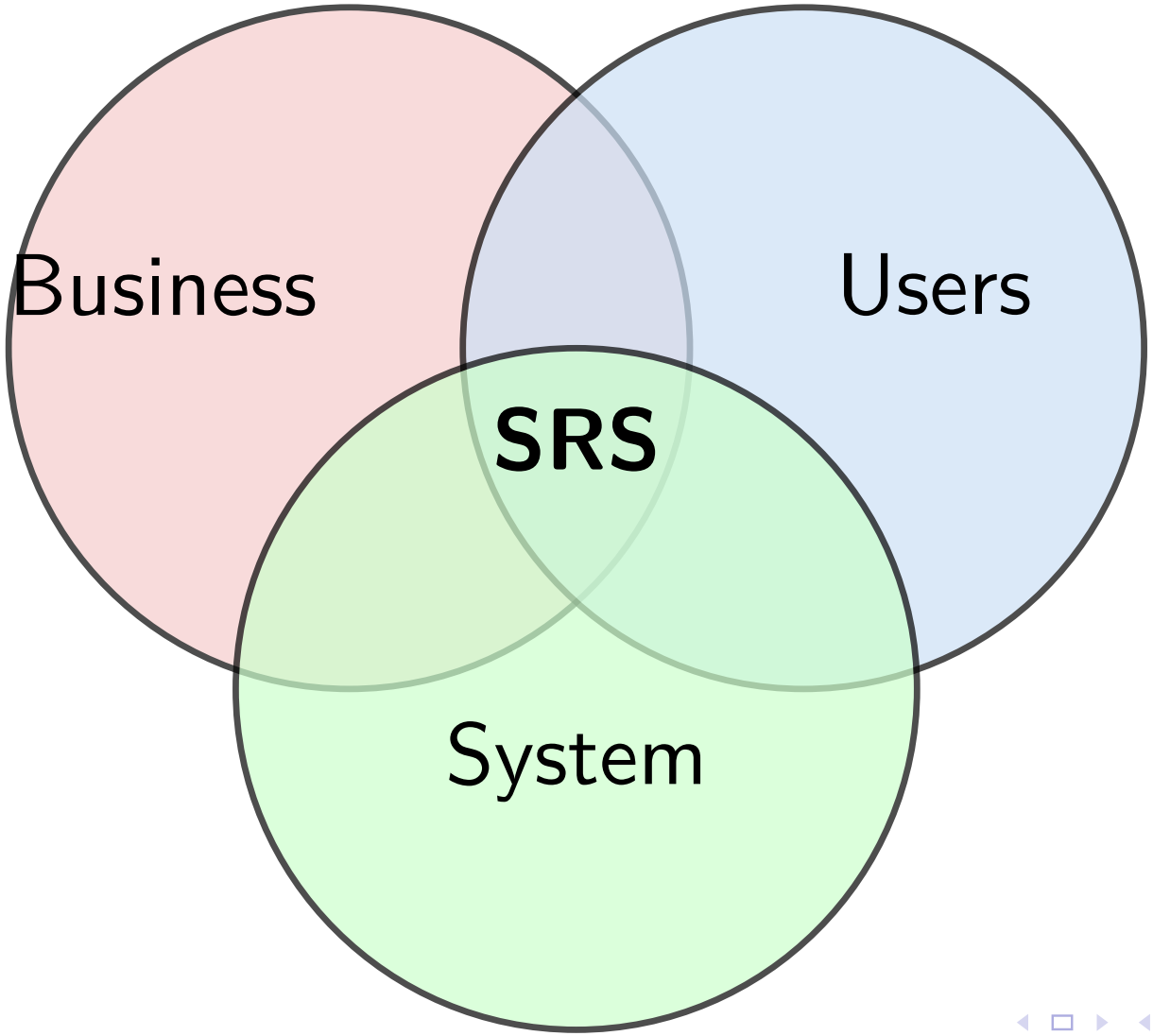
Unit 1: SE & Process Models

- **Software Engineering** is the systematic approach to development.
- **Waterfall Model:** Sequential, rigid, document-heavy.
- **Iterative & Incremental:** Builds in phases.
- **Spiral Model:** Risk-driven development.
- **Agile Methodology:** Adaptive, sprints, customer collaboration.



Unit 2: Requirements Engineering

- Requirements dictate **WHAT** the system must do.
- **Functional vs Non-Functional Requirements (NFRs).**
- **Elicitation techniques:** Interviews, surveys, prototyping.
- **IEEE 830 Standard** for Software Requirements Specification (SRS).
- The SRS is a legal contract between client and developers.



Unit 3: System Design (DFDs & UML)

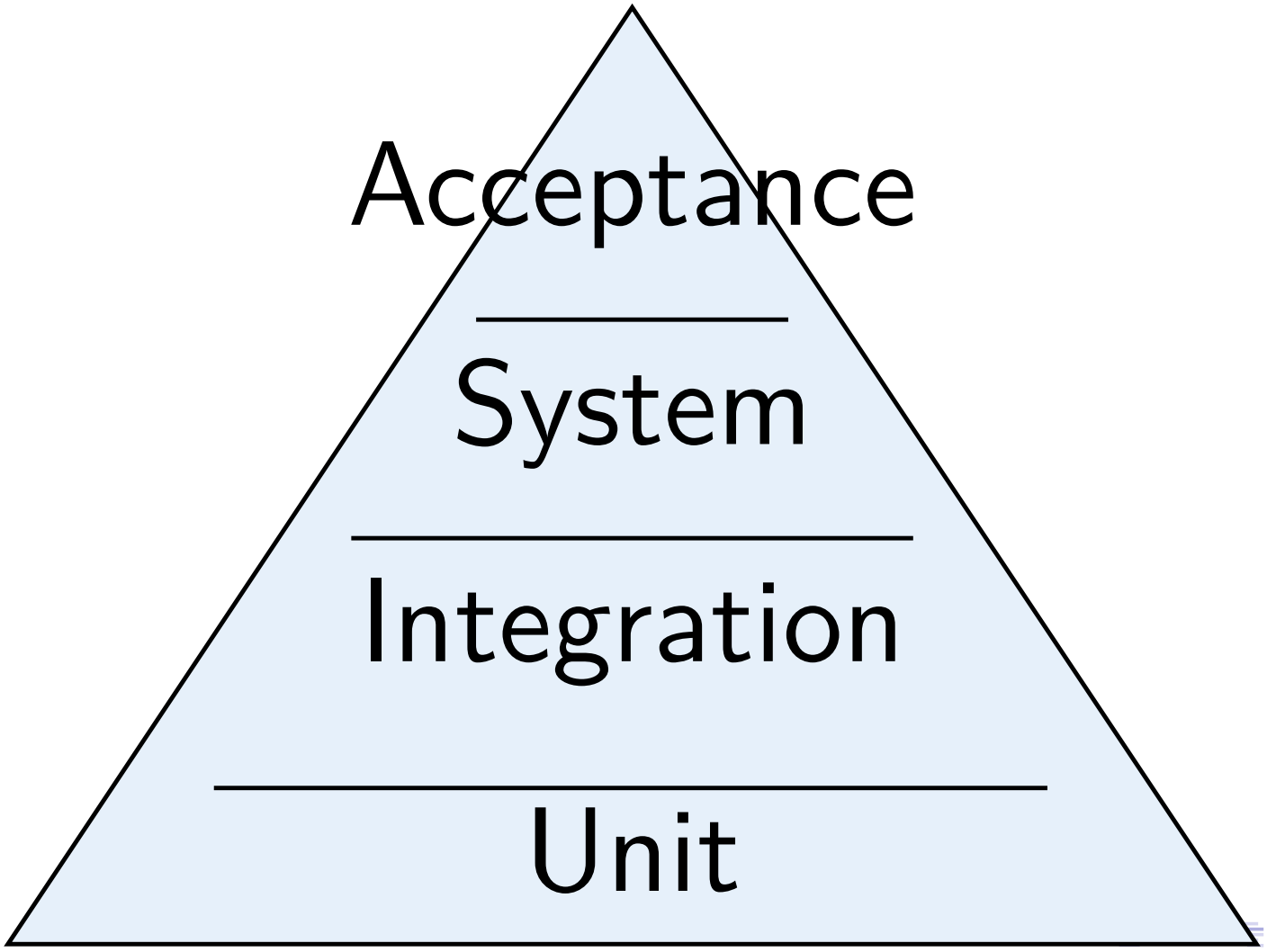
- Design dictates **HOW** the system will meet requirements.
- **DFDs:** Context (Level 0), Level 1, Level 2.
- **UML Diagrams:** Structural vs Behavioral.
- **Use Case:** Actor-System interaction.
- **Class Diagrams:** Static structure.
- **Coupling** (minimize) vs **Cohesion** (maximize).

Unit 4: Estimation & Project Management

- Software metrics: LOC vs Function Points.
- **COCOMO Model:** Organic, Semi-detached, Embedded.
- **Project Scheduling:** Gantt Charts and PERT/CPM.
- **Risk Management:** Identification, analysis, mitigation.
- **Version Control:** Git basics.

Unit 5: Coding & Testing

- **Verification** (building it right?) vs **Validation** (building right product?).
- **Black Box:** Equivalence partitioning, BVA.
- **White Box:** CFGs, Cyclomatic Complexity.
- **Levels:** Unit $\rightarrow$ Integration $\rightarrow$ System $\rightarrow$ Acceptance.
- Test Case Templates and QA models (CMMI).



Key Formulas to Remember

Important Formulas

- **Cyclomatic Complexity:** $V(G) = E - N + 2$
- **COCOMO Effort (Organic):** $E = 2.4 \times (KLOC)^{1.05}$ PM
- **COCOMO Dev Time (Organic):** $TDEV = 2.5 \times (E)^{0.38}$ Months
- **Function Point Analysis:** $FP = UFP \times CAF$
- **Statement Coverage:** (Executed/Total) $\times$ 100

Key Diagrams to Practice

- Level 0 & 1 DFD for a ‘Library Management System‘.
- UML Class Diagram for ‘E-Commerce Website‘.
- UML Use Case Diagram with ≪include≫ and ≪extend≫.
- Control Flow Graph (CFG) from a given if-else code block.
- Gantt Chart creation from task dependencies.

Career Perspectives in SE

- **Requirements:** Business Analyst, Product Owner.
- **Design:** Software Architect, UI/UX Designer.
- **Development:** Frontend, Backend, Full-stack Engineer.
- **Testing:** QA Automation Engineer, SDET.
- **Management:** Scrum Master, Project Manager, DevOps.

Summary

- SE encompasses process, design, estimation, and testing.
- Diagrams and formulas are critical for success.
- SDLC provides the foundation for tech careers.

Thank you for a great semester!

Review the slides, practice diagrams, and solve previous papers.

Best of luck on your final exams and future careers!

Any Questions?

Clarifications on the syllabus, exam formatting, or doubt resolution.