Note: ***The section/parts that are labeled by the word "Extra" represents my extra explanation and adding different examples to make the contents of the slides more clearer.***

Software Design and Architecture

[Week01 & 2 – Introduction]

Extra: Story as an introduction for Chapter1

Building software is like building a house. If the foundation is weak, cracks will appear over time, no matter how beautiful it looks.

In **software**, the foundation is **good architecture and design principles**. Without them, you'll face bugs, crashes, and hard-to-maintain code.

This course will teach you how to build strong, scalable, and maintainable software that lasts—just like a well-built house.

In the 1960s, software projects became too complex and often failed. In 1968, **software engineering** was introduced as a structured way to build reliable software, using methods similar to traditional engineering.

Lecture Outlines

- Software engineering (revised)
- Problem solving
- Software engineering design
- Why study software engineering design
- Software design challenges
- Software design process
- Software design fundamentals

Software Engineering (revised)

- **Software**

  is a collection of instructions that enable the user to interact with the computer hardwares to perform tasks.

  Extra: Example: Operating systems, applications like word processors, and web browsers.

- **Engineering**

  A systematic process" When we say something is systematic, we mean it is organized and structured" Structured" means something is arranged in a clear, orderly, and logical way, often following a specific plan or framework.", in which designers generate, and evaluate designs for devices, systems or processes whose function(s) achieve clients' objectives and users' needs while satisfying a specified set of constraints.

  Extra Example:

  Building a bridge requires careful planning to handle weight, weather, and the environment. In the same way, creating software needs a planned and organized process. This means following clear steps to make sure the software works well, is built efficiently, and meets user needs. This careful process helps reduce mistakes, handle complexity, and produce high-quality software

  Extra: User: The person who actually interacts with and uses the software

  Extra: Client: The individual or organization that pays for the development of the software.

- **Software engineering:**

  According to the IEEE, Software Engineering is:

  The application of a **systematic**, **disciplined**, and **quantifiable** approach to the **development**, **operation**, and **maintenance** of software; that is, Applying engineering principles to software.

  Extra:

  IEEE (Institute of Electrical and Electronics Engineers)- IEEE plays a significant role in setting standards, providing educational resources, and promoting professional development.

Software Engineering (revised)

- What do we mean by ***systematic***?
  - ✓ Presented or formulated as a coherent body of ideas or principles
  - ✓ Methodical in procedure or plan
- What do we mean by ***disciplined***?
  - ✓ A rule or system of rules governing conduct or activity
- What do we mean by ***quantifiable***?
  - ✓ To determine, express, or measure the quantity of deliverables

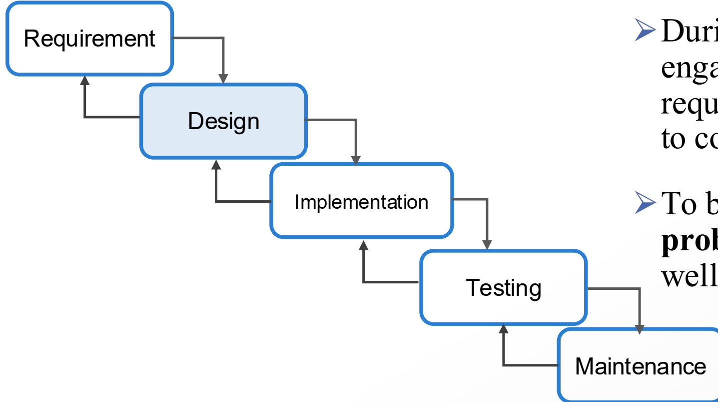
Extra: Example: Using agile methodologies to ensure iterative progress and quality control in software projects.

Software Engineering (revised)

The process of software engineering supports a ***systematic***, ***disciplined***, and ***quantifiable*** approach to software product development, and includes the following phases:

Requirements	requirements are analyzed, specified, and validated.
Design	the requirement's specification is used to create the software design which includes its architecture and detailed design.
Implementation	relies on the requirements' specification, the software architecture, the detailed design to implement the solution using a programming language
Testing	insures that the software behaves correctly and meets the specified requirements
Maintenance	Modifies software after delivery to correct faults, improve performance, or adapt it for a different environment

Software Engineering (revised)



- **Design** allows teams to organize in ***disciplined*** manner and provides a ***systematic*** approach to carefully ensure that products are built to meet their specification
- During the **design** process, engineers are constantly engaging in problem-solving activities. Their work requires them to identify, evaluate, and propose solutions to complex problems.
- To become **a good designer**, engineers must be **good problem solvers** and to become a good problem solver… well, that requires lots of time and effort

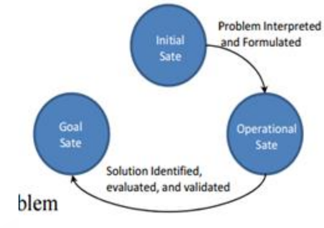
Extra: Phases of the Software Development Cycle

- **Requirements Gathering and Analysis:**
  - Identify and document the needs and expectations of stakeholders.
  - Analyze the requirements for feasibility and completeness.
  - **Example:** Conducting interviews with users to understand their needs.
- **Design:**
  - Create a blueprint for the software, including architecture and detailed design.
  - Plan how the software will be structured and how components will interact.
  - Example: Designing UML diagrams to represent the system architecture.
- **Implementation (Coding):**
  - Translate the design into executable code using a programming language.
  - **Example:** Writing code modules and integrating them into a cohesive system.

Testing:

- Verify that the software works as intended and meets the requirements.
- Perform various tests, such as unit testing, integration testing, and system testing.
- **Example:** Running automated test cases to check for bugs and ensure functionality.

- **Deployment:**
  - Release the software to users in the target environment.
  - Ensure the deployment process is smooth and the software is correctly installed.
  - **Example:** Deploying a web application to a cloud server.
- **Maintenance:**
  - Update and improve the software post-deployment to fix bugs, enhance features, or adapt to new requirements.
  - Monitor software performance and user feedback to guide maintenance activities.
  - **Example:** Releasing patches and updates to address security vulnerabilities



Engineering Problem-solving

In general, problem-solving during design occurs in **three** different states:

- **Initial State**
  - ✓ Problems are formulated and interpreted
  - ✓ In some cases, understanding the problem is a problem itself.
- **Operational State**
  - ✓ Once problem is understood, operational state begins
  - ✓ Thinking about the problem and viable solutions come to light
- **Goal State**
  - ✓ Once an appropriate solution is identified, evaluated, and validated, goal state is reached
  - ✓ Final solution found, marking the end of the problem-solving process.

Engineering Problem-solving

Try solving the following puzzle problem using the figure to the right and the following requirements:

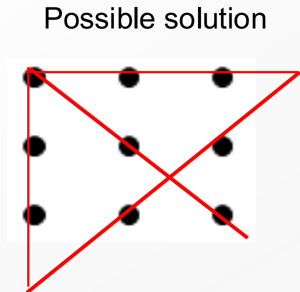
- ✓ Draw four straight lines to connect all dots.
- ✓ The pencil cannot be lifted from the paper once the line-drawing process begins.
- ✓ No lines can be retraced.
- ✓ A line must pass through every dot.

Are there any constrains??? What are they?

Engineering Problem-solving

- Most students presumed that lines should begin and end on a dot although this was not specified in the requirements from previous slide
- This behavior is an example of functional fixedness
- functional fixedness limits the ability to find solutions based on objects having a different function from their usual ones.
- Sometimes, problem-solving requires a "**think outside the box**" mentality :)
- To increase our chance in overcoming this issue, problems need to be attempted several times and considered from many different view points and unusual angles
- Overcoming functional fixedness is critical for designers attempting to provide solutions at the operational stat of problem solving.

Possible solution



Engineering Problem Solving – A holistic Approach

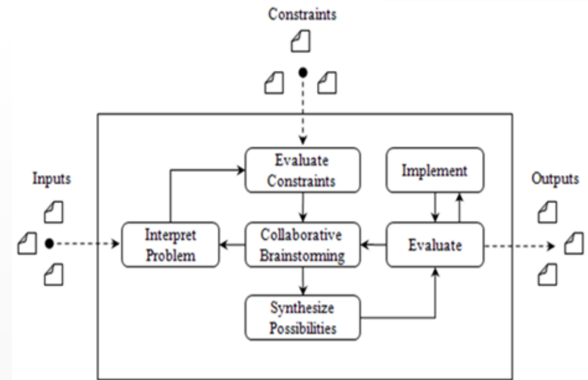
Extra: **Holistic approach looks at the whole system and how its parts interact to understand and address issues more effectively.**

- The concepts of initial, operational, and goal state of problem solving can be fused together to create a holistic problem-solving framework adequate to solving engineering problems at all stages of the development. The framework consists of the following tasks:

Initial state
- ✓ Interpreting the problem
- ✓ Evaluating the constraints

operational state
- ✓ Collaborative brainstorming
- ✓ Synthesizing "Summarizing" the possibilities:
- ✓ Evaluating the solution

Goal state
- ✓ Implementing the solution



Engineering Problem Solving – A holistic Approach

- **Interpreting the problem**
  - Performed during the Initial State of problem-solving
  - Understand and classify the problem
- **Evaluate constraints**
  - Identify external constraints" limitations" to set bounds on the solution landscape.
- **Collaborative brainstorming**
  - Performed during the Operational State of problem-solving.
  - Think about different possible solutions to the problem
- **Synthesize the possibilities**
  - Performed during the Operational State of problem-solving.
  - Try to choose one acceptable solution.
- **Evaluate solution**
  - Performed during the Operational State of problem-solving.
  - Flaws in the selected solution may trigger a transition back to the collaborative brainstorming activity.
- **Implementing solution**
  - Performed in the goal state of problem-solving
  - After finding a suitable solution, implementation can start

Software Engineering Design

- Formally, software engineering **design** is defined as:
  - ✓ (1) The **process** of identifying, and evaluating the **architectural**, and **detailed** models required to **build software** that meets its intended functional and non-functional requirements; and,
  - ✓ (2) the **product** (output or the result)of such process. (i.e. the produced design diagrams or documents)

Software Engineering Design

- In the software industry, the term design is used interchangeably to describe both the ***process*** and ***product*** resulting from such process.
  - ✓ From the ***process perspective***, it describes the phase activities and tasks required to model software's structure and behavior before construction.(This is concerned mostly with the project management's aspect of development.)

Extra: The project management's team will plan, organize, and assign tasks; ensure schedule and resources are aligned before construction.

  - ✓ From the ***product perspective***, it describes artifacts resulting from design activities, e.g., class diagram. (This is concerned mostly with the technical aspects of development.)

Extra: The Technical team will create the actual design artifacts (e.g., class diagrams, sequence diagrams) that define the software's structure and behavior.

Why Study Software Engineering Design??

Possible reasons

- 1- From the project management's perspective "process perspective", studying software design is important, mainly because:
  - Good software design help minimize effects of requirements' volatility

  Extra Example : If a new feature (e.g., multi-language support) is requested, a modular design allows for easier integration without disrupting existing components.
  - Increases efficiency in human resource allocation.

  Extra Example: Clear design documentation helps project managers allocate specific tasks to developers with relevant expertise, optimizing team productivity.
  - Shields the project from having "one guy" owning the whole system.
  - Overall, design helps improve project management: planning, organization, staffing, and tracking.

Possible reasons

2- From the “Technical Aspect”- product development’s perspective, studying software design is important, mainly, because:

- Requirements are mapped to conceptual models of software
- ***Extra:*** **Example:** Customer requirements for an online store (product listing, shopping cart, checkout process) are translated into conceptual models like user interface diagrams and flowcharts.
- Provide models that represent structure and behavior of the software system
- ***Extra:*** **Example:** Use case diagrams to show user interactions and sequence diagrams to illustrate the order of operations during a purchase.
- Main components and interfaces are identified
- ***Extra:*** **Example:** Identifying components such as the product catalog, shopping cart, payment gateway, and user authentication system.
- Provide means to evaluate quality attributes, e.g., usability, efficiency, …
- ***Extra:*** **Example:** Conducting usability testing to ensure that the checkout process is user-friendly and efficient, minimizing steps required to complete a purchase.
- Solutions can be reused in other projects
- ***Extra:*** **Example:** The payment gateway module designed for this project can be reused in other e-commerce projects or applications requiring payment processing.
- Design forms the foundation for all other development activities: Construction, Testing, Maintenance.
- ***Extra*-Example:** The database schema and API design created during the design phase guide developers during coding, help testers create relevant test cases, and assist maintainers in making updates without breaking the system.

Major Software Design Challenges

- The increasing complexity of today's software systems makes it challenging for software engineers to develop high quality software. Major design challenges include:

- **Requirements Volatility**
  - **Extra: Example:** Banking app requirements change often (e.g., add new security login).
  - **Scenario:** Initially simple banking features, later require advanced security — hard to change late in development.
- **Inconsistent Development Processes-means the teams do not follow the same clear design process**." **not uniform"**
  - **Extra : Example: Scenario:** One team uses Agile, anDifferent teams follow different methods.
  - other uses Waterfall — causes integration delays.
- **Fast, Ever-Changing Technology**
  - **Extra :Example:** Mobile OS and security updates happen frequently.
  - **Scenario:** New iOS/Android versions require quick updates to keep app working.
- **Managing Design Influences**
  - **Extra :Example:** Stakeholders have conflicting priorities.
  - **Scenario:** Executives want profit features, security team demands stronger protection.
- **Ethical and Professional Practices**
  - **Extra :Example:** Protecting sensitive customer data.
  - **Scenario:** Must ensure privacy and security even under tight deadlines.

Software Design Challenge #1 – Requirement Volatility

- Requirements volatility refers to **growth** or **changes** in requirements during a project's development lifecycle.
- Requirements changes are **costly**, particularly in the later stages of the lifecycle process, Why???
- This is because the change may require rework of the design, verification and deployment plans

Software Design Challenge #2 – Inconsistent development processes

- In the design phase, processes involve a set of activities and tasks required to bridge the gap between requirements and construction. For example:
  - ✓ Architectural and detailed designs
  - ✓ Design reviews
  - ✓ Establishing quality evaluation criteria
  - ✓ Establishing design change management and version control
  - ✓ Adopting design tools
- The problem is that in many cases, a company's design process
  - ✓ is not well established,
  - ✓ is poorly understood,
  - ✓ is focused on one form of design activity, e.g., user interface, while ignoring others, or,
  - ✓ is simply, not done at all!
  - ✓ **Extra Example :**

Imagine **building a house**:

- One team follows a **detailed plan**
- Another team works **without a plan**
- One team builds **doors first**, another builds **walls later**

➡ Result: **Nothing fits together**, work is delayed, and mistakes happen.

Software Example 💻

- Team A uses **Agile**
- Team B uses **Waterfall**
- No agreed design rules

➡ When combining their work, **systems don't match**, causing **integration delays**.

Software Design Challenge #3 – The Technology

- The **technology** for designing and implementing today' software systems continues to **evolve** to provide improved capabilities. Can you give me some examples?? **Artificial Intelligence (AI) and Machine Learning (ML):**

  Extra Example:

  Integrating AI **chatbots** into customer service platforms to provide automated, real-time responses to user queries.

  Extra

  Challenge: Developers must understand AI and ML algorithms, data preprocessing, and model training while ensuring the chatbot integrates seamlessly with existing customer databases and service protocols.

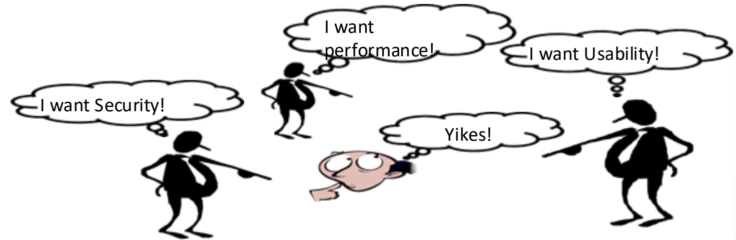
- As new technologies emerge, software designers are required to **understand** and **employ** them all at the same time!!

- This creates a demand for capable designers that can assimilate new concepts and technology quickly and effectively.

- Why this is challenging?
  - ✓ This is challenging because of the time required for both learning new technology and completing a project on-time, while making sure that the new technology interoperates well with old legacy systems.

Software Design Challenge #4 – Managing Design influences

- Software projects can have a multitude of stakeholders, each with specific wants and needs that influence the software design.
  - ✓ Some conflicting with each other!
  - ✓ Each stakeholder believes he/she is correct.
- This requires some design trade-offs to satisfy each customer.
  - ✓ Difficult to do in large-scale systems!
- This is difficult to do since design decisions need to accommodate all concerns without negatively affecting the project



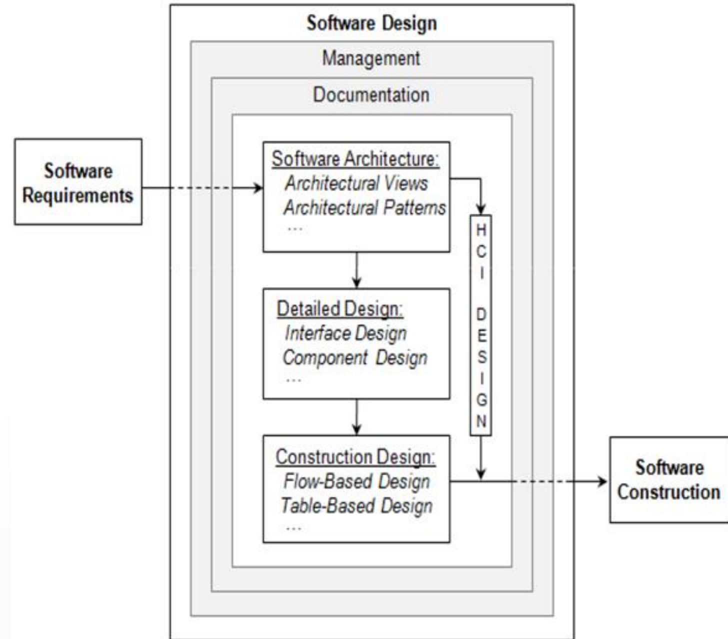
Extra Example: Stakeholders have conflicting priorities.
Scenario: Executives want profit features; security team demands stronger protection.

Software Design Challenge #5 – The Ethics

- Designers are required to consider how design decisions affect the **environment** and the **people** that uses the software.
- Designers must exhibit strong ethical and professional practices to ensure that they produce high quality systems
- This requires designers to employ strong leadership skills to:
  - ✓ Influence and negotiate with stakeholders and motivate the development team
- This is challenging under numerous pressures from different stakeholders, e.g., management, customer, peers, etc.
- An example is the ACM Code of Ethics and Professional Conduct
  - ✓ http://www.acm.org/about-acm/acm-code-of-ethics-and-professional-conduct
- **Extra: Regarding the ACM website these are the General Ethical Principles:**
  - 1.**Contribute to society and well-being, such as** Respect privacy, Be honest and fair,
  - **2. Professional Responsibilities: such as** Follow rules, Ensure security, Access resources ethically,
  - **3. Professional Leadership Principles**: such as Prioritize the public good, Enhance work quality, Apply ethical policies

Software Design Process

- Software Engineering **Design** □ process & product
- A **software design process** is a set of activities that specify how resources work together for the production of software design **products** (e.g., use cases, class diagrams, and other Unified Modeling Language (UML) models, and design documents).

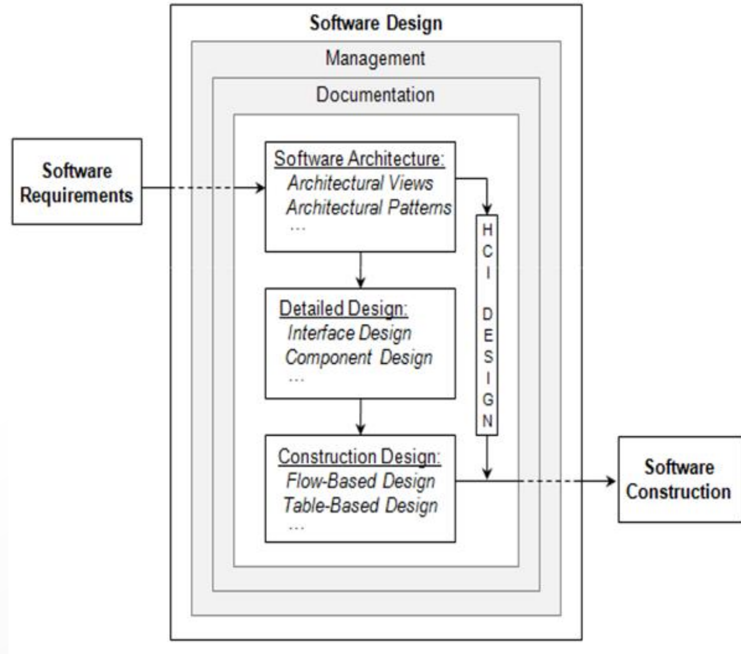


Software Design Process

➢ Software design process include major and supportive activities:

Major activities
1. **Software architecture**
2. **Detailed design**

Supporting activities
1. Construction design
2. Human-computer interface design
3. Software design documentation
4. Management activities



Software Design Process

- **Software Architecture**
  - **Corresponds** to **a macro-design approach** for creating models that **show** the quality and function of the software system
  - Provides **black-box models** used to **evaluate** the system's projected quality and functions
  - Designed using **multiple perspectives**, therefore, **allows** different stakeholders with different backgrounds to evaluate the design to ensure that it addresses their concerns
  - **Provides** the major structural components and interfaces of the system
  - **Focuses** on the quality aspects of the system before detailed design or construction can begin
  - **Serves as** an important communication, reasoning, and analysis tool that supports the development and growth of the system
  - **Lays and puts** the foundation for all subsequent work in the software engineering life-cycle

Software Design Process

- **Detailed Design**
  - ✓ **Focuses** on the internals of the systems components and interfaces.
  - ✓ **Begins** after the software architecture activity is specified, reviewed, and considered *sufficiently* complete.
  - ✓ Builds on the software architecture to **provide** a *white-box* approach to design the structure and behavior of the system.
    - – Extra: Detailed design provides a more in-depth view, describing the internal structure and behavior of the system's components.
  - ✓ **Refines** the architecture to reach a point where the software design, including architecture and detailed design, is considered sufficiently complete for the construction phase to begin.
  - ✓ **Focuses** on functional requirements, whereas the architecture focuses mostly on non-functional, or quality, requirements.

Extra:

Aspect	Software Architecture	Detailed Design
Provide	Big picture of the system: major components, how they connect, and key quality attributes (security, performance, scalability).	Internal details of each component: algorithms, data structures, workflows.
Viewpoint	**Black-box** view — what each component does and how it interacts with others.	**White-box** view — how each component is built internally.
Timing	Comes **before** detailed design or coding; foundation for the project.	Starts **after** architecture is reviewed and approved.
Focus	**Mostly Non-functional requirements** (quality attributes like security, performance, maintainability).	**Functional requirements** (exact features, logic, and data handling).
Purpose	Communication tool for all stakeholders; ensures system meets quality goals.	Guides developers on how to implement each part.
Example – Online Banking App	Defines **UI**, **Login System**, **Transaction Service**, and **Database**, and how they interact securely and efficiently.	Describes **how** the Login System checks credentials (e.g., encryption method, API calls, error handling), **how** transactions are validated, **how** database queries are structured.

Software Design Process

- Two important tasks of the **detailed design** activity include:
  - ***Component design***
    - ✓ During **architecture, major components are identified**. But during **component design,** the internal design of (the structure and behavior of) these components is created.
    - ✓ In object-oriented systems, using UML, component designs are typically in the form of **class** diagrams, **sequence** diagrams, etc.
  - ***Interface design***
    - ✓ Refers to the design activity that deals with specification of interfaces between components in the design.
- Can be focused on specifying the interfaces used internally with software components or externally across software components.

Extra:

- ✓ **Internal Interface**
  Between two parts *inside* your system
  👉 Example: How the **Login System** sends a signal to the **Transaction Service** saying "User is authenticated — allow transactions."
- ✓ **External Interface**
  Between your system and something *outside*
  👉 Example: How your banking app connects to a **third-party payment gateway** to process credit card payments.

Extra Example

Design Type	Analogy (Wedding Dress)	In Software
1- Component Design	Designing each part in detail — bodice, skirt, sleeves, etc.	Designing internal details of each component (classes, attributes, methods).
2-Interface Design	Making sure all parts connect and fit smoothly to form a full dress	Designing how components interact (method calls, data flow between modules).

Software Design Process

- Other important activities in software include:
  - ✓ **Design documentation** (why?)

Extra:

•**Records all design decisions** for future reference.
•Ensures **consistency** and **traceability** throughout the project.
•Helps **developers, testers, and maintainers** understand the system.

  - ✓ **Management activities (why?)**

Extra:

•Helps in **team coordination** and managing **design changes** effectively.
•**Plan, track, and control** the design process.
•Ensure the project stays **on time, within scope, and on budget**.

Continue…

- **Construction design** is the lowest level of detailed design that addresses the modeling and specification of function implementations.
- deals mostly with the analysis and design of algorithms.

Extra **Construction design** is :

- ✓ Focuses on **how functions are coded**.
- ✓ Designs and analysis **algorithms** for efficiency.
- ✓ It's the **deepest level of design** — turns logic into real code.

- Example: After designing a "search" feature, construction design figures out whether to use a **binary search** or a **linear search** — based on speed and performance.

- **Another important design activity is the <u>Human-computer interface design.</u>**
- Design activity where general principles are applied to optimize the interface between humans and computers.
- Extra - Human-computer interface design :
  - ✓ This design activity deals with the **interaction between users and the software**.
  - ✓ Focuses on making the **user interface (UI)** easy to use, friendly, and efficient.
  - ✓ Example: Designing a **login screen** that clearly shows errors, uses proper button sizes, and is accessible to all users.

Software Design Documentation

Similar to the specification activity of the requirements phase, software design documentation, also known as software design description (SDD), plays a big role in professional, large-scale, or software-intensive systems. Its importance is specified by the IEEE (1998, p. iii) as follows:

SDDs play a pivotal role in the development and maintenance of software systems. During its lifetime, a given design description is used by project managers, quality assurance staff, configuration managers, software designers, programmers, testers, and maintainers. Each of these users has unique needs, both in terms of required design information and optimal organization of that information. Hence, a design description must contain all the design information needed by those users.

SDD should include the necessary information that properly captures the design of the system. As part of this activity, other issues such as tools for generating design documents, validation, and configuration management must be addressed. The software design documentation activity typically begins at the design phase and continues throughout the lifetime of the software system.

Extra:

- **Software Design Documentation – Summary**

- **Timeline**: Starts in design phase and continues through the lifecycle.
- **Purpose**: Essential for large and complex software systems.
- **Role**: Supports development and maintenance.
- **Users**: Used by managers, designers, developers, testers, and maintainers.
- **Content**: Must include all required design details.
- **Activities**: Covers documenting, validating, and managing design.

Software Design Management

Management plays a big role in software engineering projects. Griffin (2010, p. 5) defines management as

A set of activities (including planning and decision making, organizing, leading, and controlling) directed at an organization's resources (human, financial, physical, and information), with the aim of achieving organizational goals in an efficient and effective manner.

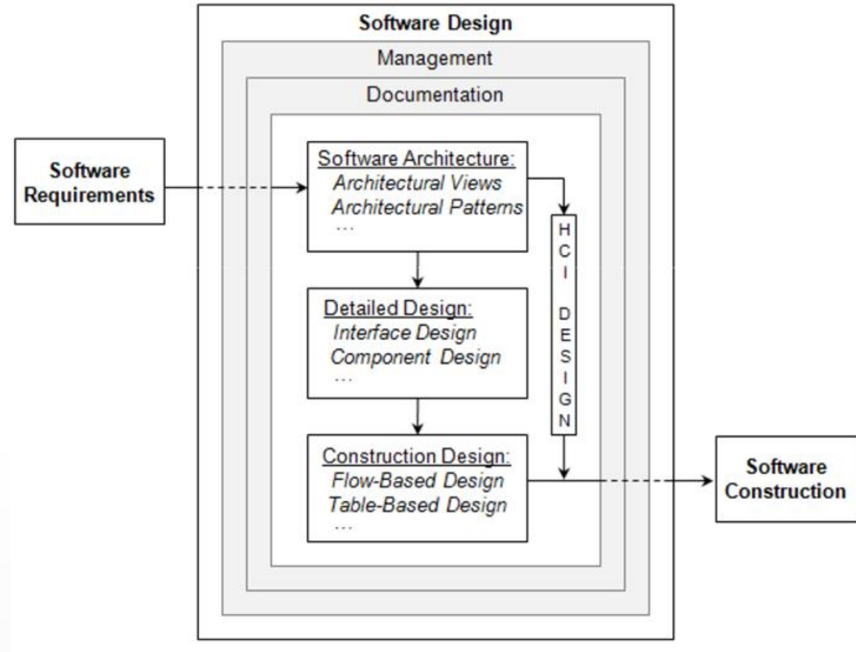
In the design phase, management refers to the set of activities required to efficiently create and implement quality design artifacts, within schedule and budget constraints. This definition encompasses a broad set of activities that are particular to specific organizations. However, at the core of every organization's management activities, quality is a focal point. The quality of software designs can be assessed in various ways. From the management's perspective, quality of software designs can be evaluated in terms of cost and scheduling. From the engineering point of view, quality in designs can be evaluated using a set of well-known design principles as well as modeling and evaluating the quality attributes that the software must exhibit, which are specified via nonfunctional, quality requirements. From the configuration management's perspective, design quality can be achieved through change management processes that control how designs are created, modified, and improved. In large-scale software projects, software design management is essential to plan, organize, staff, track, and lead the activities required to carry out successfully the software architecture and detailed design steps.

Extra: Here's a concise summary of the text provided in the image on "Software Design Management":

- **Role of Management**: Plans and controls resources to meet goals.
- **Design Phase Management**: Delivers quality designs on time and within budget.
- **Quality Focus**: Ensures design meets cost, schedule, and design standards.
- **Evaluation**: Assesses design quality from both management and engineering views.
- **Configuration Management**: Manages design changes and improvements.
- **Large-Scale Projects**: Critical for organizing and leading design activities effectively.

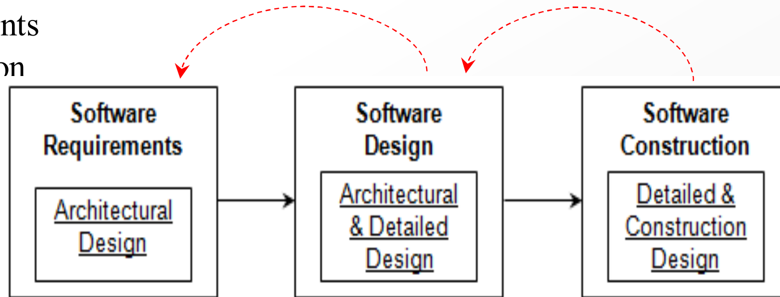
Software Design Process

- The software design phase has several activities, each having one or more tasks.
- **We will spend the rest of the semester** learning how to carry out major activities, how to document these efforts.



Software Design Process

- In a perfect world, all design work would take place during the design phase. In practice, the distribution of design activities varies throughout the software development life-cycle.
  - In most projects, most of the design efforts occur during the design phase, but some design inevitably "definitely" occurs in other life-cycle phases, e.g.,
    - Requirements
    - Construction



Role of Software Designers

- **Systems Engineer**
  - Designs the overall development process of systems as a whole including processes for development of both the software and hardware that are part of the system
- **Software Architect**
  - Design software systems using mostly a black-box modeling approach; concern is placed on the external properties of software components that determine the system's quality and support the further design of functional requirements.
- **Component Designer**
  - Focuses on designing the internal structure and behavior of software components identified during the architecture phase; typically, these designers have strong programming skills, since they implement their designs in code.
- **User Interface Designer**
  - Design the software's user interface; skilled in determining ways that increase the usability of the system

- **Extra:**
- **Systems Engineer:** Responsible for designing the **entire system**, including **software and hardware processes**.
- **Software Architect:** Designs the **high-level structure** of the software, focusing on **external behavior and quality attributes**.
- **Component Designer:** Designs and implements the **internal structure and behavior of software components**.
- **User Interface Designer:** Designs the **user interface** to improve **usability and user experience**.

أكثر شي يجي عليه اسألة

Next … Software Design Fundamentals

- **Design Principles**
  - Modularization
  - Abstraction
  - Encapsulation
  - Cohesion and coupling
  - Separation of interface and implementation
  - Sufficiency and completeness
- **Design Strategies**
  - Object-oriented vs. structured design
- **Design Considerations**
  - Minimize complexity and design for change

Software Design Fundamentals (Principles)

- Design Principle #1: ***Modularization***

  - ✓ It is the principle that drives the continuous **decomposition** of the software system until fine-grained components are created. Extra: That refers to the process of dividing a software system into separate components, each of which handles a specific part of the system's functionality. These modules can be developed, tested, and maintained independently of each other.”

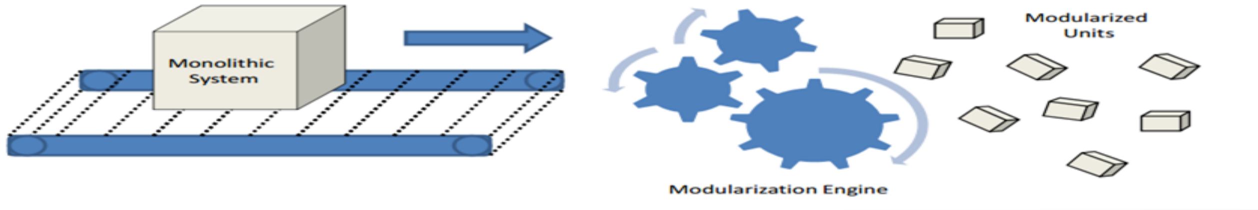
  - ✓ One of the most important design principle, since it allows software systems to be **manageable** at all phases of the development life-cycle.

  - ✓ When you modularize a design, you are also modularizing requirements, programming, test cases, etc. Extra: --Each module has its **own requirements** (what it should do).--Each module is **programmed separately**.--Each module has its **own test cases** to check if it works. So, by splitting the design into modules, you naturally split **everything else** — requirements, code, and tests — making the project **easier to build, test, and maintain**

  - ✓ Leads to designs that are easy to **understand**, resulting in systems that are easier to **develop, maintain** and **re-use**.

Software Design Fundamentals (Principles)

- Conceptual view of modularization:



Modularization is the process of continuous decomposition of the software system until fine-grained components are created.

Extra: **Fine-grained components** means the system's parts are made **small and specific** in what they do.
-Each component has a **narrow, focused responsibility**.
-Easier to **understand, reuse, and test**.
--**Example**: Instead of one big "Payment Processor" module, you have smaller ones like **Card Validator**, **Payment Authorizer**, and **Receipt Generator**.
In short — **fine-grained** = small, detailed, and specialized components.

Software Design Fundamentals (Principles)

- **Advantages of modularization:**
  - ✓ Keep the complexity of a large program manageable
  - ✓ Isolate errors
  - ✓ Eliminate redundancies
  - ✓ Encourage reuse (write libraries)
  - ✓ A modular program is
    - Easier to write / Easier to read / Easier to modify

Continue

- **But how do we justify the "modularization engine"????**
  - ✓ It turns out that two other principles can effectively guide designers during this process
    - **Abstraction** → simpler
    - **Encapsulation**

Extra: When we modularize a system (break it into smaller parts), we need a way to justify and guide how we split it.

•Abstraction is one of the main principles that helps guide this — it tells us to focus on the essential features of each module and hide the unnecessary details.

•By applying abstraction, modules become simpler, cleaner, and easier to understand while still doing their job.

In short: Abstraction justifies modularization because it ensures modules are meaningful, focused, and not cluttered with extra details.

Software Design Fundamentals (Principles)

- Design Principle #2: ***Abstraction***
  - ✓ Abstraction is the principle that focuses on essential characteristics of entities—in their active context—while deferring unnecessary details.
  - ✓ While the principle of modularization specifies what needs to be done, the principle of abstraction provides guidance as to how modularization should be done. Modularizing systems in ad-hoc manner leads to designs that are incoherent, hard to understand, and hard to maintain.
  - ✓ Abstraction can be employed to extract essential characteristics of:
    - **Data**
    - **Functions or behavior (procedures)**

Extra: Focuses on **essential features** and hides unnecessary details.

- Guides **how** to modularize a system (while modularization says **what** to separate).
- Prevents incoherent, hard-to-understand, and hard-to-maintain designs.
- Can be applied to:
- **Data** (show only needed attributes).
- **Functions/Behavior** (show only essential operations).

Example: A *Payment Gateway* hides payment processing details; the *Shopping Cart* just uses a simple interface to request payment.

Software Design Fundamentals (Principles)

- ***Functional (Procedural) abstraction***
  - Specific type of abstraction that simplifies reasoning about behavioral operations containing a sequence of steps.
  - We use this all the time, e.g., consider the statement ***"Computer 1 SENDS a message to server computer 2"***
    - Imagine if we had to say, e.g., "*Computer 1 retrieves the server's information, opens a TCP/IP connection, sends the message, waits for response, and closes the connection.*" Luckily, the procedural abstraction ***SEND*** helps simplify the operations so that we can reason about this operations more efficiently.
- ***Data abstraction***
  - Specific type of abstraction that simplifies reasoning about structural composition of data objects.
    - In the previous example, ***MESSAGE*** is an example of the data abstraction; the details of a ***MESSAGE*** can be deferred to later stages of the design phase.

Software Design Fundamentals (Principles)

- **Some advantages of abstraction**
  - Hides details (easier to use)
  - Allows us to think about the general framework (overall solution) & postpone details for later
  - Can easily replace implementations (details) by better ones

Software Design Fundamentals (Principles)

- Design Principle #3: ***Encapsulation***
  - ✓ Principle that deals with providing access to the services of *abstracted entities* by exposing only the information that is essential to carry out such services while hiding details of how the services are carried out.
  - ✓ When applied to data, encapsulation provides access only to the necessary data of abstracted entities, no more, no less.
  - ✓ Encapsulation and abstraction go **hand in hand.**
    - When we do abstraction, we hide details…
    - When we do encapsulation, we revise our abstractions to enforce that abstracted entities only expose essential information, no more, no less.
    - Encapsulation forces us to create good abstractions!

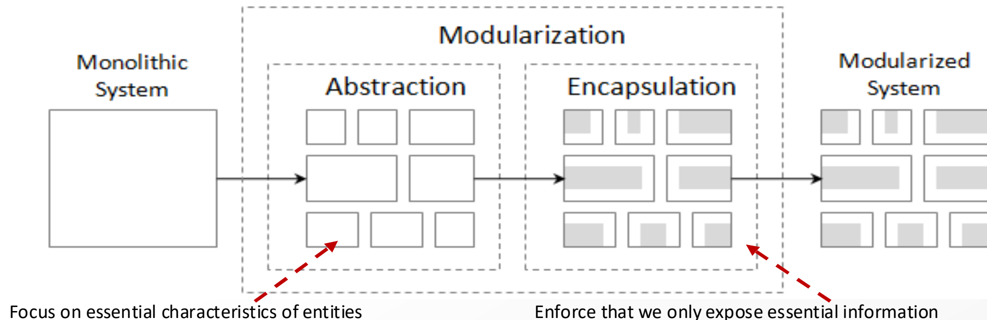
- **Extra**:
- **Hides internal details** of a module or object.
- **Exposes only** what is needed through a **well-defined interface**.
- Works **with abstraction**:
- Abstraction hides *what's not needed*.
- Encapsulation enforces *only essential info is accessible*.
- **Example**: User data (like passwords) is hidden; access is only through methods like login() or register().
- **In OOP**: Done by making attributes private and using public getter/setter methods for controlled access.

Extra :Abstraction and Encapsulation

Aspect	Abstraction	Encapsulation
Purpose	Hides unnecessary details and shows only essential features.	Hides internal implementation and restricts direct access to data.
Focus	*What* a module/class does.	*How* data and methods are accessed and protected.
Implementation	Achieved using abstract classes, interfaces, or simplified models.	Achieved using access modifiers (private, public, protected) and getter/setter methods.
Example	A payment module shows only the "processPayment" service, not its internal steps.	User password stored privately; can only be accessed via login() or getPassword() with controls.
Relationship	Defines the essential features and characteristics.	Enforces that only essential information are exposed.

Software Design Fundamentals (Principles)

- The principles of **modularization**, **abstraction**, and **encapsulation** can be summarized below.



Extra:

1 Modularization

– Breaking into parts: You take a big system and split it into smaller, separate modules. **The goal**: Make the system easier to build, test, and maintain.

Example: Breaking an e-commerce system into modules like **Login, Product Catalog, Cart, and Payment**.

2 Abstraction

– Hiding unnecessary details, focusing on what's important. Once you have modules, you decide what each module should focus on — the essential features only. You don't show all the internal steps, just what's needed to use it.

Example: For a Payment module, you show "processPayment()" but hide the 10 steps it takes inside to handle the payment.

3 Encapsulation

– Protecting and controlling access After abstraction, you protect the hidden parts so other modules cannot access them directly.

You allow interaction only through controlled interfaces or methods.

Example: In the Payment module, the credit card details are private — other modules can't touch them directly, they can only send data through methods like "pay()".

Easy way to remember:

- Modularization = Split the house into rooms.
- Abstraction = Name each room by purpose (Bedroom, Kitchen) without showing every detail inside.
- Encapsulation = Lock the rooms and give keys only for specific doors you're allowed to open. → Privet , Public] → lock

Set , get] → How to edit or Use = Master key

Exam Question

Exercise

- Exercise
- Assume we have an online student registration system. Apply the modularization principle to decompose the system into possible fine grid of modules using
- What are the possible modules that you suggest?
- Discuss the issue of abstraction in relation with modularization for the modules you suggested??
  (i.e. data abstraction, behavior abstraction)
- Discuss encapsulation options for one of the suggested modules

Extra:

Exam Solution

1 Modularization – Breaking into Fine-Grained Modules

For an online student registration system, possible modules:

1. User Management Module– handles login, registration, profile updates.
2. Course Catalog Module– displays available courses and details.
3. Registration Module – enrolls students in courses.
4. Payment Module – processes course fees.
5. Schedule & Timetable Module – generates and displays student schedules.
6. Notifications Module– sends email/SMS updates.
7. Reports & Records Module – generates transcripts, registration history.

2 Abstraction in Relation to Modularization

When modularizing, abstraction decides what each module should expose and what to hide.

Data Abstraction – Show only necessary data, hide sensitive/internal details.
  Example: In Payment Module, display payment status (“Paid/Unpaid”) but hide bank transaction IDs.

Behavior Abstraction– Show only the main operations, hide complex implementation steps.
  Example: In Registration Module, offer a method `enrollStudent(courseID)` but hide how eligibility checks and database updates happen internally.

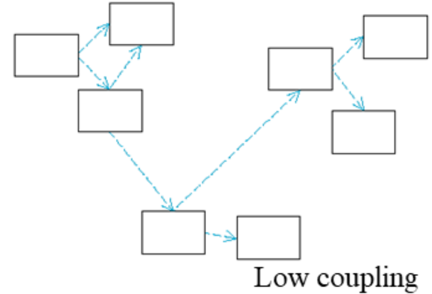
3 Encapsulation Options (Example: Payment Module)

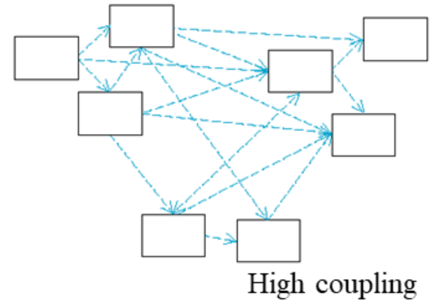
- Make sensitive data private(e.g., card numbers).
- Provide public methods like `processPayment()` and `checkPaymentStatus()` for controlled access.
- Prevent direct access to internal payment processing logic; other modules interact only through the defined API/interface.

How to write a code
→ She won't ask to write code
→ she will give the code and ask wich Type of coupling this code showing

Software Design Fundamentals (Principles)

- Design Principle #4: **Coupling**
  - Refers to the manner and degree of interdependence between software modules.
  - Measurement of dependency between units. The higher the coupling, the higher the dependency and vice versa.
  - We should aim for **lower coupling**.

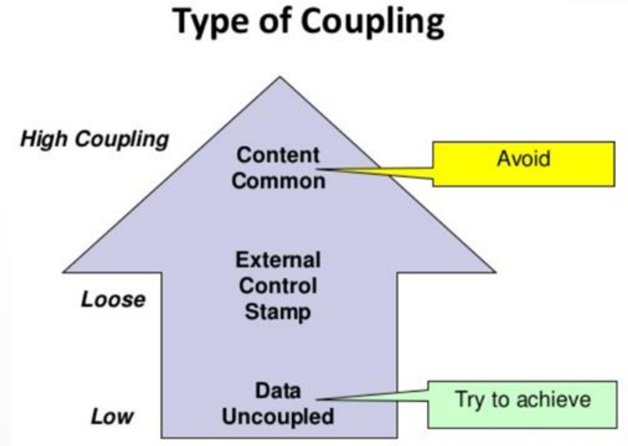




Software Design Fundamentals (Principles)

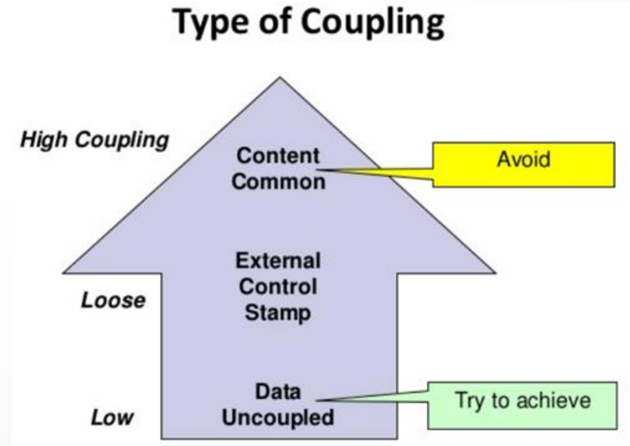
ST → St ID   Course → St ID

- Major types of coupling include:
  - ✓ **Content coupling :** the most severe and strict type, since it refers to modules that modify and rely on internal information of other modules.
  - ✓ **Common coupling:** refers to dependencies based on common access areas, e.g., global variables. When this occurs, changes to the global area causes changes in all dependent modules.



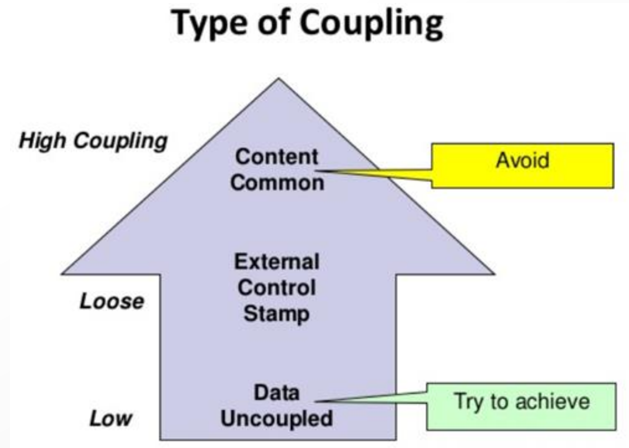
Software Design Fundamentals (Principles)

- Major types of coupling include:
  - **External coupling:** occurs when two modules share an externally imposed data format, communication protocol, or device interface.
  - **Control coupling:** is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).



Software Design Fundamentals (Principles)

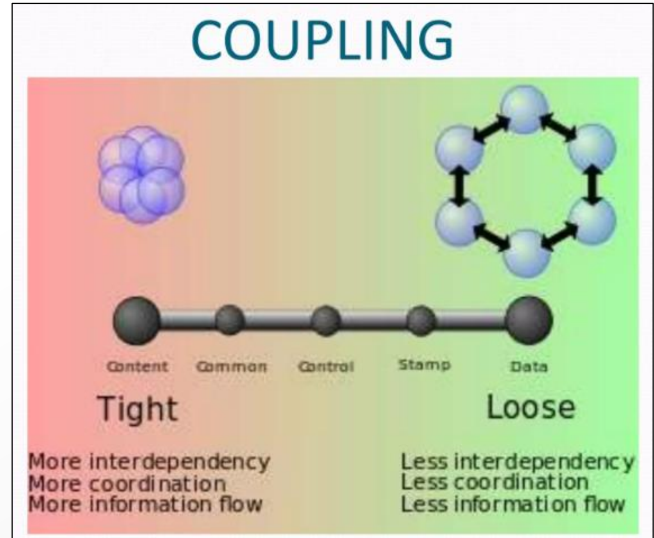
- Major types of coupling include:
  - ✓ **Stamp coupling:** occurs when modules share a composite data structure and use only parts of it, possibly different parts (e.g., passing a whole record to a function that only needs one field of it).
  - ✓ **Data coupling** occurs when modules share data through, for example, parameters.
  - ✓ **No coupling:** modules do not communicate at all with one another.



Software Design Fundamentals (Principles)

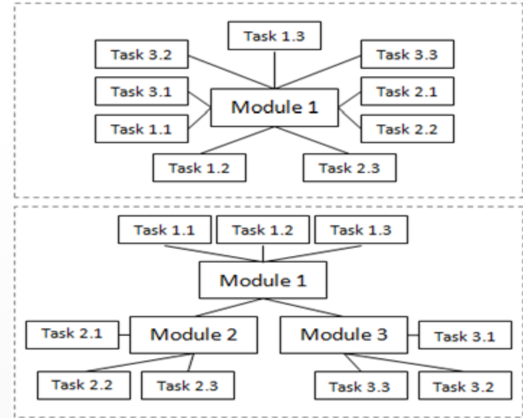
Extra Figure

- A high degree of coupling gives rise to negative side effects:
  - ✓ Quality, in terms of reusability and maintainability, decrease.
  - ✓ When coupling increase, so does complexity of managing and maintaining design units.



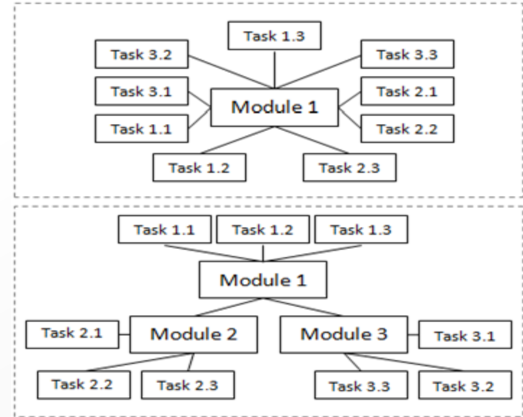
Software Design Fundamentals (Principles)

- Design Principle #5: ***Cohesion***
  - ✓ The manner and degree to which the tasks performed by a single software module are related to one another.
  - ✓ Measures how well design units are put together for achieving a particular tasks.
- **High cohesion good**, **low cohesion bad…**



Software Design Fundamentals (Principles)

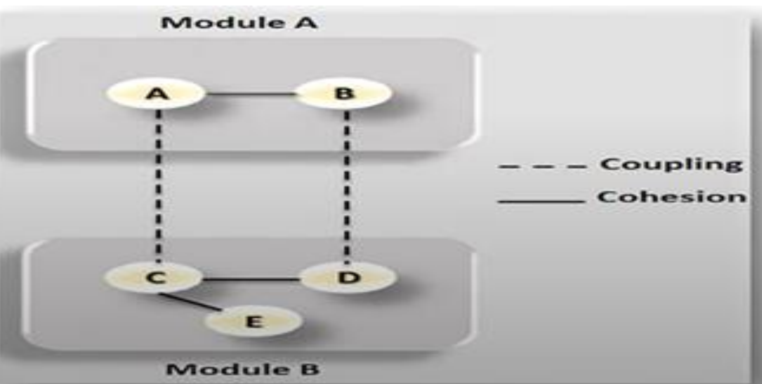
- Cohesion can be classified as:
  - ✓ **Functional cohesion** – all subunits contribute to perform a single function
  - ✓ **Procedural (or sequential) cohesion** – tasks work in steps to achieve the unit's purpose
  - ✓ **Temporal cohesion** – all tasks in a design unit are performed at a specific time
  - ✓ **Communication cohesion** – the unit's tasks produce or consume the same data



Extra:

- 1 **Functional**
  - Cohesion Definition: All parts work together to do one single task.
  - Example: A calculateTotal() method that sums prices of items in a cart.
- 2 **Procedural** (Sequential)
  - Cohesion Definition: Tasks are done in a specific sequence, where output of one step is input to the next.
  - Example: A processOrder() method that:Validates order →Calculates cost →Processes payment.
- 3 **Temporal**
  - Cohesion Definition: Tasks happen at the same time or in the same phase.
  - Example: An initializeSystem() method that sets up database connections, loads configuration, and logs startup time — all at program start.
- 4 **Communication**
  - Cohesion Definition: Tasks work on the same data.
  - Example: A generateStudentReport() method that calculates GPA, lists enrolled courses, and prints grades — all using the same student record.

Extra:



☑ **Good Design (Low coupling + High cohesion)**

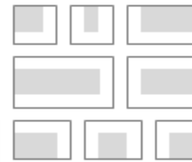
- **ShoppingCart class**:
  - Methods: addItem(), removeItem(), calculateTotal().
  - All methods are related to *cart management* → **high cohesion**.
- The cart doesn't talk directly to Database or PaymentService, but only through an **interface** → **low coupling**.
  - If tomorrow you change from SQLite to Firebase, ShoppingCart doesn't break.

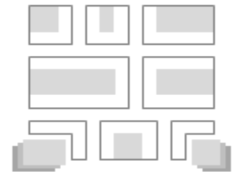
✕ **Bad Design (High coupling + Low cohesion)**

- **Utility class** called CartManager:
  - Methods: addItemToCart(), sendEmailToUser(), connectToDatabase().
  - These methods do completely different jobs → **low cohesion**.
- CartManager directly creates Database and EmailService objects → **tight coupling**.
  - If you change the database or email service, you must edit CartManager.

Software Design Fundamentals (Principles)

- Design Principle #6: ***Separation of Interface and Implementation***
  - ✓ Deals with creating modules in such way that a stable interface is identified and separated from its implementation.
  - ✓ Not the same thing as encapsulation!
  - ✓ While encapsulation dictates hiding the details of implementation, this principle dictates their separation, so that different implementations of the same interface can be provided





Extra:

Encapsulation Prevent direct access to an object's internal data and hide its implementation details.

Segregation of Interface and Implementation

•**Meaning:** Keep the **what** (interface) separate from the **how** (implementation) and allows changing the implementation **without** affecting the interface users.

Software Design Fundamentals (Principles)

- Design Principle #7: ***Completeness***
  - Measures how well design units provide the required services to a achieve their intent (**no less**)
  - Example: in detailed design, a communication class is considered complete if it provides all required services establishing /terminating a connection, sending/receiving messages. If any of these service in missing the class is considered as incomplete

- Design Principle #8: ***Sufficiency***
  - Measures how well design units are providing only the services that are sufficient for achieving their intent (no more)

Extra:

•**Completeness** = WHAT → all requirements are addressed.
•**Example:** If the requirement says *"the banking system must allow users to transfer money and view transaction history,"* the design must include **both features** (transfer + history).

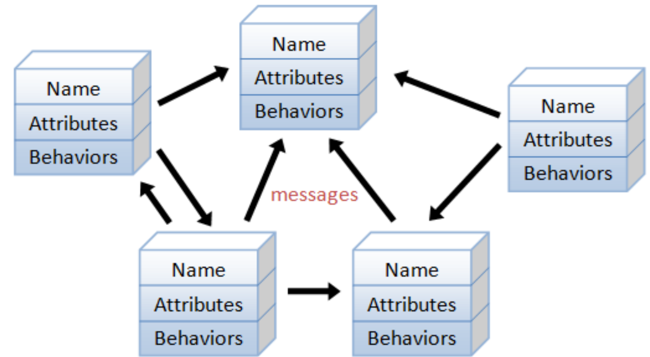
•**Sufficiency** = HOW → the design solves them effectively, efficiently, and adequately and nothing extra is there.
•**Example:** A messaging module should not also manage user profiles — that's outside its purpose.

So …. To Wrap-up

- Design **Principles**
  - ✓ **Modularization** □ Decompose into separate modules
  - ✓ **Abstraction** □ Separate the (what) from the (how)
  - ✓ **Encapsulation** □ Information hiding
  - ✓ **Coupling** □ Dependency between modules
  - ✓ **Cohesion** □ Tasks in a module
  - ✓ **Separation of Interface and Implementation** □ An interface … with different implementations
  - ✓ **Completeness** □ All required units for achieving intent
  - ✓ **Sufficiency** □ Sufficient units for achieving intent

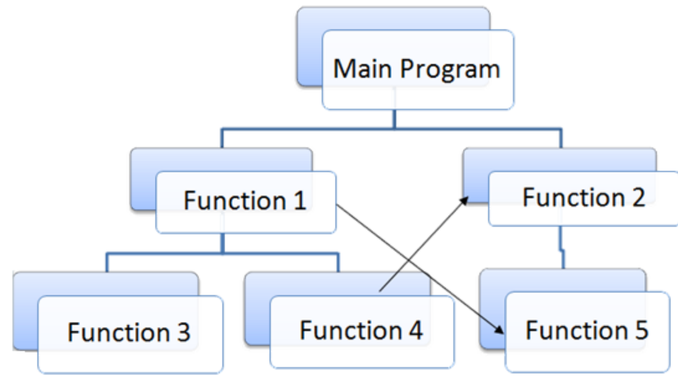
Software Design Fundamentals (Strategies)

- **Object-oriented design strategy**
  - ✓ Design strategy in which a system or component is expressed in terms of **objects** and connections between those objects.
  - ✓ Focuses on object decomposition.
    - Objects have unique identity (name)
    - Objects have state (attributes)
    - Objects have well-defined behavior
  - ✓ Supports inheritance and polymorphism
  - ✓ Appropriate for use with Object-oriented Languages



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Software Design Fundamentals (Strategies)



- **Structured (or Functional) design strategy**
  - ✓ Design strategy in which a system or component is decomposed into single-purpose, independent modules, using an iterative top-down approach.
  - ✓ Focuses on
    - The functions that the system needs to provide,
    - The decomposition of these functions, and
    - The creation of modules that incorporate these functions.
  - ✓ Inappropriate for use with object-oriented programming languages.

Extra:
Structured (Functional) Design Strategy:
- **Meaning**: Breaks the system into independent, single-purpose modules using **a top-down approach**.
- **Focuses on:**
  1-The functions the system must perform.
  2-Breaking functions into smaller ones.
  3-Creating modules that implement these functions.
- **Limitation**: Not suitable for object-oriented programming since it centers on functions, not objects.

Software Design Fundamentals (Considerations)

- **Design for minimizing complexity**
  - ✓ Design is about minimizing complexity.
  - ✓ Every decision that is made during design must take into account reducing complexity.
  - ✓ When faced with competing design option, always choose the one that minimizes complexity

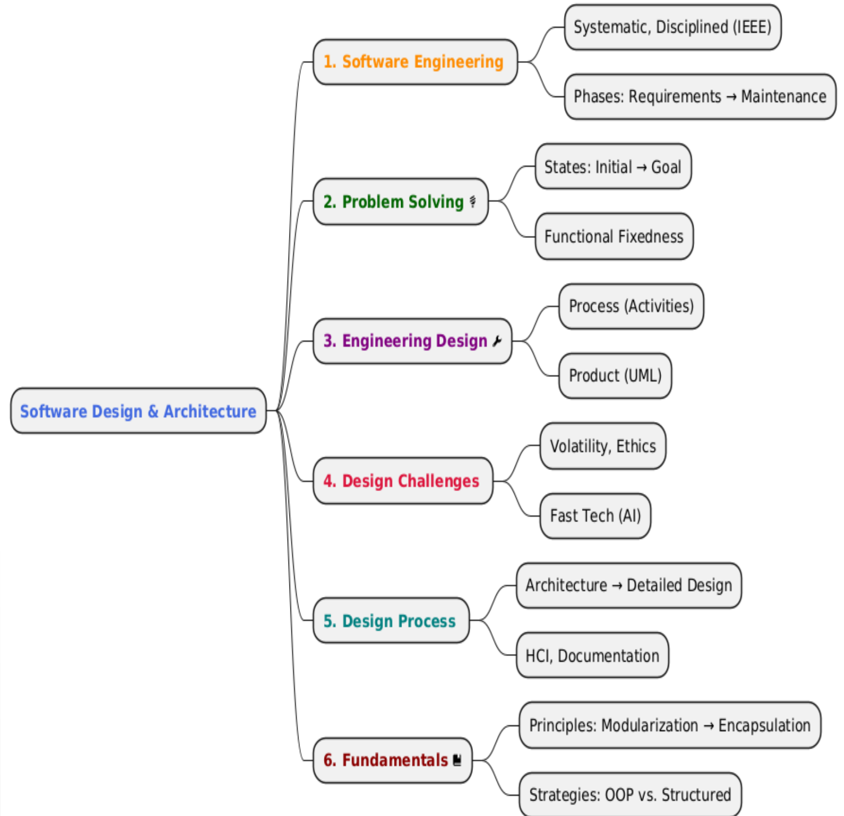
- **Design for change**
  - ✓ Software will change, design with extension in mind.
  - ✓ A variety of techniques can be employed through the design phase to achieve this.

*For Next Lecture/tutorial - Exercise

- Each student should prepare a **mind-map** for the topics that we have covered in the first chapter and how they relate with each other.
- Upload your mind-map on Moodle in the Mind-Maps forum link
- This should help you have an overall understanding of the topics we study

* This is required at the end of each chapter

Try using Use a mind-mapping tool (**XMind or Miro or Draw.io)**



Summary

- Software engineering (revised)
- Problem solving
- Software engineering design
- Why study software engineering design
- Software design challenges
- Software design process
- Software design fundamentals
- Next .. Design architecture

Summary (detailed)

- **Software engineering (revised)**
  - definitions
    - Software
    - Engineering
    - Software engineering
  - Software engineering phases
- **Problem solving**
  - Problem solving states
  - Functional fixedness problem
  - Problem solving holistic approach
- **Software engineering design**
  - Process
  - Product
  - Why study SE design
- **Software design challenges**
  - Requirements volatility
  - Inconsistent development processes
  - Fast, and ever-changing technology
  - Managing design influences
  - Ethical and Professional Practices

Summary (detailed)

- **Software design process**
  - Architecture design
  - Detailed design
  - Construction design
  - Computer-human interaction design
  - Design documentation
  - Management activities

- Software design fundamentals
  - **Design principles**
    - Modularizations
    - Abstraction
    - Encapsulation
    - Coupling
    - Cohesion
    - Separate interface from implementation
    - Completeness
    - Sufficiency
  - **Design strategies**
    - Object-oriented strategy
    - Functional (structured) strategy
  - **Design considerations**
    - Minimize complexity
    - Design for change