

OVERVIEW Introduction to Software Design and Architecture

Chapter 1 sets up the vocabulary for the whole course. It starts from what software engineering is, shows that design is a kind of problem solving (and how designers get stuck), defines design as both a process and a product, lists the five challenges that make design hard, splits the design process into major and supporting activities, and ends with the eight design principles, two strategies and two considerations used to judge every design later on. Slides labelled “Extra” are the instructor’s own explanations and examples, and they are examinable like the rest.

Design is problem solving

Problems move through an initial, operational and goal state. Functional fixedness (seeing things only in their usual role) blocks solutions in the operational state.

Process and product

Design means the activity of building models (a project-management concern) and the models it produces (a technical concern).

Architecture vs. detailed design

Architecture is the black-box, macro view focused on quality; detailed design is the white-box view of each component’s internals, focused on function.

Eight principles

Modularization, abstraction, encapsulation, coupling, cohesion, separation of interface and implementation, completeness and sufficiency.

HOW THE LECTURE IS ORGANIZED

Slides 1–8 Software engineering revised: definitions, IEEE, the life-cycle phases

Slides 9–13 Engineering problem solving: three states, functional fixedness, the holistic framework

Slides 14–18 What design is (process and product) and why it is worth studying

Slides 19–24 The five design challenges

Slides 25–39 The design process: major and supporting activities, designer roles

Slides 40–65 Design principles: modularization to sufficiency

Slides 66–68 Design strategies and considerations

Slides 69–73 Mind-map exercise and summary

EXAM FOCUS

This chapter is heavily examined. Final 252 Q1 took most of its ten MCQs from it (IEEE definition, functional fixedness, the holistic framework, requirements volatility, architecture vs. detailed design, abstraction, coupling, sufficiency). Midterm 252 asked two written questions (managing design influences, and the list of design activities), and Quiz 1 and Midterm 261 asked about functional fixedness, the design challenges, the two major activities, modularization, complexity, and coupling and cohesion types.

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.



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]

NOTES

- 1 Read this note first: any slide or box labelled “**Extra**” is the instructor’s own explanation or example, added on top of the textbook material. It is meant to make the definitions concrete, and it is fair game in quizzes.
 - 2 The chapter covers weeks 1–2 and gives the vocabulary used for the rest of SE322: what design is, why it is hard, how the design process is organised, and the principles used to judge a design.
-

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.



NOTES

- 1 The house analogy: a building can look finished and still crack later if the foundation is weak. In software the "cracks" are bugs, crashes and code that is painful to change.
- 2 The foundation of software is its **architecture** (the big structure) and the **design principles** used inside it. The whole course is about building that foundation on purpose instead of by accident.
- 3 Software engineering was named in **1968** because large projects of the 1960s kept running late, over budget or failing outright. The answer was to build software with the planned, measured methods of 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

NOTES

- 1 The lecture moves from the general to the specific: what software engineering is, then design viewed as **problem solving**, then what design is and why we study it.
 - 2 Next come the **challenges** that make design hard and the **process** (the activities designers carry out).
 - 3 It ends with the **fundamentals**: principles, strategies and considerations. These are the tools you will apply in every later chapter, so they get the most slides.
-

Software Engineering (revised)

➤ **Software**

is a collection of instructions that enable the user to interact with the computer hardware 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.



NOTES

- 1 **Software** is the set of instructions that lets a user make the hardware do useful work: operating systems, word processors and browsers are all software.
- 2 **Engineering** is a systematic process: designers generate and evaluate designs whose functions meet the clients' objectives and the users' needs while respecting constraints (cost, time, laws, hardware). The bridge example makes the point: you plan for weight and weather before you build, and software needs the same planning.
- 3 Keep **user** and **client** apart. The user is whoever actually works with the software; the client is whoever pays for it. A hospital buys a system (client) that nurses use every day (users), and the two may want different things.
- 4 The IEEE definition is the one to memorise word for word: software engineering is the application of a **systematic, disciplined, and quantifiable** approach to the development, operation, and maintenance of software, in short, applying engineering principles to software. IEEE is the professional body that publishes such standards.

Past exam · Final 252 · Q1 MCQ 1

Q1

According to the IEEE definition, which of the following best describes Software Engineering?

- A. The artistic creation of computer programs and hardware.
- B. The application of a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software.
- C. A methodical procedure for writing code in a specific programming language.
- D. A process focused solely on finding and fixing bugs after delivery.

Answer: B. The application of a systematic, disciplined, and quantifiable approach to the development, operation, and maintenance of software. — The IEEE wording: systematic, disciplined, quantifiable; covering development, operation and maintenance.

Prediction

Q2

Explain the difference between a user and a client, with an example.

Answer: The user interacts with the software; the client pays for its development. Example: a bank (client) commissions a mobile app that its customers (users) use.

SLIDE 5 / 73 Software Engineering (revised)

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.

NOTES

- 1 **Systematic** means the work is organised as a coherent body of ideas and is methodical: there is a plan and a procedure, not improvisation.
- 2 **Disciplined** means a set of rules governs the work, and the team actually follows them (coding standards, review steps, approval rules).

- 3 **Quantifiable** means you can measure the deliverables: how many requirements are done, how many defects were found, how long a response takes. If you can't measure it, you can't tell whether it is improving.
- 4 The Extra example: agile methods run in short iterations with regular checks, which is one practical way to be systematic, disciplined and measurable at the same time.

EXAM ANGLE

Prediction

Q1

"To determine, express, or measure the quantity of deliverables" defines which word in the IEEE definition?

- A. Quantifiable
- B. Systematic
- C. Disciplined
- D. Modular

Answer: A. Quantifiable

SLIDE 6 / 73 Software Engineering (revised)

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

NOTES

- 1 **Requirements:** find out what is needed, then analyse, specify and validate it. Everything later depends on this being right.
- 2 **Design** turns the requirements specification into a software design, which has two levels: the **architecture** (major parts and how they connect) and the **detailed design** (the inside of each part). This course lives in this phase.

- 3 **Implementation** writes the code, using the requirements, the architecture and the detailed design together.
- 4 **Testing** checks that the software behaves correctly and meets the specified requirements.
- 5 **Maintenance** changes the software after delivery: fixing faults, improving performance, or adapting it to a new environment. It is usually the longest and most expensive phase, which is why designs are judged by how easy they are to change.

EXAM ANGLE

Prediction

Q1

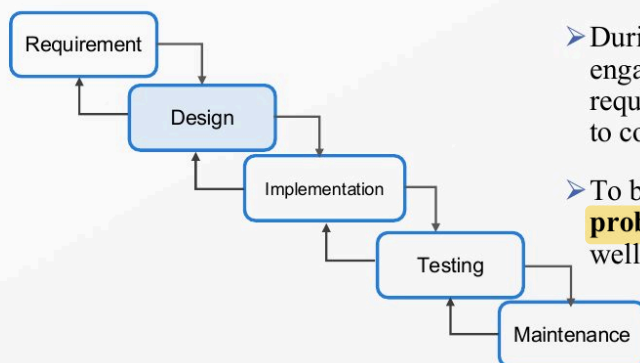
In which phase is the requirements specification used to create the software architecture and detailed design?

- A. Design
- B. Implementation
- C. Testing
- D. Maintenance

Answer: A. Design

SLIDE 7 / 73 Software Engineering (revised)

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

NOTES

- 1 Design gives a team a shared plan: who builds which part and how the parts fit. That is how it lets teams work in a **disciplined** way and keeps the product aligned with its specification.

- 2 Designing is constant **problem solving**: engineers identify problems, evaluate options and propose solutions throughout the phase, not just once at the start.
- 3 Hence the slide's conclusion: a good designer must first be a good problem solver, and that skill comes from practice over time. The next slides look at how problem solving works and where it goes wrong.

SLIDE 8 / 73 Extra: Phases of the Software Development Cycle

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.

NOTES

- 1 An expanded life cycle with an example for each phase. **Requirements**: identify and document needs, check feasibility and completeness (example: interviewing users). **Testing** (top right): unit, integration and system tests, often automated.
- 2 **Design** creates the blueprint: the architecture plus the detailed design of how components interact (example: UML diagrams).
- 3 **Implementation** translates the design into code modules and integrates them into one system.
- 4 **Deployment** is a phase this slide adds that slide 6 does not list: releasing the software into the target environment, e.g. putting a web app on a cloud server.
- 5 **Maintenance** continues after release: fix bugs, add features, adapt to new requirements, and watch performance and user feedback (example: security patches).

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.

NOTES

- 1 Problem solving in design passes through **three states**. Knowing which state you are in tells you what activity is appropriate.
- 2 **Initial state**: the problem is formulated and interpreted. Often the hardest part is understanding the problem itself (what exactly does the client need?).
- 3 **Operational state**: once the problem is understood, you think about it and possible solutions appear. This is where creativity and alternatives matter, and where functional fixedness (slide 11) does the most damage.
- 4 **Goal state**: a solution has been identified, evaluated and validated. Problem solving ends here.

EXAM ANGLE

Prediction

Q1

In which problem-solving state is the problem formulated and interpreted?

- A. Initial state
- B. Operational state
- C. Goal state
- D. Final state

Answer: A. Initial state

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 constraints??? What are they?



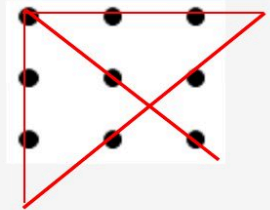
NOTES

- 1 The classic nine-dot puzzle: connect all nine dots with four straight lines without lifting the pencil or retracing a line.
- 2 Read the rules again: nothing says a line must start or end on a dot, or stay inside the square. The real constraints are only the four listed. The puzzle is hard because most people add a constraint that isn't there, which is the point of the next slide.

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 state** of problem solving.

Possible solution



NOTES

- 1 Most students assumed each line must begin and end on a dot. The requirements never said so; it was an assumption they brought with them.
- 2 **Functional fixedness:** you only see an object (or a component, or a technique) in the role you are used to, so solutions that use it differently never occur to you. It is a limit of the designer's thinking, not a property of the software.
- 3 The remedy is deliberate: attempt the problem several times, from different viewpoints and unusual angles (“think outside the box”). The possible solution on the right only works because its lines leave the square.
- 4 It matters most in the **operational state**, the state where alternative solutions are generated (slide 9).

Past exam · Final 252 · Q1 MCQ 2

Q1

In the context of engineering problem-solving, what is "functional fixedness"?

- A. A state where a solution is successfully validated and the goal is reached.
- B. A design strategy that focuses strictly on functional requirements.
- C. A cognitive bias that limits the ability to find solutions because objects are only seen in their usual function.
- D. The process of fixing errors in the final implementation phase.

Answer: C. A cognitive bias that limits the ability to find solutions because objects are only seen in their usual function. — A mental block that ties you to an object's usual role.

Past exam · Midterm 261 · Q4

Q2

A student rejects a valid design because she assumes a component can only be used in the same way she has seen it used before. Which concept best explains this limitation?

- A. Validation
- B. Modularization
- C. Abstraction
- D. Functional fixedness

Answer: D. Functional fixedness — The same question also appeared as Quiz 1 261 Q7.

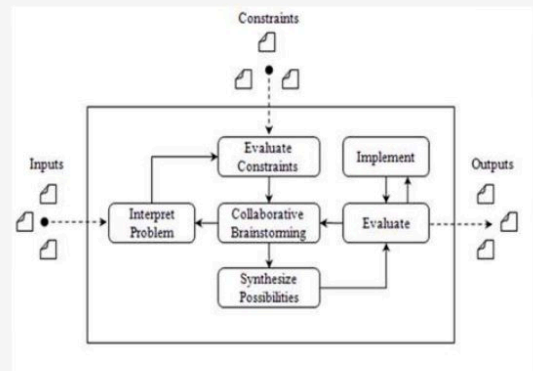
SLIDE 12 / 73 Slide 12

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 | <ul style="list-style-type: none"> ✓ Interpreting the problem ✓ Evaluating the constraints |
| operational state | <ul style="list-style-type: none"> ✓ Collaborative brainstorming ✓ Synthesizing "Summarizing" the possibilities: ✓ Evaluating the solution |
| Goal state | <ul style="list-style-type: none"> ✓ Implementing the solution |



NOTES

- 1 A **holistic** approach looks at the whole system and how its parts interact. Here the three states are fused into one framework of six tasks that fits engineering problems at every stage.
- 2 **Initial state** tasks: interpret the problem and evaluate the constraints.
- 3 **Operational state** tasks: collaborative brainstorming, synthesizing (summarising) the possibilities, and evaluating the solution.
- 4 **Goal state** task: implement the solution.

EXAM ANGLE

Past exam · Final 252 · Q1 MCQ 3

Q1

Which phase of the holistic problem-solving framework is performed during the "Initial State"?

- A. Collaborative brainstorming.
- B. Synthesizing the possibilities.
- C. Interpreting the problem and evaluating constraints.
- D. Implementing the solution.

Answer: C. Interpreting the problem and evaluating constraints. — Brainstorming and synthesizing are operational; implementing is the goal state.

SLIDE 13 / 73 Slide 13

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

NOTES

- 1 **Interpreting the problem** (initial state): understand what is being asked and classify it, for example as a known kind of problem with known solutions.

- 2 **Evaluating constraints** (initial state): find the external limits (budget, deadlines, platforms, regulations) that set the bounds of the “solution landscape”. Note the link to slide 10: only real constraints belong here, not assumed ones.
- 3 **Collaborative brainstorming** (operational state): the team proposes as many different solutions as it can before judging them.
- 4 **Synthesizing the possibilities** (operational state): narrow the ideas down and choose one acceptable solution.
- 5 **Evaluating the solution** (operational state): test the chosen idea. A flaw found here sends you back to brainstorming, so the framework is a loop, not a straight line.
- 6 **Implementing the solution** happens in the goal state, once a suitable solution has been found.

SLIDE 14 / 73 **Software Engineering Design**

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**)

NOTES

- 1 The formal definition has two halves. (1) Design is a **process**: identifying and evaluating the architectural and detailed models needed to build software that meets its functional and non-functional requirements.
- 2 (2) Design is also the **product** of that process: the design diagrams and documents it produces. When someone says “show me the design”, they mean the product; when they say “we are in design”, they mean the process.

Prediction

Q1

True or false: in software engineering, "design" refers only to the diagrams produced, not to the activity.

Answer: False — The definition covers both the process and its product.

SLIDE 15 / 73 Software Engineering Design

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.

NOTES

- 1 Because the word covers both meanings, industry uses "design" for the process and the product interchangeably. Exam questions often ask which perspective a statement belongs to.
- 2 **Process perspective:** the activities and tasks needed to model the software's structure and behaviour before construction. This is mostly a **project-management** concern: the management team plans, organises and assigns the design tasks and lines up schedule and resources.
- 3 **Product perspective:** the artifacts that design produces, such as class diagrams and sequence diagrams. This is mostly a **technical** concern: the technical team creates the actual models that define structure and behaviour.

Why Study Software Engineering Design??

NOTES

- 1 Section divider. The next two slides answer the question from both perspectives introduced on slide 15: project management (process) and technical (product).

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**.

NOTES

- 1 First reason set: the **project-management** (process) perspective.
- 2 Good design **minimises the effect of requirements volatility**. In a modular design a new feature, such as multi-language support, is added to one module without disturbing the others.
- 3 It makes **human resource allocation** efficient: once components are defined and documented, the manager can give each one to a developer with the right skills.
- 4 It protects the project from depending on **"one guy"** who owns the whole system in his head. If the design is written down, the project survives that person leaving.
- 5 Overall, design improves the four management functions: **planning, organisation, staffing and tracking**.

EXAM ANGLE

Past exam · Final 252 · Q1 MCQ 4

Q1

From a project management perspective, why is studying software design important?

- A. It ensures that only one person owns the entire system.
- B. It helps minimize the negative effects of requirements volatility.
- C. It focuses purely on technical artifacts like class diagrams.
- D. It eliminates the need for testing and maintenance phases.

Answer: B. It helps minimize the negative effects of requirements volatility. — Option a is the reverse of the slide: design shields the project from one person owning everything.

SLIDE 18 / 73 Possible reasons

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.

NOTES

- 1 Second reason set: the **technical** (product-development) perspective. The online-store example runs through all six points.
- 2 Requirements are mapped to **conceptual models**: “product listing, cart, checkout” become user-interface diagrams and flowcharts the team can reason about.
- 3 Models show the system’s **structure and behaviour**, e.g. a use case diagram for user interactions and a sequence diagram for the order of steps in a purchase.
- 4 The main **components and interfaces** are identified: catalogue, cart, payment gateway, authentication.
- 5 Design gives a way to **evaluate quality attributes** (usability, efficiency...) before code exists, such as checking how many steps checkout takes.
- 6 Designed solutions can be **reused**: the payment-gateway module can serve other e-commerce projects.
- 7 Design is the **foundation** for construction, testing and maintenance: the database schema and API design guide coders, testers and maintainers alike.

SLIDE 19 / 73 Major Software Design Challenges

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, and Different 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.

NOTES

- 1 **Requirements volatility:** the requirements change while you build. A banking app that started simple must suddenly add advanced login security late in development.
- 2 **Inconsistent development processes:** teams don’t follow the same design process. One team uses Agile, another Waterfall, and integration gets delayed.

- 3 **Fast, ever-changing technology:** platforms and libraries change constantly; new iOS and Android versions force quick updates just to keep the app working.
 - 4 **Managing design influences:** stakeholders pull in different directions; executives want profit features while the security team wants stronger protection.
 - 5 **Ethical and professional practices:** protecting customer data and privacy even under deadline pressure.
-

EXAM ANGLE

Past exam · Midterm 261 · Q2

Q1

Which FOUR of the following are considered major Software Design Challenges? (Select four.)

Answer: Fast and ever-changing technology, inconsistent development processes, managing design influences, and requirements volatility. The distractors (cost estimation, choosing a language, test data, configuring servers) are project or deployment tasks, not design challenges. The paper listed four; the slide's fifth is ethics.

Prediction

Q2

Which of these is NOT one of the major software design challenges?

- A. Choosing a programming language
- B. Requirements volatility
- C. Inconsistent development processes
- D. Managing design influences

Answer: A. Choosing a programming language

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

NOTES

- 1 **Requirements volatility** is growth or change in the requirements during the project's development life cycle. Some change is normal; the challenge is designing so it does not wreck the project.
- 2 The later a change arrives, the more it costs. Early on, a change edits a document; late on, it edits finished work.
- 3 Why: the design, the verification (test) plans and the deployment plans were all built on the old requirements, so each has to be reworked. This is the reason good design (slide 17) tries to isolate the parts most likely to change.

Past exam · Final 252 · Q1 MCQ 5

Q1

Which software design challenge refers to the growth or change in requirements during a project's development lifecycle?

- A. Inconsistent development processes.
- B. Managing design influences.
- C. Requirements volatility.
- D. Technical debt.

Answer: C. Requirements volatility.

Prediction

Q2

Why are requirement changes more costly in the later stages of the life cycle?

Answer: Because the change may force rework of the design, of the verification (test) plans and of the deployment plans, all of which were built on the old requirements.

SLIDE 21 / 73 Software Design Challenge #2 Inconsistent development processes

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**.

NOTES

- 1 In the design phase, the **process** is the set of activities that bridges the gap between requirements and construction.
- 2 Examples of those activities: architectural and detailed design, **design reviews**, setting **quality evaluation criteria**, design change management and version control, and adopting design tools.

- 3 The challenge is that in many companies this process is **not well established**, is **poorly understood**, focuses on one activity (say the user interface) while ignoring others, or is **not done at all**.
- 4 The house analogy: one team follows a plan, another works without one, one builds doors before the walls exist. In software, Team A uses Agile, Team B uses Waterfall, nobody agreed on design rules, and the pieces don't fit at integration time.

SLIDE 22 / 73 **Software Design Challenge #3 The Technology**

Software Design Challenge #3 – The Technology

- The **technology** for designing and implementing today's 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**.

NOTES

- 1 The technology used to design and build software keeps evolving to offer new capabilities. AI and machine learning are today's obvious example.
 - 2 The chatbot example shows the cost: before integrating an AI chatbot into customer service, developers must learn ML algorithms, data preparation and model training, and still make the bot work with existing customer databases.
 - 3 Designers are expected to understand and use new technologies **all at the same time**, which creates demand for designers who learn quickly.
 - 4 Why it is hard: learning takes time while the project deadline doesn't move, and the new technology must **interoperate with old legacy systems** that can't simply be replaced.
-

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.

NOTES

- 1 A project has many **stakeholders** (users, clients, managers, security, operations), and each has needs that pull the design their way: one wants performance, another usability, another security.
- 2 Some of those needs **conflict**, and each stakeholder is convinced they are right. Stronger security can make the system slower or harder to use, for instance.
- 3 The designer has to make **trade-offs** that satisfy each stakeholder as far as possible, without the decision harming the project as a whole.
- 4 This is especially difficult in **large-scale systems**, where there are more stakeholders and every decision touches more parts.

EXAM ANGLE

Past exam · Midterm 252 · Q2 · 3 marks

Q1

How could "managing design influences" be a challenge for software designers?

Answer: Software projects have many stakeholders, each with specific wants and needs that influence the design. Some of these needs conflict with each other, and each stakeholder believes they are correct. Designers must therefore make design trade-offs to satisfy each customer, which is difficult in large-scale systems, because design decisions must accommodate all concerns without negatively affecting the project.

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

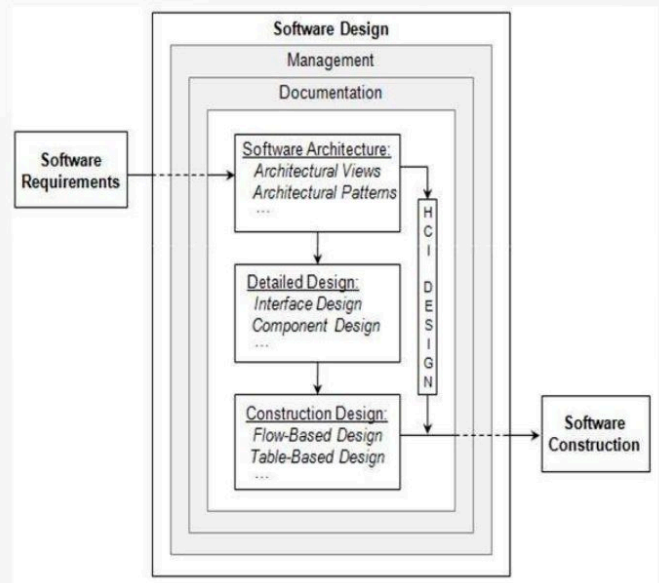
NOTES

- 1 Design decisions affect the **environment and people** who use the software: privacy, safety, accessibility, energy use. Designers are responsible for those effects.
 - 2 Designers must show strong **ethical and professional practices** so that the systems they produce are high quality, not just finished on time.
 - 3 That takes **leadership**: influencing and negotiating with stakeholders and motivating the team. It is hard because management, customers and peers all apply pressure, often toward cutting corners.
 - 4 The **ACM Code of Ethics and Professional Conduct** is the reference example. Its general principles include contributing to society and well-being (respect privacy, be honest and fair), professional responsibilities (follow rules, ensure security), and leadership principles (put the public good first).
-

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).



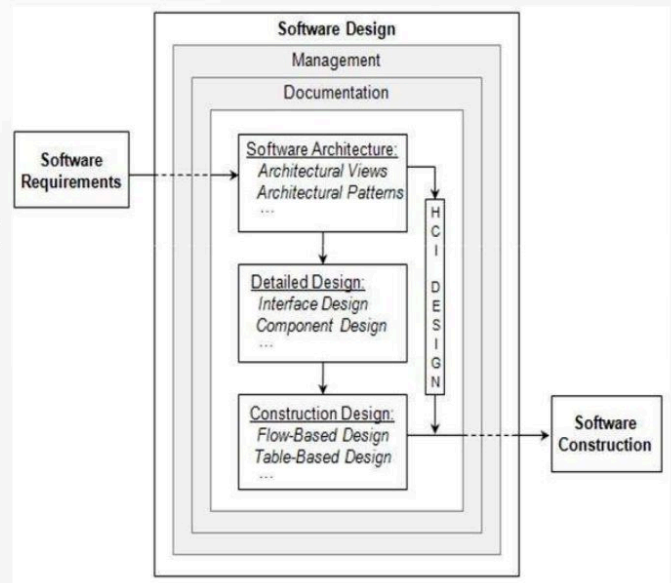
NOTES

- 1 A reminder from slide 14: design is both a process and a product. This part of the lecture is about the **process**.
- 2 A **software design process** is a set of activities that specify how resources (people, tools, time) work together to produce design products.
- 3 The products are things like use cases, class diagrams and other **UML** models, and design documents.

Software Design Process

➤ Software design process include major and supportive activities:

- | | | |
|-----------------------|---|--|
| Major activities | { | <ol style="list-style-type: none"> 1. Software architecture 2. Detailed design |
| Supporting activities | { | <ol style="list-style-type: none"> 1. Construction design 2. Human-computer interface design 3. Software design documentation 4. Management activities |



NOTES

- 1 The design process has two **major** activities: **software architecture** and **detailed design**. These two are the heart of the course (architecture in chapters 2–4, detailed design in 5–7).
- 2 And four **supporting** activities: construction design, human-computer interface design, software design documentation, and management activities. Construction design and HCI design sound “major” but are listed as supporting; that is the usual trap.

EXAM ANGLE

Past exam · Midterm 261 · Q3

Q1

Which TWO of the following are the main major activities in the Software Design Process? (Select two.)

Answer: Software Architecture and Detailed Design. Construction design, HCI design, documentation and management are the supporting activities; testing and requirements analysis are other life-cycle phases.

Past exam · Midterm 252 · Q3 · 3 marks

Q2

List the software design processes, including the major and supporting activities.

Answer: Major: **software architecture** (macro design that models the system’s quality and function) and **detailed design** (the internals of components and interfaces). Supporting: **construction design** (lowest level, how functions are implemented), **human-computer interface design** (how users interact with the system), **software design documentation**, and **management activities** (planning, organising, leading).

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

NOTES

- 1 **Software architecture** is the **macro-design** approach: it creates models that show the quality and the function of the whole system, at the level of big parts rather than code.
- 2 Its models are **black boxes**: each component is described by what it does and how it connects, not by its insides. That is enough to evaluate the system's projected quality and functions early.
- 3 It is designed from **multiple perspectives** (views), so stakeholders with different backgrounds can each check that their concerns are addressed. Chapter 2's 4+1 views come from this idea.
- 4 It defines the **major structural components and interfaces** of the system.
- 5 It focuses on the **quality aspects** (performance, security, availability...) before detailed design or construction begins, because quality is hard to add later.
- 6 It is a tool for communication, reasoning and analysis, and it lays the **foundation** for all later work in the life cycle.

Past exam · Final 252 · Q1 MCQ 6

Q1

Which design activity corresponds to a “macro-design approach” that provides black-box models to evaluate system quality?

- A. Detailed design.
- B. Software architecture.
- C. Construction design.
- D. Interface design.

Answer: B. Software architecture.

SLIDE 28 / 73 Software Design Process

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.

NOTES

- 1 **Detailed design** focuses on the **internals** of the components and interfaces that architecture identified.
- 2 It **begins after** the architecture is specified, reviewed and judged complete enough. Starting detailed design on a shaky architecture means redoing it.
- 3 It is a **white-box** approach: it opens each component and designs its internal structure and behaviour.
- 4 It refines the architecture until the whole design (architecture + detailed design) is complete enough for construction to begin.
- 5 Focus: detailed design is mostly about **functional** requirements (what the system does), while architecture is mostly about **non-functional**, quality requirements.

What is the primary focus of “Detailed Design”?

- A. External properties of software components and non-functional requirements.
- B. The overall development process for both hardware and software.
- C. The internals of system components and their behavior to fulfill functional requirements.
- D. Project management tasks like staffing and tracking.

Answer: C. The internals of system components and their behavior to fulfill functional requirements. — Option a describes architecture; option b describes the systems engineer’s role.

SLIDE 29 / 73 Extra: Software Architecture vs. Detailed Design

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.

NOTES

- 1 **Provides:** architecture gives the big picture (major components, how they connect, key quality attributes); detailed design gives the internal details of each component (algorithms, data structures, workflows).
- 2 **Viewpoint and timing:** architecture is the black-box view and comes first; detailed design is the white-box view and starts after the architecture is reviewed and approved.
- 3 **Focus and purpose:** architecture targets mostly non-functional requirements and is a communication tool for all stakeholders; detailed design targets functional requirements and tells developers how to build each part.

- 4 **Banking example:** architecture defines the UI, login system, transaction service and database and how they interact securely; detailed design says how the login checks credentials (encryption, API calls, error handling) and how transactions and queries work.

EXAM ANGLE

Prediction

Q1

Compare software architecture and detailed design on viewpoint, timing and requirement focus.

Answer: Architecture: black-box view of major components, done first, focused mostly on non-functional (quality) requirements. Detailed design: white-box view of each component's internals, done after the architecture is approved, focused on functional requirements.

SLIDE 30 / 73 Software Design Process

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.

NOTES

- 1 **Component design:** architecture only identifies the major components; component design creates their internal structure and behaviour. In object-oriented systems it is drawn in UML as class diagrams, sequence diagrams and so on. Chapter 5 is mostly this.
- 2 **Interface design:** specifying the interfaces between components, meaning the exact operations and data they exchange.
- 3 Interfaces can be **internal** (between two parts inside your system: the login system tells the transaction service "user authenticated, allow transactions") or **external** (between your system and an outside one: the banking app calls a third-party payment gateway).

Prediction

Q1

Specifying the interfaces between components in a design belongs to which detailed-design task?

- A. Interface design
- B. Component design
- C. Construction design
- D. Architecture evaluation

Answer: A. Interface design

SLIDE 31 / 73 Extra Example: Component vs. Interface Design

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).

NOTES

- 1 Wedding-dress analogy for **component design**: each part of the dress (bodice, skirt, sleeves) is designed in detail. In software: the internal details of each component (classes, attributes, methods).
- 2 For **interface design**: making sure all the parts connect and fit smoothly into one dress. In software: how components interact (method calls, data flow between modules). A beautiful bodice that doesn't attach to the skirt is a failed design.

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**.

NOTES

- 1 Why document the design?** It records every design decision (and its reason) for future reference, keeps the project consistent and traceable (each design element can be traced back to a requirement), and lets developers, testers and maintainers understand the system without asking the original designer.
- 2 Why manage the design?** Management coordinates the team and handles design changes, plans, tracks and controls the design process, and keeps the project 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.

NOTES

- 1 **Construction design** is the lowest level of detailed design: it models and specifies how individual functions are implemented. It is the step right before code.
- 2 Its main work is the analysis and design of **algorithms**, chosen for correctness and efficiency.
- 3 The example: once a "search" feature is designed, construction design decides whether it uses binary search (fast, needs sorted data) or linear search (simple, slower), based on speed and performance.

EXAM ANGLE

Prediction

Q1

Deciding whether a search function should use binary or linear search is part of which design activity?

- A. Construction design
- B. Software architecture
- C. Interface design
- D. Design management

Answer: A. Construction design

- Another important design activity is the **Human-computer interface design.**
- 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.

NOTES

- 1 **Human-computer interface design** applies general usability principles to optimise the interface between people and the computer: the UI should be easy to learn, friendly and efficient.
 - 2 The login-screen example lists concrete decisions: clear error messages, buttons of a proper size, and accessibility for all users (screen readers, colour contrast). These are design decisions, made before coding, not cosmetic touches added at the end.
-

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.

NOTES

- 1 Documentation starts in the design phase and continues through the whole life cycle, because the design keeps changing and the documents must keep up.
- 2 It is essential for large, complex systems, where no single person can hold the design in their head, and it supports both development and maintenance.
- 3 Its readers are managers, designers, developers, testers and maintainers, so it must contain all the design details each of them needs.
- 4 It is more than writing: it covers documenting, validating (checking the documents are correct) and managing the design documents.

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":

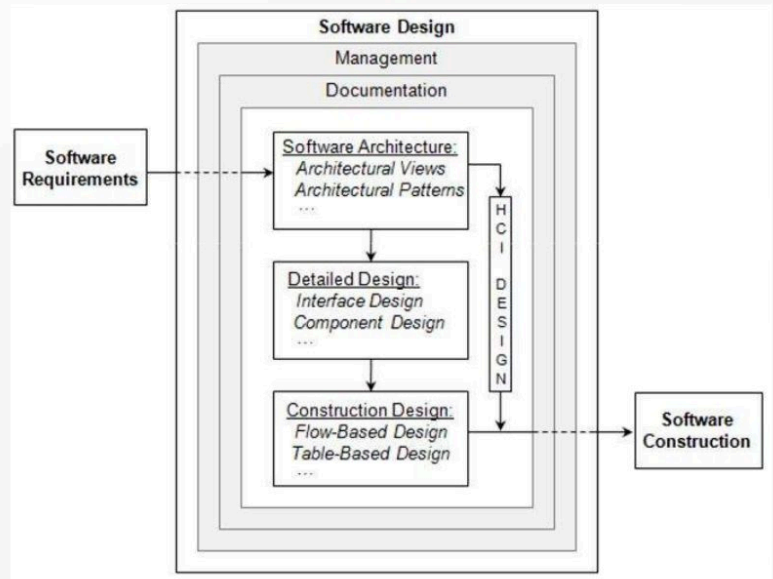
- **Role of Management:** Plans and controls resources to meet goals.
- **Design Phase Management:** Delivers quality designs on time and within budget.
- **Quality Focus:** Ensures design meets cost, schedule, and design standards.
- **Evaluation:** Assesses design quality from both management and engineering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

NOTES

- 1 **Design management** plans and controls resources so the design activities meet their goals.
- 2 In the design phase that means delivering quality designs on time and within budget, and evaluating design quality from both the management view (cost, schedule) and the engineering view (standards, correctness).
- 3 **Configuration management** controls changes to the design: which version is current, who changed what, and how improvements are approved. It is critical in large projects where many people edit the same design.

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.



NOTES

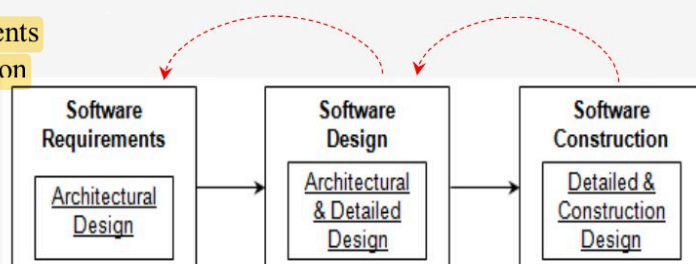
- 1 The design phase is made of several activities, and each has one or more tasks (for example, detailed design = component design + interface design).
- 2 The rest of the semester follows this map: how to carry out the major activities (architecture, then detailed design and patterns) and how to document them.

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



NOTES

- 1 Ideally all design work would happen inside the design phase. In practice design activities are spread across the life cycle.
- 2 Most design effort is still in the design phase, but some design definitely happens elsewhere: during **requirements** (for example sketching an architecture to check feasibility) and during **construction** (low-level design decisions made while coding).

SLIDE 39 / 73 Role of Software Designers

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**.

NOTES

- 1 **Systems engineer:** designs the overall development process of the whole system, including both its hardware and software parts.
- 2 **Software architect:** designs the software with a mostly **black-box** approach, concerned with the external properties of components that determine quality and support the later design of functional requirements.
- 3 **Component designer:** designs the internal structure and behaviour of the components found during architecture. These designers usually have strong programming skills, since they often implement their own designs.
- 4 **User interface designer:** designs the UI and is skilled at increasing usability.

Prediction

Q1

Who designs the internal structure and behaviour of components identified during architecture?

- A. Component designer
- B. Software architect
- C. Systems engineer
- D. User interface designer

Answer: A. Component designer

SLIDE 40 / 73 Next ... Software Design Fundamentals

Next ... Software Design Fundamentals

أكثر شيئا محي على السأله

➤ 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

NOTES

- 1 The fundamentals come in three groups. **Principles** are the rules used to judge a design: modularization, abstraction, encapsulation, cohesion and coupling, separation of interface and implementation, sufficiency and completeness.
- 2 **Strategies** are the two overall ways to decompose a system: object-oriented or structured (functional).
- 3 **Considerations** are two goals to keep in mind at every decision: minimise complexity and design for change.

Prediction

Q1

Name the eight design principles of Chapter 1.

Answer: Modularization, abstraction, encapsulation, coupling, cohesion, separation of interface and implementation, completeness, sufficiency.

SLIDE 41 / 73 Software Design Fundamentals (Principles)

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**.

NOTES

- 1 **Modularization** keeps decomposing the system into smaller components until they are fine-grained. Each module handles one part of the functionality and can be developed, tested and maintained independently.
- 2 It is one of the most important principles because it keeps the system manageable in every phase of the life cycle.
- 3 Splitting the design splits everything else with it: each module gets its own requirements, its own code and its own test cases. That is why one good decomposition helps the whole project.
- 4 The result is a design that is easier to understand, and so easier to develop, maintain and reuse.

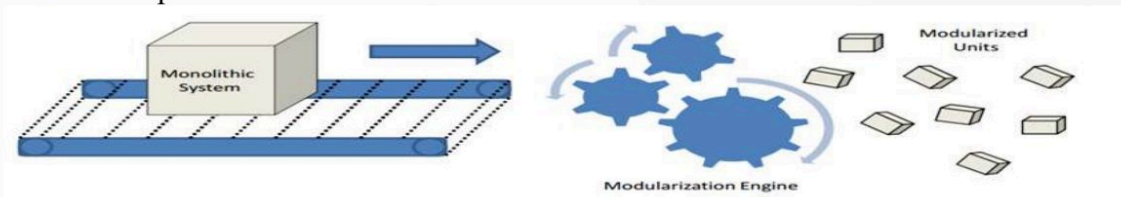
Fill in the blank: Breaking a complex software system into smaller, manageable parts is called ____.

Answer: Modularization.

SLIDE 42 / 73 Software Design Fundamentals (Principles)

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.

NOTES

- 1 **Fine-grained** means each component is small and specific: it has one narrow, focused responsibility, which makes it easier to understand, test and reuse.
- 2 Example: instead of one big Payment Processor module, use smaller ones such as Card Validator, Payment Authorizer and Receipt Generator. Each can be changed or reused without touching the others.

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

NOTES

- 1 Modules keep the **complexity** of a large program manageable: you reason about one module at a time.
 - 2 They **isolate errors**: a fault is usually confined to one module, so it is easier to find and fix.
 - 3 They **eliminate redundancy**: shared logic is written once in a module instead of copied in many places.
 - 4 They **encourage reuse**: a well-defined module becomes a library other programs can use.
 - 5 Overall a modular program is easier to write, read and 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.

NOTES

- 1 Modularization tells you to split the system, but not **where** to split it. Splitting at random produces modules that make no sense. Something has to guide and justify the cuts.
 - 2 **Abstraction** guides it: each module should represent one essential concept and hide the unnecessary details, so modules are meaningful and focused.
 - 3 **Encapsulation** completes it: once a module's essential face is decided, its internals are protected so other modules can only use the agreed operations.
-

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.

NOTES

- 1 **Abstraction** focuses on the essential characteristics of an entity, in its current context, and defers the unnecessary details to later.
- 2 Modularization says **what** must be done (split the system); abstraction says **how** to do it well. Modularizing in an ad-hoc way, without abstraction, gives incoherent designs that are hard to understand and maintain.
- 3 Abstraction applies to **data** (show only the attributes that matter) and to **functions/behaviour** (show only the essential operations). Slide 46 explains each.
- 4 Example: the payment gateway hides how payments are processed; the shopping cart just calls a simple "request payment" interface.

Past exam · Final 252 · Q1 MCQ 8

Q1

The principle of "Abstraction" is best described as:

- A. Decomposing a system into fine-grained components.
- B. Focusing on essential characteristics of entities while deferring unnecessary details.
- C. Exposing all internal data to ensure other modules can access it easily.
- D. Combining hardware and software into a single monolithic unit.

Answer: B. Focusing on essential characteristics of entities while deferring unnecessary details. — Option a is modularization.

Past exam · Quiz 1 · 261 · Q9 (true/false)

Q2

Abstraction helps designers manage complexity by focusing on essential characteristics and hiding unnecessary details.

Answer: True

SLIDE 46 / 73 Software Design Fundamentals (Principles)

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.

NOTES

- 1 **Functional (procedural) abstraction** simplifies reasoning about an operation made of many steps. "Computer 1 SENDS a message to computer 2" stands for: look up the server, open a TCP/IP connection, send, wait for the reply, close the connection. The word SEND lets you think about the operation without the steps.

- 2 **Data abstraction** simplifies reasoning about the structure of data. In the same sentence, MESSAGE is a data abstraction: you can talk about sending a message long before deciding its fields (header, body, encoding), which are deferred to later design stages.

EXAM ANGLE

Prediction

Q1

In "Computer 1 SENDS a MESSAGE to computer 2", which word is a procedural abstraction and which is a data abstraction?

Answer: SEND is a procedural (functional) abstraction: it hides the steps of the operation. MESSAGE is a data abstraction: it hides the structure of the data being sent.

SLIDE 47 / 73 Software Design Fundamentals (Principles)

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

NOTES

- 1 Abstraction hides details, so an entity is easier to use: you call `sort()` without knowing the algorithm.
- 2 It lets you design the general framework (the overall solution) first and postpone the details, which matches the move from architecture to detailed design.
- 3 Because users depend only on the abstraction, you can replace an implementation with a better one later without changing them.

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.

48

NOTES

- 1 **Encapsulation** gives access to an entity's services by exposing only the information needed to use them, and hides how the services are carried out.
- 2 Applied to data, it gives access to exactly the data needed, **no more, no less**.
- 3 Abstraction and encapsulation work together: abstraction decides what is essential and hides the rest; encapsulation then enforces it, revising the abstraction so only the essential information is reachable. In that sense encapsulation forces good abstractions.
- 4 In object-oriented code it is done with **private** attributes and public methods: a user's password is never read directly, only used through `login()` or `register()`.

EXAM ANGLE

Prediction

Q1

Exposing only the information essential to use a service, and hiding how the service is carried out, is the principle of:

- A. Encapsulation
- B. Abstraction
- C. Modularization
- D. Cohesion

Answer: A. Encapsulation

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	<i>What</i> a module/class does.	<i>How</i> 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.

NOTES

- 1 Purpose:** abstraction hides unnecessary details and shows only the essential features; encapsulation hides the internal implementation and restricts direct access to data.
- 2 Focus:** abstraction is about **what** a class or module does; encapsulation is about **how** its data and methods are accessed and protected.
- 3 Implementation:** abstraction is achieved with abstract classes, interfaces or simplified models; encapsulation with access modifiers (private, public, protected) and getter/setter methods.
- 4 Examples:** a payment module that shows only processPayment (abstraction); a password stored privately and reachable only through controlled methods (encapsulation).

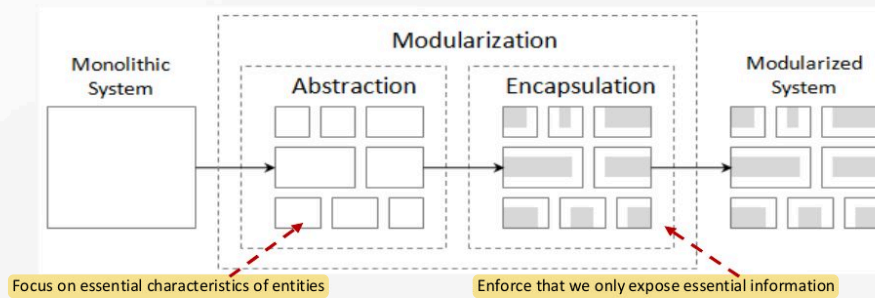
EXAM ANGLE**Prediction****Q1**

Give one difference between abstraction and encapsulation and how each is implemented in OO code.

Answer: Abstraction decides what a module shows (its essential features) and is implemented with abstract classes or interfaces; encapsulation restricts access to the internals and is implemented with private fields and public methods.

Software Design Fundamentals (Principles)

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



NOTES

- 1 The diagram chains the three principles: modularization splits the system into modules, **abstraction** decides each module's essential characteristics, and...
- 2 ...**encapsulation** enforces that each module exposes only that essential information. Remember the order: split, focus, protect.

SLIDE 51 / 73 Extra: Modularization, Abstraction and Encapsulation

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. → `Private, Public` → lock

`Self, Get` → How to edit or use → `Member` key

NOTES

- 1 **Modularization** example: an e-commerce system split into Login, Product Catalog, Cart and Payment modules, each easier to build, test and maintain.
- 2 **Abstraction** example: the Payment module shows `processPayment()` and hides the ten internal steps it takes.
- 3 **Encapsulation** example: card details inside Payment are private; other modules can only send data through methods like `pay()`.
- 4 The memory aid: modularization = split the house into rooms; abstraction = name each room by purpose without showing everything inside; encapsulation = lock the rooms and hand out keys only for the doors you may open.

SLIDE 52 / 73 Exercise

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



NOTES

- 1 The exercise applies modularization to an online student registration system: decompose it into fine-grained modules, each with one responsibility.
- 2 Then discuss abstraction for those modules, both **data** abstraction (which data each module exposes) and **behaviour** abstraction (which operations it offers).
- 3 Finally pick one module and discuss how to encapsulate it: what becomes private and which public operations remain. Slide 53 gives a model answer.

Prediction

Q1

Decompose an online student registration system into modules, and give one example each of data abstraction, behaviour abstraction and encapsulation.

Answer: Modules: user management, course catalog, registration, payment, schedule/timetable, notifications, reports. Data abstraction: Payment shows status (Paid/Unpaid) but not bank transaction IDs. Behaviour abstraction: Registration offers `enrollStudent(courseID)` and hides eligibility checks and database updates. Encapsulation: card numbers are private in Payment; only `processPayment()` and `checkPaymentStatus()` are public.

SLIDE 53 / 73 Extra: Exercise – Sample Answer

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.

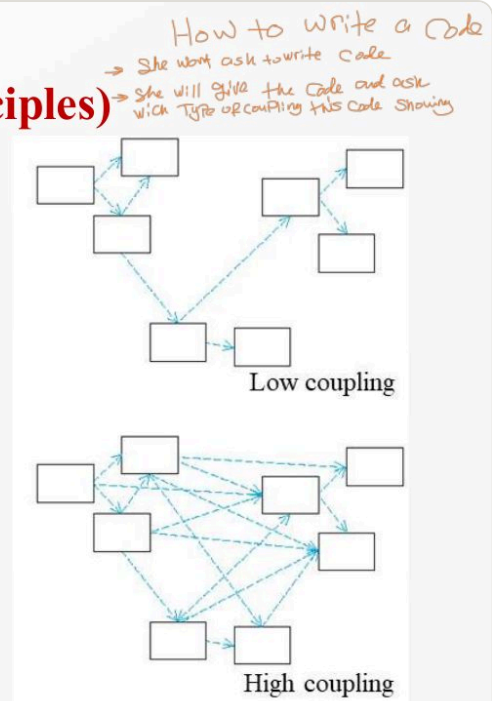
NOTES

- 1 A model decomposition into seven modules: user management, course catalog, registration, payment, schedule and timetable, notifications, and reports and records. Each has one clear job.
- 2 **Data abstraction:** the Payment module shows whether a fee is paid and hides the bank transaction IDs.
- 3 **Behaviour abstraction:** the Registration module offers `enrollStudent(courseID)` and hides how eligibility is checked and the database is updated.
- 4 **Encapsulation** for Payment: card numbers are private, the public methods are `processPayment()` and `checkPaymentStatus()`, and other modules can only go through that interface.

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**.



NOTES

- 1 **Coupling** is the manner and degree of interdependence between modules: how much one module relies on another.
- 2 It is a measurement: the higher the coupling, the stronger the dependency, so a change in one module is more likely to force changes in another.
- 3 Aim for **low coupling**. Loosely coupled modules can be changed, tested and reused separately.

EXAM ANGLE

Past exam · Final 252 · Q1 MCQ 9

Q1

In software design principles, "Coupling" refers to:

- A. The degree of interdependence between software modules.
- B. How well tasks within a single module relate to one another.
- C. Ensuring a module provides all required services.
- D. Hiding the implementation details of a class.

Answer: A. The degree of interdependence between software modules. — Option b is cohesion, c is completeness, d is encapsulation.

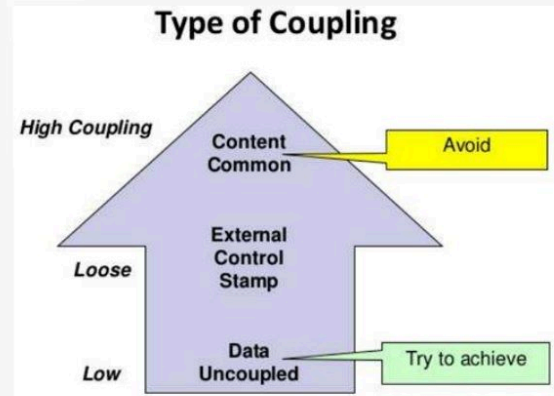
Software Design Fundamentals (Principles)

ST
↓
st+ID

Change
↓
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.



NOTES

- 1 **Content coupling** is the most severe type: one module modifies or relies on the **internal** data or code of another (e.g. directly changing another class's private field). Any change inside the second module can break the first.
- 2 **Common coupling**: modules depend on a shared common area, such as **global variables**. Change the global, and every module that uses it may change behaviour, which makes faults hard to trace.

EXAM ANGLE

Past exam · Midterm 261 · Q7 (word bank)

Q1

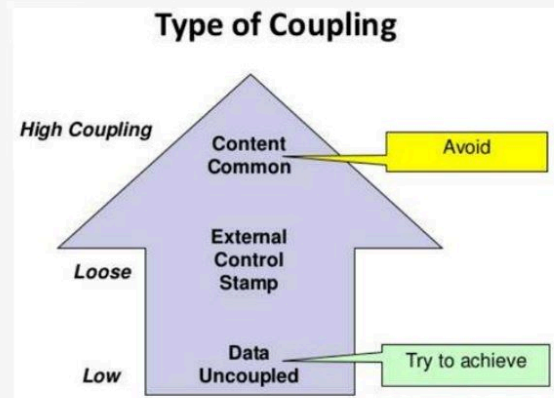
When multiple modules directly access the same global data, this is ___ coupling.

Answer: Common coupling.

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).



NOTES

- 1 **External coupling:** two modules share an externally imposed data format, communication protocol or device interface. If the external format changes, both modules must change.
- 2 **Control coupling:** one module controls the flow of another by passing it information on what to do, for example a flag like `mode = PRINT_OR_SAVE`. The receiver's behaviour now depends on the caller's decisions.

EXAM ANGLE

Past exam · Midterm 261 · Q7 (word bank)

Q1

When one module sends information that determines or controls the internal behavior of another module, this is __ coupling.

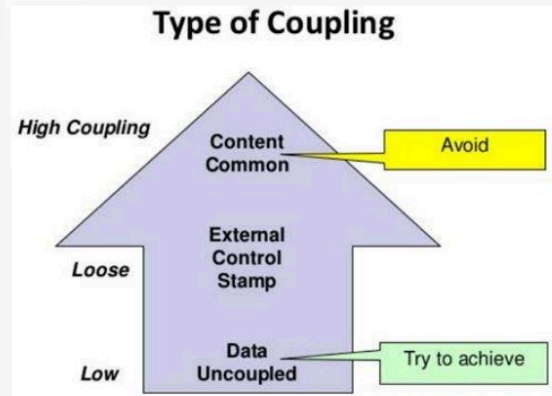
Answer: Control coupling.

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.

→ `Music { }---`
 → `calculator { }---`



NOTES

- 1 **Stamp coupling:** modules share a composite data structure but each uses only part of it, such as passing a whole student record to a function that needs only the ID. The function now depends on the whole record's structure.
- 2 **Data coupling:** modules share only the data they need, through parameters (pass the ID, not the record). This is the acceptable form of coupling.
- 3 **No coupling:** modules do not communicate at all.
- 4 Order from worst to best, as on slides 55–57: content, common, external, control, stamp, data, none.

Prediction

Q1

Passing a whole student record to a function that only needs the student ID is an example of:

- A. Stamp coupling
- B. Data coupling
- C. Content coupling
- D. Control coupling

Answer: A. Stamp coupling

Prediction

Q2

Which is the most severe type of coupling?

- A. Content coupling
- B. Common coupling
- C. Stamp coupling
- D. Data coupling

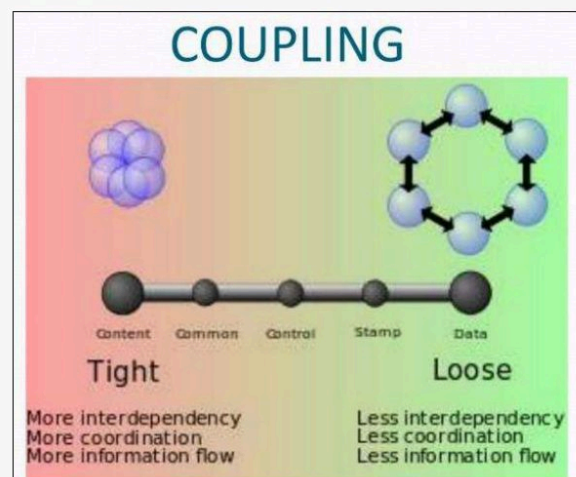
Answer: A. Content coupling

SLIDE 58 / 73 Software Design Fundamentals (Principles)

Software Design Fundamentals (Principles)

- 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.

Extra Figure



NOTES

- 1 High coupling lowers quality in two ways: **reusability** drops (a module can't be taken elsewhere without its partners) and **maintainability** drops (a change ripples into dependent modules).

- 2 As coupling rises, so does the **complexity** of managing and maintaining the design units, because you have to understand several modules to change one.

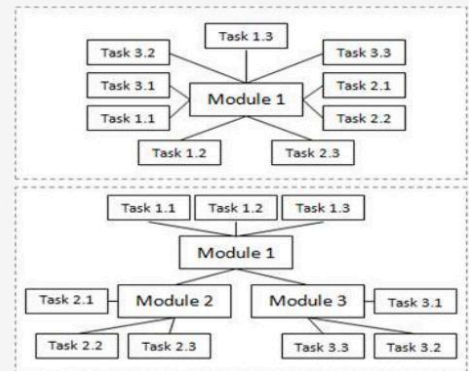
SLIDE 59 / 73 **Software Design Fundamentals (Principles)**

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...**



NOTES

- 1 **Cohesion** is the manner and degree to which the tasks performed by one module are related to each other.
- 2 It measures how well a design unit's parts belong together to achieve a particular task. A cohesive module is easy to name in a few words.
- 3 **High cohesion is good, low cohesion is bad.** Together with slide 54 this gives the golden rule: low coupling, high cohesion.

EXAM ANGLE

Prediction

Q1

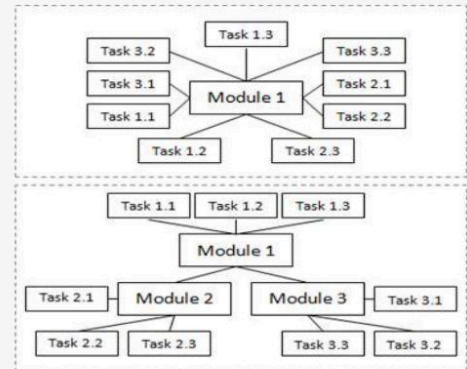
True or false: good design aims for high coupling and low cohesion.

Answer: False — It aims for low coupling and high cohesion.

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



NOTES

- 1 **Functional** cohesion (the best): every part of the unit contributes to one single function.
- 2 **Procedural (sequential)** cohesion: the tasks are steps done in order to achieve the unit's purpose, often the output of one step feeding the next.
- 3 **Temporal** cohesion: the tasks are grouped because they run at the same time (e.g. at start-up), not because they are related.
- 4 **Communication(al)** cohesion: the tasks produce or consume the same data.

EXAM ANGLE

Past exam · Midterm 261 · Q7 (word bank)

Q1

When several operations in a module work on the same input data or contribute to the same output, this is ___ cohesion.

Answer: Communicational cohesion.

Past exam · Midterm 261 · Q7 (word bank)

Q2

When module elements are grouped because they need to execute at approximately the same time, this is ___ cohesion.

Answer: Temporal cohesion.

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.

NOTES

- 1** `calculateTotal()` sums the prices in a cart: every line serves one task, so it is **functional** cohesion.
- 2** `processOrder()` validates the order, then calculates the cost, then processes payment: steps in sequence, so **procedural (sequential)**.
- 3** `initializeSystem()` opens database connections, loads configuration and logs the start-up time: unrelated tasks grouped because they happen at program start, so **temporal**.
- 4** `generateStudentReport()` calculates GPA, lists courses and prints grades, all from the same student record: **communicational**.

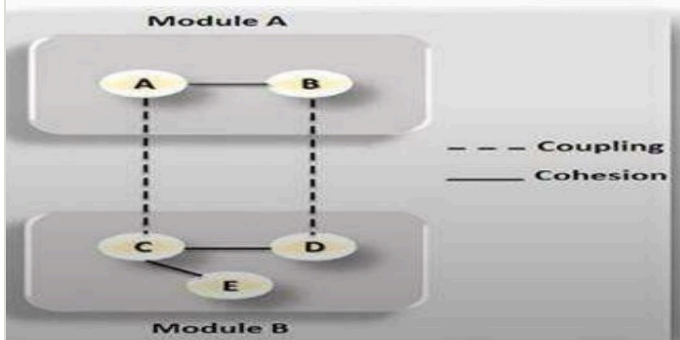
EXAM ANGLE**Prediction****Q1**

An `initializeSystem()` method opens database connections, loads configuration and logs the start time. Which cohesion is this?

- A. Temporal
- B. Functional
- C. Communicational
- D. Procedural

Answer: A. Temporal

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.

NOTES

- 1 Good design: ShoppingCart has `addItem()`, `removeItem()` and `calculateTotal()`, all about managing the cart, so cohesion is **high**.
- 2 It reaches the database and payment service only through an interface, so coupling is **low**: switching from SQLite to Firebase doesn't break the cart.
- 3 Bad design: CartManager mixes `addItemToCart()`, `sendEmailToUser()` and `connectToDatabase()`, three unrelated jobs, so cohesion is **low**.
- 4 It also creates the Database and EmailService objects itself, so coupling is **tight**: changing either service means editing CartManager.

EXAM ANGLE

Prediction

Q1

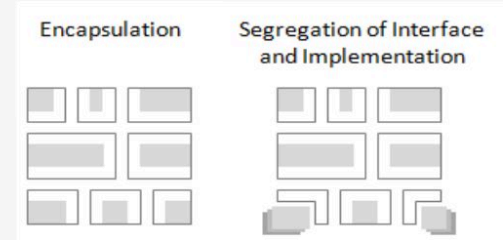
A CartManager class adds items to the cart, sends emails to users and connects to the database, and creates its own Database object. Identify the cohesion and coupling problems and suggest a fix.

Answer: Low cohesion: three unrelated responsibilities. Tight coupling: it creates concrete Database and EmailService objects itself. Fix: split into ShoppingCart, an email/notification service and a data-access class, and let ShoppingCart depend on interfaces that are passed in.

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.

NOTES

- 1 **Separation of interface and implementation:** design each module so that a **stable interface** (what it offers) is identified and kept separate from its implementation (how it does it).
- 2 It is **not** the same as encapsulation. Encapsulation hides the implementation details; this principle **separates** them, so the interface exists on its own.
- 3 The payoff: **different implementations** of the same interface can be provided and swapped (a `PaymentGateway` interface with Visa and PayPal implementations) without affecting the code that uses the interface.

EXAM ANGLE

Prediction

Q1

How does separation of interface and implementation differ from encapsulation?

Answer: Encapsulation hides how a module works. Separation of interface and implementation keeps the interface as a separate, stable element so that several implementations can be provided and swapped without affecting its users.

Software Design Fundamentals (Principles)

➤ Design Principle #7: **Completeness**

- ✓ Measures how well design units provide the required services to 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 services is missing, the class is considered 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.

NOTES

- 1 **Completeness** measures whether a design unit provides **all** the services it needs to achieve its intent: no less.
- 2 Example: a communication class is complete only if it can establish and terminate a connection and send and receive messages. Missing any one makes it incomplete.
- 3 **Sufficiency** measures whether the unit provides **only** the services sufficient for its intent: no more. Completeness asks “is anything missing?”, sufficiency asks “is anything extra?”.
- 4 Example of insufficiency in the opposite sense: a messaging module that also manages user profiles has taken on a service outside its purpose. The banking example on the slide shows completeness: the transfer and history requirements must both appear in the design.

EXAM ANGLE

Past exam · Final 252 · Q1 MCQ 10

Q1

Which design principle measures how well a unit provides only the services necessary for its intent, ensuring no extra, unnecessary features are included?

- A. Completeness.
- B. Sufficiency.
- C. Cohesion.
- D. Modularization.

Answer: B. Sufficiency. — Completeness = no less; sufficiency = no more.

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

NOTES

- 1 Modularization = decompose into separate modules.
- 2 Abstraction = separate the **what** from the **how**.
- 3 Encapsulation = information hiding.
- 4 Coupling = dependency **between** modules; cohesion = how related the tasks **in** a module are. Separation of interface and implementation = one interface, possibly several implementations.
- 5 Completeness = every required service is there; sufficiency = only the services needed. A one-line definition of each is a likely short question.

EXAM ANGLE

Prediction

Q1

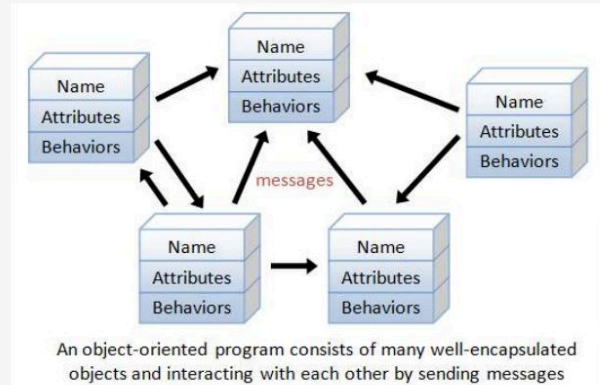
Match each principle to its one-line meaning: modularization, abstraction, encapsulation, coupling, cohesion, completeness.

Answer: Modularization: decompose into modules. Abstraction: separate what from how. Encapsulation: information hiding. Coupling: dependency between modules. Cohesion: relatedness of the tasks in a module. Completeness: all required services are provided.

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



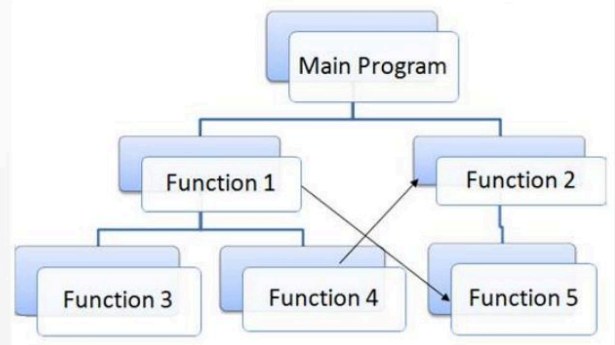
NOTES

- 1 The **object-oriented** strategy expresses a system or component as objects and the connections between them, so the decomposition is by objects rather than by functions.
- 2 Every object has a unique **identity** (a name), a **state** (its attribute values) and well-defined **behaviour** (its operations).
- 3 It supports inheritance and polymorphism and suits object-oriented languages such as Java. Chapters 5–7 use this strategy throughout.

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.

NOTES

- 1 The **structured (functional)** strategy decomposes a system into single-purpose, independent modules using an iterative **top-down** approach: start with the whole function and keep breaking it down.
- 2 It focuses on the functions the system must provide, their decomposition into smaller functions, and the modules that implement them.
- 3 It is **inappropriate for object-oriented** languages because it is centred on functions, not objects. It fits procedural languages such as C.

EXAM ANGLE

Prediction

Q1

Compare the object-oriented and structured design strategies.

Answer: Object-oriented: decomposes the system into objects (identity, state, behaviour) and their connections; supports inheritance and polymorphism; suits OO languages. Structured: decomposes top-down into single-purpose functional modules; centred on functions; inappropriate for OO languages.

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.

NOTES

- 1 Minimise complexity:** design is fundamentally about reducing complexity, and every decision should take that into account.
- 2** When two design options compete and both meet the requirements, choose the one that is less complex (fewer parts, fewer dependencies).
- 3 Design for change:** software will change, so design with extension in mind. Several techniques (modularity, interfaces, the principles of chapter 5) help achieve this.

EXAM ANGLE

Past exam · Quiz 1 · 261 · Q10 (fill in)

Q1

Fill in the blank: When two designs satisfy the same requirements, the design with fewer unnecessary dependencies is generally preferred because software design should minimize ____.

Answer: Complexity.

*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

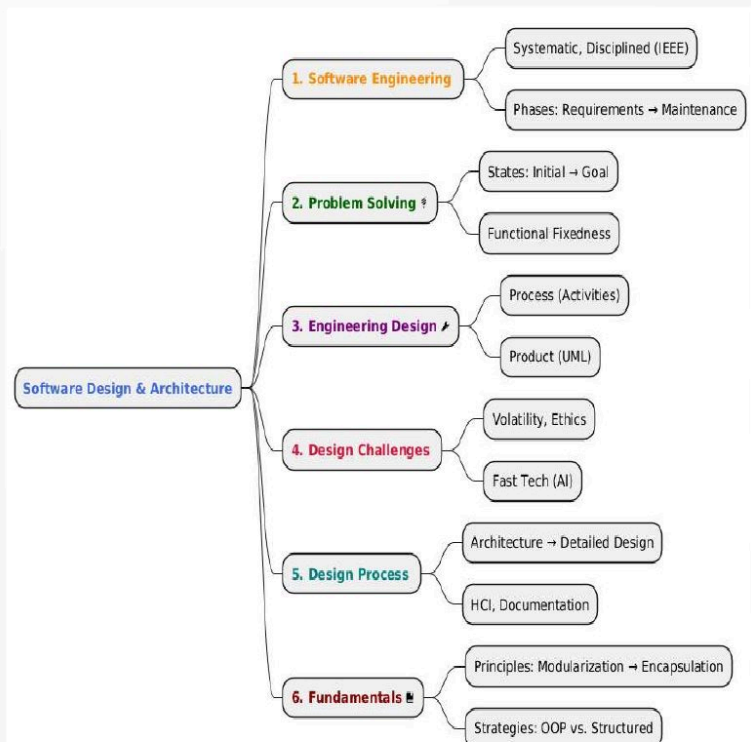
* This is required at the **end of each chapter**

NOTES

- 1 The exercise: draw a mind-map of the chapter's topics and how they relate, and upload it to the Moodle forum. The value is in the connecting lines, not in listing terms.
- 2 It is required at the end of every chapter, so it doubles as a revision sheet before each exam.

SLIDE 70 / 73 Mind-mapping tools

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



NOTES

- 1 Any mind-mapping tool works: XMind, Miro or draw.io. Put the central topic (software design) in the middle and branch to problem solving, process, challenges, principles and strategies.
-

SLIDE 71 / 73 Summary

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

NOTES

- 1 The summary repeats the seven topics of slide 3. A good check: for each one, can you give the definition and one example without looking?
 - 2 The next chapter starts the first major design activity: software 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

NOTES

- 1 The detailed summary, first half: SE definitions and phases, problem solving (states, functional fixedness, the holistic approach), design as process and product, why study design.
- 2 The five challenges: requirements volatility, inconsistent processes, fast-changing technology, managing design influences, ethics.

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

NOTES

- 1 Second half: the design process, with its major activities (architecture, detailed design) and supporting activities (construction design, HCI design, documentation, management).
 - 2 The eight principles, from modularization to sufficiency.
 - 3 The two strategies (object-oriented, structured) and the two considerations (minimise complexity, design for change).
-

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#).
Notes and questions are study aids. Made by Shoug Alomran.