CS2SD Software Systems Design

Introduction to Object-Oriented Design
aided by Methodology of UML

Lecture Notes

Table of Contents

Session 1: What is Software Systems Design About?

1.1 Introduction

Software systems design is the disciplined process of defining the architecture, components, interfaces, and behaviour of a system so that it satisfies specified requirements. Unlike ad-hoc programming, systems design takes a holistic view: it considers not only what a piece of software should do, but how it fits within a larger organisational, technical, and human context. In the CS2SD module we adopt the Unified Modelling Language (UML) as the principal modelling notation and couple it with object-oriented principles to bridge the gap between user needs and working software.

This first session sets the stage by exploring the fundamental question every software engineer must confront: how do we ensure that the systems we build truly serve the people who use them? We examine the tension between user expectations and technical solutions, introduce the idea of 'building the right system' versus merely 'building the system right', and survey the principal challenges that make software systems design both intellectually demanding and practically important.

1.2 User Needs vs Solutions: The Requirements Gap

A recurring theme in software engineering is the mismatch between what users truly need and what developers ultimately deliver. This gap arises because users often express their needs in domain-specific, informal language, while developers think in terms of data structures, algorithms, and APIs. The classic illustration of this problem is the 'tree swing' cartoon, in which every stakeholder in a project interprets the requirement differently, and the final product bears little resemblance to the customer's original vision.

The requirements gap can be attributed to several factors. First, users may not fully understand their own needs until they see a working prototype. Second, requirements are often ambiguous: natural-language statements like 'the system should be fast' lack the precision needed for implementation. Third, requirements evolve over time as the business environment changes, new regulations appear, or user demographics shift. Software systems design provides structured techniques — notably use-case analysis and iterative prototyping — to progressively narrow this gap.

Understanding the distinction between what a user really wants and what is eventually built is a cornerstone of effective systems design. A successful software project is one that closes this requirements gap through disciplined elicitation, analysis, and validation of requirements at every stage of development.

1.3 Building the System Right AND Building the Right System

'Building the system right' refers to verification — ensuring that the software conforms to its specification, is free of defects, and meets quality standards such as performance, reliability, and security. In contrast, 'building the right system' refers to validation — confirming that the software actually addresses the user's real-world problem. A system can be impeccably engineered yet utterly useless if it solves the wrong problem.

The dual challenge for software engineers is therefore twofold. On the one hand, they must apply sound engineering practices — structured design, code reviews, testing, and continuous integration — to build systems that are robust and maintainable. On the other hand, they must engage deeply with stakeholders to understand the problem domain, validate assumptions early, and remain responsive to changing needs. The UML methodology introduced in this module provides a framework for tackling both challenges simultaneously: its diagrams serve as communication tools with stakeholders (validation) and as precise specifications for developers (verification).

1.4 Challenges in Building the Right Systems

Designing software systems that genuinely meet user needs is fraught with challenges. The following subsections explore the principal difficulties, drawing on themes introduced in the lecture slides.

1.4.1 Capturing Accurate Requirements

Requirements capture is the foundation of any software project. It involves understanding both functional requirements (what the system should do) and non-functional requirements (qualities such as performance, usability, and security). A key difficulty is that user needs are often tacit: users know what they want only in broad terms and may not articulate constraints or edge cases. Moreover, requirements evolve as the business environment changes and as users interact with early prototypes.

Effective requirements engineering combines several techniques: interviews and workshops with stakeholders, observation of existing workflows, analysis of competing systems, and formal specification methods such as use-case modelling. The goal is to produce a requirements specification that is complete, consistent, unambiguous, and verifiable — while also accounting for evolving user needs and environmental constraints.

1.4.2 Designing for Performance and Scalability

A system that works well for ten users may collapse under the load of ten thousand. Designing for performance and scalability requires careful attention to system architecture from the outset. Common pitfalls include bottlenecks in database queries, synchronous processing where asynchronous approaches would be more appropriate, and monolithic architectures that cannot be scaled independently.

Architects must ensure responsiveness under variable loads and integrate efficient data processing strategies. Increasingly, energy-aware computation is also a design consideration: processing that is needlessly resource-intensive not only degrades performance but also increases the environmental footprint of the software.

1.4.3 Ensuring Maintainability and Modularity

Software systems typically spend far more of their lifetime being maintained than being initially developed. Tightly coupled components dramatically increase the cost and risk of changes, because a modification in one module can have unexpected ripple effects elsewhere. Good design promotes loose coupling and high cohesion: each module should have a single, well-defined responsibility and interact with other modules through narrow, stable interfaces.

Established design patterns — such as the microservices architecture at the system level and design patterns like Strategy, Observer, and Factory at the class level — provide proven solutions for achieving modularity. The object-oriented principles of encapsulation and abstraction, discussed in Session 2, are the intellectual foundation on which these patterns rest.

1.4.4 Integrating Sustainability and Ethical Considerations

Modern software design increasingly recognises sustainability and ethics as first-class concerns. Sustainability encompasses minimising energy consumption and carbon footprint throughout a system's lifecycle — from data-centre power usage to the energy cost of training and running AI models. Ethical design demands attention to privacy (data minimisation, informed consent), fairness (avoiding algorithmic bias), and inclusivity (accessible interfaces).

Designers must balance short-term performance goals with long-term environmental impact, ensuring that software serves not only its immediate users but also the wider community and planet.

1.4.5 Facilitating Critical Analysis and Iterative Improvement

A design is rarely correct on the first attempt. Effective systems design therefore builds in mechanisms for analysis, critique, and refinement. Feedback loops — from user testing, automated analysis tools, and AI-assisted review — enable designers to identify weaknesses early and iterate towards a better solution.

Aligning conceptual understanding with practical implementation is another challenge: a UML model may look elegant on paper yet prove impractical when translated to code. Bridging this gap requires experience, peer review, and a willingness to revise designs in light of implementation feedback.

1.4.6 Managing Complexity

Large-scale software systems contain thousands of interacting components, each with its own state, behaviour, and dependencies. Managing this complexity is arguably the central intellectual challenge of software engineering. Strategies include decomposition (breaking a system into manageable subsystems), abstraction (hiding unnecessary detail), and the use of modelling languages such as UML to maintain clarity in documentation.

Coordination across teams, domains, and technologies adds a further layer of complexity. A well-defined design methodology — like the five-step UML methodology introduced in Session 3 — provides a shared vocabulary and process that helps diverse teams collaborate effectively.

1.5 Systems Thinking Approach for Problem-Solving

1.5.1 Holistic Systems View

Systems thinking is an approach to problem-solving that views a system as a whole rather than a collection of isolated parts. In the context of software design, this means understanding how components influence one another within a specific domain context. Rather than optimising individual modules in isolation, systems thinking considers the purpose of the overall system, the interdependencies between its parts, the collaborations that occur during operation, and the interactions with external stakeholders.

A hallmark of systems thinking is the feedback loop: design decisions have consequences that feed back into the system, potentially affecting performance, sustainability, and user satisfaction. By modelling these feedback loops explicitly, designers can anticipate unintended side-effects and design for resilience. Systems thinking also emphasises leveraging agility — making targeted changes that achieve optimal outcomes while maintaining the completeness and coherence of the system.

Furthermore, systems thinking supports the optimisation of AI integration into existing or new business applications. As organisations increasingly adopt AI capabilities, understanding the system-level implications — data flows, ethical guardrails, performance trade-offs — becomes essential for productive and responsible deployment.

1.5.2 Applying Systems Thinking to Software Development

When applied to software development, systems thinking encourages developers to focus on smaller subsystems (components) while keeping the 'whole picture' in view. This dual perspective is essential for avoiding the trap of local optimisation at the expense of global coherence.

In practice, this means designing systems that can be configured, re-configured, and de-configured in response to innovation and competitive pressure. Modern architectural approaches — particularly microservices and API-driven design — embody this philosophy. In a microservices architecture, small, independent computing services communicate through well-defined APIs, enabling teams to develop, deploy, and scale services independently.

The benefits are significant: increased extensibility (new services can be added without disrupting existing ones), cost-effective development (teams work in parallel on different services), and resilience (failure in one service does not cascade to the entire system). However, microservices also introduce complexity in areas such as service discovery, distributed data management, and network reliability — reinforcing the need for the systematic, model-driven approach that UML provides.

1.6 Using LLM and Conceptual Prompts to Aid Design

The emergence of large language models (LLMs) — such as OpenAI's ChatGPT, Google Gemini, and Microsoft Copilot — offers new opportunities for software systems design. In this module, AI agents are used not as a replacement for critical thinking but as tools to empower learning and improve design quality.

The aims of using AI to support systems design include:

- Enhance knowledge acquisition and research — LLMs can rapidly summarise best practices, surface relevant design patterns, and explain unfamiliar concepts, supporting the learning of software systems design principles.
- Enable critical analysis and reasoning — by prompting an LLM with design scenarios, students can evaluate designs for performance, sustainability, and maintainability, developing their analytical skills.
- Provide iterative feedback — LLMs can review UML diagrams (described textually), suggest improvements, and promote conceptual understanding through continuous reflection and refinement.
- Increase productivity and quality — AI assistance helps ensure that design outcomes are complete, adequate, and well-structured.
- Support long-term skills development — rather than creating dependency, the aim is to develop capabilities for independent, sustainable learning and professional growth.

The key phrase is 'conceptual prompts': carefully crafted prompts that engage the LLM at the level of design concepts rather than asking it to generate code directly. This approach ensures that students develop genuine understanding rather than outsourcing their thinking to a machine.

Session 2: Key Object-Oriented Concepts Adopted by UML

2.1 Introduction to the Object-Oriented Paradigm

The object-oriented (OO) paradigm models software as a collection of interacting objects, each of which encapsulates data and behaviour. This approach mirrors the way humans naturally conceptualise the world: we think in terms of things (objects) that have properties (attributes) and can perform actions (methods). The four pillars of object orientation — abstraction, encapsulation, inheritance, and polymorphism — provide the conceptual foundation for the Unified Modelling Language (UML) and for modern programming languages such as Java, C++, and Python.

Understanding these principles is essential for software systems design because they directly influence how we decompose a problem, organise code, manage complexity, and prepare systems for future change. Each principle is explored in detail below, together with illustrative examples.

2.2 Abstraction

Key Concept — Abstraction

Abstraction is the process of focusing on the essential features of an entity while hiding the background details and complexities. It allows designers to capture the most important aspects of a system without being overwhelmed by implementation minutiae.

In object-oriented design, abstraction means defining a class that represents a real-world concept at an appropriate level of detail. For example, consider a Vehicle class in a fleet-management system. The essential attributes might include registration number, current location, and fuel level; the essential behaviours might include start(), stop(), and reportPosition(). Details such as the precise engine-management firmware or the colour of the dashboard are irrelevant at this level and are omitted.

Abstraction enables designers to manage complexity by working at the right level of detail for each stage of design. At the architectural level, entire subsystems can be abstracted as single components; at the detailed design level, individual classes and methods are specified. UML class diagrams are the primary tool for expressing abstraction: they show classes, their attributes and operations, and the relationships between them, all at a chosen level of abstraction.

2.3 Encapsulation

Key Concept — Encapsulation

Encapsulation is the mechanism for bundling data (attributes) and methods (functions or procedures) into a single unit (a class), while hiding the internal state from the outside world and restricting direct access to data or methods across connected units.

Encapsulation enforces the principle of information hiding. A class exposes a public interface — a set of methods through which other objects can interact with it — while keeping its internal data private. This separation has two major benefits. First, it reduces coupling: because external code depends only on the interface, the internal implementation can change without affecting the rest of the system. Second, it protects data integrity: by controlling access through methods (getters and setters), the class can enforce invariants — for example, ensuring that a bank account balance never becomes negative.

Consider a BankAccount class with a private attribute balance. External code cannot modify balance directly; instead, it must call deposit(amount) or withdraw(amount), which include validation logic. This pattern is fundamental to building reliable, maintainable systems.

2.4 Inheritance (Specialisation and Generalisation)

Key Concept — Inheritance

Inheritance is the mechanism by which one class (the subclass or child) inherits the attributes and methods of another class (the superclass or parent). It promotes code reuse through two complementary relationships: specialisation (a subclass adds or overrides features specific to a particular kind of entity) and generalisation (a superclass represents common features shared by a set of related classes).

For example, consider a Shape superclass with attributes such as colour and position, and a method draw(). Subclasses such as Circle, Rectangle, and Triangle inherit these common features but add specialised attributes (e.g., radius for Circle) and override the draw() method to render themselves correctly. The superclass captures the generalisation — all shapes have a colour, a position, and can be drawn — while each subclass provides the specialisation.

Inheritance hierarchies are depicted in UML class diagrams using a solid line with a hollow arrowhead pointing from the subclass to the superclass. When designing an inheritance hierarchy, it is important to ensure that the 'is-a' relationship genuinely holds: a Circle 'is a' Shape, which is correct; but modelling a Stack as a subclass of ArrayList would be inappropriate because a Stack is not conceptually a kind of list — it merely uses one internally.

2.5 Polymorphism

Key Concept — Polymorphism

Polymorphism (from the Greek for 'many forms') allows objects of different classes to respond to the same method call in class-specific ways. It enhances system flexibility through dynamic method dispatch at runtime, meaning the correct method implementation is selected based on the actual type of the object, not the declared type of the reference.

Continuing the Shape example: if a list of Shape objects contains a Circle, a Rectangle, and a Triangle, calling draw() on each element will invoke the appropriate subclass method — Circle.draw(), Rectangle.draw(), or Triangle.draw() — without the calling code needing to know the specific type. This is the essence of polymorphism: one interface, multiple implementations.

Polymorphism enhances both flexibility and extensibility. To add a new shape — say, Pentagon — one simply creates a new subclass with its own draw() method; no existing code needs to change. This adherence to the Open/Closed Principle (open for extension, closed for modification) is a hallmark of well-designed object-oriented systems.

2.6 UML in Object-Oriented Design

2.6.1 What is UML?

The Unified Modelling Language (UML) is a standardised, general-purpose modelling language used in software engineering. It provides a pictorial language for creating systems design 'blueprints' — visual representations of the artefacts of a software system. UML supports four key activities:

- Specifying — precisely defining the structure and behaviour of a system.
- Visualising — creating diagrams that make complex systems comprehensible.
- Constructing — using UML models as the basis for generating code.
- Documenting — recording design decisions for future reference and communication.

2.6.2 UML as Realisation of Systems Thinking

UML embodies the systems thinking approach discussed in Session 1. Each UML diagram type offers a different perspective on the system — structural, behavioural, or interactional — enabling designers to consider the system holistically while focusing on specific aspects as needed. Use-case diagrams capture the system's purpose from the stakeholder's perspective; class diagrams model its data structure; sequence diagrams trace functional workflows. Together, they provide a multi-faceted view that no single diagram could offer.

2.6.3 From UML to Implementation

One of UML's most practical benefits is that well-formed UML models can be translated relatively straightforwardly into implementation code using object-oriented programming languages such as Java, C++, or Python. A UML class diagram, for instance, maps directly to class definitions: each class box becomes a class declaration, attributes become fields, and operations become methods. Relationships (association, aggregation, inheritance) translate to fields, collections, and extends/implements keywords respectively.

This tight coupling between model and code is a major advantage: it means that the UML diagrams produced during design remain relevant throughout implementation and maintenance, serving as living documentation of the system's architecture.

Session 3: UML Methodology and UML Techniques

3.1 Recap: Software Architecture (Three-Tier)

Modern software systems are commonly structured using a three-tier (or n-tier) architecture that separates concerns into distinct layers. This separation improves maintainability, testability, and scalability by ensuring that changes in one layer do not ripple uncontrollably into others. The three standard tiers are summarised in the table below.

Tier	Responsibility	Typical Technologies
Presentation Tier (User Interface)	Handles all user interaction: displaying information and capturing user input. Concerned with usability, accessibility, and responsive design.	HTML/CSS/JavaScript, React, Angular, Mobile UI frameworks
Application Tier (Business Logic)	Implements the core business rules and processing logic. Validates data, enforces workflows, coordinates transactions.	Java EE, Spring, .NET, Django, Node.js
Data Tier (Persistence)	Manages data storage, retrieval, and integrity. Provides an interface for querying and updating persistent data.	SQL databases, NoSQL (MongoDB), Cloud storage services

Understanding this architecture is critical for UML modelling because the domain aspects that we identify during analysis (see Section 3.2) map onto these tiers. Use-case diagrams and activity diagrams primarily address the presentation and application tiers; class diagrams span the application and data tiers; sequence diagrams trace interactions across all three.

3.2 Determining Domain Aspects for Design

Before selecting specific UML techniques, designers must identify the key domain aspects that the system must address. Each aspect corresponds to a different modelling concern and guides the choice of UML diagram. The table below summarises the main domain aspects and their design implications.

Domain Aspect	Design Concern	Primary UML Technique
Functional requirements	What the system must do; actors and their goals	Use Case Diagrams
Business processes / workflows	How tasks are performed; sequencing, decisions, parallelism	Activity Diagrams

Data structures / entities	What information the system manages; relationships between entities	Class Diagrams
Object lifecycle / state	How an individual object's state changes in response to events	State Transition Diagrams
Inter-object communication	How objects collaborate to fulfil a use case at runtime	Sequence Diagrams

3.3 UML Techniques for System Modelling

UML defines a rich set of diagram types. In this module, we focus on five core techniques that together cover the essential perspectives of a software system: structure, behaviour, and interaction.

3.3.1 Use Case Diagrams

Use case diagrams model what the system does and who interacts with it. They capture functional requirements by identifying actors (users or external systems) and use cases (discrete pieces of functionality that deliver value to an actor). Use case diagrams are typically the first UML artefact produced in a project because they establish the scope of the system and serve as the basis for all subsequent modelling.

A use case diagram consists of a system boundary (a rectangle), actors (stick figures) positioned outside the boundary, and use cases (ovals) inside. Lines connect actors to the use cases they participate in. Relationships between use cases — such as 'include' (mandatory sub-functionality) and 'extend' (optional behaviour) — are shown with dashed arrows.

3.3.2 Activity Diagrams

Activity diagrams model how processes are carried out. They are similar to traditional flowcharts but include support for concurrent activities (fork/join bars) and swim lanes that allocate activities to different actors or system components. Activity diagrams are particularly useful for modelling business workflows and the detailed flow of events within a use case.

Key elements include: initial node (filled circle), final node (circle with inner filled circle), action nodes (rounded rectangles), decision nodes (diamonds), and flow edges (arrows). By detailing the scoped flow of events, activity diagrams bridge the gap between high-level use cases and the detailed logic required for implementation.

3.3.3 Class Diagrams

Class diagrams model the static data structure of a system. They are the backbone of object-oriented design, showing classes (with their attributes and operations), and the relationships between them: association, aggregation, composition, and inheritance. Class diagrams answer the question 'what data does the system manage, and how are the entities related?'

Each class is represented as a rectangle divided into three compartments: the class name (top), attributes (middle), and operations (bottom). Visibility markers (+ for public, - for private, # for protected) indicate the access level of each member. Multiplicities on associations (e.g., 1..*, 0..1) specify how many instances of one class can be related to instances of another.

3.3.4 State Transition Diagrams

State transition diagrams (also called state machine diagrams or statechart diagrams) model the dynamics of an individual object — specifically, the states it can be in and the events that cause transitions between states. They are particularly useful for objects with complex lifecycles, such as an Order that progresses through states like Created, Confirmed, Shipped, Delivered, and Cancelled.

The diagram consists of states (rounded rectangles), transitions (arrows labelled with triggering events and optional guard conditions), an initial pseudo-state (filled circle), and one or more final states. Understanding the state behaviour of key domain objects is essential for ensuring that the system handles all possible scenarios correctly.

3.3.5 Sequence Diagrams

Sequence diagrams model the functional workflow of a system by showing how objects interact over time to fulfil a specific use case. They depict objects (or actors) as vertical lifelines and messages between them as horizontal arrows, arranged chronologically from top to bottom.

Sequence diagrams are invaluable for understanding the runtime behaviour of a system: they reveal the order of method calls, the data passed between objects, and the conditions under which certain paths are taken. Combined fragments (such as alt for alternatives and loop for iteration) add expressive power for modelling conditional and repetitive behaviour.

The following table provides a summary of the five UML techniques, what each models, and the primary output or artefact it produces.

UML Technique	What It Models	Primary Output
Use Case Diagram	What the system does and who interacts with it	System scope and functional requirements specification
Activity Diagram	How processes and workflows are performed	Detailed flow of events for each use case
Class Diagram	Static data structure and entity relationships	Domain model / design-level class structure
State Transition Diagram	Dynamics of an individual object's lifecycle	State model for key domain objects
Sequence Diagram	Runtime interactions between objects over time	Functional workflow for each use case scenario

3.4 The UML Methodology: A Five-Step Process

The five UML techniques introduced above are not used in isolation; they form a coherent, step-by-step methodology for designing software systems. The five steps are:

- Step 1 — Use Case Diagrams: Adopt the system scope; identify actors, use cases, and their relationships. Extract events and conditions from the problem description.
- Step 2 — Activity Diagrams: Detail the scoped flow of events for each use case, capturing sequencing, decisions, and parallel activities.
- Step 3 — Class Diagrams: Structure the scoped data by identifying classes, attributes, operations, and relationships.
- Step 4 — State Transition Diagrams: Model the behaviour of key objects as they respond to events during processing.
- Step 5 — Sequence Diagrams: Transform the flow of events into functional workflows, showing how objects collaborate to fulfil each use case.

This methodology proceeds from the general to the specific, and from the user-facing view to the implementation-facing view. Each step builds on the outputs of the previous one: use cases drive activity diagrams, which inform class identification, which feeds into state and sequence modelling. The result is a comprehensive set of design artefacts that can be validated against stakeholder requirements and translated into implementation code.

3.5 The Problem-Solving Process

The UML methodology operates within a broader problem-solving process. This process begins with stakeholders describing a problem in domain-specific terms. The system analyst — a key role in software design — translates this problem description into a structured requirements specification using conceptual thinking: reasoning about the problem at a level of abstraction above code.

The process is problem-driven (starting from the problem to be solved rather than from a technology to be applied) and requirement-driven (every design decision traces back to a stated requirement). Validation feedback loops ensure that the design remains aligned with stakeholder needs throughout the process.

3.6 Assessable Learning Outcomes

The CS2SD module assesses understanding through two main assignments that correspond to the UML methodology steps:

- Group assignment — typically covering the earlier methodology steps (use case diagrams, activity diagrams, and class diagrams), where teams collaborate to produce a coherent design for a given problem.
- Individual assignment — typically covering the later steps (state transition diagrams and sequence diagrams), where each student demonstrates their individual ability to model object behaviour and interactions.

Both assignments require students to demonstrate not only technical proficiency with UML notation but also critical analysis — evaluating design choices, justifying decisions, and reflecting on the quality and completeness of their models.

3.7 What to Do Next

Following this introductory lecture, students should:

- Review the key object-oriented concepts (abstraction, encapsulation, inheritance, polymorphism) and ensure a solid understanding before moving to UML modelling.
- Carry out the dedicated practical tasks and instructions available on Blackboard (BB), which provide hands-on experience with UML diagram creation.
- Conduct the dedicated formative assessment on Blackboard to test and enhance understanding of OO design for software systems.
- Critically use conceptual prompts with LLMs (e.g., ChatGPT, Gemini, Copilot) to improve and refine systems design quality and productivity — always engaging critically with the AI output rather than accepting it uncritically.

References and Further Reading

- Fowler, M. (2004) UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd edn. Boston: Addison-Wesley.
- Larman, C. (2004) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd edn. Upper Saddle River, NJ: Prentice Hall.
- Pressman, R.S. and Maxim, B.R. (2020) Software Engineering: A Practitioner's Approach. 9th edn. New York: McGraw-Hill.
- Sommerville, I. (2016) Software Engineering. 10th edn. Harlow: Pearson.
- Booch, G., Rumbaugh, J. and Jacobson, I. (2005) The Unified Modeling Language User Guide. 2nd edn. Boston: Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.
- Object Management Group (2017) Unified Modeling Language Specification, Version 2.5.1. Available at: https://www.omg.org/spec/UML/ (Accessed: February 2026).