Software Systems Design — Terminology Sheet

CS2SD — Introduction to Object-Oriented Design (Lecture 1)

1. Core Object-Oriented Concepts

Term	Definition	Source
Object-Oriented Paradigm	A software modelling approach that represents a system as a collection of interacting objects, each encapsulating data (attributes) and behaviour (methods). It mirrors how humans conceptualise the world in terms of things with properties that can perform actions.	Both
Abstraction	The process of focusing on the essential features of an entity while hiding background details and complexities. In OO design, it means defining classes at an appropriate level of detail, capturing only the most important aspects of the system.	Both
Encapsulation	The mechanism for bundling data (attributes) and methods (functions/procedures) into a single unit (a class), while hiding internal state from the outside world and restricting direct access to data or methods across connected units. Enforces the principle of information hiding.	Both
Inheritance	The mechanism by which one class (subclass/child) inherits the attributes and methods of another class (superclass/parent). Promotes code reuse through the complementary relationships of specialisation and generalisation.	Both
Specialisation	A relationship in which a subclass possesses specialised characteristics particular to it, adding or overriding features from its superclass. E.g., Circle specialises Shape by adding a radius attribute.	Both
Generalisation	A relationship that represents common characteristics extracted from a set of related classes into a shared superclass. E.g., Shape generalises Circle, Rectangle, and Triangle.	Both
Polymorphism	The ability of objects of different classes to respond to the same method call in class-specific ways. Uses dynamic method dispatch at runtime so the correct implementation is selected based on the actual object type, not the declared reference type.	Both
Dynamic Method Dispatch	The runtime mechanism that selects the correct method implementation based on the actual object type, enabling polymorphic behaviour. Central to flexibility and extensibility in OO systems.	PDF
Open/Closed Principle	A design principle stating that software entities should be open for extension but closed for modification.	PDF

	Polymorphism supports this by allowing new subclasses to be added without changing existing code.	
Information Hiding	The principle that a class exposes a public interface while keeping its internal data private. Reduces coupling and protects data integrity by controlling access through methods (getters/setters).	PDF
Class	A blueprint or template that defines a set of attributes (data) and operations (methods) shared by all objects of that type. Represented in UML as a rectangle with three compartments: name, attributes, and operations.	Both
Object	A specific instance of a class, possessing concrete attribute values and capable of performing the operations defined by its class. Objects interact by sending messages (method calls) to one another.	Both
Attribute	A named property of a class that holds data describing the state of its objects. E.g., registrationNumber, currentLocation, fuelLevel in a Vehicle class.	Both
Method (Operation)	A function or procedure defined within a class that specifies a behaviour the object can perform. E.g., start(), stop(), draw(). Also referred to as an operation in UML.	Both
Superclass (Parent Class)	A class from which other classes (subclasses) inherit attributes and methods. Represents the general concept in a generalisation hierarchy.	Both
Subclass (Child Class)	A class that inherits from a superclass and may add or override attributes and methods to provide specialised behaviour.	Both

2. Software Systems Design Fundamentals

Term	Definition	Source
Software Systems Design	The disciplined process of defining the architecture, components, interfaces, and behaviour of a system so that it satisfies specified requirements. Takes a holistic view considering organisational, technical, and human context.	Both
Requirements Gap	The mismatch between what users truly need and what developers ultimately deliver, arising because users express needs in informal domain-specific language while developers think in terms of data structures, algorithms, and APIs.	Both
Functional Requirements	Specifications of what the system should do — the features, capabilities, and services it must provide to its users. Captured primarily through use-case modelling.	Both
Non-Functional Requirements	Quality attributes the system must exhibit, such as performance, usability, security, scalability, reliability, and sustainability. Constrain how the system delivers its	Both

	functional requirements.	
Verification	The process of ensuring that the software conforms to its specification, is free of defects, and meets quality standards ("building the system right").	PDF
Validation	The process of confirming that the software actually addresses the user's real-world problem ("building the right system"). Often involves stakeholder feedback.	PDF
Requirements Engineering	The systematic process of eliciting, analysing, specifying, and validating requirements. Combines interviews, workshops, observation, competitor analysis, and formal specification methods.	PDF
Scalability	The ability of a system to handle growing amounts of work or to be enlarged to accommodate growth. Requires careful architectural attention to avoid bottlenecks.	Both
Maintainability	The ease with which a system can be modified to correct faults, improve performance, or adapt to a changed environment. Promoted by loose coupling and high cohesion.	Both
Modularity	The degree to which a system is composed of discrete, self-contained modules with well-defined interfaces. Supports reusability, independent development, and easier maintenance.	Both
Loose Coupling	A design quality where modules depend on each other through narrow, stable interfaces, minimising the ripple effect of changes. Opposite of tight coupling.	PDF
High Cohesion	A design quality where each module has a single, well-defined responsibility. All elements within the module are closely related and work together toward that responsibility.	PDF
Sustainability (in Software Design)	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.	Both
Ethical Design	Designing systems that respect privacy (data minimisation, informed consent), fairness (avoiding algorithmic bias), and inclusivity (accessible interfaces).	Both
Feedback Loop	A mechanism in which design decisions produce consequences that feed back into the system, affecting performance, sustainability, and user satisfaction. Used to identify weaknesses and iterate toward better solutions.	Both

3. Systems Thinking and Architecture

Term	Definition	Source

Systems Thinking	An approach to problem-solving that views a system as a whole rather than a collection of isolated parts. Considers purpose, interdependencies, collaborations, interactions with stakeholders, and feedback loops within a specific domain context.	Both
Holistic Systems View	Understanding the system as a complete entity with its purpose, interdependencies, collaborations, and stakeholder interactions — rather than optimising individual modules in isolation.	Both
Three-Tier Architecture	A software architecture pattern that separates a system into three layers: Presentation Tier (UI), Application Tier (business logic), and Data Tier (persistence). Improves maintainability, testability, and scalability.	PDF
Presentation Tier	The layer that handles all user interaction: displaying information and capturing user input. Concerned with usability, accessibility, and responsive design.	PDF
Application Tier (Business Logic)	The layer that implements core business rules and processing logic. Validates data, enforces workflows, and coordinates transactions.	PDF
Data Tier (Persistence)	The layer that manages data storage, retrieval, and integrity. Provides an interface for querying and updating persistent data.	PDF
Microservices Architecture	An architectural style where small, independent computing services communicate through well-defined APIs. Enables independent development, deployment, and scaling. Increases extensibility but introduces complexity in service discovery and distributed data management.	Both
API-Driven Design	An approach where system components communicate through Application Programming Interfaces (APIs), enabling seamless integration, reconfiguration, and extensibility.	Both
Decomposition	The strategy of breaking a large system into smaller, manageable subsystems or components. A key technique for managing complexity.	PDF
Design Patterns	Proven, reusable solutions to commonly occurring problems in software design. Examples include Strategy, Observer, and Factory at the class level, and microservices at the system level.	PDF

4. Unified Modelling Language (UML)

Term	Definition	Source
UML (Unified Modelling Language)	A standardised, general-purpose pictorial modelling language used in software engineering to create systems design blueprints. Supports specifying, visualising, constructing, and documenting the artefacts of a software	Both

	system.	
UML Methodology	A five-step design process: (1) Use Case Diagrams, (2) Activity Diagrams, (3) Class Diagrams, (4) State Transition Diagrams, (5) Sequence Diagrams. Proceeds from general to specific, from user-facing to implementation-facing views.	Both
Use Case Diagram	Models what the system does and who interacts with it. Identifies actors (users/external systems) and use cases (discrete functionality). Establishes system scope and functional requirements. Contains a system boundary, actors (stick figures), and use cases (ovals).	Both
Actor	A user, person, or external system that interacts with the system being designed. Represented as a stick figure positioned outside the system boundary in a use case diagram.	Both
Use Case	A discrete piece of functionality that delivers value to an actor. Represented as an oval inside the system boundary. Connected to actors via lines.	Both
Include Relationship	A relationship between use cases indicating mandatory sub-functionality that is always executed as part of the base use case. Shown with a dashed arrow labelled «include».	PDF
Extend Relationship	A relationship between use cases indicating optional behaviour that may be triggered under certain conditions. Shown with a dashed arrow labelled «extend».	PDF
Activity Diagram	Models how processes and workflows are carried out. Similar to flowcharts but supports concurrent activities (fork/join bars) and swim lanes. Bridges the gap between high-level use cases and detailed implementation logic.	Both
Swim Lane	A partition in an activity diagram that allocates activities to different actors or system components, showing who is responsible for each action.	PDF
Fork/Join Bar	Elements in an activity diagram representing the splitting (fork) and synchronisation (join) of concurrent flows of activity.	PDF
Class Diagram	Models the static data structure of a system. Shows classes (with attributes and operations) and relationships: association, aggregation, composition, and inheritance. The backbone of OO design.	Both
Association	A structural relationship between classes indicating that instances of one class are connected to instances of another. Shown as a solid line in class diagrams.	PDF
Aggregation	A "whole–part" relationship where the part can exist independently of the whole. A weaker form of composition. Shown as a line with a hollow diamond at the whole end.	PDF
Composition	A strong "whole–part" relationship where the part cannot exist without the whole. If the whole is destroyed, so are	PDF

	its parts. Shown with a filled diamond at the whole end.	
Multiplicity	A notation on associations in class diagrams specifying how many instances of one class can be related to instances of another. E.g., 1..* (one or more), 0..1 (zero or one).	PDF
Visibility Markers	Symbols in class diagrams indicating access level of attributes/operations: + (public), – (private), # (protected).	PDF
State Transition Diagram	Models the dynamics of an individual object — the states it can be in and the events that cause transitions between states. Also called state machine or statechart diagrams. Useful for objects with complex lifecycles.	Both
State	A condition during the life of an object during which it satisfies some condition, performs some action, or waits for some event. Represented as a rounded rectangle.	PDF
Transition	A change from one state to another, triggered by an event and optionally constrained by a guard condition. Represented as an arrow between states.	PDF
Guard Condition	A Boolean expression associated with a transition that must be true for the transition to fire when its triggering event occurs.	PDF
Sequence Diagram	Models the functional workflow by showing how objects interact over time to fulfil a specific use case. Depicts objects as vertical lifelines and messages as horizontal arrows arranged chronologically from top to bottom.	Both
Lifeline	A vertical dashed line in a sequence diagram representing the existence of an object over time. Messages are sent and received between lifelines.	PDF
Combined Fragment	A construct in sequence diagrams for modelling conditional and repetitive behaviour. E.g., alt (alternatives), loop (iteration).	PDF
System Boundary	A rectangle in a use case diagram that delineates what is inside the system (use cases) from what is outside (actors).	PDF

5. Design Process and Methodology

Term	Definition	Source
System Analyst	A key role in software design responsible for translating problem descriptions from stakeholders into structured requirements specifications using conceptual thinking.	Both
Stakeholder	Any person, group, or organisation that has an interest in or is affected by the software system — including users, clients, developers, and management.	Both
Problem-Driven Design	An approach that starts from the problem to be solved rather than from a technology to be applied. Every design decision traces back to a stated requirement.	Both
Requirement-Driven Design	An approach where every design decision is justified by	Both

	and traceable to a documented requirement from stakeholders.	
Conceptual Thinking	Reasoning about a problem at a level of abstraction above code — translating domain-specific problem descriptions into structured specifications.	Both
Domain Aspect	A key concern within the problem domain that the system must address. Each aspect (e.g., functional requirements, workflows, data structures, object lifecycle, inter-object communication) maps to a specific UML technique.	PDF
Iterative Prototyping	A technique for progressively narrowing the requirements gap by building successive versions of the system that users can evaluate and provide feedback on.	PDF
Validation Feedback Loop	A mechanism ensuring that the design remains aligned with stakeholder needs throughout the development process by continuously checking design outputs against requirements.	Both
Conceptual Prompts	Carefully crafted prompts that engage an LLM at the level of design concepts (rather than code generation) to support learning, critical analysis, and design refinement.	Both

6. Quality Attributes and Architectural Concerns

Term	Definition	Source
Performance	The responsiveness and throughput of a system under given conditions. Designing for performance requires avoiding bottlenecks and ensuring responsiveness under variable loads.	Both
Extensibility	The ability to add new functionality to a system with minimal impact on existing components. Supported by polymorphism, modularity, and API-driven design.	Both
Reusability	The degree to which existing software components can be used in new contexts. Promoted by inheritance, encapsulation, and well-defined interfaces.	Both
Resilience	The ability of a system to continue operating when individual components fail. In microservices, failure in one service should not cascade to the entire system.	PDF
Energy-Aware Computation	A design consideration that accounts for the energy cost of processing, recognising that needlessly resource-intensive computation degrades both performance and environmental footprint.	Both
Tight Coupling	A design anti-pattern where components are highly dependent on each other's internal details, dramatically increasing the cost and risk of changes due to unexpected ripple effects.	Both
Service Discovery	The mechanism by which microservices locate and communicate with one another in a distributed system. A	PDF

	source of added complexity in microservices architectures.	