Unified Modelling Language (UML)

A Comprehensive Structured Resource

Diagrams, Relationships, Notation, and Techniques

Structured Reference Notes

Table of Contents

Session 1: Introduction to the Unified Modelling Language

1.1 What is UML?

The Unified Modelling Language (UML) is a standardised, general-purpose visual modelling language adopted across the software engineering industry for specifying, visualising, constructing, and documenting the artefacts of software systems. Maintained by the Object Management Group (OMG), UML provides a common vocabulary and set of diagrammatic conventions that enable analysts, architects, developers, and stakeholders to communicate design decisions clearly and unambiguously. The current version, UML 2.5, defines fourteen diagram types that collectively address the structural, behavioural, and interactional aspects of any software system.

UML is not a programming language, nor is it a methodology in itself. Rather, it is a notation — a set of graphical symbols and rules for combining them — that can be used within any software development methodology, whether waterfall, iterative, agile, or hybrid. Its power lies in its ability to represent complex systems at multiple levels of abstraction, from high-level architectural overviews to fine-grained implementation details, using a consistent visual vocabulary that all team members can understand.

Key Concept — UML

UML is a standardised visual modelling language for specifying, visualising, constructing, and documenting the artefacts of software-intensive systems. It is not a methodology but a notation that can be applied within any development process.

1.2 History and Evolution of UML

UML emerged in the mid-1990s from the unification of three influential object-oriented modelling approaches: Grady Booch's Booch method, James Rumbaugh's Object Modelling Technique (OMT), and Ivar Jacobson's Object-Oriented Software Engineering (OOSE). These three pioneers — often referred to as the 'Three Amigos' — joined forces at Rational Software to create a single, unified language that combined the strengths of their respective approaches. Version 1.0 of UML was submitted to the OMG in 1997, and by 1997 it became an OMG standard.

The language has evolved significantly since then. UML 2.0, released in 2005, introduced major improvements including interaction overview diagrams, timing diagrams, and composite structure diagrams. The current specification, UML 2.5.1 (released 2017), refined the metamodel and clarified numerous aspects of the notation. Throughout its evolution, UML has remained backwards-compatible, ensuring that models created in earlier versions remain valid.

Version	Year	Key Contribution
UML 0.9	1996	Initial unification of Booch, OMT, and OOSE methods

UML 1.0	1997	First OMG-adopted standard
UML 1.4	2001	Refinements to action semantics and profiles
UML 2.0	2005	Major revision: new diagram types, improved metamodel
UML 2.5.1	2017	Current specification: clarified semantics and notation

1.3 Purpose and Benefits of UML

UML serves four fundamental purposes in software engineering, each of which contributes to the overall quality and success of a development project. These purposes are not mutually exclusive; a single diagram may simultaneously fulfil several of them.

- Specifying — UML provides precise, unambiguous definitions of system structure and behaviour. A well-formed UML model eliminates the vagueness inherent in natural-language descriptions, enabling developers to implement the system exactly as designed.
- Visualising — UML diagrams make complex systems comprehensible by presenting information graphically. Humans process visual information far more efficiently than dense text, and UML exploits this by representing classes, relationships, and interactions as intuitive graphical elements.
- Constructing — UML models can serve as blueprints for code generation. Many modern CASE (Computer-Aided Software Engineering) tools can generate skeleton code from class diagrams and round-trip between models and code.
- Documenting — UML diagrams form a durable record of design decisions, system architecture, and inter-component relationships. This documentation is invaluable during maintenance, onboarding of new team members, and future system evolution.

1.4 UML Diagram Categories

UML 2.5 defines fourteen diagram types, organised into two broad categories: structure diagrams (which model the static aspects of a system) and behaviour diagrams (which model the dynamic aspects). Within behaviour diagrams, a sub-category of interaction diagrams specifically addresses how objects communicate at runtime. Understanding this taxonomy is essential for selecting the right diagram for a given modelling task.

Category	Diagram Type	Primary Purpose
Structure	Class Diagram	Classes, attributes, operations, and relationships
Structure	Object Diagram	Snapshot of object instances at a point in time
Structure	Package Diagram	Organisation of model elements into packages

Structure	Component Diagram	Physical components and their dependencies
Structure	Composite Structure Diagram	Internal structure of a classifier
Structure	Deployment Diagram	Hardware nodes and software deployment
Structure	Profile Diagram	Extensions to the UML metamodel
Behaviour	Use Case Diagram	Actors, use cases, and system scope
Behaviour	Activity Diagram	Workflows, business processes, algorithms
Behaviour	State Machine Diagram	Object lifecycle and state transitions
Interaction	Sequence Diagram	Object interactions ordered by time
Interaction	Communication Diagram	Object interactions emphasising links
Interaction	Interaction Overview Diagram	High-level flow of interactions
Interaction	Timing Diagram	State changes along a time axis

In practice, most projects use a subset of these fourteen diagrams. The five most commonly used — and the ones most frequently assessed in university modules — are class diagrams, use case diagrams, activity diagrams, state machine diagrams, and sequence diagrams. These five collectively cover the essential structural, behavioural, and interactional perspectives of a system.

Key Concept — Diagram Categories

UML diagrams fall into two main categories: structure diagrams (static view) and behaviour diagrams (dynamic view). Interaction diagrams are a specialised subset of behaviour diagrams. Selecting the right diagram depends on which aspect of the system you wish to model.

Session 2: Class Diagrams — Structure and Notation

2.1 Overview of Class Diagrams

The class diagram is arguably the most important and widely used of all UML diagram types. It models the static structure of a system by depicting classes, their attributes and operations, and the relationships between them. Class diagrams serve as the backbone of object-oriented design: they define the data model, establish the vocabulary of the problem domain, and provide the blueprint from which implementation code is derived.

A class diagram can be produced at different levels of abstraction. A conceptual or domain model shows real-world entities and their relationships without implementation detail. A design-level class diagram adds visibility markers, data types, method signatures, and navigation directions, ready for translation into code. Both levels use the same basic notation; the difference lies in the amount of detail included.

2.2 Class Notation

In UML, a class is represented as a rectangle divided into three horizontal compartments. The top compartment contains the class name (in bold, centred). The middle compartment lists the attributes (data fields), and the bottom compartment lists the operations (methods). If a compartment is empty, it may be omitted or left blank, but the class name compartment is always present.

Key Concept — Class Box Structure

A UML class is drawn as a three-compartment rectangle:
• Top: Class name (bold, centred; abstract class names are italicised)
• Middle: Attributes in the form visibility name : type [= default]
• Bottom: Operations in the form visibility name(params) : returnType

2.2.1 Visibility Markers

Each attribute and operation in a class diagram is prefixed with a visibility marker that indicates its access level. UML defines four standard visibility levels, each represented by a single symbol. Understanding visibility is essential for encapsulation — one of the four pillars of object-oriented design.

Symbol	Visibility	Meaning
+	Public	Accessible to all classes; any object can access this member.
−	Private	Accessible only within the declaring class; hidden from all

		other classes.
#	Protected	Accessible within the declaring class and its subclasses (children).
~	Package	Accessible to classes within the same package (language-dependent).

As a general design principle, attributes should be private (−) and accessed through public (+) getter and setter methods. This enforces encapsulation: the internal representation of data can change without affecting external code that uses the class. Operations that form the public interface of the class are marked public (+), while internal helper methods are marked private (−) or protected (#).

2.2.2 Attribute and Operation Syntax

UML prescribes a specific syntax for declaring attributes and operations within a class box. These conventions ensure consistency across diagrams and teams.

Attributes follow the pattern: visibility name : type [multiplicity] = defaultValue. For example, '− balance : double = 0.0' declares a private attribute named 'balance' of type double with a default value of 0.0. The multiplicity (e.g., [0..*]) indicates how many values the attribute can hold.

Operations follow the pattern: visibility name(paramName : paramType, ...) : returnType. For example, '+ withdraw(amount : double) : boolean' declares a public method that takes a double parameter and returns a boolean. Static members are underlined in UML, and abstract operations are rendered in italics.

2.2.3 Abstract Classes and Interfaces

An abstract class is a class that cannot be instantiated directly and typically contains one or more abstract operations (operations without an implementation). In UML, abstract classes are indicated by writing the class name in italics or by placing the stereotype «abstract» above the name. Abstract operations are also italicised.

An interface is a classifier that declares a set of operations without providing implementations. It is depicted as a class box with the stereotype «interface» above the name. Alternatively, an interface may be shown as a small circle (lollipop notation) connected to the implementing class. A class that implements an interface is connected to it by a dashed line with a hollow arrowhead (realisation relationship).

2.3 Multiplicity

Multiplicity specifies how many instances of one class may be associated with instances of another class. It is written at each end of an association line and is one of the most critical pieces of information in a class diagram, as it directly influences database schema design and implementation data structures.

Notation	Meaning	Example

1	Exactly one	Each Order has exactly one Customer
0..1	Zero or one (optional)	A Person may have zero or one Passport
* or 0..*	Zero or more	A Customer may place zero or more Orders
1..*	One or more	An Order must contain at least one OrderLine
n..m	Between n and m (inclusive)	A Committee has 3..10 Members

Key Concept — Reading Multiplicity

Always read multiplicity from the perspective of the opposite class. If an association between Customer and Order shows '1' at the Customer end and '*' at the Order end, it means: each Order is associated with exactly 1 Customer, and each Customer may be associated with zero or more (*) Orders.

Session 3: UML Relationships and Arrow Types

3.1 Overview of UML Relationships

Relationships are the connective tissue of UML diagrams. They express how model elements are related to one another — structurally, behaviourally, or through dependency. UML defines a precise set of relationship types, each with its own line style, arrowhead, and semantics. Mastering these relationships and their visual representations is essential for both reading and creating UML diagrams correctly.

This session provides a comprehensive catalogue of every major relationship type used in UML, along with the corresponding arrow notation, its meaning, and guidance on when to use it. The relationships are grouped by the diagram contexts in which they most commonly appear.

3.2 Class Diagram Relationships

3.2.1 Association

An association represents a structural relationship between two classes, indicating that instances of the classes are connected in some meaningful way. It is the most general and most common relationship type in class diagrams. Associations are drawn as a solid line connecting two classes, optionally adorned with a name, role names at each end, multiplicity indicators, and navigation arrows.

For example, a 'Customer places Order' association connects the Customer and Order classes. The association name 'places' appears on the line, with multiplicity '1' at the Customer end and '*' at the Order end. A small arrowhead (triangle) next to the association name indicates the reading direction.

Navigability may be shown with an open arrowhead at one or both ends. If an arrowhead appears only at the Order end, it means that Customer can navigate to its Orders, but not vice versa. If no arrowheads are shown, navigability is unspecified (or bidirectional, depending on convention).

Association — Notation

Line style: Solid line
Arrowhead: Open arrowhead (→) for navigability (optional)
Adornments: Association name, role names, multiplicity at each end
Meaning: Instances of the connected classes are structurally linked.

3.2.2 Aggregation (Shared Aggregation)

Aggregation is a specialised form of association that represents a 'whole–part' relationship where the part can exist independently of the whole. It indicates that one class (the whole) contains or is composed of instances of another class (the part), but the part's lifecycle is not tied to the whole. If the whole is destroyed, the parts can continue to exist.

For example, a Department aggregates Employees. If the Department is dissolved, the Employee objects are not destroyed — they can be reassigned to other departments. Aggregation is drawn as a solid line with a hollow (unfilled) diamond at the whole end.

Aggregation — Notation

Line style: Solid line
Symbol: Hollow (unfilled) diamond (◇) at the 'whole' end
Meaning: 'Has-a' relationship; the part can exist independently of the whole.
Example: Department ◇—— Employee

3.2.3 Composition (Composite Aggregation)

Composition is a stronger form of aggregation in which the part's lifecycle is entirely dependent on the whole. When the whole is destroyed, all of its composite parts are also destroyed. Composition implies exclusive ownership: a part can belong to only one whole at any given time.

For example, a House is composed of Rooms. If the House is demolished, the Rooms cease to exist. Similarly, an Order is composed of OrderLines — if the Order is deleted, its OrderLines are deleted with it. Composition is drawn as a solid line with a filled (solid) diamond at the whole end.

Composition — Notation

Line style: Solid line
Symbol: Filled (solid) diamond (◆) at the 'whole' end
Meaning: Strong 'owns-a' relationship; parts cannot exist without the whole.
Example: House ◆—— Room

3.2.4 Inheritance (Generalisation)

Generalisation (commonly called inheritance) represents an 'is-a' relationship between a more general class (the superclass or parent) and a more specific class (the subclass or child). The subclass inherits all attributes and operations of the superclass and may add new ones or override inherited operations to provide specialised behaviour.

Inheritance is drawn as a solid line from the subclass to the superclass, terminated with a hollow (unfilled) triangular arrowhead at the superclass end. The arrow always points from child to parent, indicating the direction of generalisation. Multiple subclasses inheriting from the same superclass are often drawn with a shared arrowhead for visual clarity.

Inheritance (Generalisation) — Notation

Line style: Solid line
Arrowhead: Hollow (unfilled) triangle (△) pointing to the superclass
Meaning: 'Is-a' relationship; subclass inherits from superclass.
Example: Circle —▷ Shape (Circle is a Shape)

3.2.5 Realisation (Interface Implementation)

Realisation is the relationship between an interface and the class that implements it. The implementing class provides concrete method bodies for all abstract operations declared by the interface. Realisation is drawn as a dashed line with a hollow triangular arrowhead pointing to the interface.

For example, if an interface Sortable declares a method compareTo(), and the class Student implements Sortable, a dashed arrow from Student to Sortable with a hollow triangle indicates that Student provides the implementation of compareTo(). The dashed line distinguishes realisation from generalisation (which uses a solid line).

Realisation — Notation

Line style: Dashed line
Arrowhead: Hollow (unfilled) triangle (△) pointing to the interface
Meaning: The class implements (realises) the interface's contract.
Example: Student - - -▷ «interface» Sortable

3.2.6 Dependency

A dependency indicates that one class (the client) uses or depends on another class (the supplier) in some way, but without a persistent structural link. Common scenarios include: a method parameter is of the supplier's type, a method locally creates an instance of the supplier, or a method calls a static method on the supplier. Dependencies are the weakest form of relationship and are drawn as a dashed line with an open arrowhead pointing from the client to the supplier.

Dependency — Notation

Line style: Dashed line
Arrowhead: Open (stick) arrowhead (→) pointing to the supplier
Meaning: The client uses or depends on the supplier transiently.
Common stereotypes: «use», «create», «call», «instantiate»

3.3 Complete Arrow Reference Table

The following table provides a single, consolidated reference for all major relationship types and their arrow notations used across UML class diagrams. This is one of the most important tables in UML — memorising it will enable you to read and draw class diagrams with confidence.

Relationship	Line Style	Arrowhead / Symbol	Direction	Meaning
Association	Solid	Open arrowhead (optional) or none	Optional navigation	'Knows-about'; structural link between classes
Aggregation	Solid	Hollow diamond at	Whole to part	'Has-a'; part exists

		whole end		independently of whole
Composition	Solid	Filled diamond at whole end	Whole to part	'Owns-a'; part's lifecycle tied to whole
Generalisation (Inheritance)	Solid	Hollow triangle at superclass	Subclass → Superclass	'Is-a'; subclass inherits from superclass
Realisation (Implementation)	Dashed	Hollow triangle at interface	Class → Interface	Class implements interface contract
Dependency	Dashed	Open arrowhead at supplier	Client → Supplier	Transient usage; weakest relationship

3.4 Use Case Diagram Relationships

Use case diagrams employ a distinct set of relationships to connect actors and use cases. These are separate from the class diagram relationships described above, though they share some visual conventions.

Relationship	Notation	Meaning
Association (Actor ↔ Use Case)	Solid line (no arrowhead)	The actor participates in the use case
Include	Dashed arrow with «include» stereotype from base to included use case	The base use case always incorporates the behaviour of the included use case (mandatory sub-functionality)
Extend	Dashed arrow with «extend» stereotype from extending to base use case	The extending use case optionally adds behaviour to the base use case (conditional sub-functionality)
Generalisation (Actor or Use Case)	Solid line with hollow triangle pointing to the general element	A specialised actor/use case inherits from a general one

Key Concept — Include vs Extend

Include means the base use case ALWAYS uses the included use case (mandatory). The arrow points FROM the base TO the included use case.
Extend means the extending use case SOMETIMES adds to the base (optional/conditional). The arrow points FROM the extending use case TO the base use case.
A common mnemonic: Include = always; Extend = sometimes.

3.5 Sequence Diagram Relationships

Sequence diagrams use a different set of arrow styles to represent messages exchanged between objects (lifelines) over time. Each arrow style conveys distinct information about the nature of the communication.

Arrow Style	Notation	Meaning
Synchronous message	Solid line with filled (solid) arrowhead →	The sender waits for the receiver to process the message and return before continuing (like a method call)
Asynchronous message	Solid line with open (stick) arrowhead →	The sender continues without waiting for a response (like posting to a message queue)
Return message	Dashed line with open arrowhead ←	The response returned from the receiver back to the sender (often optional if the return is obvious)
Self-message	Arrow looping back to same lifeline	An object sends a message to itself (internal method call)
Create message	Dashed arrow with «create» to object header	Creates a new object; the arrow points to the new lifeline's header box
Destroy message	Arrow to an X on the lifeline	Destroys the receiving object; the lifeline ends with an X symbol

3.6 Activity Diagram Relationships

Activity diagrams use flow edges (arrows) to connect activity nodes. While simpler than class or sequence diagram arrows, they carry important information about control flow and concurrency.

Element	Notation	Meaning
Control flow	Solid arrow between actions	The flow of control from one action to the next
Object flow	Solid arrow passing through an object node (rectangle)	Data or an object produced by one action and consumed by another
Decision node	Diamond with one incoming and multiple outgoing arrows with guard conditions [condition]	A branching point where the flow takes one of several paths based on a condition
Merge node	Diamond with multiple incoming and one outgoing arrow	Brings together multiple alternative flows into one
Fork bar	Thick horizontal bar: one incoming, multiple outgoing	Splits a single flow into multiple concurrent (parallel) flows

	arrows	
Join bar	Thick horizontal bar: multiple incoming, one outgoing arrow	Synchronises concurrent flows; all must complete before continuing

3.7 State Machine Diagram Relationships

State machine diagrams (also called state transition diagrams or statechart diagrams) use transitions — arrows between states — to model how an object responds to events. Each transition is labelled with a trigger event, an optional guard condition, and an optional effect (action).

Element	Notation	Meaning
Transition	Solid arrow from source state to target state Label: event [guard] / action	The object changes from one state to another when the event occurs and the guard is true
Initial pseudo-state	Filled black circle with an arrow to the first state	The starting point of the state machine
Final state	Filled circle inside an outer circle (bull's-eye)	The termination point of the state machine
Self-transition	Arrow looping back to the same state	An event is handled without leaving the current state

The transition label follows the syntax: event [guard] / action. The event is what triggers the transition (e.g., 'paymentReceived'); the guard is a boolean condition that must be true for the transition to fire (e.g., '[amount >= total]'); and the action is what happens during the transition (e.g., '/ issueReceipt()'). All three components are optional: a transition may have only an event, only a guard, or only an action.

Session 4: Use Case Diagrams

4.1 Purpose of Use Case Diagrams

Use case diagrams are typically the first UML artefact produced during analysis. They capture the functional requirements of a system by identifying who interacts with the system (actors) and what the system does for them (use cases). By establishing the system scope and the interactions at its boundary, use case diagrams provide the foundation upon which all subsequent UML modelling is built.

Use case diagrams are deliberately high-level: they show what the system does, not how it does it. This makes them ideal for communicating with non-technical stakeholders, who can verify that the system's proposed functionality aligns with their needs without needing to understand implementation details.

4.2 Key Elements

4.2.1 Actors

An actor is any entity that interacts with the system from outside its boundary. Actors are most commonly human users, but they can also be external systems, hardware devices, or time-based triggers (e.g., a scheduled batch process). Actors are drawn as stick figures positioned outside the system boundary, with the actor's role name below.

Primary actors initiate interactions with the system to achieve a goal (e.g., 'Customer' placing an order). Secondary actors are called upon by the system to fulfil a use case (e.g., 'Payment Gateway' processing a payment). Identifying all actors is a critical first step because it defines the external interfaces of the system.

4.2.2 Use Cases

A use case represents a discrete piece of functionality that the system provides to an actor, delivering observable value. Use cases are drawn as ovals (ellipses) inside the system boundary rectangle, with a short descriptive name that follows the pattern 'verb + noun' (e.g., 'Place Order', 'Generate Report', 'Authenticate User').

Each use case should represent a complete, meaningful interaction — not a single step in a process. For example, 'Enter Password' is too granular to be a use case; 'Authenticate User' (which encompasses entering credentials, validating them, and responding) is the appropriate level of granularity.

4.2.3 System Boundary

The system boundary is drawn as a rectangle enclosing all use cases. It represents the scope of the system under design. Anything inside the rectangle is part of the system; anything outside (actors and external systems) is not. The system name is written at the top of the rectangle.

Key Concept — Use Case Granularity

A well-formed use case represents a complete user goal that delivers observable value. It should be neither too broad (an entire subsystem) nor too narrow (a single button click). The 'Elementary Business Process' test is useful: a use case should describe a task performed by one person, in one place, at one time, in response to a business event, that leaves the data in a consistent state.

4.3 Use Case Relationships in Detail

4.3.1 Include Relationship

The include relationship indicates that a base use case unconditionally incorporates the behaviour of another (included) use case. It is used to extract common functionality shared by multiple use cases, thereby avoiding duplication. For example, both 'Place Order' and 'Cancel Order' might include 'Authenticate User', because authentication is mandatory before either action.

Notation: a dashed arrow labelled «include» from the base use case to the included use case. The direction of the arrow is crucial: it points towards the included (shared) behaviour.

4.3.2 Extend Relationship

The extend relationship indicates that an extending use case may optionally add behaviour to a base use case under certain conditions. The base use case is complete on its own; the extension provides additional, conditional functionality. For example, 'Apply Discount' might extend 'Place Order', triggered only when a discount code is entered.

Notation: a dashed arrow labelled «extend» from the extending use case to the base use case. Note the direction: the arrow points from the extension towards the base, the reverse of include.

4.3.3 Generalisation

Generalisation between use cases (or between actors) works the same way as inheritance in class diagrams. A specialised use case inherits the behaviour of a general use case and may add or override steps. For example, 'Pay by Credit Card' and 'Pay by Direct Debit' might both generalise from a parent use case 'Make Payment'. The notation is a solid line with a hollow triangular arrowhead pointing to the general (parent) use case.

Relationship	Arrow Direction	When to Use
Include	Base → Included	Common, mandatory sub-behaviour shared by multiple use cases
Extend	Extension → Base	Optional, conditional behaviour that augments the base
Generalisation	Specific → General	Specialised variant that inherits from a general use case

Session 5: Activity Diagrams

5.1 Purpose and Context

Activity diagrams model the flow of control (and optionally the flow of data) through a process, workflow, or algorithm. They are similar to traditional flowcharts but extend them with support for concurrency (parallel activities), swim lanes (responsibility allocation), and object flows. Activity diagrams are particularly valuable for modelling business processes and for detailing the flow of events within a use case.

In the UML methodology, activity diagrams are the second step: after identifying use cases (Step 1), the designer creates an activity diagram for each non-trivial use case to elaborate the detailed flow of events, including decisions, loops, and parallel activities.

5.2 Key Notation Elements

Element	Symbol	Description
Initial node	Filled black circle (●)	The starting point of the activity; there is exactly one initial node
Activity final node	Filled circle inside outer circle (⨀)	Terminates the entire activity; reaching this node ends all flows
Flow final node	Circle with X inside (⊗)	Terminates a single flow without ending the entire activity
Action node	Rounded rectangle	A single, atomic step in the process (e.g., 'Validate Input')
Decision node	Diamond with one incoming edge and multiple outgoing edges	A conditional branch; each outgoing edge has a guard condition [...]
Merge node	Diamond with multiple incoming edges and one outgoing edge	Combines multiple alternative paths back into one
Fork bar	Thick horizontal/vertical bar (1 in, many out)	Splits flow into concurrent (parallel) threads
Join bar	Thick horizontal/vertical bar (many in, 1 out)	Synchronises concurrent threads; waits for all to complete
Swim lane	Vertical or horizontal partition	Allocates actions to a responsible actor, role, or system component

Key Concept — Decision vs Merge, Fork vs Join

Decision and Merge nodes both use the diamond shape but serve opposite purposes: Decision splits one flow into alternatives (based on guards); Merge brings alternatives back together.

Fork and Join both use the thick bar but serve opposite purposes: Fork splits one flow into concurrent threads; Join waits for all concurrent threads to finish.

5.3 Guard Conditions

Outgoing edges from a decision node must be labelled with guard conditions, written in square brackets (e.g., [amount > 0], [approved], [else]). Guard conditions must be mutually exclusive and exhaustive — that is, exactly one guard must evaluate to true for any given scenario. Using [else] as a catch-all on one outgoing edge ensures exhaustiveness.

5.4 Swim Lanes (Activity Partitions)

Swim lanes divide an activity diagram into vertical (or horizontal) bands, each representing a different actor, role, or system component responsible for the actions within that lane. Swim lanes make it immediately clear who is responsible for each step in a process, which is invaluable for business process modelling and for identifying the interfaces between system components.

For example, in an online order process, swim lanes might be labelled 'Customer', 'Web Application', 'Payment Service', and 'Warehouse'. Each action is placed in the lane of the responsible party, and flow edges crossing lane boundaries represent interactions between parties.

Session 6: State Machine Diagrams

6.1 Purpose and Context

State machine diagrams (also known as state transition diagrams or statechart diagrams) model the lifecycle of a single object, showing the discrete states it can be in and the events that cause transitions between those states. They are the fourth step in the UML methodology: after modelling the system's data structure with class diagrams (Step 3), the designer identifies objects with complex lifecycles and creates state machine diagrams to capture their behaviour.

State machine diagrams are most useful for objects whose behaviour depends significantly on their current state. An Order object, for example, behaves differently when it is in the 'Pending' state versus the 'Shipped' state: certain operations are valid only in certain states, and the same event may produce different effects depending on the current state.

6.2 Key Notation Elements

Element	Symbol	Description
State	Rounded rectangle with name	A condition or situation during the life of an object
Initial pseudo-state	Filled black circle (●)	The entry point of the state machine; exactly one
Final state	Bull's-eye (◉)	The termination point; the object's lifecycle ends here
Transition	Solid arrow between states	A change from one state to another, triggered by an event
Self-transition	Arrow looping back to same state	An event is processed without leaving the current state
Composite state	State containing nested sub-states	A state that has its own internal state machine

6.3 Transition Syntax

The label on a transition arrow follows the syntax: event [guard] / action. Each component is optional, but at least one should be present to give the transition meaning.

- Event — the trigger that causes the transition (e.g., 'paymentReceived', 'timeout', 'cancel'). If omitted, the transition fires automatically when any entry actions of the source state are complete.
- Guard — a boolean condition in square brackets that must be true for the transition to fire (e.g., '[balance >= amount]'). If the guard is false, the transition does not occur even if the event is received.

- Action — a behaviour that executes during the transition, prefixed with '/' (e.g., '/ sendConfirmation()'). Actions are considered instantaneous and non-interruptible.

6.4 Internal Activities

States may also have internal activities that occur while the object remains in that state, without causing a transition. UML defines three standard internal activity labels:

Label	When It Executes	Example
entry /	Immediately upon entering the state	entry / startTimer()
exit /	Immediately before leaving the state	exit / stopTimer()
do /	Continuously while the object remains in the state	do / monitorSensor()

Key Concept — State vs Transition

A state represents a period of time during which an object satisfies some condition, performs some activity, or waits for an event. A transition represents an instantaneous change from one state to another, triggered by an event. States have duration; transitions do not.

Session 7: Sequence Diagrams

7.1 Purpose and Context

Sequence diagrams model the runtime interactions between objects (or actors) ordered by time. They are the fifth and final step in the UML methodology: having identified use cases, detailed their flows (activity diagrams), modelled the data structure (class diagrams), and captured object lifecycles (state machine diagrams), the designer now shows how objects collaborate to fulfil each use case scenario.

Sequence diagrams are read from top to bottom, with time progressing downwards. Each vertical dashed line represents the lifeline of an object (or actor), and horizontal arrows represent messages passed between them. The strength of sequence diagrams lies in their ability to make the temporal ordering of interactions explicit, which is crucial for understanding concurrency, synchronisation, and the detailed mechanics of use case fulfilment.

7.2 Key Notation Elements

Element	Symbol	Description
Lifeline	Rectangle (header) atop a vertical dashed line	Represents an object or actor participating in the interaction; labelled objectName : ClassName
Activation bar (execution specification)	Thin rectangle on the lifeline	Indicates the period during which the object is actively processing a message (the method is on the call stack)
Synchronous message	Solid line with filled arrowhead (▶)	A call where the sender waits for the receiver to finish (standard method call)
Asynchronous message	Solid line with open arrowhead (▷)	A call where the sender does not wait for a response (fire-and-forget)
Return message	Dashed line with open arrowhead	The value or control returned from a method call; can be omitted if obvious
Self-message	Arrow looping back to same lifeline	An object invokes one of its own methods
Object creation	Dashed arrow to the header of a new lifeline with «create»	Creates a new object during the interaction
Object destruction	Arrow ending with an X on the lifeline	Destroys the receiving object

7.3 Combined Fragments

Combined fragments add control-flow logic to sequence diagrams, enabling the representation of conditions, loops, and alternatives within a single diagram. A combined fragment is drawn as a rectangle overlaying part of the sequence diagram, with an operator keyword in the top-left corner.

Operator	Name	Meaning
alt	Alternative	Models if–else logic: the fragment is divided into operands separated by dashed lines; each operand has a guard condition, and exactly one executes
opt	Option	Models an if-without-else: the fragment's content executes only if the guard condition is true
loop	Loop	Models iteration: the fragment's content repeats while the guard is true (e.g., loop [items remaining])
par	Parallel	Models concurrent execution: multiple operands execute simultaneously
break	Break	Models an exception or early exit: if the guard is true, the enclosing fragment is abandoned
ref	Reference	Refers to another interaction diagram, promoting reuse and reducing diagram clutter

Key Concept — Reading a Sequence Diagram

Read top-to-bottom (time flows downward). Each horizontal arrow is a message: solid with filled arrowhead = synchronous call; solid with open arrowhead = asynchronous; dashed = return. The activation bar shows when an object is executing. Combined fragments (alt, opt, loop, par) add control-flow logic.

7.4 From Use Case to Sequence Diagram

Sequence diagrams are derived from use case descriptions and activity diagrams. For each use case scenario (main success scenario and each significant alternative), the designer identifies the participating objects (from the class diagram), then traces the flow of events step by step, showing which object sends which message to which other object.

This process often reveals missing operations on classes: if a sequence diagram requires object A to send a 'calculateTotal()' message to object B, then class B must have a calculateTotal() operation. In this way, sequence diagrams serve as a validation mechanism for the class diagram, ensuring that the static structure supports the dynamic behaviour required by each use case.

Session 8: UML Techniques for System Modelling

8.1 The Five-Step UML Methodology

UML is most effective when its diagram types are used together within a coherent methodology. The five-step process introduced here provides a systematic approach to system modelling, progressing from high-level requirements to detailed design. Each step produces specific artefacts that feed into subsequent steps, creating a traceable chain from user needs to implementation-ready specifications.

Step	UML Technique	Input	Output
1	Use Case Diagram	Problem description, stakeholder interviews	System scope, actors, use cases, include/extend relationships
2	Activity Diagram	Use case descriptions	Detailed flow of events for each use case, including decisions and parallelism
3	Class Diagram	Use cases, activity diagrams, domain knowledge	Domain model: classes, attributes, operations, and relationships
4	State Machine Diagram	Class diagram (objects with complex lifecycles)	State models showing how key objects respond to events over their lifetime
5	Sequence Diagram	Use cases, class diagram, activity diagrams	Functional workflows showing object interactions for each use case scenario

Key Concept — The Five-Step Methodology

Step 1 (Use Case) → Step 2 (Activity) → Step 3 (Class) → Step 4 (State Machine) → Step 5 (Sequence). Each step builds on the previous: use cases drive activities, activities reveal classes, classes inform state models, and all feed into sequence diagrams.

8.2 Step 1: Establishing System Scope with Use Case Diagrams

The modelling process begins with the problem description provided by stakeholders. The system analyst reads this description carefully and identifies the actors (who or what interacts with the system), the use cases (what the system must do), and the relationships between them (associations, include, extend, and generalisation).

A practical technique for identifying use cases is to look for verbs in the problem description that represent actions the system must perform. For identifying actors, look for nouns that represent roles or external systems. Events and conditions mentioned in the description often correspond to include/extend relationships or guard conditions in subsequent diagrams.

- Identify all actors and their roles with respect to the system.
- List all use cases as verb-noun phrases (e.g., 'Place Order', 'Generate Report').
- Determine include relationships for mandatory shared sub-functionality.
- Determine extend relationships for optional or conditional behaviour.
- Draw the use case diagram with a clear system boundary.

8.3 Step 2: Detailing Workflows with Activity Diagrams

For each non-trivial use case identified in Step 1, the designer creates an activity diagram to elaborate the detailed flow of events. This step transforms the 'what' of use cases into the 'how' of step-by-step processes. Activity diagrams capture the sequencing of actions, decision points, parallel activities, and the allocation of responsibilities through swim lanes.

The activity diagram should cover the main success scenario as well as significant alternative and exception paths. Each decision node corresponds to a condition identified in the problem description, and each action node corresponds to a discrete step in the process. Swim lanes should be used when multiple actors or system components are involved, to clarify responsibility.

8.4 Step 3: Structuring Data with Class Diagrams

Step 3 transitions from behavioural modelling to structural modelling. The designer examines the use cases, activity diagrams, and domain knowledge to identify the key entities (classes) that the system must manage, their attributes, operations, and relationships.

A practical technique for identifying classes is to look for nouns in the problem description and use case narratives. Nouns that represent tangible or conceptual entities (e.g., 'Customer', 'Order', 'Product', 'Payment') are strong candidates for classes. Verbs associated with these nouns suggest operations (e.g., 'Customer places Order' suggests a placeOrder() method). Adjectives and values suggest attributes (e.g., 'order total' suggests a 'total' attribute on the Order class).

- Identify classes from nouns in the problem description and use case narratives.
- Determine attributes from adjectives, values, and data items mentioned.
- Derive operations from verbs and the actions in activity diagrams.
- Establish relationships: association, aggregation, composition, and inheritance.
- Add multiplicities to all associations.
- Apply visibility markers (public, private, protected) to enforce encapsulation.

8.5 Step 4: Modelling Object Behaviour with State Machine Diagrams

Not every class warrants a state machine diagram — only those whose instances exhibit significantly different behaviour depending on their state. Typical candidates include domain objects that progress through a lifecycle (e.g., Order, Account, Ticket, Request) and controller objects that manage multi-step processes.

To create a state machine diagram, the designer identifies the distinct states the object can be in, the events that trigger transitions between states, the guard conditions that control whether transitions fire, and the actions performed during transitions or within states. The events typically come from the use case descriptions and activity diagrams; the states emerge from the domain knowledge and the class diagram.

8.6 Step 5: Capturing Interactions with Sequence Diagrams

The final step brings everything together. For each use case scenario, the designer creates a sequence diagram that shows how the participating objects (identified in the class diagram) interact over time to fulfil the use case. The messages in the sequence diagram correspond to operations on the receiving class, creating a direct, verifiable link between the static class model and the dynamic interaction model.

Sequence diagrams often reveal gaps in earlier artefacts. If a sequence diagram requires an operation that does not exist on the target class, the class diagram must be updated. If the required interaction pattern does not match the activity diagram's flow, one or both must be revised. This iterative refinement across diagram types is a hallmark of a mature UML design process.

8.7 Traceability Across Diagram Types

One of the greatest strengths of the five-step methodology is the traceability it provides. Every element in a later diagram can be traced back to an element in an earlier diagram, and ultimately to a stakeholder requirement. This traceability chain is summarised below.

From	To	Traceability Link
Use Case	Activity Diagram	Each use case is elaborated by one or more activity diagrams
Activity Diagram	Class Diagram	Action nodes suggest operations; data items suggest attributes and classes
Use Case + Class	State Machine	Events from use cases trigger transitions on stateful objects from the class diagram
Use Case + Class	Sequence Diagram	Use case scenarios are realised by interactions between objects from the class diagram

Sequence Diagram	Class Operations	Each message in a sequence diagram must correspond to an operation on the receiving class

Key Concept — Traceability

Traceability means that every design element can be traced back to a requirement. In UML, this is achieved by ensuring that use cases drive activity diagrams, which inform class diagrams, which feed state machines and sequence diagrams. If an element cannot be traced to a requirement, it may be unnecessary; if a requirement cannot be traced to a design element, it may have been overlooked.

Session 9: Practical Modelling Guidelines and Common Pitfalls

9.1 General Best Practices

Effective UML modelling is as much about judgement and experience as it is about notation. The following guidelines, drawn from industry practice and academic literature, will help you create models that are clear, correct, and useful.

- Model with purpose — Every diagram should have a clear objective. Do not create diagrams for their own sake; each should answer a specific question about the system (e.g., 'What are the main use cases?', 'How does an Order change state?').
- Keep diagrams focused — A single diagram that tries to show everything becomes unreadable. Use multiple, focused diagrams rather than one monolithic one. A class diagram should typically contain 5–15 classes; a sequence diagram should cover one scenario of one use case.
- Use consistent naming — Class names should be singular nouns in PascalCase (e.g., 'Customer', not 'customers'). Operation names should be verbs in camelCase (e.g., 'calculateTotal()'). Use case names should be verb-noun phrases (e.g., 'Place Order').
- Show multiplicities on all associations — Omitting multiplicities leaves the model ambiguous. Always specify the cardinality at both ends of every association.
- Validate across diagrams — Ensure consistency: every message in a sequence diagram must correspond to an operation on the receiving class; every use case should have at least one activity diagram; every class should participate in at least one interaction.

9.2 Common Mistakes and How to Avoid Them

Common Mistake	Why It Is Wrong	Correction
Using association where composition is needed	Fails to express lifecycle dependency; parts may be incorrectly shared or survive deletion of the whole	If the part cannot exist without the whole (e.g., OrderLine without Order), use composition (filled diamond)
Confusing include and extend arrow directions	Reversing the arrows changes the semantics entirely; a common source of marks lost in assessments	Include: arrow from BASE to INCLUDED Extend: arrow from EXTENSION to BASE
Using inheritance when association is appropriate	Inheritance represents 'is-a', not 'has-a' or 'uses-a'; misuse creates fragile hierarchies	Apply the 'is-a' test rigorously. If a Car has an Engine, use association, not inheritance
Omitting multiplicities	Leaves the model ambiguous; implementations	Always specify multiplicity at both ends

	will vary based on developer interpretation	of every association
Mixing abstraction levels in one diagram	High-level and low-level elements in the same diagram confuse readers	Keep each diagram at a consistent level of abstraction (conceptual, design, or implementation)
States that describe actions, not conditions	A state should be a condition (e.g., 'Pending'), not an action (e.g., 'Processing payment')	Name states as adjectives or nouns representing conditions: 'Pending', 'Approved', 'Shipped'

9.3 Choosing the Right Diagram

With fourteen diagram types available, selecting the appropriate one for a given modelling task is itself a design decision. The following decision guide maps common questions to the diagram types best suited to answer them.

Question You Want to Answer	Recommended Diagram
What does the system do and who uses it?	Use Case Diagram
How is a process or workflow performed?	Activity Diagram
What data does the system manage and how are entities related?	Class Diagram
How does a specific object change state over its lifetime?	State Machine Diagram
How do objects interact to fulfil a use case?	Sequence Diagram
How are components deployed on hardware?	Deployment Diagram
How are model elements organised into groups?	Package Diagram
What are the physical components and their dependencies?	Component Diagram

Key Concept — Diagram Selection

No single UML diagram shows everything about a system. Each diagram type provides a specific perspective. The five core diagrams (use case, activity, class, state machine, sequence) together provide a comprehensive view covering scope (what), process (how), structure (data), lifecycle (state), and interaction (collaboration).

9.4 From UML to Code

A well-formed UML class diagram can be translated to code in a systematic, almost mechanical fashion. The following table summarises the mapping from UML constructs to typical object-oriented programming language constructs (using Java as the reference language).

UML Construct	Java Equivalent	Notes
Class	class ClassName { }	One .java file per class
Attribute (– name : String)	private String name;	Visibility maps to access modifier
Operation (+ getName() : String)	public String getName() { }	Return type and parameters map directly
Association (1..*)	private List<Order> orders;	Collections for * multiplicities
Composition	private List<OrderLine> lines; (created/destroyed with Order)	Managed in constructor and lifecycle methods
Inheritance	class Circle extends Shape { }	extends keyword for generalisation
Interface realisation	class Student implements Sortable { }	implements keyword for realisation
Abstract class	abstract class Shape { }	abstract keyword; cannot be instantiated

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.
- Booch, G., Rumbaugh, J. and Jacobson, I. (2005) The Unified Modeling Language User Guide. 2nd edn. Boston: Addison-Wesley.
- Rumbaugh, J., Jacobson, I. and Booch, G. (2004) The Unified Modeling Language Reference Manual. 2nd edn. Boston: Addison-Wesley.
- 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.
- 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).
- Miles, R. and Hamilton, K. (2006) Learning UML 2.0. Sebastopol, CA: O'Reilly Media.
- Pilone, D. and Pitman, N. (2005) UML 2.0 in a Nutshell. Sebastopol, CA: O'Reilly Media.