CS2SD Software Systems Design

Use Case Diagrams

Lecture Notes

Table of Contents

Session 1: Use Case Diagrams — Design Role

1.1 Introduction and Learning Objectives

Use case diagrams occupy a pivotal position in the Unified Modelling Language (UML) methodology. They are the first modelling technique applied during systems analysis and design, and they serve as the foundation upon which all subsequent UML artefacts are built. In the CS2SD module, use case diagrams represent the starting point of the five-step UML methodology introduced in earlier sessions: they capture the functional requirements of a system by identifying who interacts with it and what the system must do in response.

The learning objectives for these sessions on use case diagrams are threefold. First, students should appreciate the design role that use case diagrams play in relation to a system's requirements — understanding not merely what use case diagrams look like, but why they are indispensable. Second, students should be able to articulate and capture users' needs by representing requirements in terms of actors (both internal and external), their roles, their responsibilities and behaviours in use cases, and the information consumed during the course of action. Third, students should be able to apply the principles and style guide of use case diagrams: modelling the functionalities the intended system should perform and documenting the system's requirements with clear design input and design output.

These objectives underscore a fundamental principle of software systems design: the system must be designed from the perspective of those who will use it. Use case diagrams enforce this user-centred perspective by requiring designers to think in terms of actors and the value that each piece of functionality delivers to them.

1.2 Recap: The UML Methodology

Before examining use case diagrams in detail, it is helpful to recall where they sit within the broader UML methodology. As introduced in the previous lecture, the methodology comprises five sequential steps, each producing a distinct set of design artefacts. Use case diagrams constitute Step 1 of this process and are therefore the entry point for all system modelling work.

Step	UML Technique	Primary Purpose
Step 1	Use Case Diagrams	Adopt the system scope; identify actors, use cases, and their relationships
Step 2	Activity Diagrams	Detail the scoped flow of events for each use case
Step 3	Class Diagrams	Structure the scoped data by identifying classes, attributes, and relationships
Step 4	State Transition Diagrams	Model the behaviour of key objects as they respond to events

Step 5	Sequence Diagrams	Transform the flow of events into functional workflows between objects

Each step builds on the outputs of its predecessor. The actors and use cases identified in Step 1 drive the activity diagrams in Step 2; the events and information elements discovered through use case analysis inform the class identification in Step 3; and so on. Because use case diagrams sit at the top of this chain, any errors or omissions at this stage propagate throughout the entire design. This is why rigorous use case modelling is essential.

1.3 Design Role of Use Case Diagrams

1.3.1 Who Develops Use Case Diagrams?

Use case diagrams are developed collaboratively by systems analysts and end-users. The systems analyst brings expertise in modelling techniques and the UML notation, while the end-users (and other stakeholders such as business managers, domain experts, and operations staff) contribute knowledge of the problem domain, the existing business processes, and the requirements for the new or enhanced system. This collaborative approach ensures that the resulting use case model accurately reflects what users actually need, rather than what developers assume they need.

The partnership between analysts and users is critical because of the requirements gap discussed in earlier sessions. Users express their needs in domain-specific, informal language; analysts must translate these into precise, structured models. Use case diagrams provide an accessible visual notation that serves as a shared language between these two groups, enabling effective communication and validation.

1.3.2 What Purpose Do Use Case Diagrams Serve?

Use case diagrams serve a broad range of purposes within the software development lifecycle. At their core, they are a tool for functional requirements analysis and the conceptual design of a system. However, their utility extends well beyond initial requirements capture. The principal purposes are summarised below.

- Specifying and clarifying the system scope in a domain context — the system boundary rectangle in a use case diagram explicitly delineates what is inside the system and what is external to it.
- Identifying and capturing functional requirements for the system — each use case represents a discrete piece of functionality that delivers value to an actor.
- Modelling user interactions — the associations between actors and use cases document who interacts with what functionality.
- Facilitating requirement prioritisation — use cases can be ranked by importance (high, medium, low) to guide iterative development and release planning.
- Guiding implementation — use cases provide a natural unit of work for development teams, often mapping to user stories or feature increments.
- Aiding in test case development — each use case, together with its flow of events and alternative scenarios, provides a basis for systematic testing.

Beyond these immediate purposes, use case diagrams also support forward engineering — the renovation and reclamation of existing systems to produce new ones. When an organisation decides to modernise a legacy system, analysts can model the existing functionality as use cases, identify gaps and opportunities, and design the target system's use case model accordingly. This makes use case diagrams valuable not only for greenfield development but also for system migration and re-engineering.

Perhaps most importantly, the design output of use case diagramming feeds directly into subsequent modelling steps. The context, actors, use cases, events, flows, conditions, and information elements discovered during use case analysis become the input for producing activity diagrams (Step 2), class diagrams (Step 3), and state transition diagrams (Step 4). In this sense, use case diagrams are the seed from which the entire system design grows.

Key Concept — Design Role of Use Case Diagrams

Use case diagrams serve as the foundation of the UML methodology. Developed collaboratively by systems analysts and end-users, they capture the functional requirements of a system by identifying actors, use cases, and their relationships. Their design output — actors, events, flows, conditions, and information elements — feeds directly into activity diagrams, class diagrams, and state transition diagrams in subsequent methodology steps.

1.4 System Levels of Use Cases

Use cases can be considered at different levels of abstraction, depending on whether the focus is on business processes or on the technical behaviour of a software application. In the context of this module, we are primarily concerned with system use cases — those that describe what a software system does in response to interactions with its users and other external systems.

System use cases are technology-dependent: they describe the automated functions performed within a software application. The scope of a system use case is defined by the technical behaviour of the application, and the actors involved are the application's end-users. The focus is on what the system does to produce information or insights for its users, rather than on the broader business processes in which the system participates.

It is worth noting the connection between use case diagramming and sequence diagramming at the system level. While use case diagrams capture what functional requirements the system must fulfil (a static, declarative view), sequence diagrams capture how the functional workflow is carried out at runtime (a dynamic, procedural view). The two techniques are complementary: use case diagrams define the 'what', and sequence diagrams detail the 'how'. Together, they provide a comprehensive specification of the system's functional behaviour.

Level	Focus	Actors	Output
Business Use Cases	Business processes and organisational goals	Business actors (roles, departments, external organisations)	Business process models
System Use Cases	Automated functions within a software	Application end-users	Functional requirements

	application	and external systems	specification

Key Concept — System Use Cases

System use cases describe the technology-dependent automated functions that a software application performs. They define the scope of the application’s technical behaviour, focusing on what the system does for its end-users. System use cases feed directly into sequence diagrams, which model how those functions are carried out at runtime.

Session 2: Use Case Diagrams — Concepts and Design Principles

2.1 Introduction

Having established the design role that use case diagrams play within the UML methodology, this session turns to the core concepts, notations, and design principles that underpin use case modelling. A firm grasp of these fundamentals is essential for producing use case diagrams that are both syntactically correct and semantically meaningful. This session covers four primary concepts — the system boundary, use cases, actors, and interactions — before examining the structured technique for documenting use cases in textual form.

2.2 Core Concepts and Notations

2.2.1 System Boundary

The system boundary is represented in a use case diagram as a rectangle that encloses all use cases belonging to the system under design. The name of the system is typically written at the top of the rectangle. Everything inside the boundary represents functionality that the system provides; everything outside represents external entities (actors) that interact with the system.

The system boundary serves a critical design function: it explicitly defines the scope of the system. In practice, scope definition is one of the most important — and most contested — decisions in any software project. By drawing the boundary, the design team makes a visible commitment about what the system will and will not do. This helps to manage stakeholder expectations, prevent scope creep, and provide a clear basis for contractual agreements. According to the UML 2.5 specification, the subject (system boundary) of a use case could be a system, a component, or any other element that may have behaviour, giving designers flexibility in defining the appropriate level of scope.

2.2.2 Use Case

A use case is depicted as an ellipse (oval) containing the name of the use case. It represents a specific function or behaviour that the system must perform in response to an actor's interaction. Crucially, a use case should provide some value to the actor — it is not merely a technical operation but a meaningful unit of functionality from the user's perspective.

The UML specification defines a use case as a specification of behaviour: an instance of a use case refers to an occurrence of the emergent behaviour that conforms to the corresponding use case description. In practical terms, this means that a use case captures a complete interaction scenario — from the moment an actor triggers it to the point where the system returns to a stable state. The use case must always be completed for it to be considered successful; after execution, the system should be in a state where no further inputs or actions are expected and the use case can be initiated again, or the system is in a defined error state.

When naming use cases, adopt a verb–noun phrase convention (for example, ‘Make Order’, ‘Process Payment’, or ‘Generate Report’). This convention ensures that the name conveys the action being performed and the object upon which it acts, making the use case model immediately readable.

Key Concept — Use Case

A use case represents a specific function that the system performs in response to an actor’s interaction. It must deliver observable value to the actor and describe a complete behaviour sequence — from trigger to stable outcome. Use cases are named with verb–noun phrases (e.g. ‘Make Order’) and depicted as ellipses within the system boundary.

2.2.3 Actor

An actor represents a user or another system that interacts with the system being designed. Actors are drawn as stick figures positioned outside the system boundary, with the actor’s name written beneath. It is important to understand that an actor does not represent a specific individual but rather a role that an entity plays when interacting with the system.

Actors can be classified into three broad categories. A human user is the most common type — for example, a customer, an administrator, or a warehouse manager. An external system is another software application or service that communicates with the system under design — for example, a payment gateway or an email server. A hardware device is a physical entity that provides input to or receives output from the system — for example, a barcode scanner or a sensor array.

The distinction between primary and secondary actors is also significant. A primary actor initiates the interaction with the system: it has a goal that the system helps to achieve. A secondary (or supporting) actor is called upon by the system during the execution of a use case — for example, a credit-checking service that is invoked when processing an order. Understanding which actors are primary and which are secondary clarifies the direction of interaction and helps analysts ensure that all stakeholder needs are captured.

Actor Type	Description	Example
Human User	A person who interacts directly with the system through its user interface	Customer, Administrator, Warehouse Manager
External System	Another software application or service that communicates with the system	Payment Gateway, Email Server, ERP System
Hardware Device	A physical device that provides input to or receives output from the system	Barcode Scanner, Sensor Array, Printer

2.2.4 Interactions and Relationships

Interactions in a use case diagram are represented by connections between actors and use cases, or between use cases themselves. These connections take several forms, each with a distinct semantic meaning. Understanding the different relationship types is essential for building accurate and expressive use case models.

The four principal relationship types are association, generalisation, «include», and «extend». Each serves a different design purpose and is represented with a distinct notational convention. The table below summarises these relationship types.

Relationship	Notation	Connects	Meaning
Association	Solid line	Actor ↔ Use Case	The actor participates in (communicates with) the use case
Generalisation	Solid line with hollow arrowhead	Actor ↔ Actor or Use Case ↔ Use Case	Inheritance: the child inherits the behaviour of the parent
«include»	Dashed arrow with «include» label	Use Case → Use Case	The base use case always includes the behaviour of the included use case
«extend»	Dashed arrow with «extend» label	Use Case → Use Case	The extending use case optionally adds behaviour to the base use case

Key Concept — Use Case Relationships

Four relationship types structure a use case diagram: association (actor participates in a use case), generalisation (inheritance between actors or use cases), «include» (mandatory sub-functionality always executed), and «extend» (optional behaviour added under specified conditions). Correct use of these relationships ensures the model is both precise and maintainable.

2.3 Technique for Documenting Use Cases

While use case diagrams provide a valuable high-level visual overview of a system's functionality, they do not capture the detail needed for implementation. Each use case must therefore be accompanied by a structured textual description that elaborates on the behaviour, conditions, and interactions in full detail. This technique for documenting use cases is a systematic approach comprising several standard sections, each capturing a specific aspect of the use case.

2.3.1 Use Case Name and ID

Every use case description begins with a name and an identifier. The name follows the verb–noun convention discussed earlier (e.g. 'Make Order', 'Register Customer'). The ID is a unique reference number (e.g. UC-001) that enables the design team to track design decisions on specific requirements throughout the project. Traceability is a core principle of requirements engineering:

every requirement should be identifiable and traceable from elicitation through to implementation and testing.

2.3.2 Importance Level

The importance level section explicitly prioritises each use case relative to others. Typical levels are high, medium, and low. This prioritisation enables users and analysts to agree on which use cases should be implemented in the earliest versions of the system and which can be deferred to later releases. In iterative and agile development processes, this prioritisation directly influences sprint planning and backlog ordering.

The importance level is not merely a developer convenience — it is a business decision. Stakeholders determine priority based on factors such as business value, regulatory obligations, technical risk, and user demand. By recording importance levels explicitly, the team ensures that these decisions are transparent, documented, and revisitable.

2.3.3 Stakeholders, Primary Actors, and Description

The stakeholders and their interests section identifies who cares about this use case and why. Stakeholders are not limited to direct users; they include anyone with an interest in the use case's outcome — business managers, regulators, customers, and support staff. The focus here is on the values and benefits that the use case brings to the business.

The primary actor is the entity that directly initiates the interaction with the system. For each use case there is typically one primary actor, although the use case may involve multiple supporting actors. The description section provides a concise statement of the aim that the use case intends to achieve — a one- or two-sentence summary of the use case's purpose that serves as a quick reference for the design team.

2.3.4 Trigger

The trigger section identifies the event that causes the use case to execute its sequence of actions. Triggers fall into two categories. An external trigger originates from an actor outside the system — for example, a customer placing an order or an administrator requesting a report. A temporal trigger is generated by the system itself based on a schedule — for example, a system timer that initiates a data backup at midnight every day.

Clearly identifying triggers is important because they define the entry conditions for each use case. If a trigger is missing or ambiguous, the design team may fail to account for all the ways in which a use case can be invoked, leading to incomplete or incorrect behaviour in the final system.

2.3.5 Conditions

The conditions section specifies the business rules that control the use case's behaviour. Conditions define the constraints under which certain actions are performed or certain transitions occur. For example, a condition for closing an order might be: 'when the payment is received AND the goods are received by the customer, then the order must be closed.'

Business rules expressed as conditions are essential for ensuring that the system behaves correctly under all circumstances. They capture the domain logic that governs how the system responds to different combinations of events and states. During implementation, these conditions translate directly into conditional statements, validation rules, and workflow logic in the application code.

2.3.6 Relationships in Use Case Descriptions

The relationship section of a use case description documents the range of possible relationships that this particular use case has with actors and other use cases. This section complements the visual representation in the diagram by providing textual detail about each relationship.

An association indicates that an actor is linked to the use case — that is, the actor communicates with and participates in the use case. In the diagram, this is shown as a solid line (a communication association) connecting the actor to the use case ellipse.

An «include» relationship indicates that the base use case always incorporates the behaviour of another use case. This is used when a specific behaviour is common to multiple use cases and should be factored out to avoid duplication. For example, both 'Withdraw Cash' and 'Transfer Funds' might include 'Authenticate User'. The included use case is not optional — it is always executed as part of the base use case.

An «extend» relationship indicates that a use case may optionally extend the behaviour of a base use case under certain conditions. The extending use case adds functionality that is not always needed. For instance, an 'Online Help' use case might extend 'Perform ATM Transaction' only when the customer selects the help button. Importantly, the arrow in an «extend» relationship is drawn from the extending use case to the base use case, and the conditions under which the extension occurs are defined and captured in the base use case.

A generalisation relationship represents inheritance between actors or between use cases. When use cases are related by generalisation, a child use case inherits the behaviour of the parent use case and may add or override specific behaviours. For example, 'Make Withdrawal' and 'Make Deposit' might both be specialisations of a general 'Make Transaction' use case, inheriting common steps such as authentication and receipt generation while defining their own specific processing logic.

Key Concept — «include» vs. «extend»

The «include» relationship denotes mandatory sub-functionality: the included use case is always executed as part of the base use case. The «extend» relationship denotes optional behaviour: the extending use case adds functionality only when specified conditions are met. In both cases, the design intent is to promote reuse and reduce duplication, but the semantics differ — «include» is unconditional, whereas «extend» is conditional.

2.3.7 Flow of Events, Sub-Flows, and Alternative Events

The flow of events section describes the dynamic behaviour exhibited during the execution of the use case. It provides a step-by-step account of the activities that take place, typically written in structured natural language (for example, 'if x then perform activity1, else perform activity2'). The flow of events is the primary input for activity diagramming in Step 2 of the UML methodology.

Sub-flows identify common constituent behaviours that appear in more than one use case. A sub-flow is a part of the flow of events that can be shared by other use cases, reducing duplication and improving consistency. For example, the sub-flow ‘validate payment details’ might appear in both ‘Make Order’ and ‘Process Refund’.

The alternative and exceptional events section captures potential events that may not have been anticipated by the users initially but are specified by the systems analysts based on their expertise. These include error conditions, boundary cases, and recovery procedures. For example, an alternative event might describe an auto-backup mechanism that activates in case of interruption by power failure. Documenting these exceptional paths is essential for building robust systems that handle failure gracefully.

2.4 Summary of Use Case Description Sections

The table below provides a consolidated summary of all sections in a use case description, together with their purpose and a brief example to illustrate each section.

Section	Purpose	Example
Use Case Name	Identifies the use case with a verb–noun phrase	Make Order
ID	Unique identifier for traceability and tracking	UC-001
Importance Level	Prioritises the use case (high, medium, low)	High
Stakeholders & Interests	Identifies who benefits and what value is delivered	Customer: wants to place orders quickly and reliably
Primary Actor	The actor who initiates the interaction	Registered Customer
Description	Concise statement of the use case’s aim	Allows a customer to place a new product order
Trigger	The event that causes execution to begin	Customer clicks ‘Place Order’ button (external trigger)
Conditions	Business rules controlling behaviour	Payment received AND goods delivered → close order
Relationships	Links to actors and other use cases	Association with Customer; «include» Validate Payment
Flow of Events	Step-by-step description of dynamic behaviour	1. Customer selects items 2. System calculates total ...
Sub-Flows	Reusable constituent behaviours shared across use cases	Validate Payment Details (shared with Process Refund)
Alternative Events	Exceptional or unanticipated paths	Power failure triggers auto-backup of order state

Session 3: Use Case Diagramming — Guidance for Modelling

3.1 Recap of Concepts and Notations

Before proceeding to the modelling guidance, it is worth consolidating the four core concepts introduced in Session 2: the system boundary, use cases, actors, and interactions (relationships). These concepts form the vocabulary of every use case diagram, and fluency with them is a prerequisite for effective modelling.

The system boundary (a named rectangle) defines the scope of the system. Use cases (ellipses inside the boundary) represent the functions the system performs. Actors (stick figures outside the boundary) represent the users or external systems that interact with it. Relationships connect these elements: associations link actors to use cases; «include» and «extend» link use cases to one another; and generalisations express inheritance between actors or between use cases. Together, these elements provide a complete, unambiguous visual specification of the system's functional scope.

Concept	Notation	Location	Purpose
System Boundary	Named rectangle	Encloses all use cases	Defines the scope of the system
Use Case	Ellipse (oval)	Inside the boundary	Represents a specific system function
Actor	Stick figure	Outside the boundary	Represents a user, external system, or device
Association	Solid line	Actor ↔ Use Case	Indicates participation in a use case
«include»	Dashed arrow	Use Case → Use Case	Mandatory sub-functionality
«extend»	Dashed arrow	Use Case → Use Case	Optional conditional behaviour
Generalisation	Solid line, hollow arrowhead	Actor ↔ Actor or UC ↔ UC	Inheritance of behaviour

3.2 Guidance for Use Case Diagramming

Producing a high-quality use case diagram requires more than knowing the notation; it demands a disciplined approach to requirements analysis and modelling. The following guidance distils best practice into actionable principles that students should follow when constructing their own use case models.

3.2.1 Adopt the Functional Requirements Document

The starting point for use case diagramming is always the functional requirements document (or problem description). From this document, the analyst must extract four key elements: the scope of the intended system (its purpose and goal), the primary actors (both internal and external) and the use cases they initiate, the behaviours of each use case, and the business rules that control those behaviours.

This extraction process is inherently analytical. The analyst reads the problem description critically, identifying nouns that suggest actors and entities, verbs that suggest use cases and actions, and conditional statements that suggest business rules. This 'noun–verb analysis' technique is a well-established method in object-oriented analysis and applies directly to use case identification.

3.2.2 Apply Use Case Diagramming Principles

When constructing the diagram itself, several principles should be observed. First, there must be no duplicated use cases in the model: if two use cases describe the same functionality, they should be merged or one should be decomposed using «include». Duplication leads to inconsistency and confusion, particularly as the model evolves over time.

Second, suitable relationships must be used to connect the diagram elements. Associations connect actors to the use cases they participate in. Generalisations express inheritance between actors (e.g. 'Registered Customer' generalises 'Customer') or between use cases (e.g. 'Pay by Credit Card' and 'Pay by Debit Card' generalise 'Pay'). The «include» relationship factors out mandatory shared behaviour, while the «extend» relationship captures optional extensions.

Third, every use case should be connected to at least one actor. An orphan use case — one with no actor association — suggests either a missing actor or a use case that does not deliver value to any stakeholder. Similarly, every actor should be associated with at least one use case; an isolated actor indicates an incomplete analysis.

Key Concept — Use Case Modelling Principles

Three core principles guide use case diagramming: (1) no duplicated use cases — each function appears exactly once; (2) appropriate relationships — use association, generalisation, «include», and «extend» correctly; and (3) completeness — every use case has at least one actor, and every actor has at least one use case.

3.3 Requirements Acquisition and Articulation

Requirements acquisition is the process of discovering what the system must do. In the context of use case modelling, this means identifying actors and the functional requirements they impose on the system. A systematic approach to requirements acquisition involves structured interviews with stakeholders, observation of existing workflows, review of existing documentation, and — increasingly — the use of AI-assisted prompting to explore requirements systematically.

Developing well-structured prompt sets can guide the discovery of actors and functional requirements. For example, prompts such as 'Who are the primary users of the system?', 'What external systems must the application interact with?', and 'What are the core functions that deliver value to each actor?' help analysts explore the problem space methodically. When used with large language models (LLMs), these conceptual prompts can accelerate requirements discovery while encouraging critical thinking about the completeness and accuracy of the resulting model.

Requirements articulation refers to the process of expressing discovered requirements in a clear, structured, and unambiguous form. Use case descriptions (as detailed in Session 2) are the primary vehicle for articulating functional requirements. The use case description technique provides a standardised template that ensures consistency across the project and facilitates review and validation by stakeholders.

3.4 Utilising the Use Case Description Technique

The use case description technique introduced in Session 2 is not merely a documentation exercise; it is a design tool that deepens conceptual understanding and supports modelling. By systematically working through each section of the description — name, ID, importance, stakeholders, actors, description, trigger, conditions, relationships, flow of events, sub-flows, and alternative events — the analyst is forced to think rigorously about every aspect of the use case.

Designing prompt sets to support use case description is a powerful technique. For instance, prompts such as 'What triggers this use case?', 'What business rules govern its behaviour?', and 'What alternative or exceptional paths might occur?' guide the analyst through the description process systematically. When used with LLMs, these prompts can generate initial drafts of use case descriptions that the analyst then refines critically, ensuring accuracy and completeness.

3.5 Design Steps — Building a Use Case Model

In practice, a use case model is built incrementally by designing individual components and then integrating them into a coherent whole. The lecture slides demonstrate this process using an e-commerce example with several actor components. Each component focuses on a specific actor or group of related actors and their associated use cases, before all components are brought together in an integrated use case diagram.

3.5.1 Component: Non-Registered Customer

The first component considers the non-registered customer — a user who visits the system without an existing account. Typical use cases for this actor might include 'Browse Catalogue', 'Search Products', and 'Register Account'. The non-registered customer represents the broadest audience and often serves as the entry point for new users. Modelling this component ensures that the system accommodates first-time visitors and provides them with a clear path towards registration and full functionality.

3.5.2 Component: Registered Customer

The registered customer component builds upon the non-registered customer by adding use cases that require authentication — for example, 'Place Order', 'View Order History', 'Update Profile', and 'Track Delivery'. A generalisation relationship between 'Registered Customer' and 'Non-Registered

Customer' may be appropriate if the registered customer inherits all the browsing capabilities of the non-registered customer while gaining additional transactional functionality.

3.5.3 Component: Contracted Suppliers

The contracted supplier component introduces an external actor that interacts with the system from the supply side rather than the demand side. Typical use cases might include 'Update Stock Levels', 'Process Purchase Order', and 'Confirm Shipment'. Modelling supplier interactions ensures that the system accounts for the full supply chain, not just the customer-facing functionality. This component often introduces «include» relationships where supplier interactions depend on shared sub-processes such as 'Validate Credentials'.

3.5.4 Component: Payment Processing (Generalisation)

The payment component is a particularly instructive example of the generalisation relationship between use cases. Instead of modelling separate, unrelated use cases for each payment method, the designer creates a general 'Pay' use case and specialises it into 'Pay by Credit Card', 'Pay by Debit Card', 'Pay by PayPal', and so on. This application of the object-oriented concept of generalisation reduces duplication, clarifies the relationship between payment methods, and makes it straightforward to add new payment methods in the future.

The generalisation approach to payment processing also illustrates the open–closed principle at the requirements level: the payment model is open for extension (new payment methods can be added as new child use cases) but closed for modification (the base 'Pay' use case and its relationships remain unchanged).

Key Concept — Component-Based Use Case Design

Building a use case model component by component — one actor or actor group at a time — is a practical strategy for managing complexity. Each component is designed and validated independently before being integrated into the complete model. This incremental approach reduces errors, facilitates stakeholder review, and ensures that all actors and their use cases are accounted for.

3.6 Integrating the Design Components

Once all individual actor components have been designed, they must be integrated into a single, coherent use case diagram. Integration involves placing all actors and use cases within or around a single system boundary, resolving any overlaps or conflicts between components, and ensuring that shared use cases (those accessed by multiple actors) are represented once and connected to all relevant actors.

During integration, the analyst should verify several properties of the model. First, every use case should be unique — no two ellipses should represent the same functionality. Second, all relationships should be correctly typed: associations for actor–use-case links, «include» for mandatory sub-behaviours, «extend» for optional extensions, and generalisations for inheritance hierarchies. Third, the model should be complete: every known functional requirement should be represented as a use case, and every stakeholder role should be represented as an actor.

The integrated diagram is the primary deliverable of Step 1 in the UML methodology. It provides a single-page overview of the system's functional scope that can be presented to stakeholders for validation. Stakeholders can review the diagram to confirm that all their requirements are represented, that the system boundary is correctly drawn, and that the relationships between actors and use cases accurately reflect their understanding of the domain.

3.7 The Use Case Modelling Process

The overall use case modelling process can be understood as a cycle of problem analysis, design reasoning, and validation. The process begins with the problem context: the analyst examines the problem description, domain knowledge, and stakeholder input to understand what the system must accomplish. This understanding is then formalised through use case modelling — identifying actors, defining use cases, documenting descriptions, and constructing the diagram.

Reasoning for design is a critical component of this process. The analyst does not simply transcribe requirements; they make deliberate design decisions about how to structure the model, which relationships to use, how to decompose complex behaviours, and where to apply generalisation. These decisions should be justifiable and documented, as they directly affect the quality and maintainability of the resulting design.

Validation closes the loop. The completed use case model is reviewed against the original requirements and stakeholder expectations. Prompt sets designed for validation can be particularly useful here — for example, 'Does every actor have at least one associated use case?', 'Is every functional requirement represented by a use case?', and 'Are the «include» and «extend» relationships correctly applied?' Using LLMs to validate models can provide rapid feedback, although the analyst must always apply critical judgement to the AI's suggestions.

Key Concept — The Use Case Modelling Process

Use case modelling follows a cycle of problem analysis, design reasoning, and validation. The analyst extracts requirements from the problem context, makes deliberate design decisions to structure the model, and validates the result against stakeholder expectations. This process is iterative: validation feedback may reveal gaps or errors that require revisiting earlier steps.

3.8 What to Do Next

Following the three sessions on use case diagrams, students should consolidate their understanding through a combination of review, practice, and critical engagement with AI tools. The recommended next steps are as follows.

- Review the key messages for use case diagrams — ensure a solid understanding of the system boundary, actors, use cases, relationships («include», «extend», generalisation), and the use case description technique.

- Conduct the dedicated formative assessment on Blackboard to enhance understanding of use case modelling and to receive feedback on any areas of weakness.
- Carry out the dedicated practical tasks and instructions on Blackboard, which provide hands-on experience with constructing use case diagrams and writing use case descriptions for realistic problem scenarios.
- Critically apply conceptual prompts to LLMs (e.g. ChatGPT, Gemini, Copilot) to enhance the accuracy of user requirements modelling and accelerate the design process — always engaging critically with the AI output rather than accepting it uncritically.

As students progress through the module, the use case models produced in these sessions will serve as the foundation for activity diagrams (Step 2), class diagrams (Step 3), state transition diagrams (Step 4), and sequence diagrams (Step 5). Building a strong use case model now will pay dividends throughout the remainder of the module and in the summative assessments.

References and Further Reading