UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURE 04

Requirements Engineering: Use Case Diagrams

Prof. Luigi Libero Lucio Starace

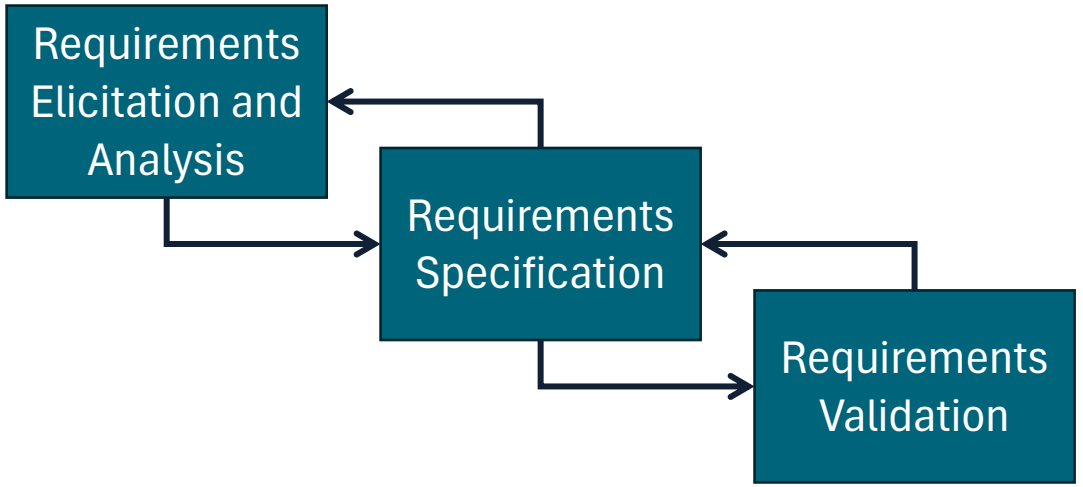
luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace

Previously, on Software Engineering

- We've seen what Requirement Engineering is and why it's important in the SDLC
- We got an overview of the main challenges and understand the main RE processes
- We looked into Requirements Elicitation and Analysis



The Requirement Engineering Process

The Software Life Cycle

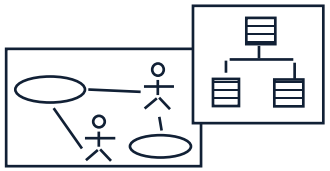
Requirements Engineering → System Design → Software and UI/UX Design → Implemen-tation → Testing → Operation and Maintenance

Requirements collected via:

- Interviews with Stakeholders
- Personas
- Stories and Scenarios

Specified using:

- Use Cases
- Natural Language
- Domain Models
- Mock-ups



Define System Architecture

- Requirements are allocated to software sub-systems
- Sub-systems are allocated to hardware resources
- Architectural Patterns

Define Subsystems

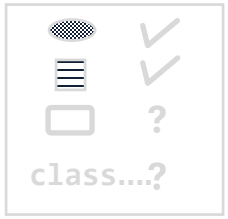
- Objects required to realize each subsystem are defined.
- Software Design Patterns
- Usability Engineering
- High-fidelity Wireframing

Each Subsystem is implemented

- Source code and other artifacts
- Clean Code
- Frameworks and ORMs
- Focus on Software Quality

Ensure the Software satisfies customers

- Code inspections
- Functional Testing (unit, integration, system testing)
- Usability Testing



System is put into practical use

Maintance will be required at some point

- To fix errors that were not discovered in previous phases
- To adapt the software to changes in requirements on in its environment

Requirements Specification

Requirement Specification

- The process of writing down the user and system requirements in a **requirements document**.
- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development
  - It is therefore important that these are as complete and detailed as possible.

Requirements and Design

- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, **requirements** and **design** are **inseparable**
  - Requirements may be structured and organized based on a high-level system architecture
  - The system may inter-operate with other systems that generate design requirements
  - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement

Requirements Specification

Different approaches are possible:

- **Natural Language**: express requirements as numbered sentences in natural language. Each sentence should express one requirement.
- **Structured Natural Language**: Use a standardized form or template
- **Semi-formal notations and models**: UML Use Case Diagrams and other domain models, typically supplemented by natural language annotations
- **Formal Specification**: These notations are based on mathematical concepts such as finite and infinite state machines, temporal logics

Requirements Specifications

Different approaches are used in different domains

- When engineering **safety-critical** systems, it is common to use formal specifications and structured natural language
- When engineering a to-do list app, one might use unstructured natural language to express requirements
- When engineering a medium-to-large sized information system, leveraging semi-formal notations and models might be a good compromise

Natural Language (NL) Specifications

- Requirements are written as natural language sentences, optionally supplemented by diagrams and tables.
- Used for writing requirements because it is **expressive**, **intuitive** and **universal**.
  - This means that the requirements can be understood by users and customers.
- Can express **functional** and **non-functional** requirements

Guidelines for NL Specification

- Define a “standard” format and use it for all requirements.
  - E.g.: <part of the system> shall <requirement> (<rationale>)
  - E.g.: The overall monthly enrollment report shall include statistics about the gender of enrolled students (this is required to fill the gender equality report)
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.
- Include an explanation (rationale) of why a requirement is necessary.

Problems with Natural Language

- **Lack of clarity**
  - Precision is difficult without making the document difficult to read.
- **Requirements confusion**
  - Functional and non-functional requirements tend to be mixed-up.
- **Requirements amalgamation**
  - Several different requirements may be expressed together.

Structured Specifications

- An approach to writing requirements where the freedom of the requirements writer is limited and a standardized way of writing requirements is enforced.
- This works well for some types of requirements (e.g.: requirements for embedded control systems) but is sometimes too rigid for writing business system requirements.

Structured Specification Example

Insulin Pump/Control Software/SRS/3.3.2	
Function	Compute insulin dose: Safe sugar level.
Description	Computes the dose of insulin to be delivered when the current measured sugar level between 3 and 7 units.
Inputs	Current sugar reading (r_2), the previous two readings (r_0 and r_1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose – the dose in insulin to be delivered.
Action	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requires	Two previous readings, so the rate of change of sugar level can be computed
Precondition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Postcondition	In the memory, r_0 is replaced by r_1, then r_1 is replaced by r_2
Side effects	None

Use Case Diagrams

Use Cases

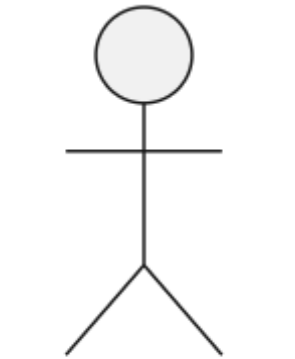
- Use Cases are a way to describe interactions between users and a system using a graphical model and structured natural language text.
- Are a key part of the Unified Modelling Language (UML), and can be used to represent the set of **functional requirements** of a system
- Use cases identify:
  - **Actors** – Categories of users (not necessarily humans) of the system
  - **Use Cases** – Types of interactions (or features) offered by the system
- Additional information on the interactions can be provided as (structured) textual descriptions, or by means of one or more semi-formal models (e.g.: UML Sequence Diagrams or Statecharts)

Use Cases

- Is a way to support communication with the client to define system functionalities
  - Should be as simple as possible
- Does not define «how» the system is implemented, but what the system should do
  - From the point of view of users (the system is a black box)
- Use cases are often documented using a high-level **Use Case Diagram (UCD)**

Actors

- Actors are represented using **stick figures**
- They represent external entities that interact with the **System Under Development** (**SUD**)
  - Classes of (human) users
  - Other systems
  - Physical Environment
- Every actor has a unique name
- Actors are more coarse-grained than Personas
  - One actor may be associated to multiple Personas



Use Cases

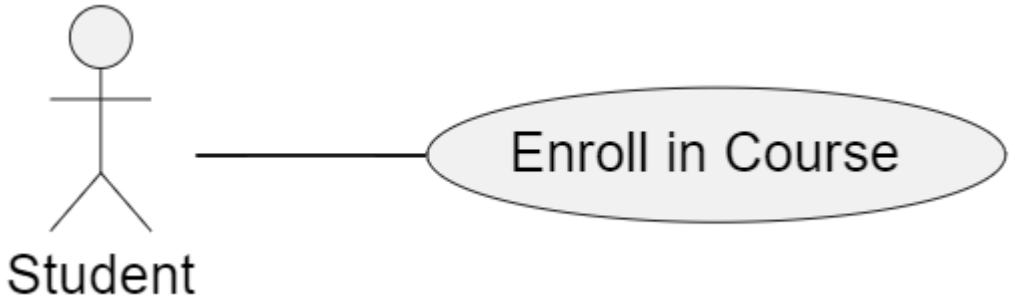
- Use cases are represented as named ellipses
- Correspond to **features** offered by the system, providing some **benefit** or **utility** to actors.
- Use cases model **functional requirements**.
- A use case abstracts many possible **scenarios** (sequences of actions) for a given feature
  - A scenario can be seen as an instance of a use case
  - A use case represents a class of scenarios that aim at using the same functionality
  - E.g.: there might be multiple ways to enroll in a course

Heuristics to Identify Actors

- To identify actors, you may ask yourselves:
  - What groups of users are supported by the SUD in their job?
  - What groups of users perform the main features offered by the SUD?
  - What groups of users perform the secondary functions of the SUD, such as administration?
  - Will the SUD interact with external systems or software?
    - Each external system or software with wich the SUD interacts will be an actor
- Actors do not correspond to a single entity, but rather represent a class of users that can have the same role
  - A user can have different roles in the same system
    - The Administrator of a website can also visit the website as a non-authenticated user

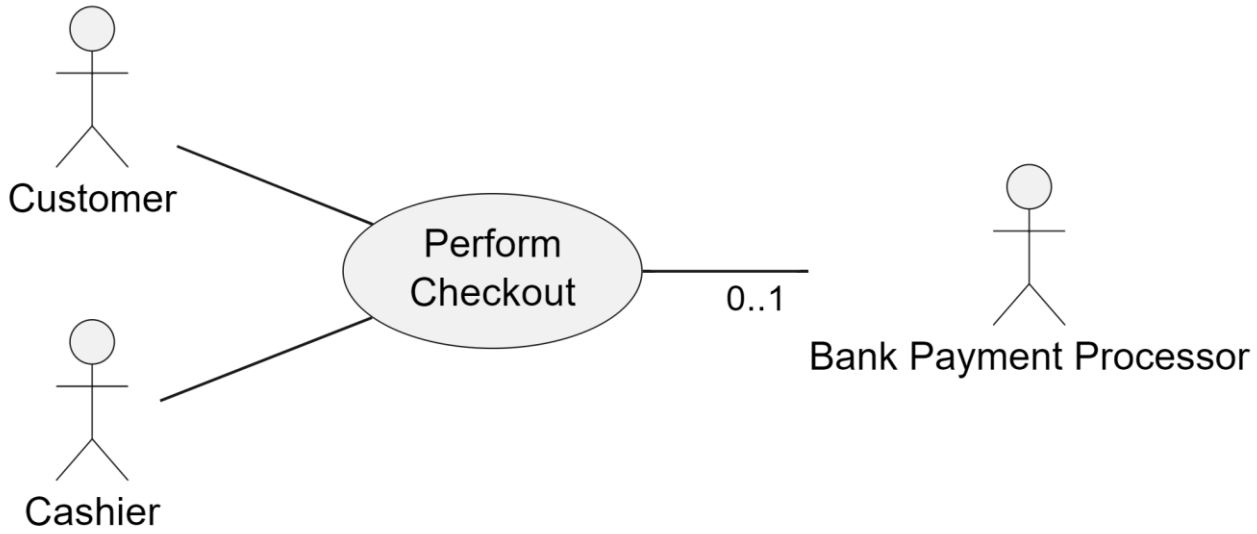
UCD: Associations

- In addition to actors and use cases, use case diagrams include different types of relations between them
- An association between an actor and a use case indicates that the actor can perform the use case
  - Graphically, it's represented as a line connecting an actor and a use case

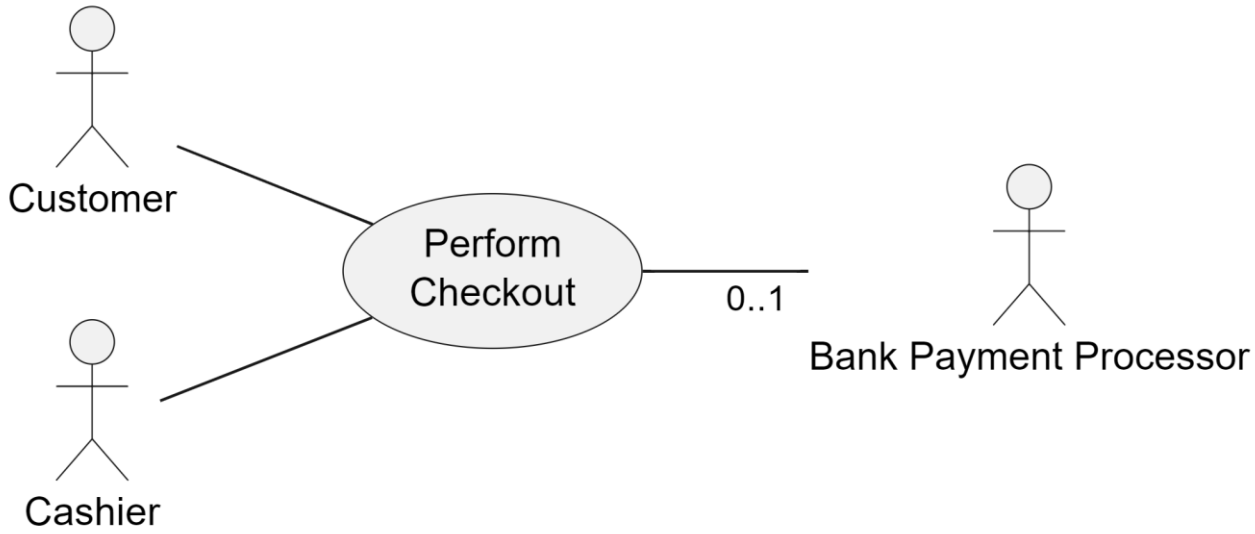


UCD: Secondary Actors

- A Use Case may be associated with multiple actors
- The semantics is that multiple actors to need **collaborate** in some way to perform that use case
- Cardinalities are supported in UML 2.5



UCD: Secondary Actors



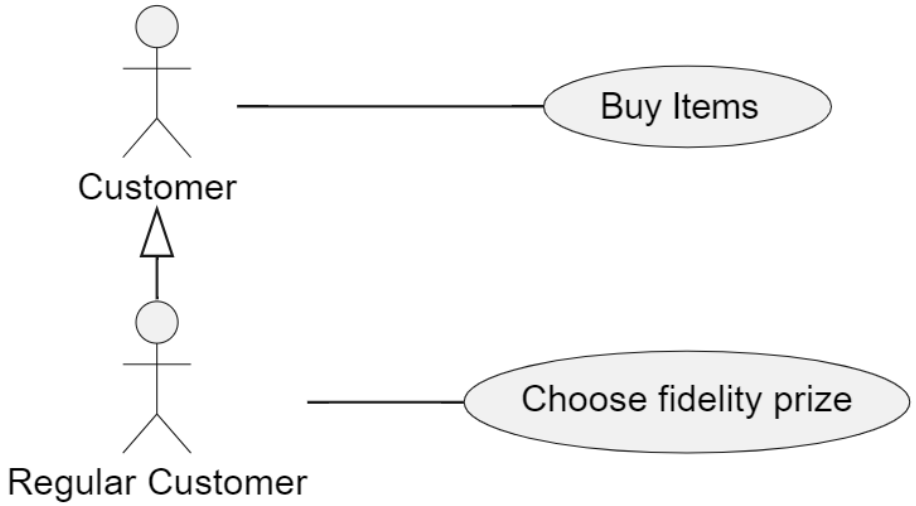
- Customer and Cashier need to cooperate to perform the checkout
- Optionally (cardinality is 0…1), a Bank Payment Processor (an external system) might be involved
  - E.g.: when the customer is paying using a credit/debit card

UCD: Secondary Actors

- UML UCD do not include facilities to specify **how** different actors involved in a use case
- Interactions modes and different responsibilities can be specified with additional descriptions (we'll see in a while)

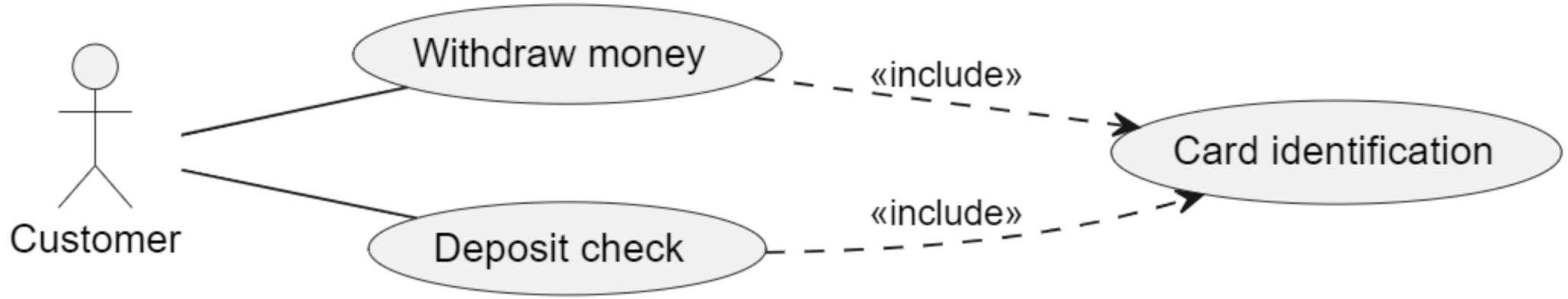
UCD: Actor Generalizations

- Generalization between actors can be applied when one actor is a sub-type of another actor
- Same concept and notation as in UML Class Diagrams
  - Graphically represented as an arrow with an «empty» head
- Specialized actor can perform any use case the parent can perform



UCD: Use Cases «include» Relation

- The «*include*» relationship is intended to be used when there are common parts of the behavior of two or more Use Cases.
- This common part is then extracted to a separate Use Case, to be included by all the base Use Cases having this part in common.

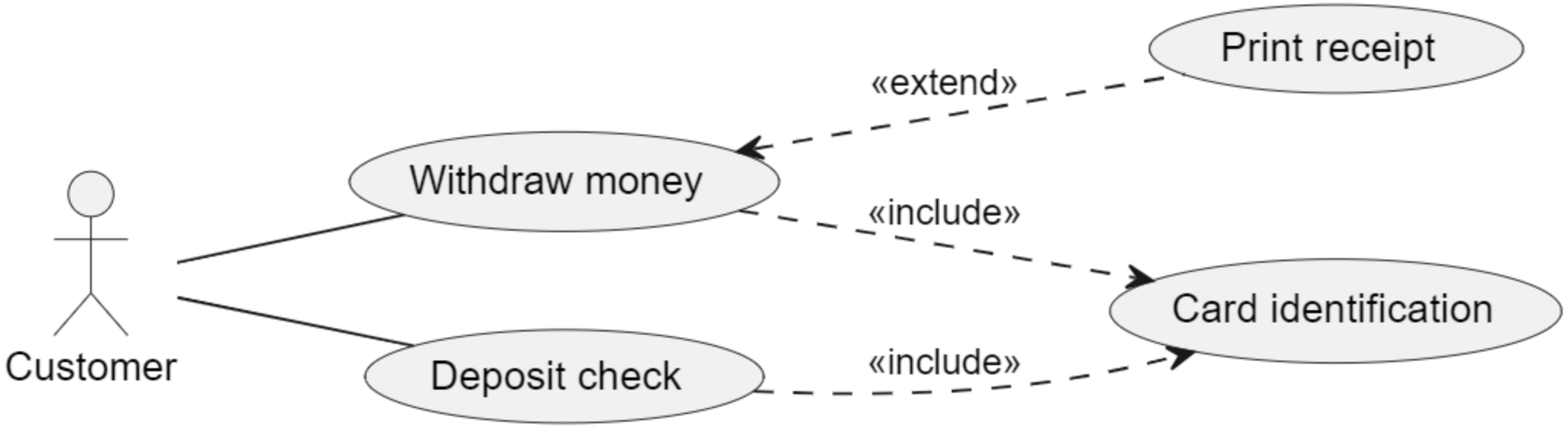


UCD: Use Cases «include» Relation

- As the primary use of the «include» relationship is for reuse of common parts, what is left in a base Use Case is usually not complete in itself but dependent on the included parts to be meaningful.
- This is reflected in the direction of the relationship, indicating that the base Use Case depends on the addition but not vice versa.
- This relation can be useful to:
  - **Decompose** a complex interaction into smaller, more manageable interactions
  - **Factor common sequences** of steps across different use cases

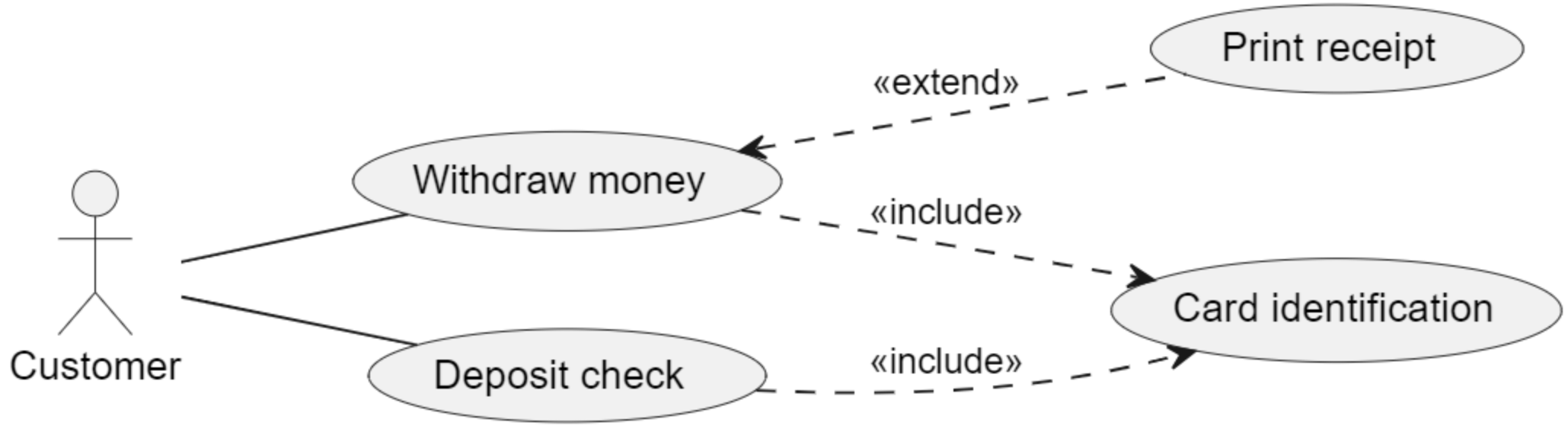
UCD: Use Cases «extend» Relation

- The «*extend*» relationship is intended to be used when there is some additional behavior that **may** be added, **possibly conditionally**, to the behavior defined in one or more Use Cases.



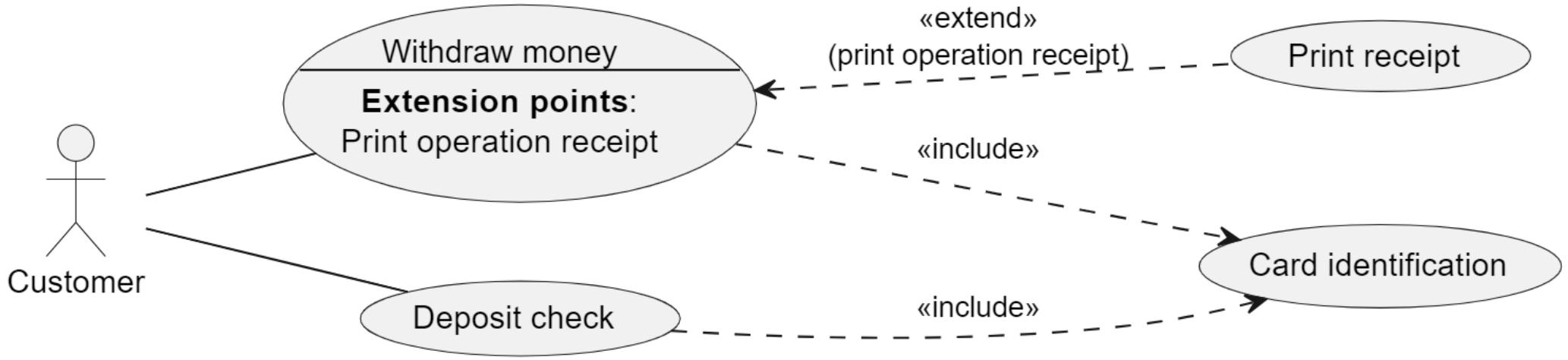
UCD: Use Cases «extend» Relation

- The extended Use Case is defined independently of the extending Use Case and is meaningful independently of the extending Use Case
- On the other hand, the extending Use Case typically defines behavior that may not necessarily be meaningful by itself.



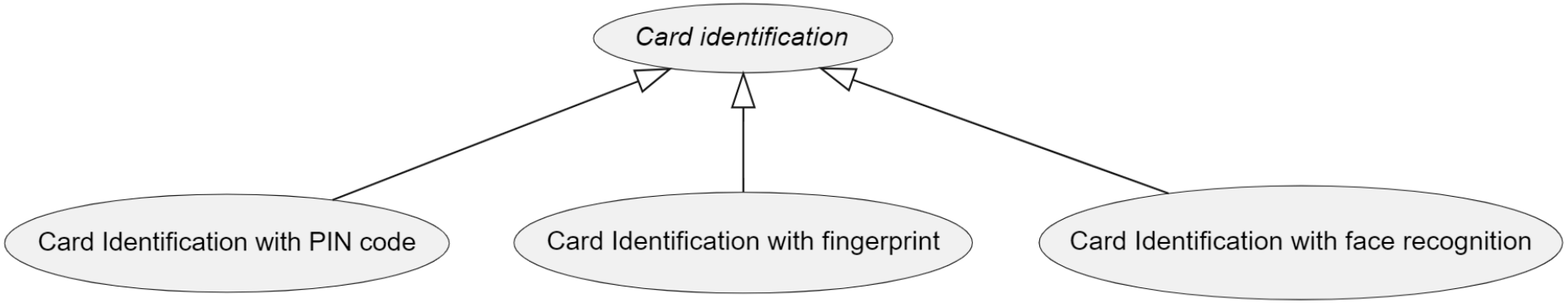
UCD: Use Cases Extension Points

- Extension points identify the point in the base Use Case where the behavior of an extension Use Case can be inserted
- They can be useful when a Use Case can be extended in multiple points

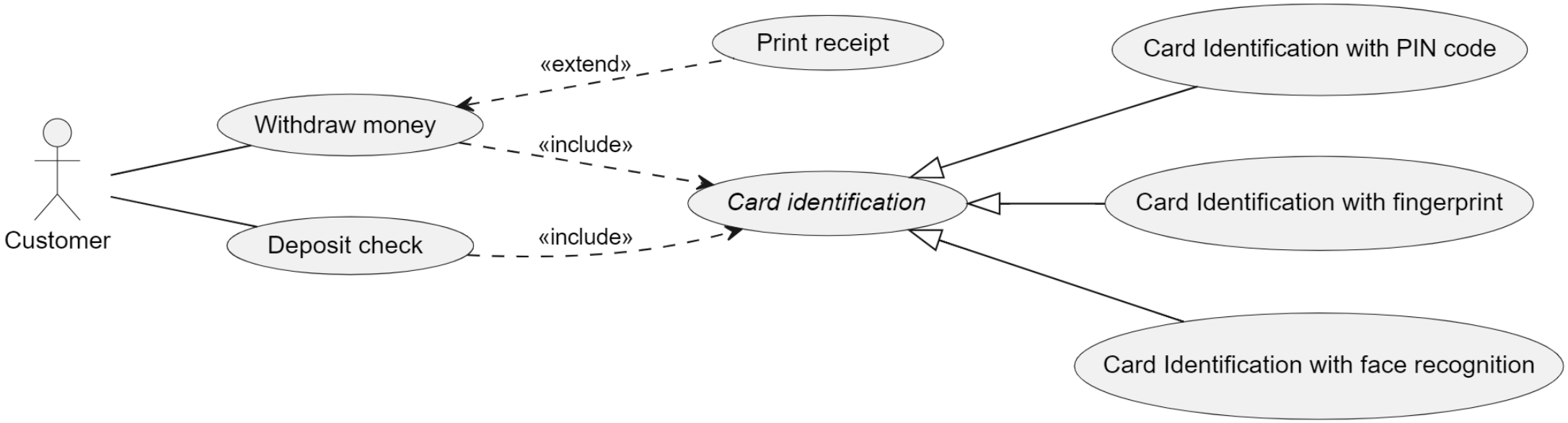


UCD: Use Cases Generalization

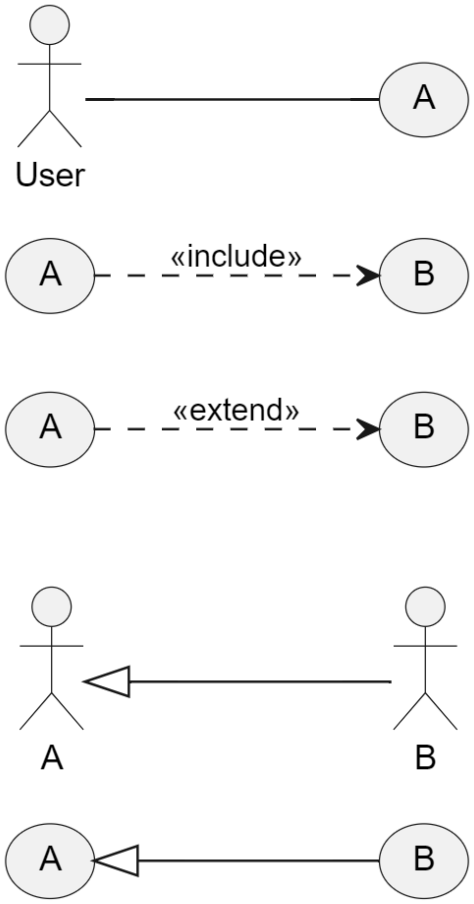
- Generalization between use cases is intended to be used when a Use Case is a specialization of another
- Differently from «extend» dependecies, there is no precise points in which the specialized use cases deviate from the parent
- Specializations can be very different w.r.t. the parent use cases.
- Notation is the standard UML notation for specialization



UCD: Use Cases Generalization

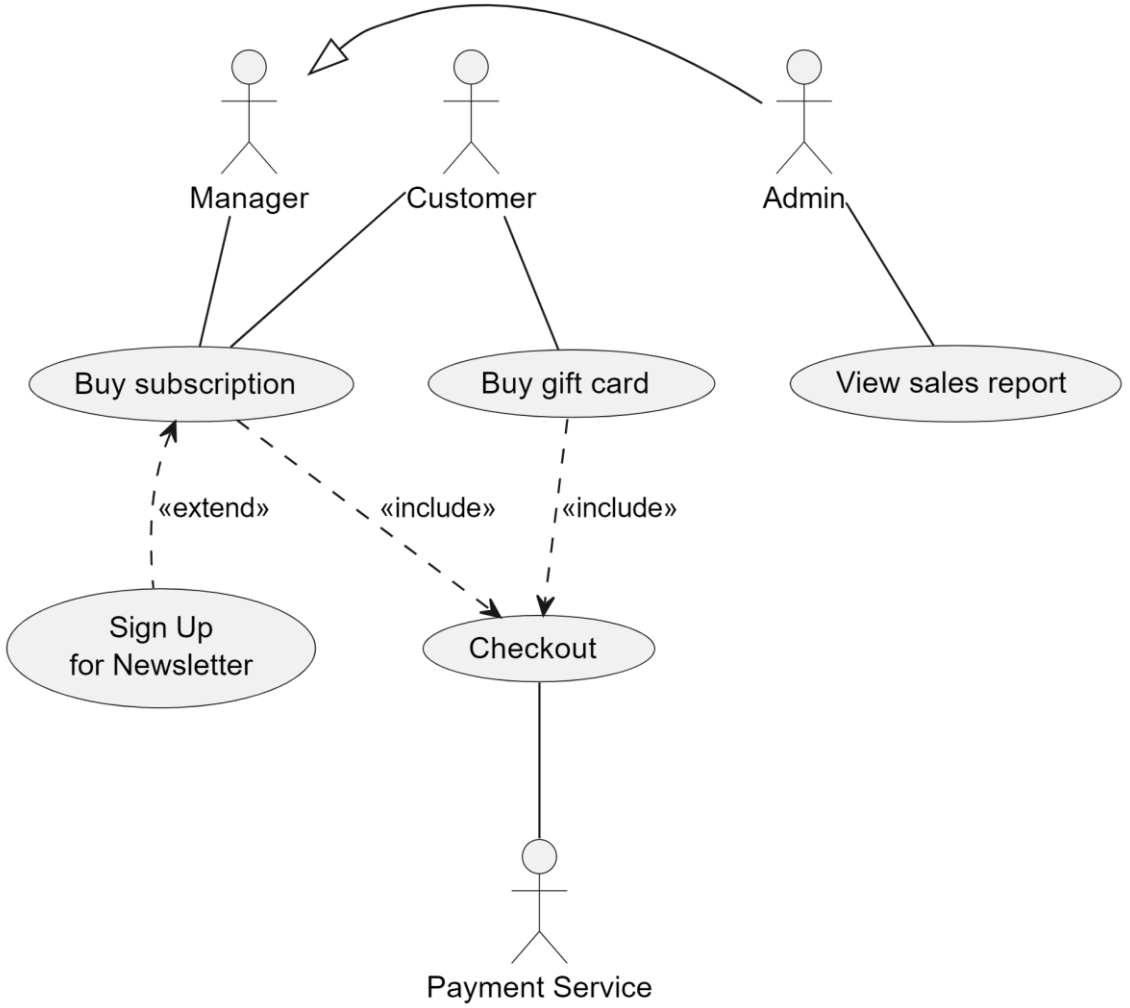


UCD: Relations Overview

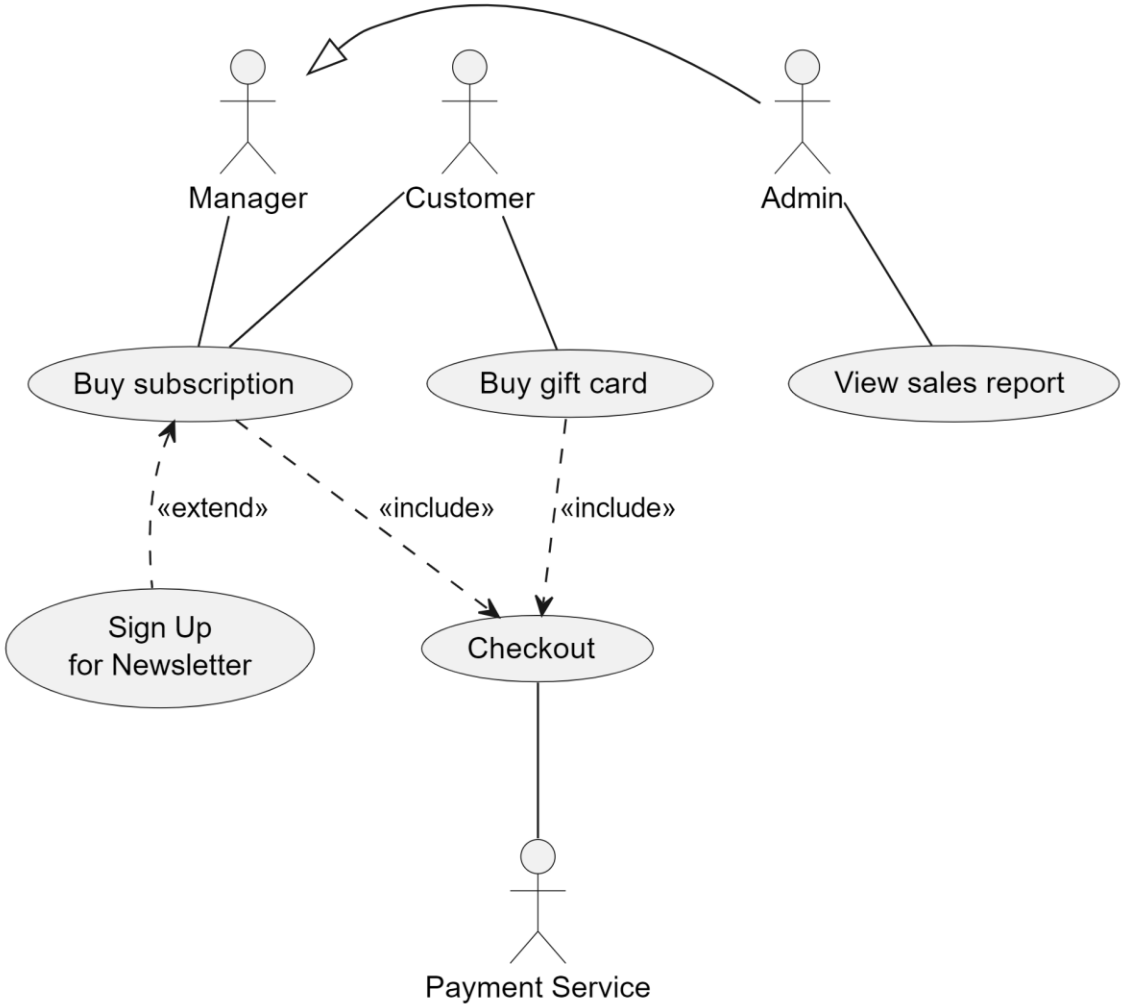


- **Association**: an actor is associated to a use cases
  - User can perform UC A
- **Include**: Common behaviour is factorized
  - UC A includes steps defined in UC B
- **Extend**: Identifies (possibly optional) variations.
  - UC A can include the sequence of steps in UC B
- **Generalization**: can be applied to actors and use cases.
  - B inherits the behaviour of A
  - A can be replaced with an instance of B

Reading a Use Case Diagram



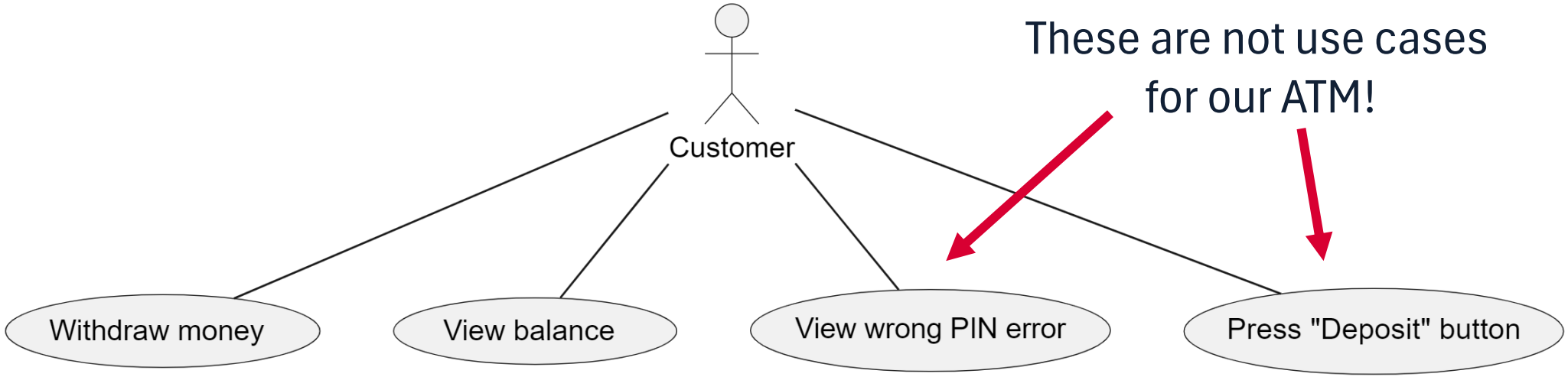
Reading a Use Case Diagram



- Customers can use the system to buy a gift card or a subscription
  - Both use cases include a common checkout behaviour (sequence of steps) for payment
    - This sequence of steps requires collaboration with an external Payment Service
  - To Buy a subscription, the collaboration of a manager (or an admin) is required (e.g.: they may need to approve the purchase)
    - Optionally, when buying a subscription, the customer can sign up for the newsletter
- Admins are the only one that can view sales reports

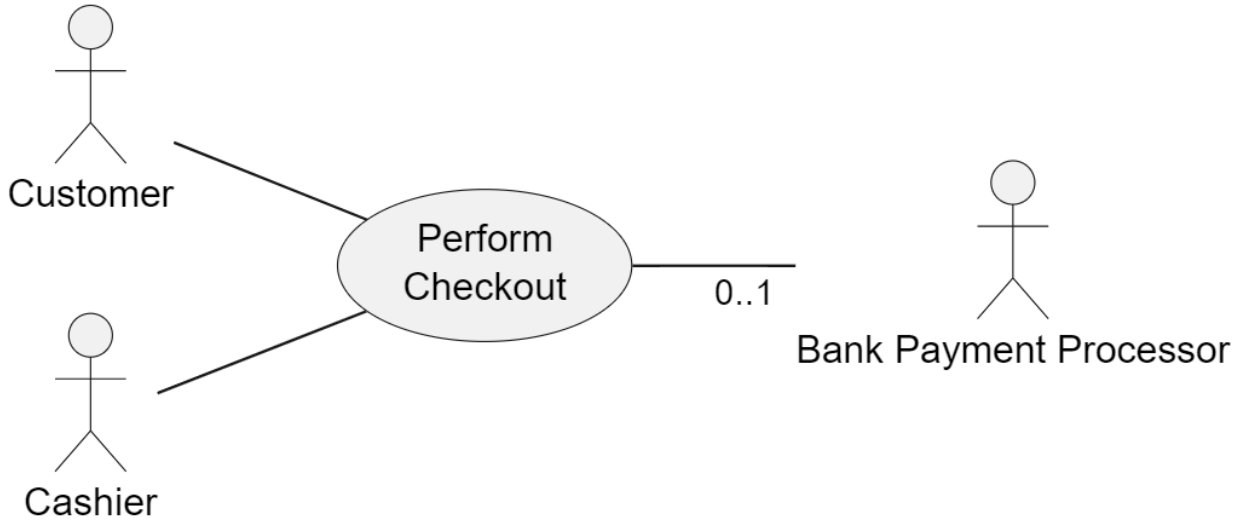
Use Cases: Rookie Mistakes – I

- Use cases should provide some benefit to the actor, help the actor complete his job or achieve some goal.
  - Typically, Use Case names should include a verb
  - Typically, Actor names should be nouns



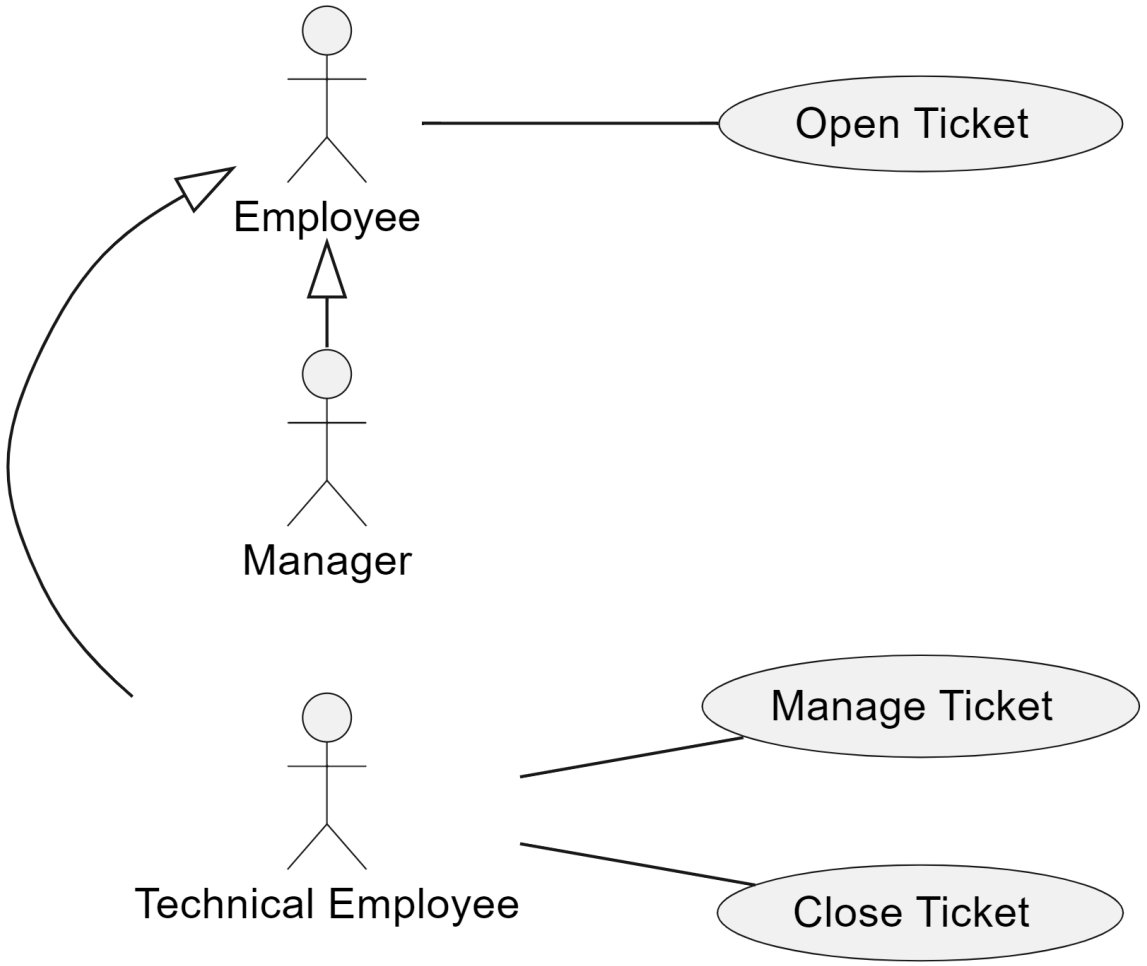
Use Cases: Rookie Mistakes – II

- If two actors are associated with the same use case (with a non-zero cardinality), it means that the two actors are involved (and need to collaborate) in each instance (scenario) of that use case
  - It does not mean that both actors can perform that use case independently!



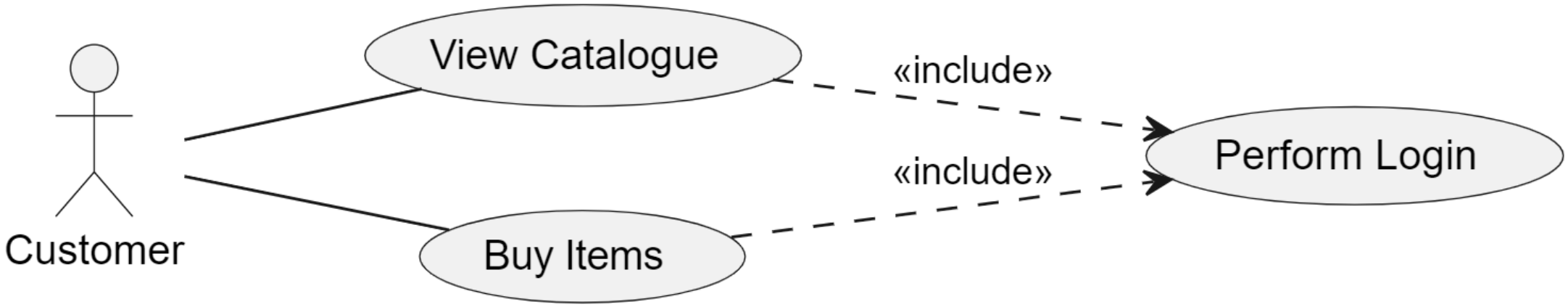
Use Cases: Rookie Mistakes – III

- Beware of improper generalization
  - Every actor should have their own use case(s)
  - Specialized actors can already perform all the use cases of their ancestors
  - If the specialized actors does not have some use cases of their own, the generalization might be useless, or some use cases might be missing



Use Cases: Rookie Mistakes – IV

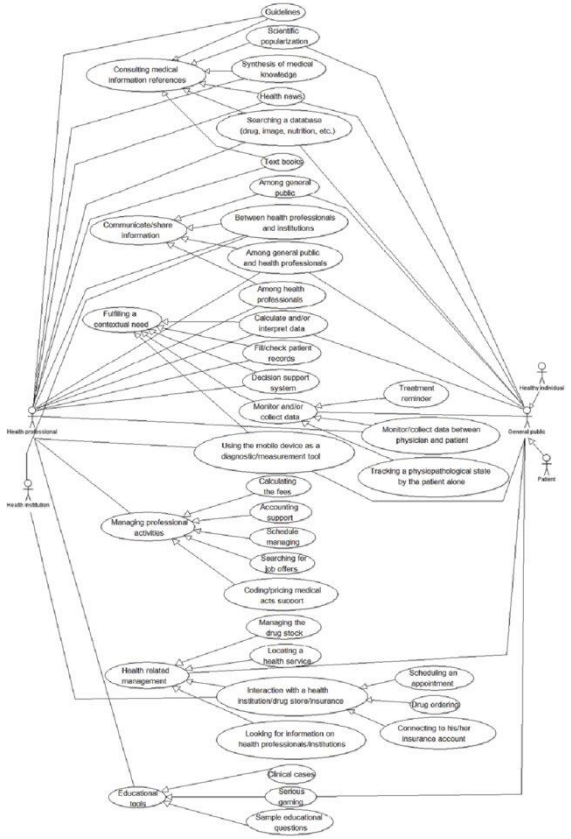
- *«include»* is not a good way to represent temporal relations



  - The above diagram indicates that **Customer** needs to perform the sequence of steps in the **Perform Login** UC every time they want to **Buy Items** or **View Catalogue**!
  - **Preconditions** in the text description of the UC should be used instead

Use Cases: Rookie Mistakes – IV

- Use case diagrams should not get too complex and messy
  - Use relationships between use cases and generalizations between actors with moderation
  - The modeling should be at a sufficiently high degree of abstraction
  - You need to be orderly (try to avoid intersecting lines, etc…)
- A **complex diagram** is an indication of a **bad analyst.**



Yasini, M., & Marchand, G. (2015). Toward a use case based classification of mobile health applications.

Exercises

Volunteers needed!

Exercise #1

Si vuole realizzare un sistema informativo per la gestione di segnalazioni di guasti informatici all'interno di una rete aziendale. Un impiegato dell'azienda, previa autenticazione, può compilare un form specificando una descrizione del problema, un livello di priorità della riparazione, ed il codice di inventario dell'apparecchio guasto. Un tecnico IT ha la possibilità di visualizzare tutte le segnalazioni pendenti, di prenderne in carico una, specificando una data prevista di soluzione, e di segnalarne la chiusura in seguito ad un intervento. In quest'ultimo caso, inserirà una descrizione dell'intervento effettuato, e una stima del tempo impiegato. Infine, un amministratore può visualizzare diversi report, quali ad esempio il numero di segnalazioni evase nell'ultimo mese o settimana.

Exercise #2

Un sistema gestionale per supportare il processo di raccolta e gestione dei requisiti in un'azienda software permette di gestire più progetti software. Per ciascun progetto, è possibile definire uno o più stakeholder, caratterizzati da un nome (e.g.: "Cassiere", "Cliente") e da una eventuale descrizione. Inoltre, per un certo progetto, il sistema permette di specificare uno o più requisiti. Ciascun requisito è caratterizzato da un titolo, da una descrizione, da una lista non vuota di stakeholder interessati, e da un livello di priorità. I Project Manager possono creare nuovi progetti, caratterizzati da un titolo, un committente (e.g.: azienda, pubblica amministrazione, etc.), da una durata prevista, e da un budget (in Euro). I Project Manager possono inoltre associare a ciascun progetto i Requirement Engineer allocati su quel progetto. I Requirement Engineer possono quindi visualizzare soltanto i progetti cui sono assegnati, e creare/modificare requisiti per quei progetti. Quando un Requirement Engineer crea/modifica un requisito, il sistema controlla automaticamente la presenza di ambiguità o inconsistenze nel titolo e nella descrizione, sfruttando il software esterno "Req-GPT". Se Req-GPT rileva potenziali ambiguità, il sistema non permette di salvare il requisito.

Exercise #3

Si vuole realizzare un sistema per la gestione di prestiti di libri.Gli utenti avranno la possibilità di registrarsi al sito inserendo le proprie informazioni personali come nome, cognome, e-mail e password. Ogni utente avrà un profilo univoco nel sistema. Una volta registrati, gli utenti potranno cercare libri nel catalogo inserendo il titolo, l'autore o il genere desiderato. Il sistema mostrerà quindi i dettagli dei libri trovati, inclusi titolo, autore, anno di pubblicazione e stato attuale (disponibile o in prestito). Gli utenti avranno la possibilità di richiedere il prestito di un libro disponibile. Una volta selezionato il libro di cui richiedere il prestito, l'utente potrà procedere con la richiesta di prestito, che sarà completata una volta effettuato il pagamento del contributo fisso di € 3,00. Per il pagamento, l’utente deve inserire il numero della propria carta di credito, con intestatario della stessa, scadenza, e codice CCV. Le informazioni vengono sul pagamento vengono inviate al servizio esterno UninaPay, che processa il pagamento.