Use Case & Domain Modeling with UML

L.EIC-ES-2021-22

J. Pascoal Faria, Ademar Aguiar

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

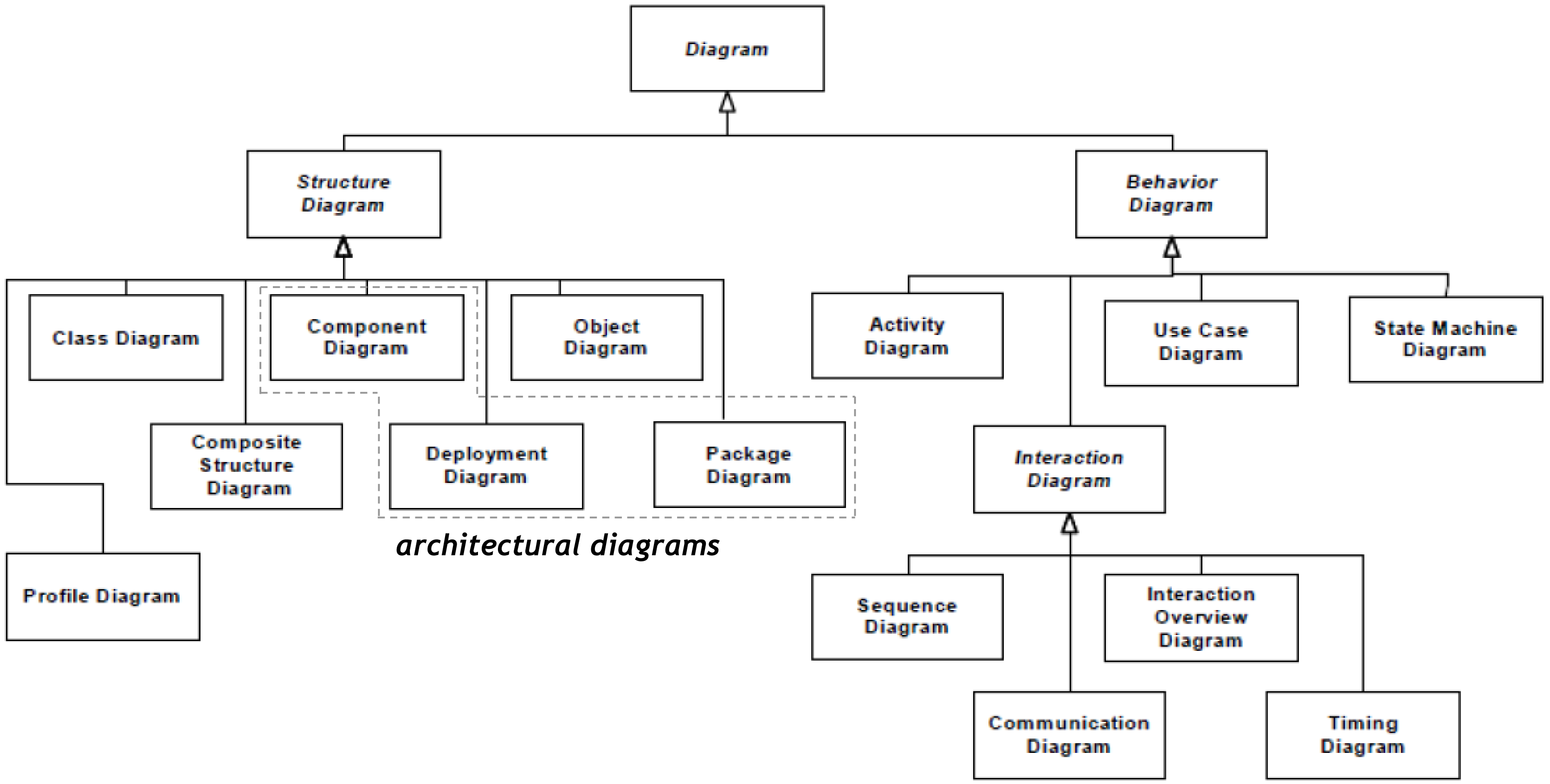
Contents

Introduction

System models in requirements engineering

- A system model is a simplified representation of a system (as-is or to-be) from a certain perspective
- Models are used in many fields of engineering to tackle complexity through abstraction
- UML is the modeling standard in software engineering
- In requirements engineering, (semi-formal) models help removing the ambiguity of natural language descriptions
- The following are helpful in requirements engineering:
  - Use case model – for clarifying the system boundaries & scope and organizing functional requirements (user view)
  - Domain model – for organizing the vocabulary and capturing information requirements in the problem domain

UML diagrams



UML modeling tools

Examples:

- draw.io
  - UML diagrams
  - Simple, online, free, collaborative
- Enterprise Architect
  - UML diagrams and associated documentation
  - More complex, commercial (academic license available)
- Visual Paradigm
- ...

You can use any tool of your choice for drawing UML diagrams for your project.

Running example: ticketing system

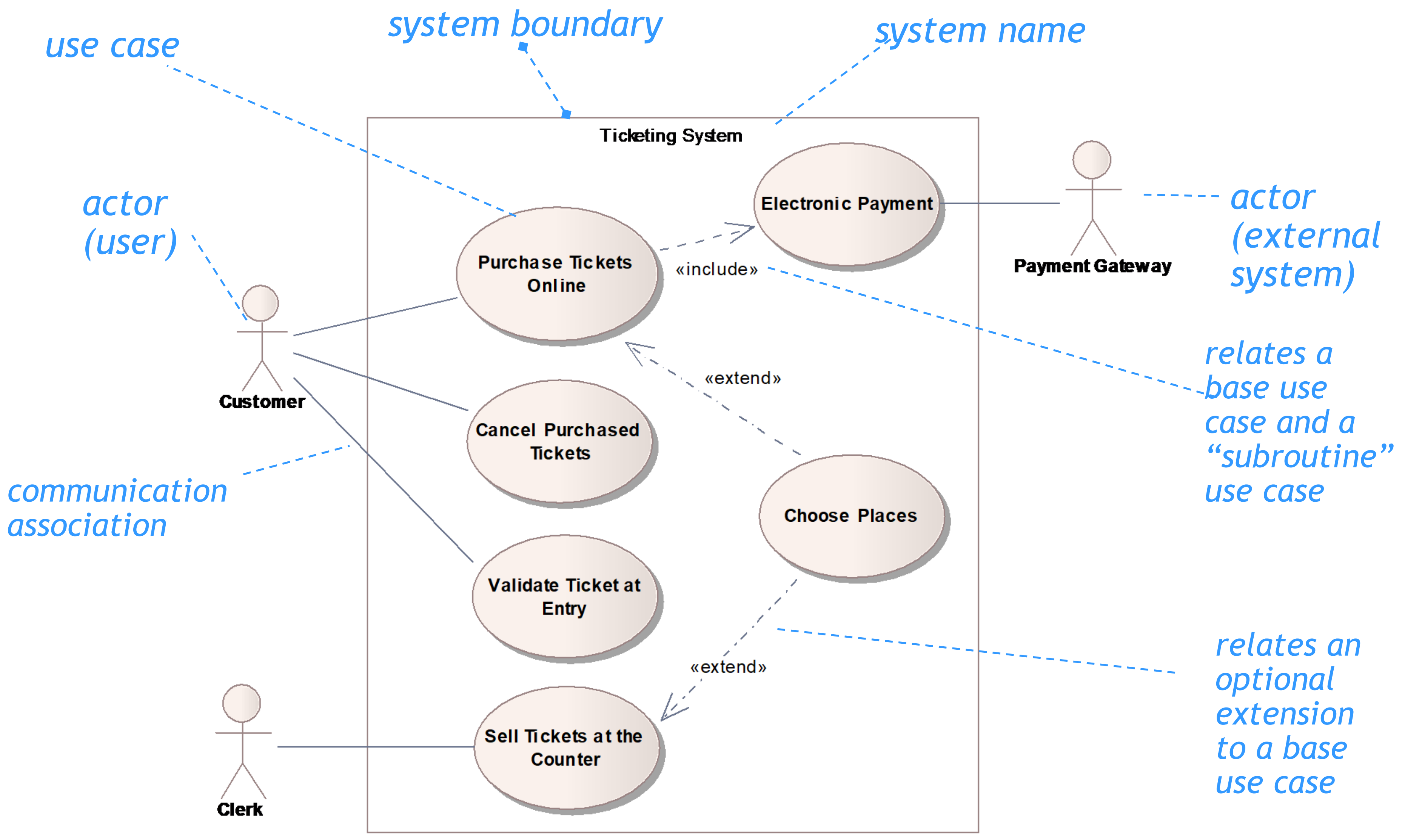
- Assume one wants to develop a ticketing system for various types of events with the following requirements:
  - R1. Customers can purchase tickets online, using common online payment methods (supported by payment gateway).
  - R2. A QR code is assigned to each ticket, allowing quick validation at the entrance of event (in self-service mode).
  - R3. A customer may purchase multiple tickets at once.
  - R4. Optionally, the user may choose places.
  - R5. Reservations can be cancelled in advance.
  - R6. There is also the possibility to buy tickets over the counter.
  - R7. Tickets are anonymous.
  - R8. The system should prevent fraud (forgery or duplication).

Use Case Modeling

Use case diagrams

- Use case model = use case diagram(s) + associated documentation
- Shows:
  - **Actors**: user roles or external systems
  - **Use cases**: system functionalities or services as perceived by users; types of interactions between actors and the system
  - Relationships between them: participation (A/U), generalization (A/A, U/U), include (U/U), extend (U/U).
- Purpose:
  - Specify the system context (actors)
  - Show the system purpose (used by whom for what?)
  - Capture functional requirements (through use cases)
- Applicable not only to software systems
- Prepared by analysts and domain experts

Running example: ticketing system



Use case documentation template

Name	Use case name, showing the actor's intent.
Actor	Name of the actor that initiates the use case.
Description	Brief description highlighting the utility (possibly including results and responsibilities).
Preconditions	Initial conditions (usually about the initial state of the system and actors and input parameters) to be able to execute the use case successfully.
Postconditions	Effects of use case execution, usually in the form of outputs produced and changes in the state of the system and actors (particularly those not visible in the flow of events).
Normal flow	Textual description (numbered list) of the of normal events (actions of the system and actors).
Alternative flows and exceptions	Alternative or exceptional flows of events, described in relation to the normal flow.

For further (optional) fields, see reference on "Use Case Modeling".

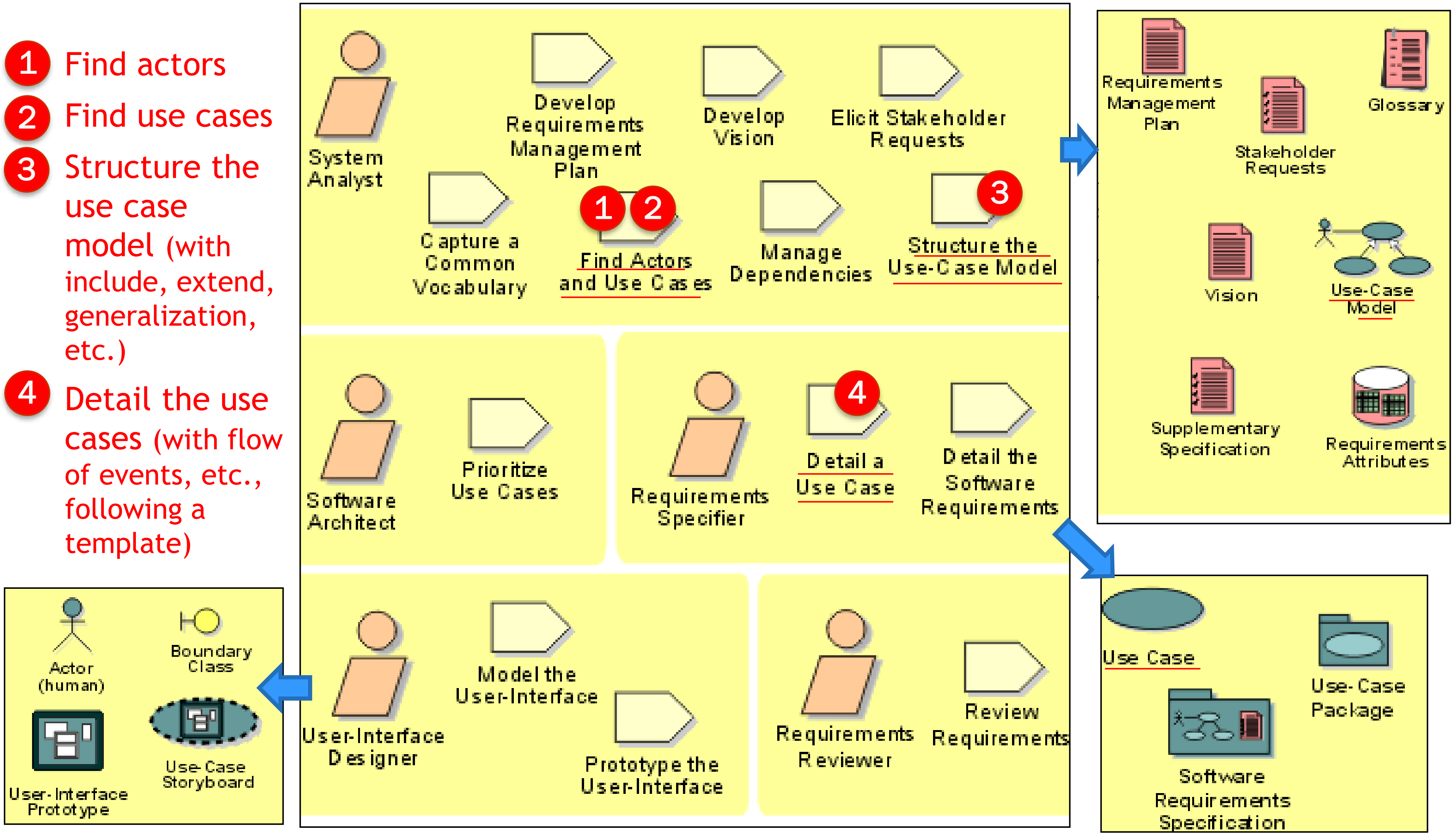
Running example: ticketing system

Name	Purchase tickets online
Actor	Customer
Description	The customer purchases one or more tickets for an event, using an electronic payment method, having the option to choose places. (1)
Preconditions	• The customer has electronic payment means. • The event has tickets available on-sale.
Postconditions	• The customer gets the electronic tickets with a QR code. • The customer is charged of the tickets' cost, and the seller credited. • Information of tickets sold & seats available for the event is updated.
Normal flow	1. The customer accesses the web page of the ticketing system. 2. The system shows the list of events with tickets on-sale. 3. The customer selects the event and the number of tickets. 4. If wanted, the costumer may <u>Choose Places</u>. 5. The system shows the total price to pay. 6. The system redirects the customer to <u>Electronic Payment</u>. 7. The system delivers the electronic tickets to the customer with a unique identifier and QR code.
Alternative flows and exceptions	1. [Payment failure] If, in step 6 of the normal flow the payment fails, the system gives the user the possibility to cancel or retry.

(1) Can also be written as a user story: As a … I want … so that …

Use-case driven requirements engineering in RUP

1. Find actors
2. Find use cases
3. Structure the use case model (with include, extend, generalization, etc.)
4. Detail the use cases (with flow of events, etc., following a template)

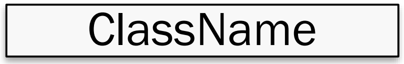
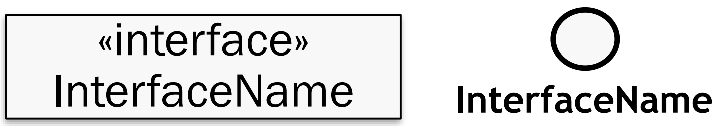
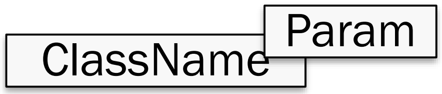
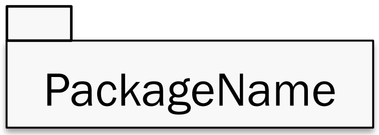
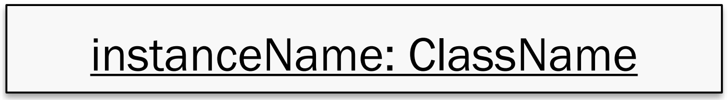


Domain Modeling

Domain models

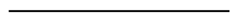
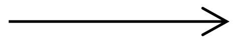
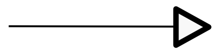
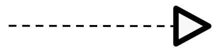
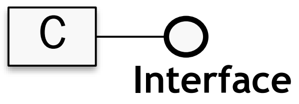
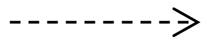
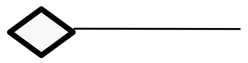
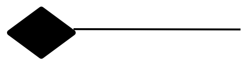
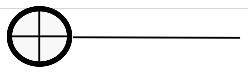
- May be represented by **UML class diagrams** and associated documentation (e.g., class descriptions and constraints)
- Used to organize the **vocabulary** of the problem domain
  - Classes represent concepts
  - Concepts may be described textually in a glossary
- Used to capture **information requirements**
  - Classes represent domain entities (things, events, etc.)
  - Information is conveyed in attributes and relationships
- In a complex system, classes may be partitioned into packages, with a different color per package

Class diagrams - notation summary (1)

Notation	Description
	Class
	Interface
	Parameterized class (generic)
	Package
	Object (instance of a class)

Arrows indicate the relevant notation for domain modeling.

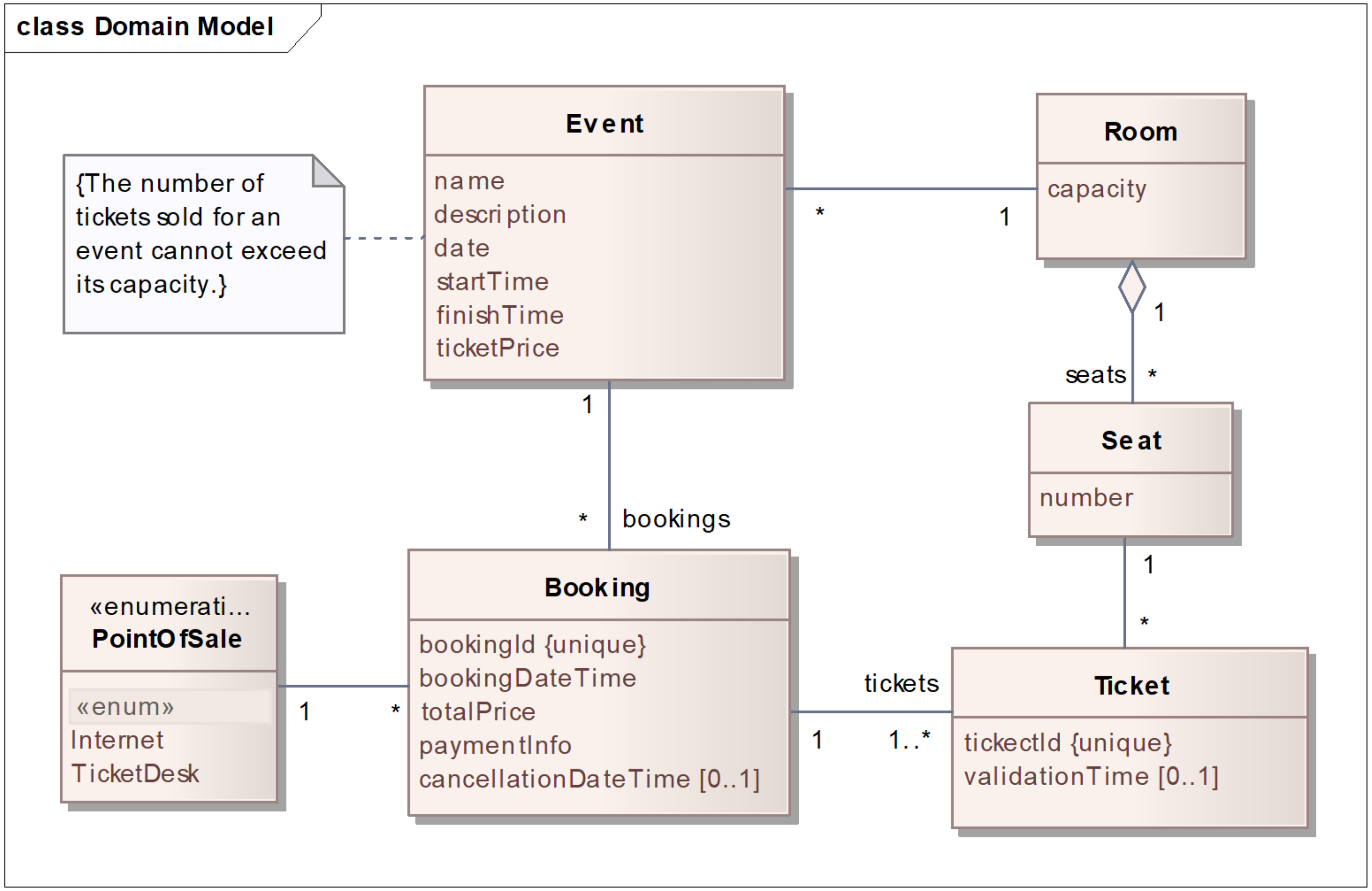
Class diagrams - notation summary (2)

Notation	Description
	Association
	Navigable association (object reference)
	Generalization (extends in Java)
	Realization (implements in Java) Alternative notation: 
	Dependency (particularly, between packages)
	Aggregation ("parts" may exist independently of a "whole")
	Composition ("parts" only exist within a "whole")
	Nesting (inner classes)

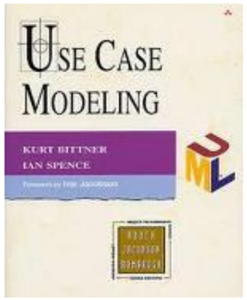
Class diagrams - notation summary (3)

Category	Notation	Description
Modifiers	*italic*	Abstract classes and operations
	underlined	Static attributes and operations
	/name	Derived (calculated) attribute
Visibility	+ - # ~	public, private, protected, package
Multiplicity	*, 0..*, 1, 0..1, 1..*	0 or more, 0 or more, 1, 0 or 1, 1 or more
Constraints	{constraint}	Restricts the valid instances; is usually defined in the context of a class.

Running example: ticketing system



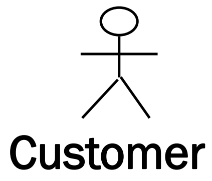
References and further info



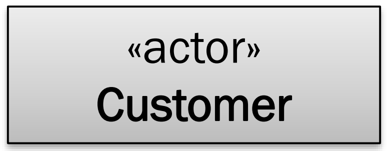
- Use Case Modeling, K. Bittner, I. Spencer, Addison-Wesley, 2003
- UML Tutorial: http://www.sparxsystems.com.au/resources/uml2_tutorial/
- UML Specification: http://www.uml.org/
- Software engineering, 10th edition, Ian Sommerville, Chapter 5 – System Modeling
- http://www.agilemodeling.com

Supplementary materials

Actors



or



- Actor = user role or external system

- Actor = role
  - An actor (in relation to a system) is a role that someone or something of the surrounding environment plays when interacting with the system

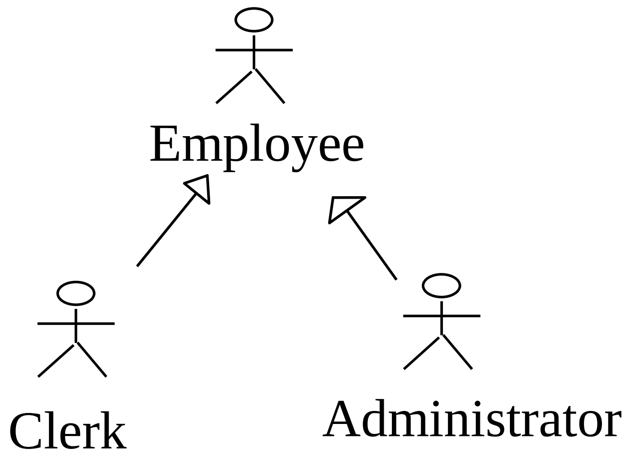
- Actor ≠ individual
  - the same individual can interact with the system in various roles (such as customer, supplier, etc.)

Use cases

Purchase tickets

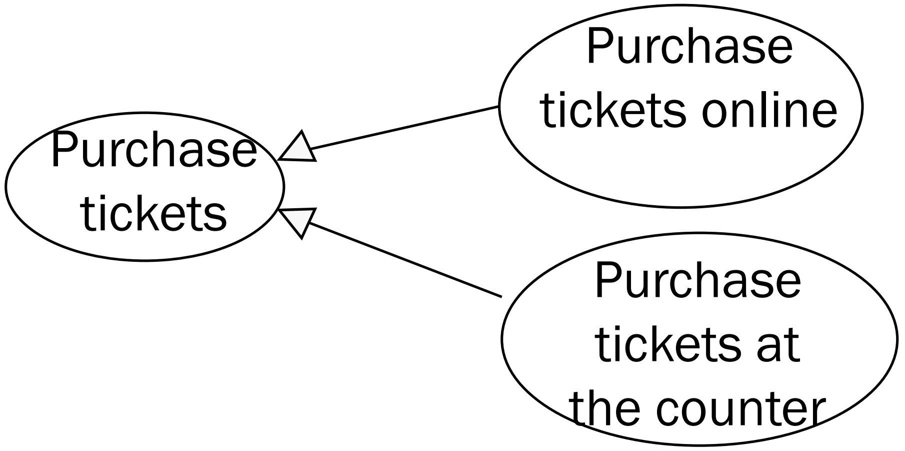
- Definitions:
  - Functionality or service as perceived by users
  - Type of interaction between actors and the system
  - Sequence of actions, including variants, resulting in an observable result with value for an actor
- Name:
  - Must show the purpose
  - Should be given from the perspective of the actor
- Granularity:
  - "Enter card" in ATM -> Not ok, has no value in isolation
  - “Withdraw money" -> Ok, has value for the cardholder
  - It includes preparatory and finalization actions: “Withdraw money: from the introduction of the card to the collection of the card, the receipt and the money"

Relationships: generalization (actors)



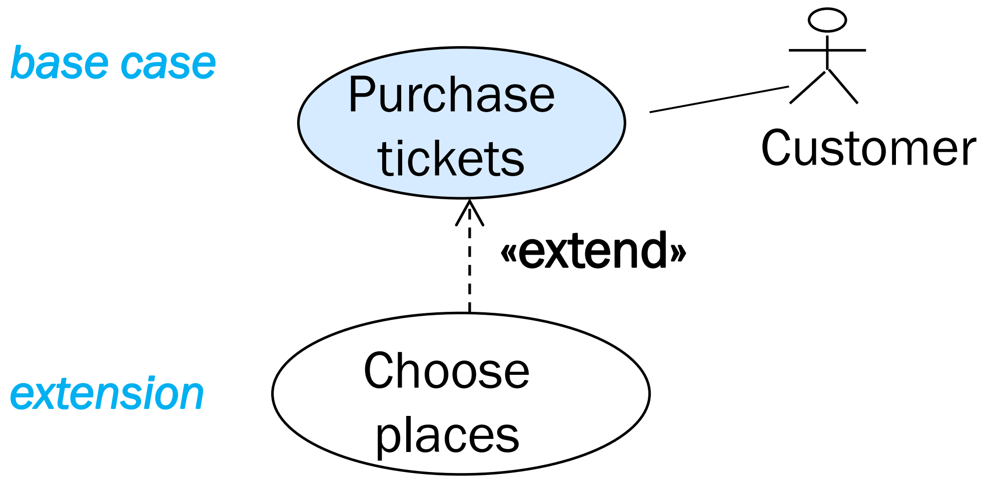
- Generalization relationship: between a more general concept and a more specialized concept
- It should be possible to read "is a (special case of)"
- Specialized actors inherit use cases of more generic actors
- Allows simplifying and structuring diagrams

Relationships: generalization (use cases)



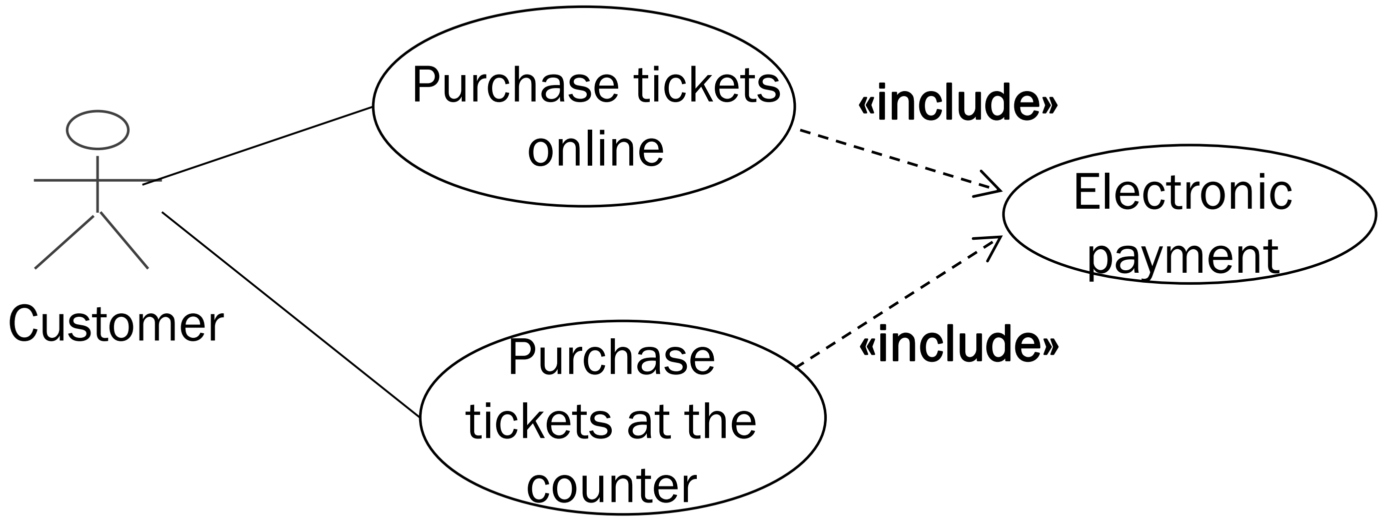
- The specialized use case inherits the behavior, meaning and actors from the generic use case, and may add behavior
- It should be possible to read “is a (special case of)”

Relationships: Extend



- Extensions to base cases indicate conditionally added behaviors
- They allow to highlight optional features and distinguish what is mandatory or essential from what is optional or exceptional
- Actors interact with the base case, which should make sense alone

Relationships: Include



- When several use cases share some common behavior, that common behavior can be separated and described in a new use case which is included by the first ones
- Inclusion is mandatory
- In the textual description, write: include (Electronic payment)

Integrity constraints

- Class diagrams can be semantically enriched with **integrity constraints** (also called **invariants**) attached to each class, to restrict valid object states (attribute values, links, etc.)
- There is an OMG standard language for formalizing such constrains (the OCL - Object Constraint Language), but it is out of the scope of this introductory course.
- Common types of integrity constraints:
  - Domain constraints (on the set of values of an attribute)
  - Uniqueness constraints (keys)
  - Constraints related to loops in associations
  - Time-related constraints
  - Constraints that define derived elements
  - Existence constraints (on the existence of objects or values)
  - Generic business rules and constraints

Running example: ticketing system

- Domain constraints (on the set of values of an attribute)
  - The capacity must be a positive integer
- Uniqueness constraints (keys):
  - There cannot exist two tickets with the same ticketId
- Constraints related to loops in associations:
  - The two navigation paths from Ticket to Event must the lead to the same event
- Time-related constraints:
  - For any event, startTime < finishTime
- Constraints that define derived attributes:
  - The totalPrice of a Booking must equal the number of tickets multiplied by their unit price
- Generic business rules and constraints:
  - The number of tickets sold for an event cannot exceed the event capacity

Model-Driven Engineering (MDE)

- In this course, UML models are use as (semi-formal) documentation
- In MDE, models are used as “first class citizens”, from which implementations can be generated (as opposed to using models only for documentation purposes), e.g.,
  - Database schema (or DDL scripts) generation from models
  - Code generation from models
  - Test case generation from models
  - Etc.
- MDE is outside the scope of this introductory course.