

Modeling with UML

Presented by:

Basma Hussien

Contents

1

Introduction

2

What is UML?

3

UML Diagrams

4

Class Diagram

5

Use Case Diagram

What is Modeling?

- **Modeling consists of building an abstraction of reality.**
- **Abstractions are simplifications because:**
 - ✓ They ignore irrelevant details
 - ✓ They only represent the relevant details.
- **What is *relevant* or *irrelevant* depends on the purpose of the model.**

Example: map



Example: street map (2)



Why model software?

- **Software is getting increasingly more complex**
 - ✓ Windows XP > 40 millions lines of code.
 - ✓ A single programmer cannot manage this amount of code in its entirety.
- **Code is not easily understandable by developers who did not write it.**
- **We need simpler representations for complex systems**
 - ✓ Modeling is a mean for dealing with complexity.
- **Modeling is the only way to visualize the design and check it against requirements before one starts to code.**

Systems, Models and Views

- A *system* is an organized set of communicating parts
- A *model* is an abstraction describing a subset of a system
- A *view* depicts selected aspects of a model

Examples:

- **System:** Aircraft
- **Models:** Flight simulator, scale model
- **Views:** All blueprints, electrical wiring, fuel system

12/21/2022



Concepts and Phenomena

- **Phenomenon**

- ✓ An object in the world of a domain as you perceive it
- ✓ *Example*: The lecture you are attending
- ✓ *Example*: You!!

- **Concept**

- ✓ Describes the properties of phenomena that are common.
- ✓ *Example*: Lectures on software engineering
- ✓ *Example*: Trainee

- **Concept is a 3-tuple:**

- ✓ *Name* (To distinguish it from other concepts)
- ✓ *Purpose* (Properties that determine if a phenomenon is a member of a concept)
- ✓ *Members* (The set of phenomena which are part of the concept)

Concepts and Phenomena in OO analysis

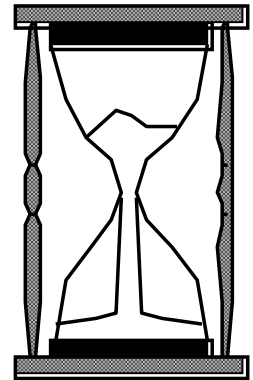
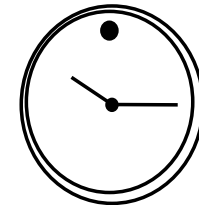
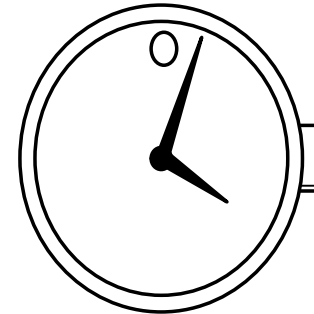
Name

Purpose

Members

Clock

**A device that
measures
time.**



- **Abstraction**
 - ✓ Classification of phenomena into concepts
- **Modeling**
 - ✓ Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Example

- **Q. If you want to make systems that deal with real world problems, how do you get your hands around real world complexities?**
- Ans. The key is to organize the design process in a way that clients, analysts, programmers and others involved in system development can understand and agree on.
- UML is a key in providing this organization.

What is UML?

- **UML (Unified Modeling Language)**
 - ✓ Modeling(visual) language not a *method* for modeling and communicating about systems through the use of diagrams and supporting text.
 - ✓ An emerging standard for modeling object-oriented software.
- **Supported by several CASE tools**
 - ✓ ArgoUML (Open Source)
 - ✓ Rational ROSE (IBM)
 - ✓ TogetherJ (Borland)
 - ✓ Visio (Microsoft)

What is UML? (Cont')

- **UML is by far the most exciting thing to happen to the software industry in recent years as every other engineering discipline has a standard method of documentation**
 - ✓ Electronic engineers have schematic diagrams
 - ✓ Architects and mechanical engineers have blueprints and mechanical diagrams
 - ✓ The software industry now has UML!



Is UML process dependent?

The **UML** is a **process-independent** notation system - that is, there is **no dependency** on a particular development process, like Agile Programming or the Unified Process, **both of which may be used effectively with the UML.**

Historical Background

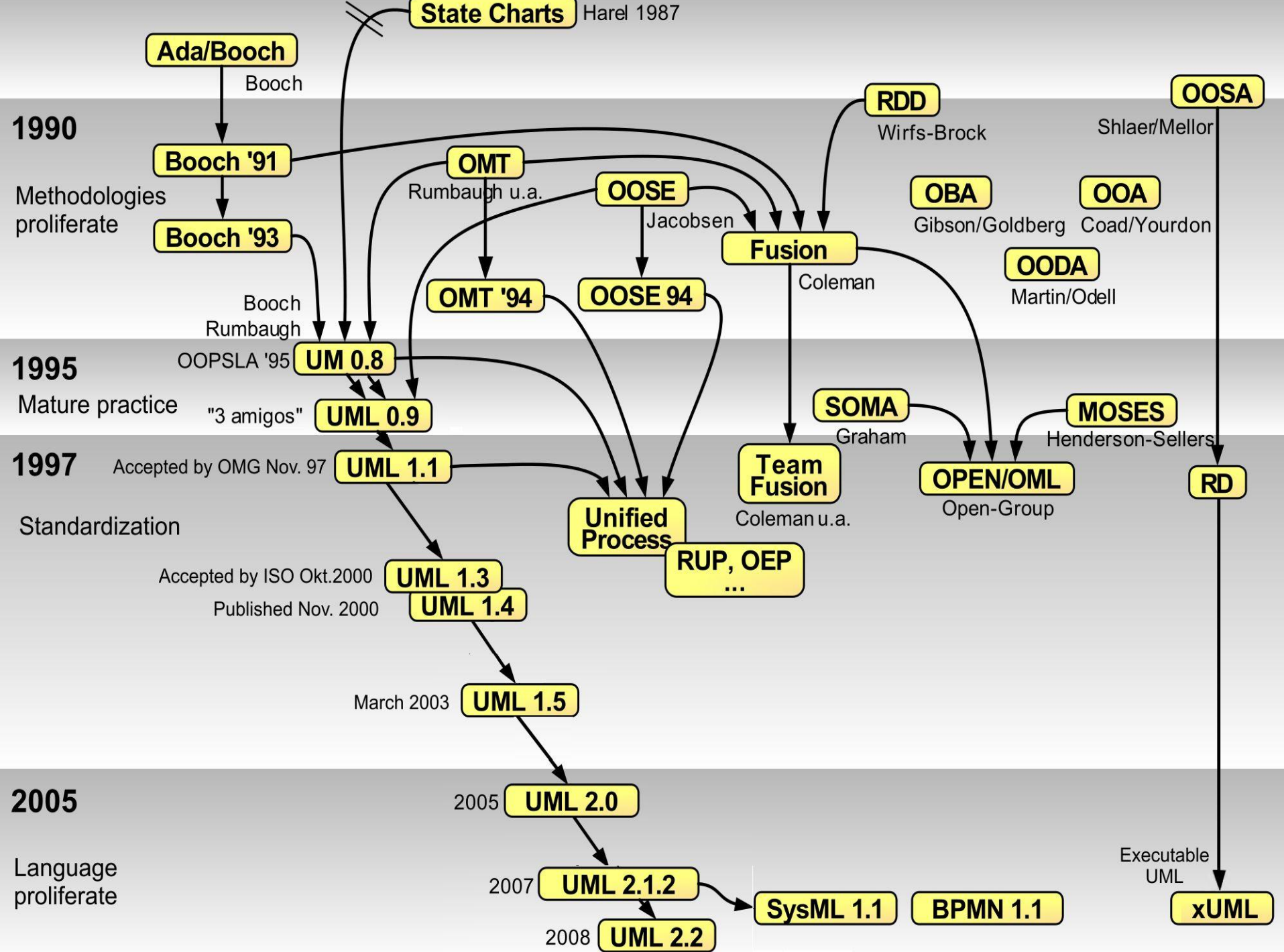
- **1970 – Object-oriented modeling languages began to appear.**
- **1996 – Release of UML 0.9 by by Grady Booch, Jim Rumbaugh of Rational Software Corporation, Ivar Jacobson of Objectory company.**
- **1996 – Release of UML 1.0 by Digital Equipment, HP, ILogix, IntelliCorp, IBM, ICON, MCI, Microsoft, Oracle, Rational, TI and Unisys.**

Historical Background (Cont')

- **1997 – Release of UML 1.1 by IBM, ObjecTime, Platinum, Ptech, Taskon, Reich and Softeam**
- **After the first release a task force was formed to improve the language, which released several minor revisions, 1.3, 1.4, and 1.5**
- **2001 – Work on UML 2.0 specifications.**
- **2007 - Although UML 2.1 was never released as a formal specification, versions 2.1.1 and 2.1.2 appeared in 2007, followed by UML 2.2 in February 2009.**

Historical Background (Cont')

- **2010 – UML 2.3 was formally released in May 2010.**
- **2011 – UML 2.4.1 was formally released in August 2011.**
- **2012 – UML 2.5 was released in October 2012 as an "In progress" version and was officially released in June 2015.**
- **2017 – Formal version 2.5.1 was adopted in December 2017.**



Benefits of UML

- **Software systems are professionally designed and documented before they are coded so that all stakeholders know exactly what they are getting, in advance.**
- **UML design enables you to check your system against user requirements even before start coding, this reduces software development costs by eliminating re-work in all areas of the life cycle.**

Benefits of UML (Cont')

- **Since system design comes first, UML enables reusable code to be easily identified and coded with the highest efficiency, thus reducing software development costs.**
- **UML enables logic 'holes' to be spotted in design drawings so that software will behave as expected.**
- **UML diagrams assist in providing efficient training to new members of the development team member.**

Benefits of UML (Cont')

- **UML enables ease of maintenance by providing more effective visual representations of the system. Consequently, maintenance costs are reduced.**
- **UML improve clarity of communication with both internal and external stakeholders as it documents the system much more efficiently.**

UML Diagrams

The slide features a blue wavy banner at the top with the title 'UML Diagrams' in white bold text. Three blue spheres of varying sizes are positioned around the banner. The background is light blue with faint, stylized UML diagrams and binary code (0s and 1s) visible.

Software characteristics

- **A software system has three characteristics:**
 - ✓ **Static:** The static characteristic of a system is essentially the structural aspect of the system. The static characteristics define *what parts the system is made up of*.
 - ✓ **Dynamic:** The behavioral features of a system; for example, *the ways a system behaves in response to certain events or actions* are the dynamic characteristics of a system.
 - ✓ **Implementation:** The implementation characteristic of a system is an entirely new feature that describes *the different elements required for deploying a system*.

UML Diagram Classification

Static

- Class diagram
- Package diagram
- Object diagram
- Component diagram
- Composite Structure diagram
- Profile diagram

Dynamic

- Use Case diagram
- Activity diagram
- Sequence diagram
- State diagram
- Communication diagram
- Collaboration diagram
- Interaction Overview diagram

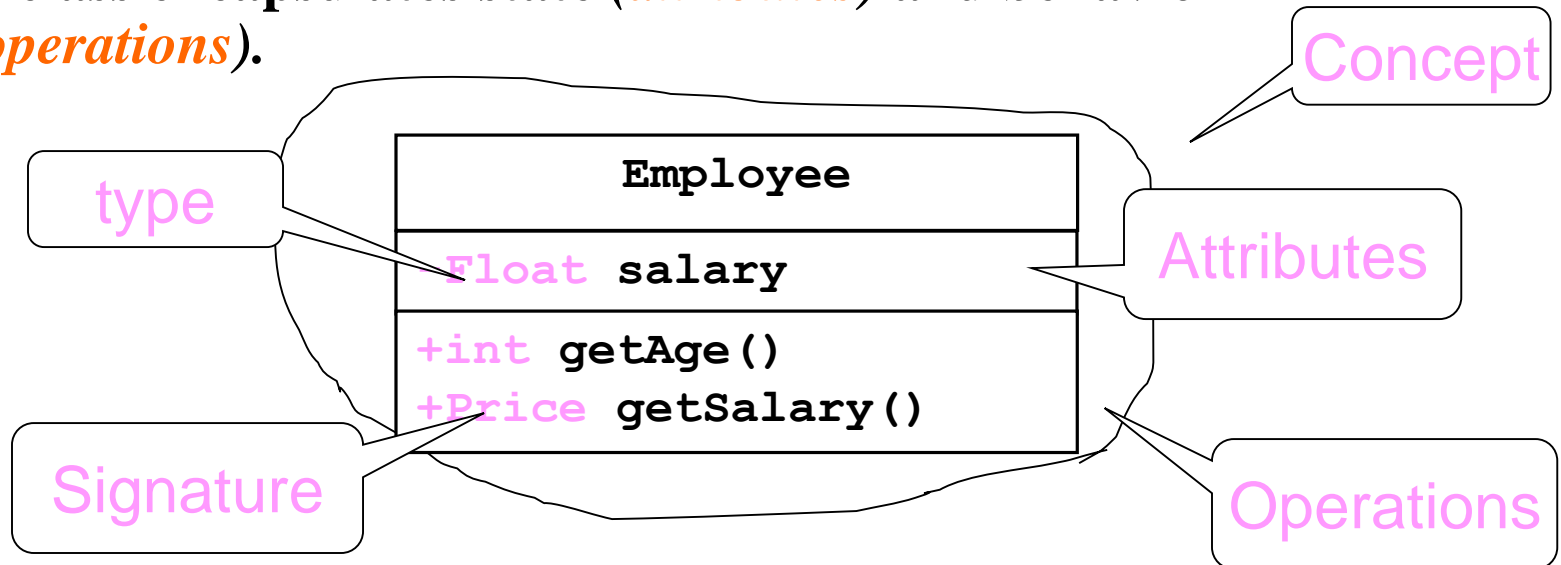
Implementation

- Deployment diagram
- Timing diagram

Class Diagrams

Classes

- A *class* represent a concept?!!
- A class encapsulates state (*attributes*) and behavior (*operations*).



- Each attribute has a *type*.
- Each attribute has a *Access Specifier*.(+,-,#)
- Each operation has a *signature*.
- Each operation has a *Access Specifier*.(+,-,#)
- The class name is the only mandatory information.

Class representation

- Class is represented with a rectangular box divided into *compartments*.
- The first compartment holds the **name of the class**, the second holds **attributes**, and the third is used for **operations**.
- You can hide any compartment of the class if that increases the readability of your diagram.
- You may add compartments to a class to show additional information, such as exceptions or events.
- UML suggests that the class name:
 - ✓ Start with a capital letter
 - ✓ Be centered in the top compartment
 - ✓ Be written in a boldface font
 - ✓ Be written in italics if the class is *abstract*.

Class Diagram (cont.)

Attributes:

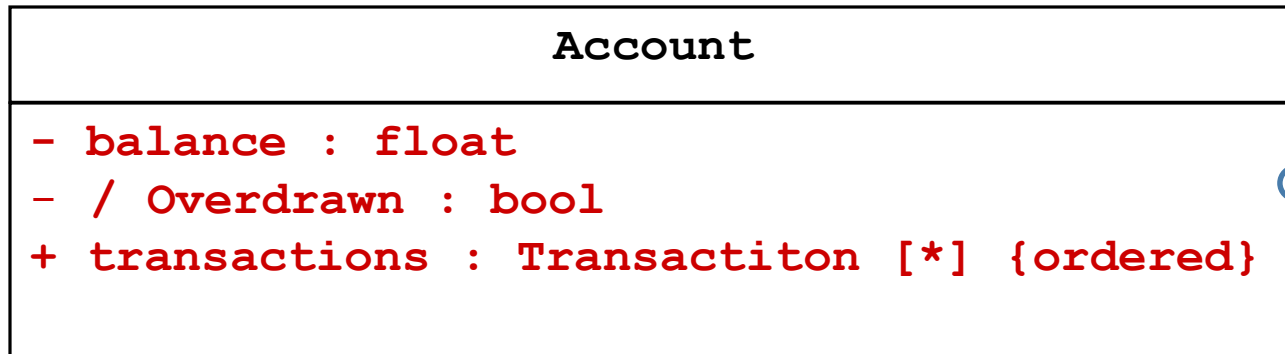
- Can be simple primitive types (integers, floating-point numbers, etc.) or relationships to other, complex objects.
- Can be shown using two different notations: **inlined** or **relationships** between classes
- Notation is available to show such things as **multiplicity**, **uniqueness**, and **ordering**.
- There is no semantic difference between inlined attributes and attributes by relationship; it's simply a matter of how much detail you want to present.
- Static attributes are represented by underlining their specification.

Class Diagram (cont.)

Inlined Attributes:

- Lists a class's attributes right in rectangle notation
- It uses the following notation:

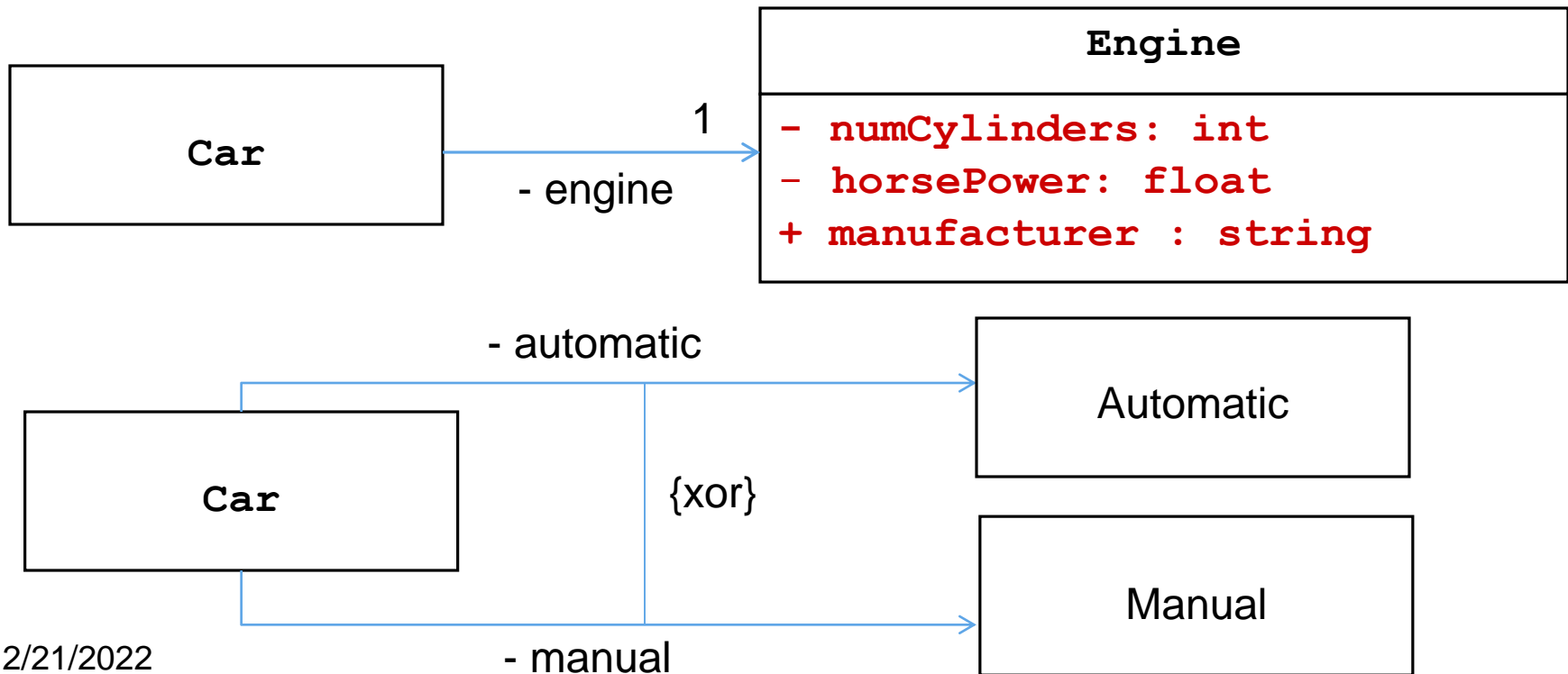
visibility / name : type multiplicity = default {property strings and constraints}
{ + | - | # } [lower..upper]



Class Diagram (cont.)

Attributes by Relationship:

- Results in a larger class diagram, but it can provide greater detail for complex attribute types
- Conveys exactly how the attribute is contained within a class

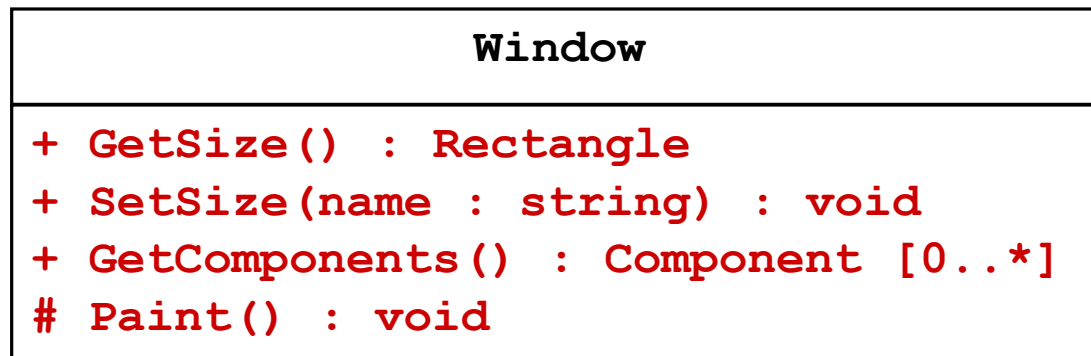


Class Diagram (cont.)

Operations:

- Features of classes that specify how to invoke a particular behavior.
- Placed in a separate compartment with the following syntax:

visibility name (parameters) : return-type { constraints }



- **Operation Constraints:**

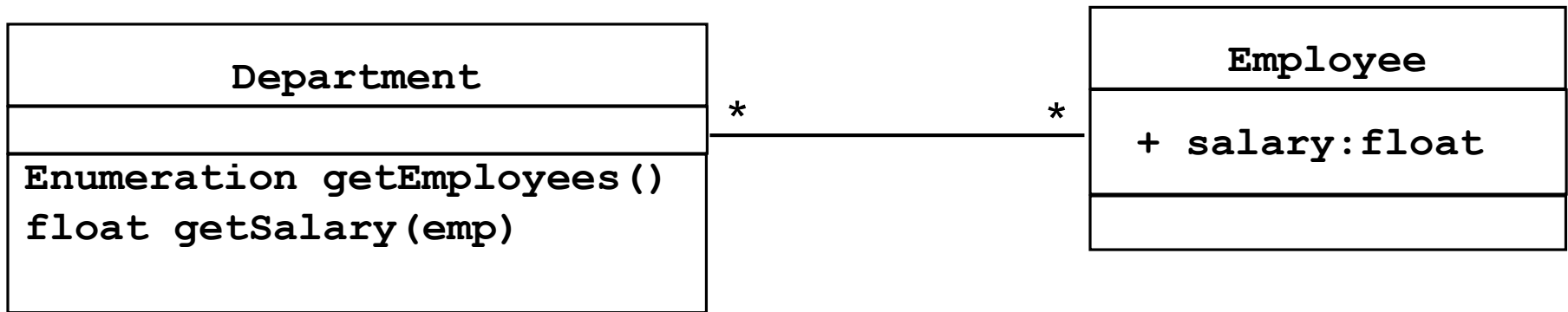
{Preconditions – Postconditions - Body conditions - Query operations - Exceptions}

Objects

<u>Mohamed:Employee</u>
salary = 500.20

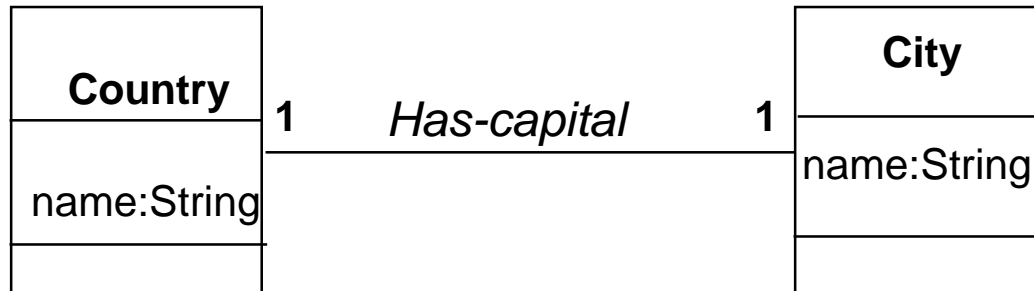
- An **Object** represents a phenomenon....??!!!
- The name of an instance is **underlined** and can contain the class of the instance.
- The attributes are represented with their ***values***.

Associations

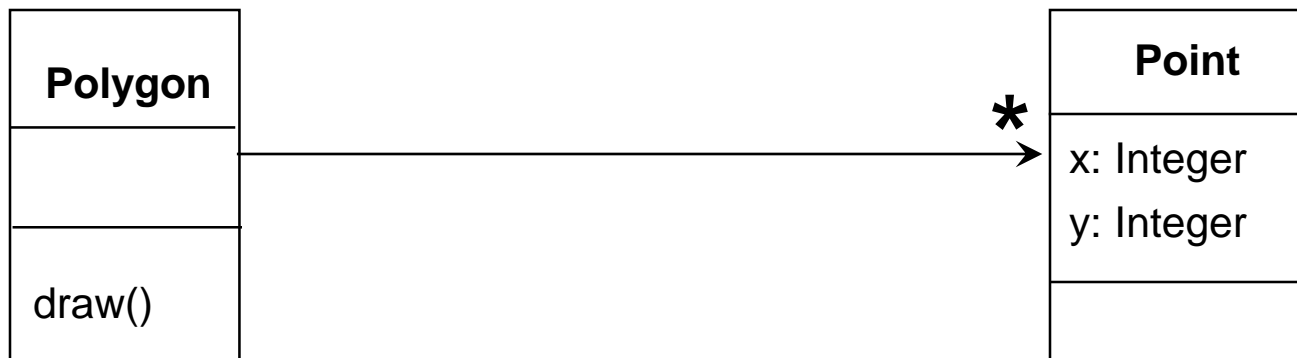


- **Associations denote relationships between classes.**
- **The multiplicity of an association end denotes how many objects the source object can legitimately reference.**

1-to-1 and 1-to-many Associations



One-to-one association



One-to-many association

From Problem Statement To Object Model

*Problem Statement: A stock exchange lists many companies.
Each company is uniquely identified by a ticker symbol*

Class Diagram:



From Problem Statement to Code

*Problem Statement: A stock exchange lists many companies.
Each company is uniquely identified by a ticker symbol*



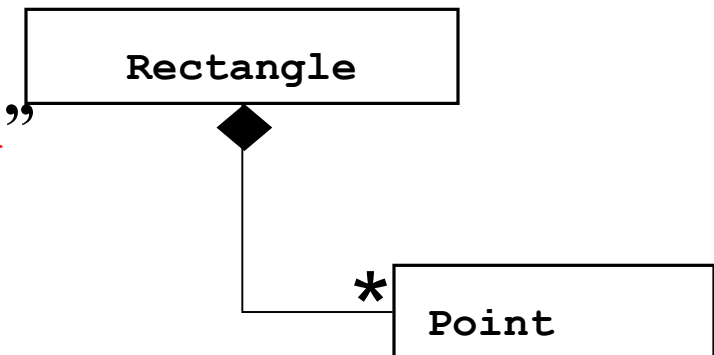
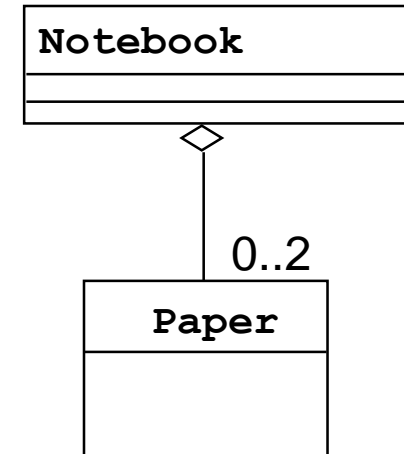
C# Code

```
public class StockExchange
{
    private List<Company> m_Company = new List<Company>();
}

public class Company
{
    public int m_tickerSymbol;
    private List<StockExchange> m_StockExchange = new List<StockExchange>();
};
```

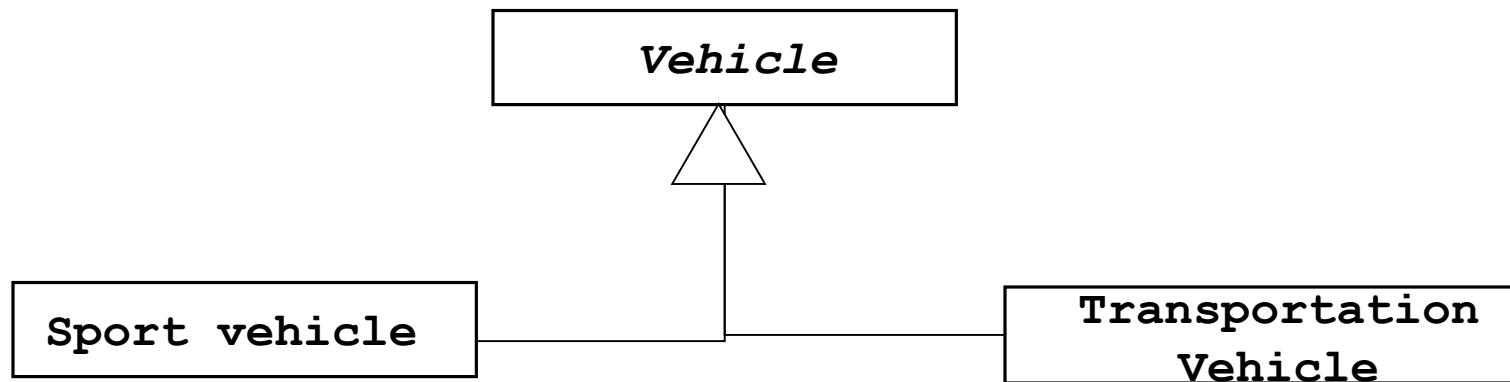

Aggregation

- An **aggregation** is a special case of association denoting a “**consists of**” hierarchy.
- The **aggregate** is the parent class, the *components* are the children class.
- A solid diamond denotes **composition**, a strong form of aggregation denoting a “**Contain**” hierarchy
- **Note: Rectangle contain Object from Point**



Inheritance relationship

- Inheritance simplifies the model by eliminating redundancy.
- The children classes inherit the attributes and operations of the parent class.
- The inheritance denote **Kind-Of** relationship



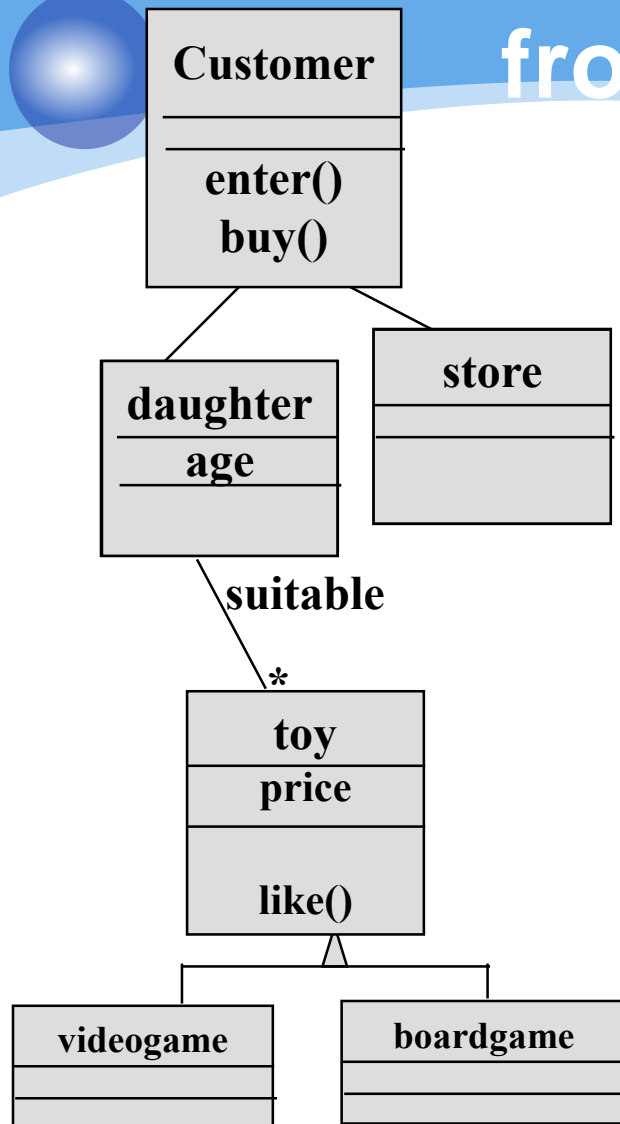
How do you find classes?

- The customer enters the store to buy a toy. It has to be a toy that his daughter likes and it must cost less than 50 Euro. He tries a videogame, which uses a data glove and a head-mounted display. He likes it. An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely a boardgame. The customer buy the game and leaves the store

Example: Flow of events

- **The customer enters the store to buy a toy.**
- **It has to be a toy that his daughter likes and it must cost less than 50 Euro.**
- **He tries a videogame, which uses a data glove and a head-mounted display. He likes it.**
- **An assistant helps him.**
- **The suitability of the game depends on the age of the child. His daughter is only 3 years old.**
- **The assistant recommends another type of toy, namely the board game "Monopoly".**

Generation of a class diagram from flow of events



The **customer enters** the **store** to **buy** a **toy**. It has to be a toy that his **daughter likes** and it must cost **less than 50 Euro**. He tries a **videogame**, which uses a data glove and a head-mounted display. He likes it. An assistant helps him. The suitability of the game **depends** on the **age** of the child. His daughter is only 3 years old. The assistant recommends another **type of toy**, namely a **boardgame**. The customer **buy** the game and **leaves the store**

Mapping parts of speech to object model components

<i>Part of speech</i>	<i>Model component</i>	<i>Example</i>
Proper noun	object	Jim Smith
Improper noun	class	Toy, doll
Doing verb	method	Buy, recommend
being verb	inheritance	is-a (kind-of)
having verb	Aggregation, Association	has an
modal verb	constraint	must be
adjective	attribute	3 years old
transitive verb	method	enter
intransitive verb	method (event)	depends on

Who uses class diagrams?

- a. Main Purpose of Class diagrams is to describe the static properties of a system.
- b. The customer and end users are often not interested in class diagrams. They usually focus more on system functionality.
- c. The developer uses class diagrams during the development of a system, that is, during analysis, system design, object design and implementation.

ERD & Class Diagram

❖ Is Entity Relationship Diagram a UML diagram?

- Both are similar...
-
- **But**, A **Class diagram** is a **UML diagram**.
An **ERD**, on the other hand, is **NOT** a UML diagram, it is a diagram for data modeling (attributes and relationships).




Use Case Diagrams





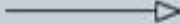
Use Case Diagrams

- **A way to capture system functionality and requirements.**
- **It consists of:**
 - ✓ Use cases
 - ✓ Actors
 - ✓ Subjects

Use Case Modeling: Core Elements

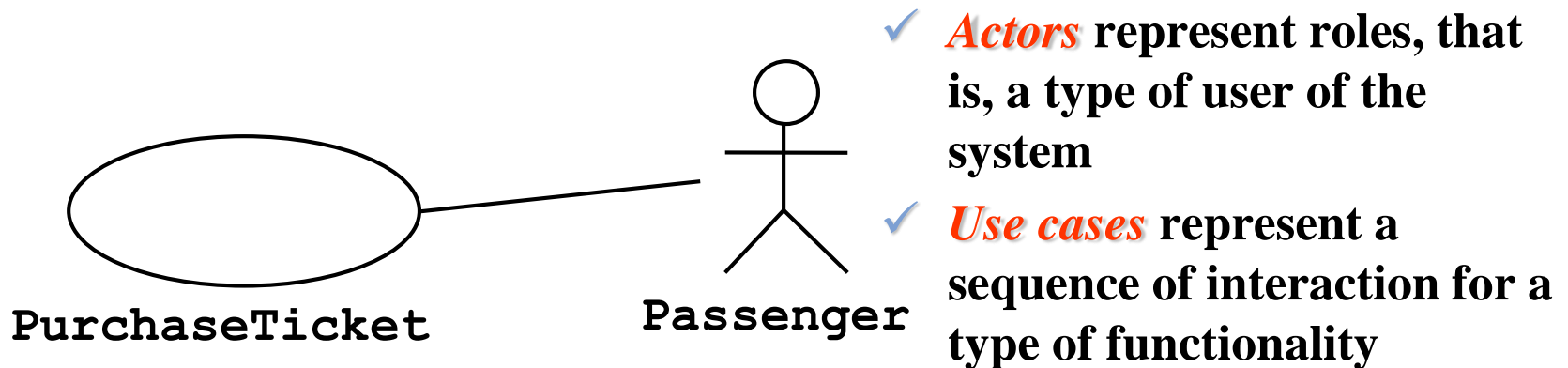
Construct	Description	Syntax
use case	A sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.	
actor	A coherent set of roles that users of use cases play when interacting with these use cases.	
system boundary	Represents the boundary between the physical system and the actors who interact with the physical system.	

Use Case Modeling: Core Elements Cont.'

Construct	Description	Syntax
association	The participation of an actor in a use case. i.e., instance of an actor and instances of a use case communicate with each other.	
extend	A relationship from an <i>extension</i> use case to a <i>base</i> use case, specifying how the behavior for the extension use case can be inserted into the behavior defined for the base use case.	
generalization	A taxonomic relationship between a more general use case and a more specific use case.	

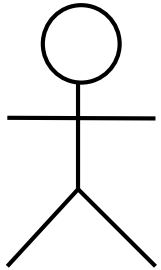
Use Case Diagram

The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment



Use Case diagram used during requirements elicitation to represent external behavior

Actors

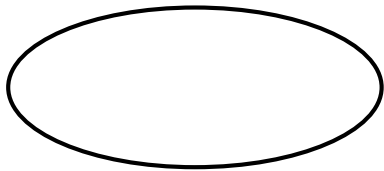


Passenger

- **An actor models an external entity which communicates with the system:**
 - ✓ User
 - ✓ External system
- **An actor has a unique name and an optional description.**
- **Examples:**
 - ✓ **Passenger:** A person in the train
 - ✓ **GPS satellite:** Provides the system with GPS coordinates

Use Case

- **A use case represents a class of functionality provided by the system as an event flow.**



PurchaseTicket

A use case consists of:

- ✓ Unique name
- ✓ Participating actors
- ✓ Entry conditions
- ✓ Flow of events
- ✓ Exit conditions
- ✓ Special requirements

Use Case Definition: Example

Name: Purchase ticket

Participating actor: Passenger

Entry condition:

- **Passenger** standing in front of ticket distributor.
- **Passenger** has sufficient money to purchase ticket.

Exit condition:

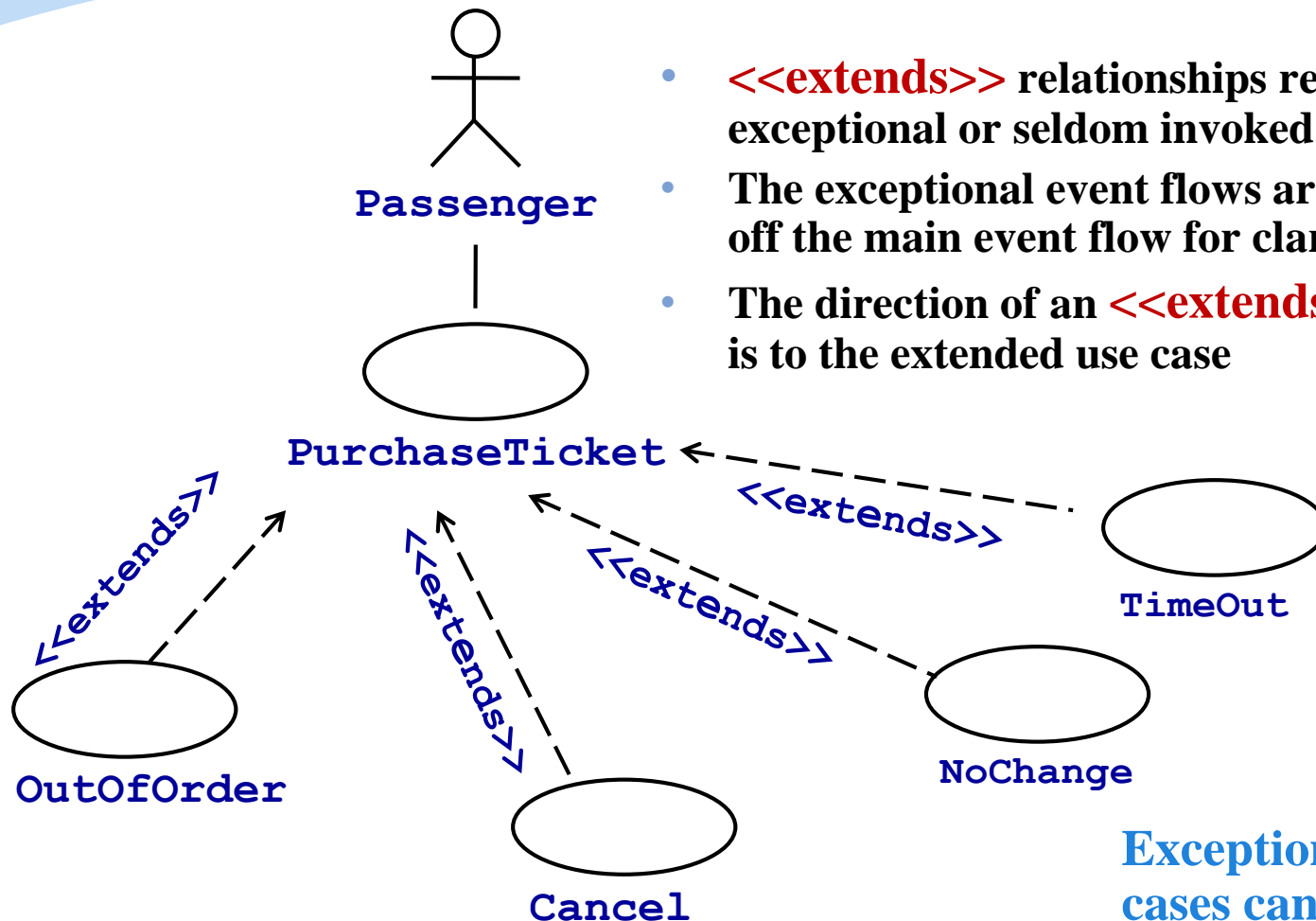
- **Passenger** has ticket.

Event flow:

1. **Passenger** selects the number of zones to be traveled.
2. **Distributor** displays the amount due.
3. **Passenger** inserts money, of at least the amount due.
4. **Distributor** returns change.
5. **Distributor** issues ticket.

**What about
Exceptional cases?**

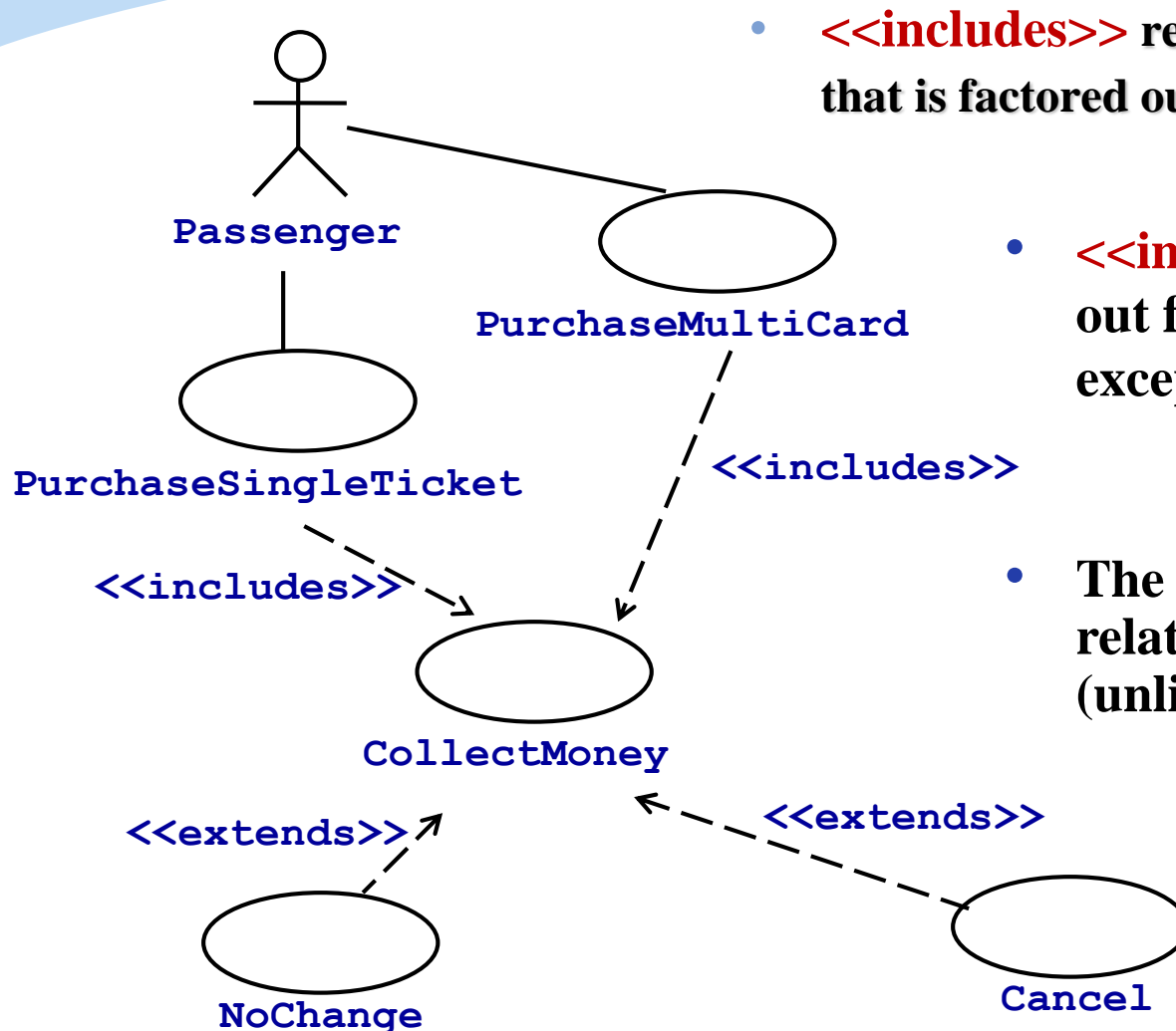
The <<extends>> Relationship



- <<extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out off the main event flow for clarity.
- The direction of an <<extends>> relationship is to the extended use case

Exceptional flows use cases can extend more than one use case.

The <<includes>> Relationship



- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of an <<includes>> relationship is to the using use case (unlike <<extends>> relationships).



Thanks