COIS 2240
Software Design & Modelling

Lecture 5

UML Modelling I - Modelling with Classes

Taher Ghaleb

TRENT UNIVERSITY

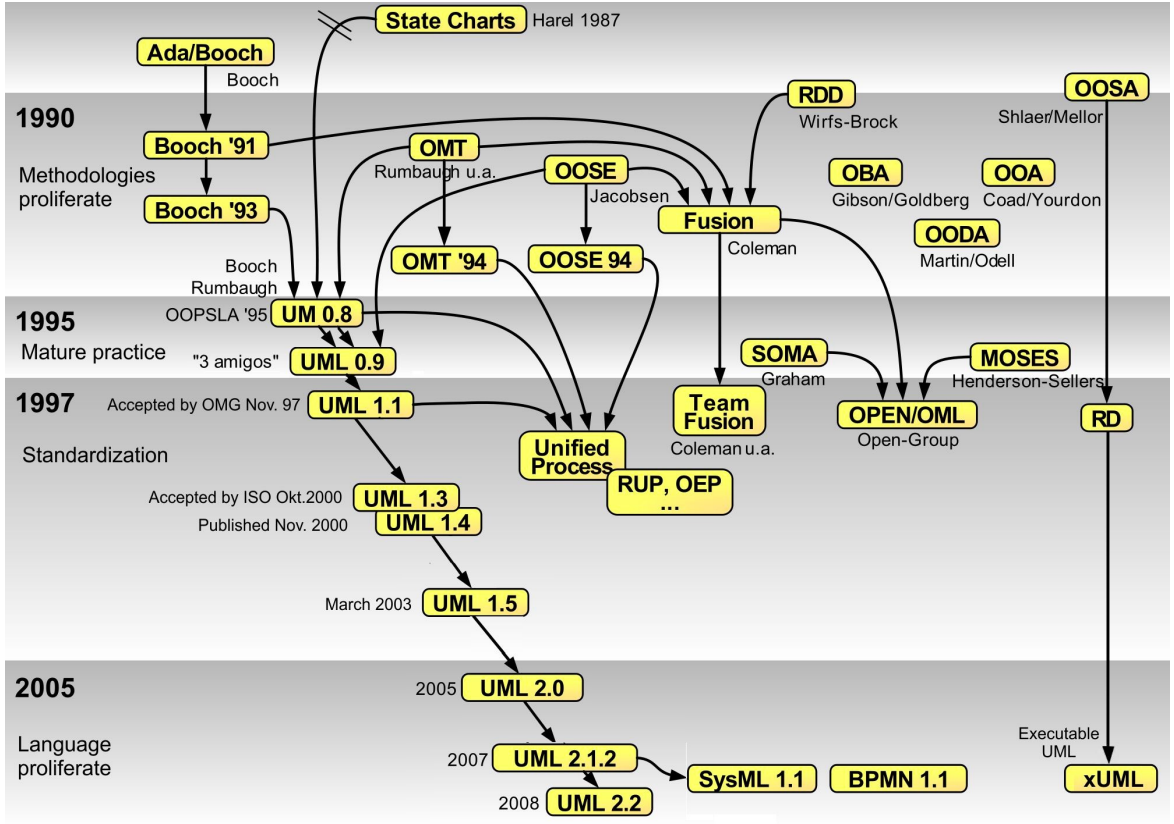
Parts of the content on the slides were sourced from / inspired by the textbook slides

What is UML?

The **U**nified **M**odelling **L**anguage is a standard graphical language for modelling software

- Developed by *Rumbaugh*, *Booch* and *Jacobson*
- Based on earlier languages they had each developed
- They worked together at the Rational Software Corporation, later bought by IBM
  - Much development of UML was done at IBM Rational Ottawa

Modeling Language Genealogy



UML 2.5 (2017)

UML Diagrams

- Class Diagrams (data and relationships)
- Sequence Diagrams (interactions)
- Communication Diagrams (interactions)
- State Diagrams (behavior driven by events)
- Activity Diagrams (behavior and parallel algorithms)
- Use Case Diagrams (user interactions)
- Component Diagrams (structure and connection among system parts)
- Deployment Diagrams (allocation of software to hardware)

A *model* goes beyond a mere set of diagrams

Software Model

- A model captures an interrelated set of information about the system
- A diagram is simply one view of that information
- Several diagrams can present the same information in slightly different ways, either with different notations or with different levels of detail
- An element can be deleted from a diagram, but still kept it in the model; deleting an element from the model should make it disappear from all diagram
- A model gives software engineers insights about the system; they can analyze the model (manually or using tools) to discover problems or other properties
- Being a standard notation enables anyone to interpret it the same way

UML Class Diagrams

The main symbols shown on class diagrams are:

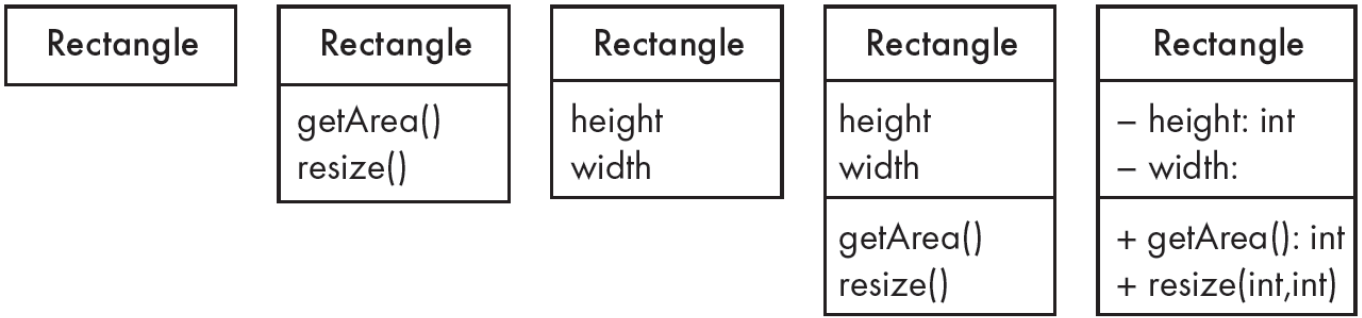
- *Classes*
  - types of data themselves
- *Associations*
  - links between instances of classes
- *Attributes*
  - simple data found in classes and their instances
- *Operations*
  - abstract and concrete methods performed by the classes and their instances
- *Generalizations*
  - group classes into inheritance hierarchies

Classes

A class is simply represented as a box with the name of the class inside

- The diagram may also show the attributes and operations
- The complete signature of an operation is:

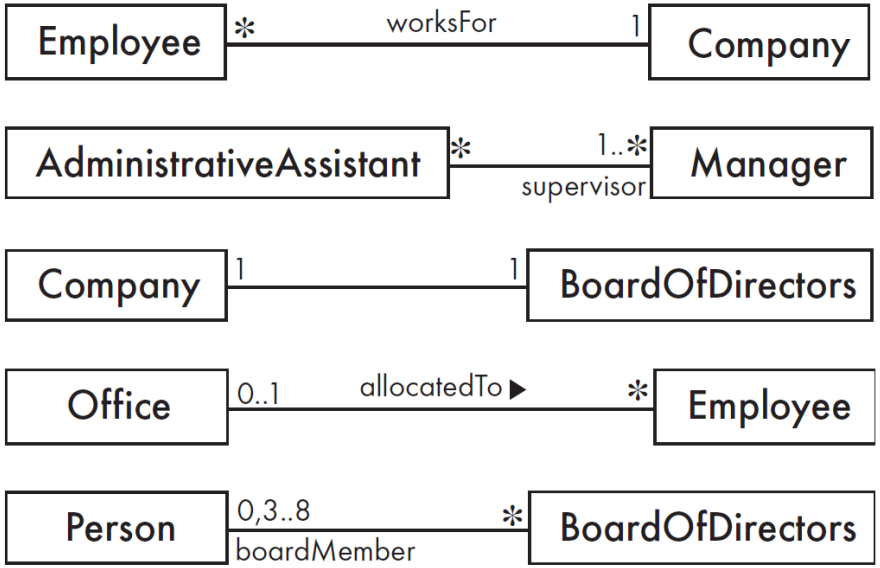
```
operationName(parameterName: parameterType …): returnType
```



Associations and Multiplicity

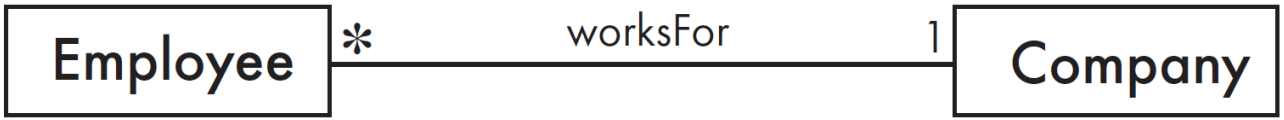
An association is used to show how two classes are related to each other

- Symbols indicating multiplicity are shown at each end of the association
- Each association can be labelled, to make explicit the nature of the association



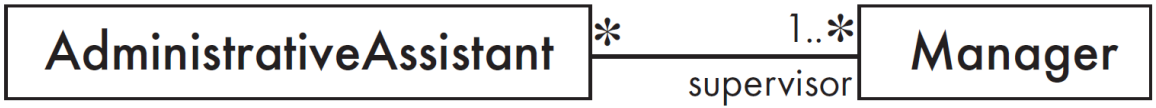
Many-to-one associations

- A company has many employees
- An employee can only work for one company
- A company can have zero employees
- It is not possible to be an employee unless you work for a company



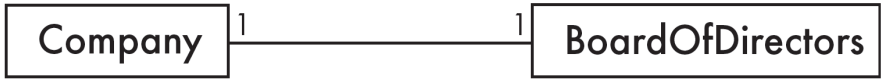
Many-to-many associations

- An assistant can work for many managers
- A manager can have many or a group of assistants
- Assistants can work in pools
- Managers can have a group of assistants
- Some managers might have zero assistants.
- Is it possible for an assistant to have, perhaps temporarily, zero managers?

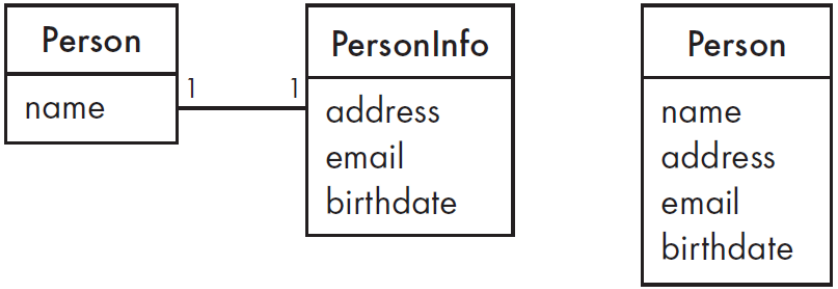


One-to-one associations

- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company



- Avoid unnecessary one-to-one associations



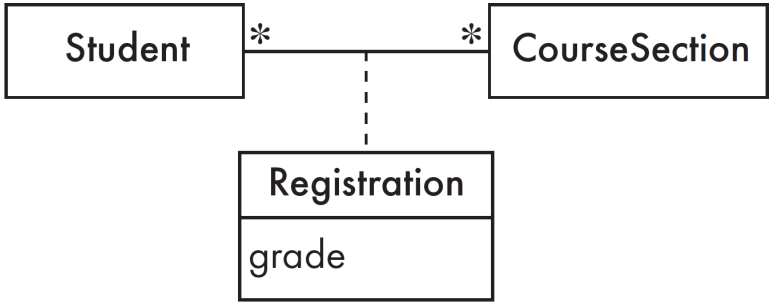
A more complex example

- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could never involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking

- The frame around this diagram is an optional feature of UML 2.0

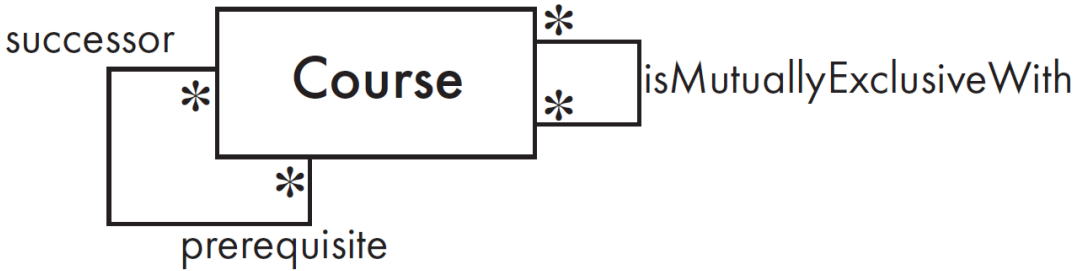
Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent:



Reflexive associations

It is possible for an association to connect a class to itself

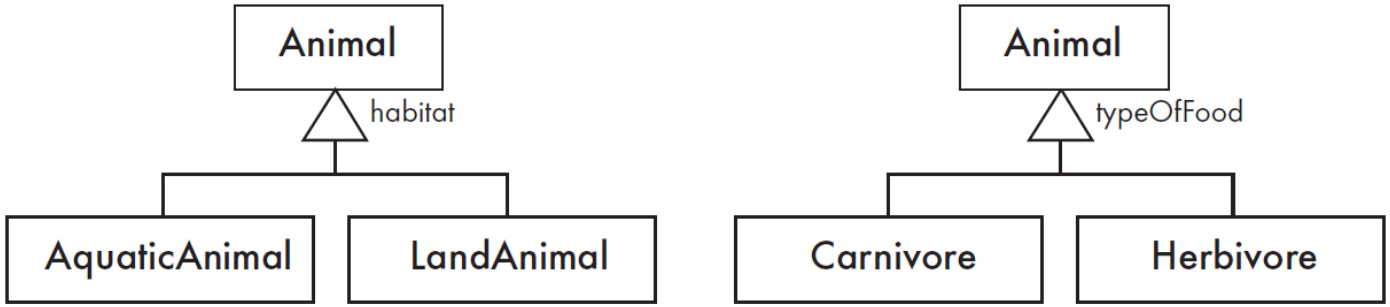


Directionality in associations

- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end

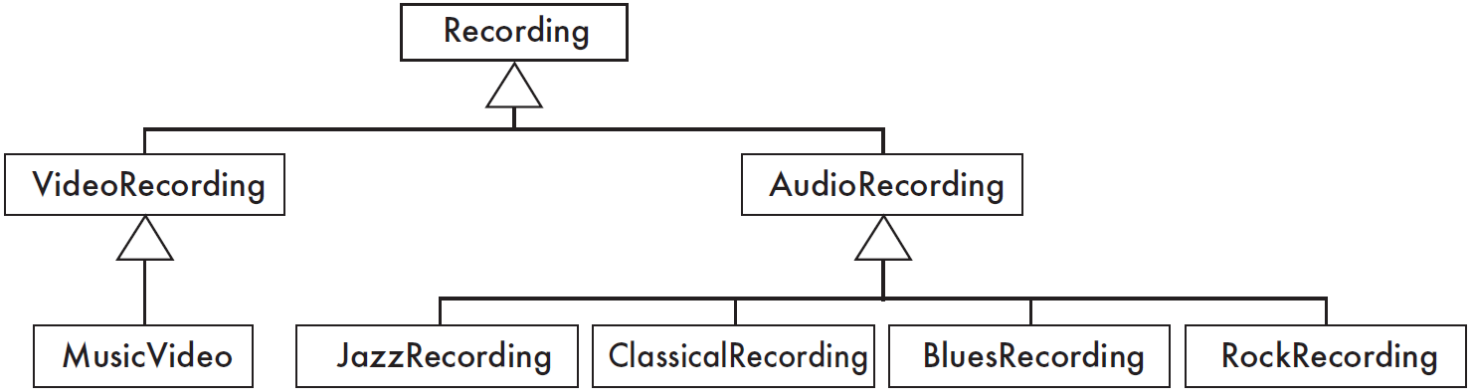
Generalization

- Specializing a superclass into two or more subclasses
  - A generalization set is a labelled group of generalizations with a common superclass
  - The label (sometimes called the discriminator) describes the criteria used in the specialization

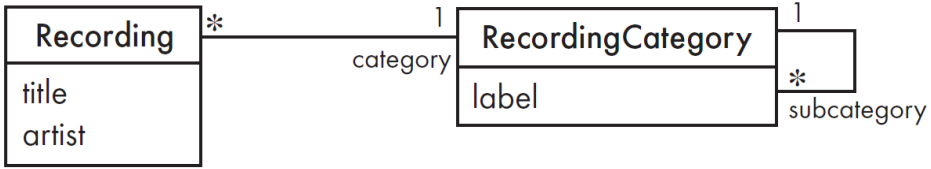


Avoid unnecessary generalizations

- Inappropriate hierarchy of classes, which should be instances

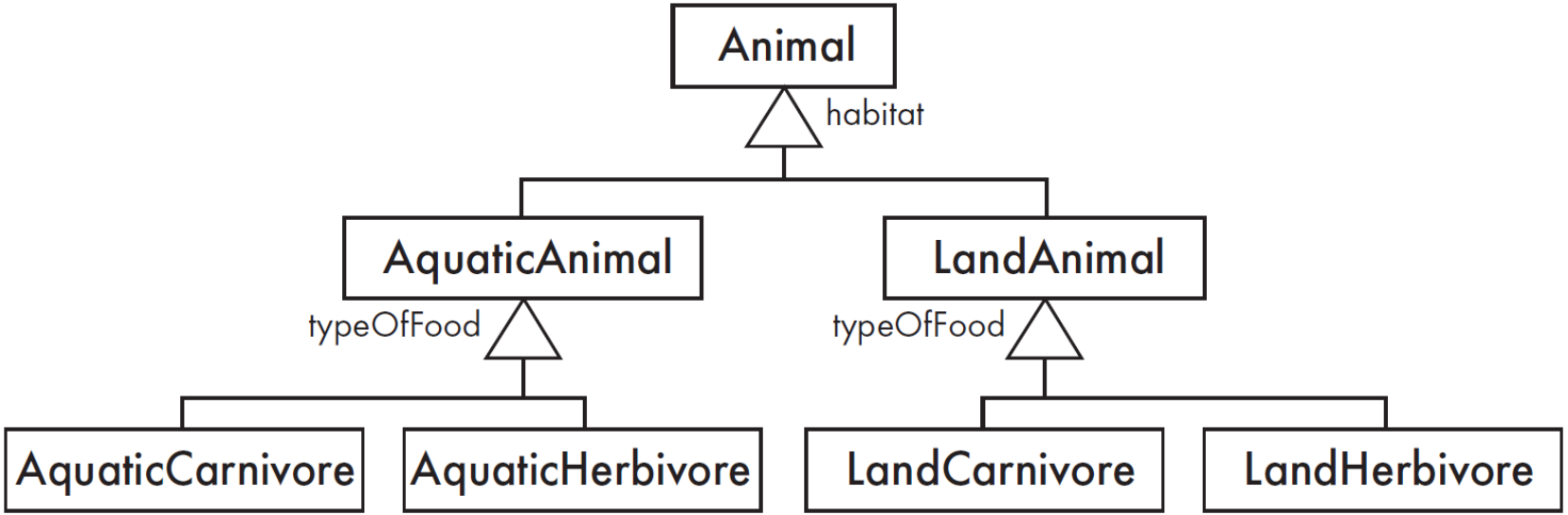


- Should be like:



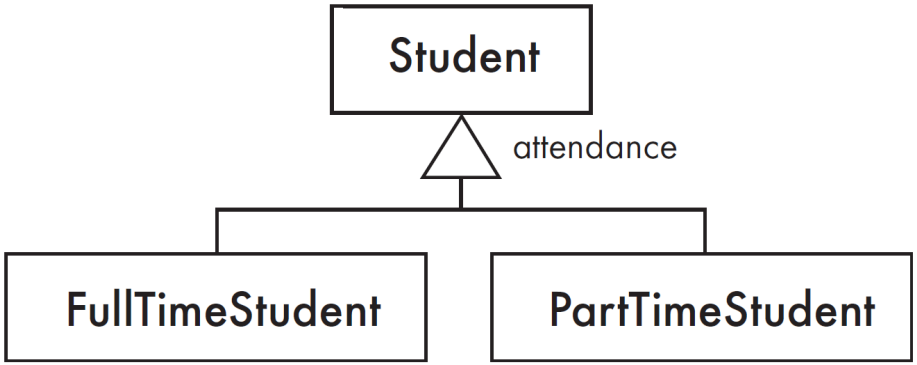
Avoid instances change their type or class

- Creating higher-level generalization



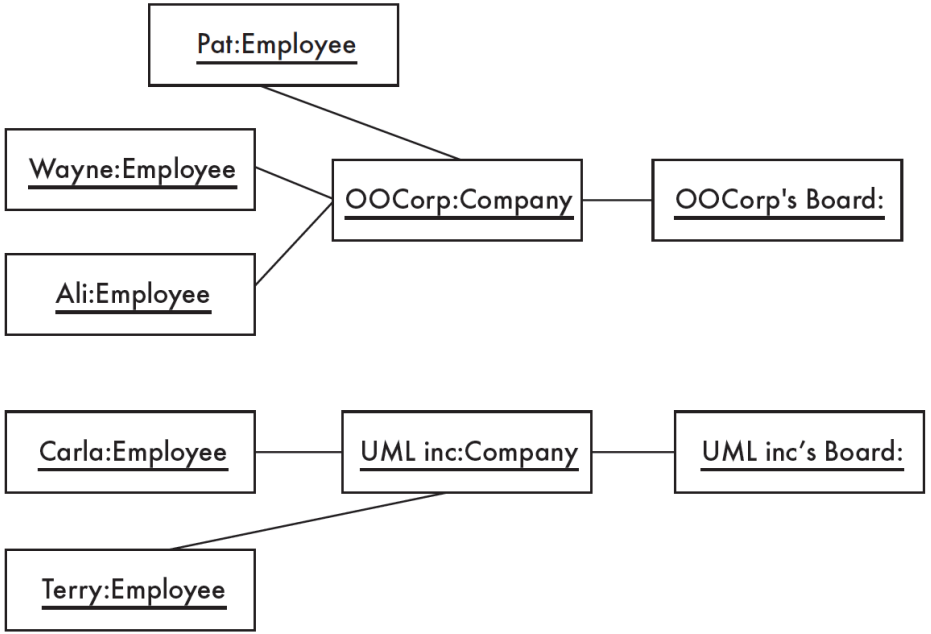
Avoiding having instances change class

- An instance should not need to change its original class



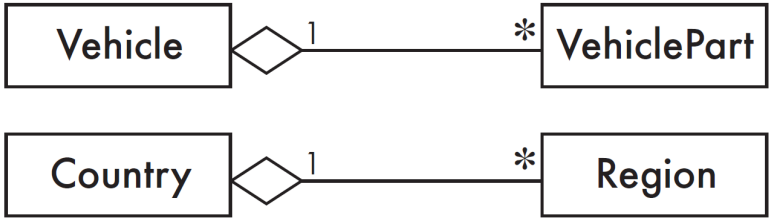
Object Diagrams

- A *link* is an instance of an association
  - In the same way that we say an object is an instance of a class



Aggregation

- Aggregations are special associations that represent ‘part-whole’ relationships
  - The ‘whole’ side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named `isPartOf`

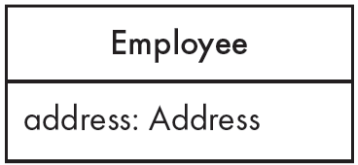


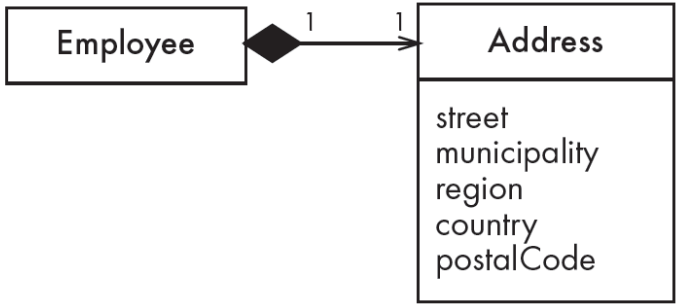
- An association can be marked as an aggregation if the following are true:
  - You can state that:
    - the parts ‘are part of’ the aggregate
    - or the aggregate ‘is composed of’ the parts
  - When something owns or controls the aggregate, they also own or control the parts

Composition

- A composition is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well

- Two alternatives for addresses



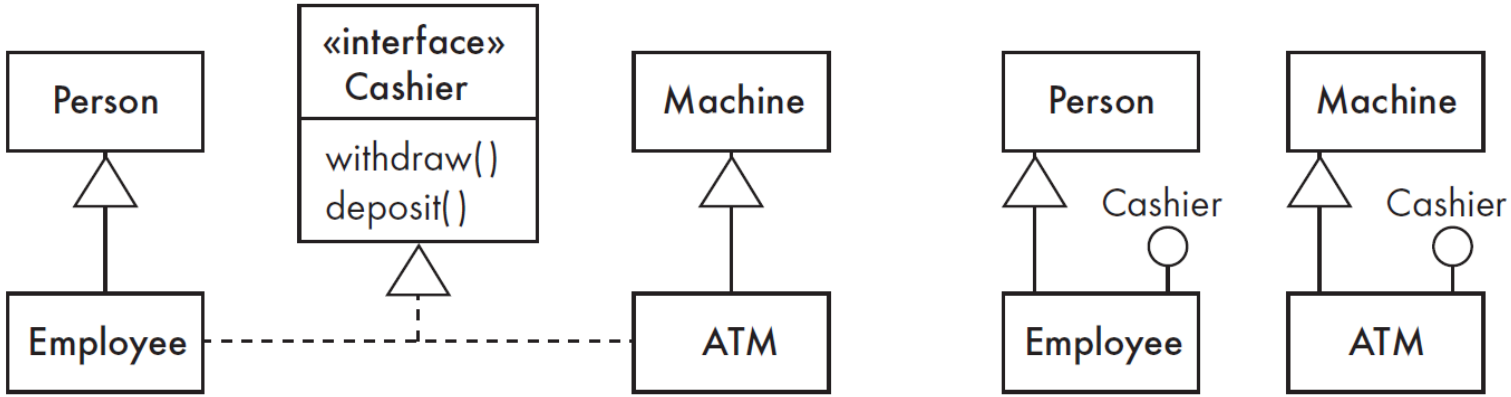


Propagation

- A mechanism where an operation in an aggregate is implemented by having the aggregate perform that operation on its parts
- At the same time, properties of the parts are often propagated back to the aggregate
- Propagation is to aggregation as inheritance is to generalization.
  - The major difference is:
    - inheritance is an implicit mechanism
    - propagation has to be programmed when required

Interfaces

- An interface describes a portion of the visible behavior of a set of objects
  - An *interface* is similar to a class, except it lacks instance variables and implemented methods



Notes and descriptive text

- Descriptive text and other diagrams:
  - Embed your diagrams in a larger document
  - Text can explain aspects of the system using any notation you like
  - Highlight and expand on important features, and give rationale

- Notes:
  - A note is a small block of text embedded in a UML diagram
  - It acts like a comment in a programming language

Object Constraint Language (OCL)

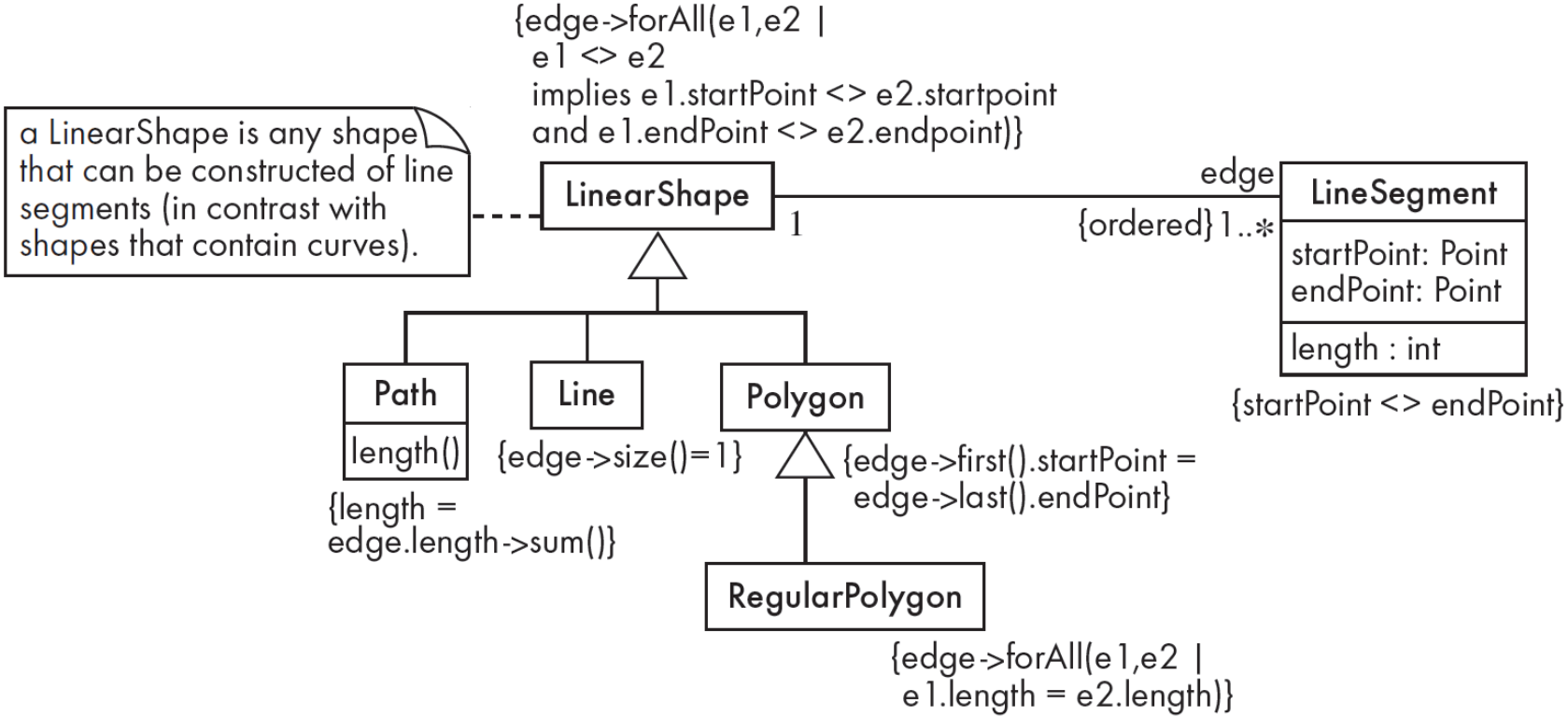
OCL is a specification language designed to formally specify constraints in software modules

- An OCL expression simply specifies a logical fact (a constraint) about the system that must remain true
- A constraint cannot have any side-effects
  - it cannot compute a non-Boolean result nor modify any data.
- OCL statements in class diagrams can specify what the values of attributes and associations must be

OCL statements

- OCL statements can be built from:
  - References to role names, association names, attributes and the results of operations
  - The logical values true and false
  - Logical operators such as `and`, `or`, `=`, `>`, `<`, or `<>` (not equals)
  - String values such as: `'a string'`
  - Integers and real numbers
  - Arithmetic operations `*`, `/`, `+`, `-`

Example: Constraints on Polygons



Suggested design approach

- Identify a first set of candidate ***classes***
- Focus on the core 1-2 classes initially
  - Add ***associations*** and ***attributes***
  - Find ***generalizations***
  - Iterate for the other classes
- List the main ***responsibilities*** of each class
- Decide on specific ***operations***
- ***Iterate*** until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify *interfaces*
  - Apply *design patterns*
- Don’t be too disorganized. Don’t be too rigid either.

Discovering classes

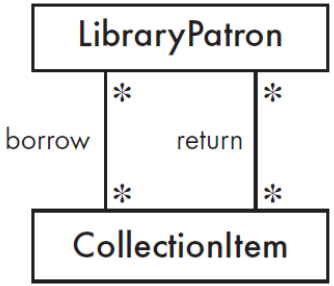
- Look at a source material such as a description of requirements
- Extract the nouns and noun phrases
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application
- Pay attention to classes in a model that represent types of users or other actors

Identifying associations and attributes

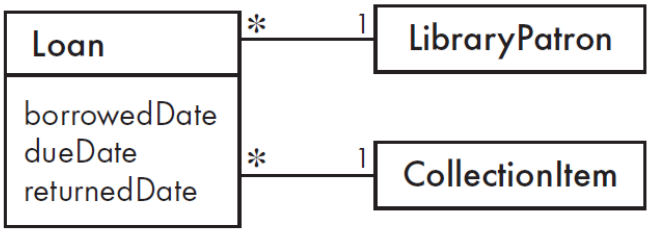
- Start with classes you think are most central and important
- Decide on the clear and obvious data it must contain and its relationships to other classes
- Work outwards towards the classes that are less important
- Avoid adding many associations and attributes to a class
  - A system is simpler if it manipulates less information
- An association should exist if a class
  - Possesses, controls, is connected to, is related to, is a part of, has as parts, is a member of, or has as members some other class in your model
- Specify the multiplicity at both ends
- Label it clearly

Actions versus associations

A common mistake is to represent ***actions*** as if they were ***associations***



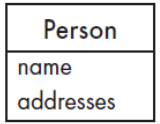
Bad, due to the use of associations that are actions



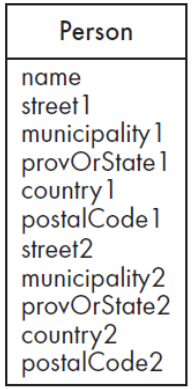
Better. The `borrow` operation creates a `Loan`, and the `return` operation sets the `returnedDate` attribute

Identifying attributes

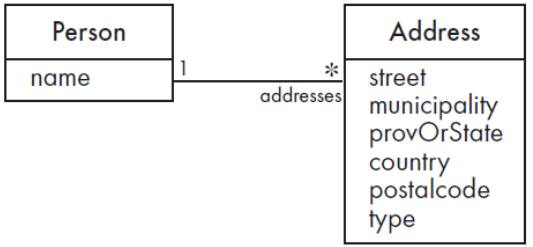
- Look for information that must be maintained about each class
- An attribute should generally contain a simple value (e.g. string, number, etc.)
- It is not good to have many duplicate attributes
- If a subset of a class's attributes form a coherent group, then create a distinct class containing these attributes



Bad, due to a plural attribute



Bad, due to too many attributes, and the inability to add more addresses



Good solution. The type indicates whether it is a home address, business address etc.

Identifying generalizations and interfaces

- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
- Create an *interface*, instead of a superclass if
  - The classes are very dissimilar except for having a few operations in common
  - One or more of the classes already have their own superclasses
  - Different implementations of the same class might be available

Allocating responsibilities to classes

A responsibility is something that the system is required to do.

- Each functional requirement must be attributed to one of the classes
  - All the responsibilities of a given class should be clearly related.
  - If a class has too many responsibilities, consider splitting it into distinct classes
  - If a class has no responsibilities attached to it, then it is probably useless
  - When a responsibility cannot be attributed to any of the existing classes, then a new class should be created
- To determine responsibilities
  - Perform use case analysis
  - Look for verbs and nouns describing actions in the system description

Categories of responsibilities

Written by a code generator such as **Umple**

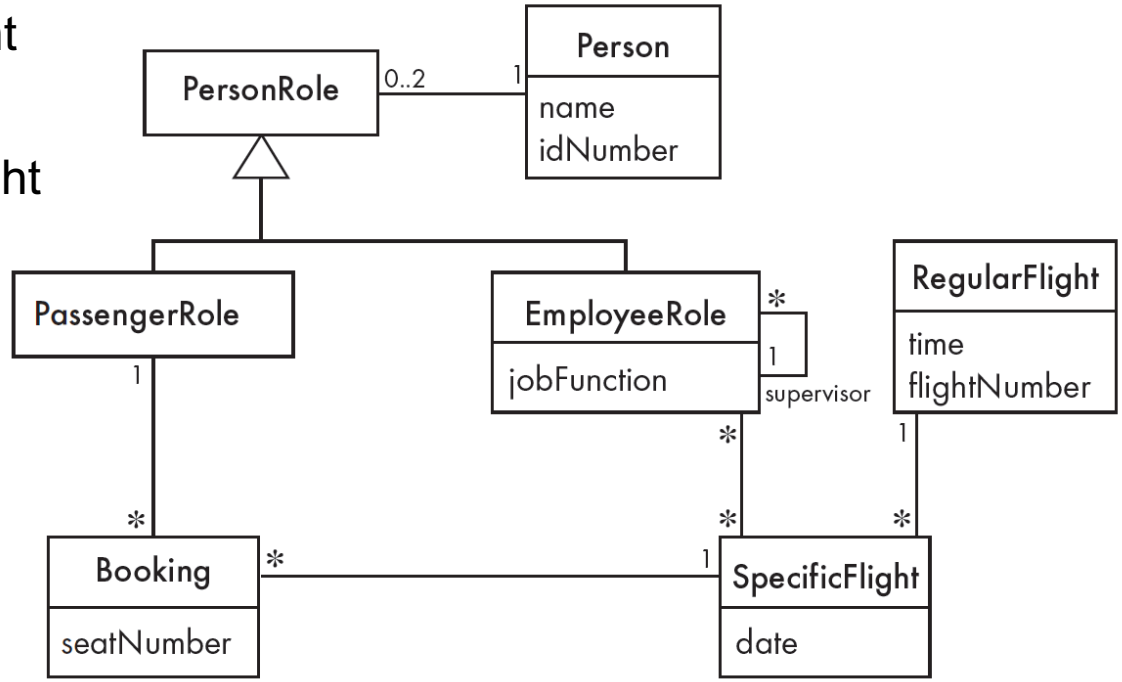
- **Setting** and **getting** the values of attributes
- **Creating** and **initializing** new instances
- **Destroying** instances
- **Adding** and **deleting** links of associations

May require specialized code generation or manual coding

- **Loading** to and **saving** from persistent storage
- **Copying**, **converting**, **transforming**, **transmitting** or **outputting**
- **Computing** numerical results
- **Navigating** and **searching**
- Other specialized work

Example: responsibilities

- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking



Identifying operations

Operations are needed to realize the responsibilities of each class

- There may be several operations per responsibility
- The main operations that implement a responsibility are normally declared public
- Other methods that collaborate to perform the responsibility must be as private as possible