UML for Data Modeling

Vu Tuyet Trinh
trinhvt@soict.hust.edu.vn

Department of Information Systems
SoICT-HUST

What is UML?

- The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling.

UML is a language for

- **Visualizing**: Graphical models with precise semantics
- **Specifying**: Models are precise, unambiguous and complete to capture all important Analysis, Design, and Implementation decisions.
- **Constructing**:   Models can be directly connected to programming languages, allowing forward and reverse engineering
- **Documenting**:   Diagrams capture all pieces of information collected by development team, allowing to share and communicate the embedded knowledge.

Unified Modeling Language (UML)

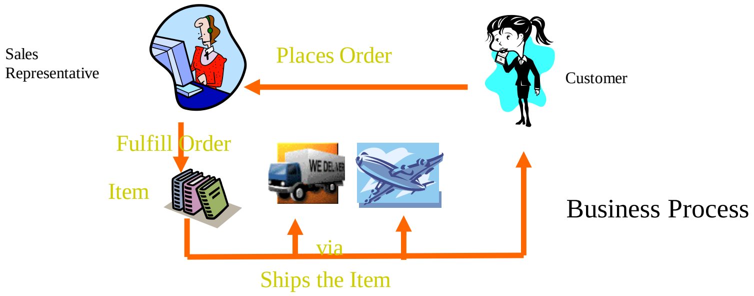
- An effort by IBM (Rational) – OMG to standardize OOA&D notation
- Combine the best of the best from
  - Data Modeling (Entity Relationship Diagrams);
    Business Modeling (work flow); Object Modeling
  - Component Modeling (development and reuse - middleware, COTS/GOTS/OSS/…:)
- Offers vocabulary and rules for communication
- *Not* a process but a language

UML is for Visual Modeling

- standard graphical notations: Semi-formal

A picture is worth a thousand words!

- for modeling enterprise info. systems, distributed Web-based applications, real time embedded systems, …



Sales Representative

Places Order

Customer

Fulfill Order

Item

via

Ships the Item

Business Process

- *Specifying & Documenting:* models that are precise, unambiguous, complete
  - □ UML symbols are based on well-defined syntax and semantics.
  - □ analysis, architecture/design, implementation, testing decisions.
- *Construction:* mapping between a UML model and OOPL.

UML diagrams

- Class diagrams
  - describe classes and their relationships
- Interaction diagrams
  - show the behaviour of systems in terms of how objects interact with each other
- State diagrams and activity diagrams
  - show how systems behave internally
- Component and deployment diagrams
  - show how the various components of systems are arranged logically and physically

Class Diagrams

Essentials of UML Class Diagrams

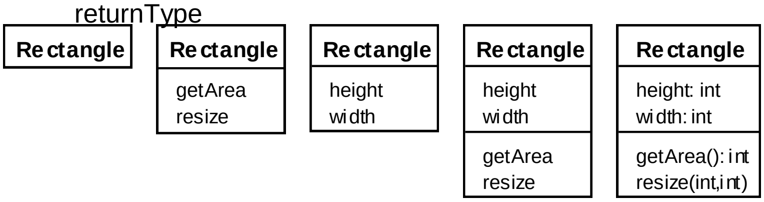
□ *The main symbols shown on class diagrams are:*

- *Classes*
  - represent the types of data themselves
- *Associations*
  - represent linkages between instances of classes
- *Attributes*
  - are simple data found in classes and their instances
- *Operations*
  - represent the functions 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



Rectangle

Rectangle
getArea
resize

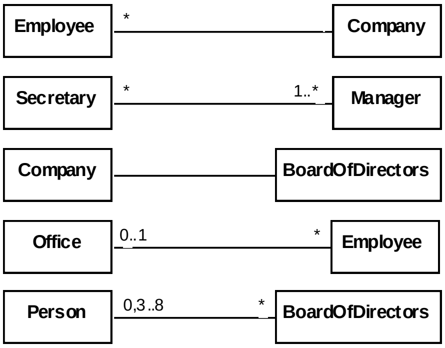
Rectangle
height
width

Rectangle
height
width
getArea
resize

Rectangle
height: int
width: int
getArea(): int
resize(int,int)

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



Employee * — Company

Secretary * — 1..* Manager

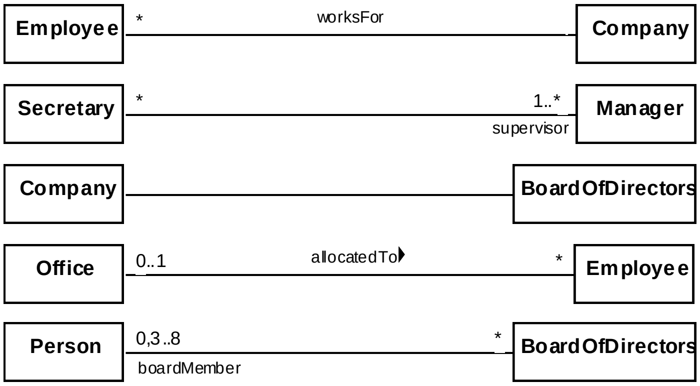
Company — BoardOfDirectors

Office 0..1 — * Employee

Person 0,3..8 — * BoardOfDirectors

Labelling associations

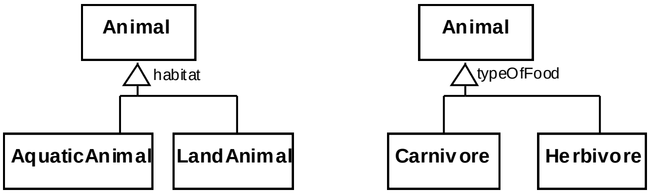
- Each association can be labelled, to make explicit the nature of the association



Generalization

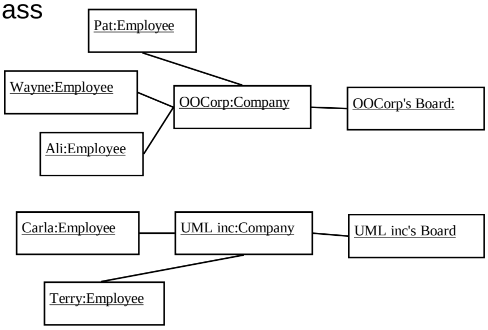
□Specializing a superclass into two or more subclasses

- The *discriminator* is a label that describes the criteria used in the specialization



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



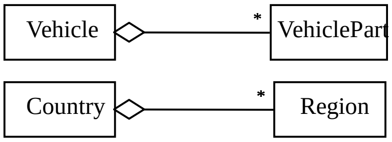
Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between *instances* at run time.
  - When you show an object diagram generated from a class diagram, there will be an instance of *both* classes joined by an association

- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in object diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses

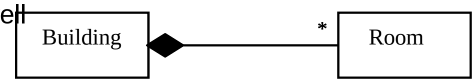
More Advanced Features: Aggregation

- 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`

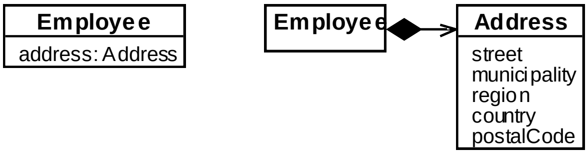


Composition

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



- Two alternatives for addresses



The Process of Developing Class Diagrams

□ You can create UML models at different stages and with different purposes and levels of details

- **Exploratory domain model**:
  - Developed in domain analysis to learn about the domain
- **System domain model**:
  - Models aspects of the domain represented by the system
- **System model**:
  - Includes also classes used to build the user interface and system architecture

Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process 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.*

A simple technique for discovering domain 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 domain model that represent *types of users* or other actors

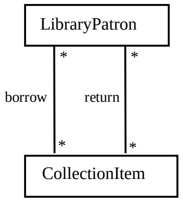
Tips about identifying and specifying valid associations

- 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*

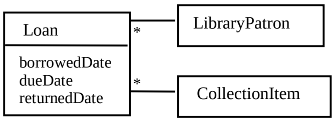
  some other class in your model
- Specify the multiplicity at both ends
- Label it clearly.

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



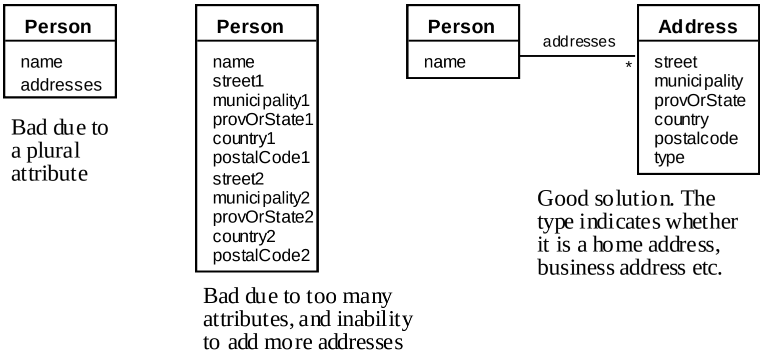
Better: The *borrow* operation creates a *Loan*, and the *return* operation sets the returnedDate attribute

Identifying attributes

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number

Tips about identifying and specifying valid attributes

- 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



Person

name
addresses

Bad due to a plural attribute

Person

name
street1
municipality1
provOrState1
country1
postalCode1
street2
municipality2
provOrState2
country2
postalCode2

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

Person

name

addresses

*

Address

street
municipality
provOrState
country
postalcode
type

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

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

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