Object-Oriented Programming

Object modelling and relations between objects

Anders Jonsson & Federico Heras
2023-24

Course topics

Topic 1 Introduction and the concept of objects

Topic 2 The object-oriented programming paradigm

Topic 3 Object modelling and relations between objects

Topic 4 Inheritance and polymorphism

Topic 5 Abstract classes and interfaces

Topic 6 Reuse and study of problems solved using objects

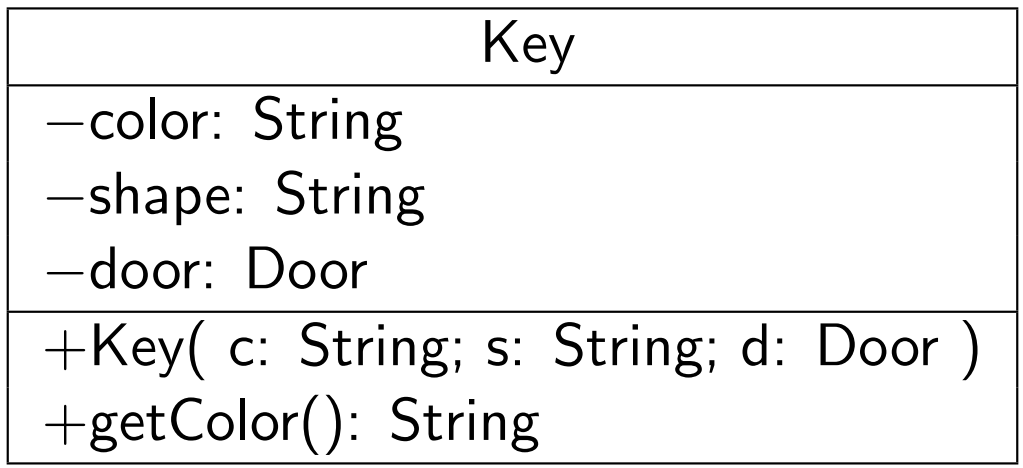
Classes and instances

- Fundamental units of object-oriented programming
- **Class**:
  - Abstract idea
  - Represents an object family
  - Defines the attributes and methods that are shared by all objects in this family
- **Instance**:
  - Concrete object
  - Belongs to an object family (that is, a class)
  - Assigns a concrete value to each attribute of the class

Instance members

- **Attribute**:
  - Describes a category that is common to all objects of the class
  - Implemented as a *variable*
  - When defining the class the *value* of the attribute is unknown!
- **Method**:
  - Describes a behavior that is common to all objects of the class
  - Implemented as a *procedure* or a *function*
  - Acts on the attributes, without knowing their concrete value

Class definition



Theory session 3

Class members

Object modelling

Relations among classes

Intuition

- Instance members are associated with individual instances
  - Each instance has its own copy of each attribute
  - A method can only be applied on a concrete instance
- There exists the possibility of associating members with classes instead of instances

Class member

- Class member: attribute or method that belongs to a class
- In Java and C++, a class member is implemented using the keyword static
- A class member is not associated with individual instances!
- A class member is accessed using the name of the class itself:
  - `System.out`
  - `Math.cos( Math.PI )`

The keyword `final`

- Modifier for classes, methods and attributes
- Meaning for classes and methods explained in later sessions
- For attributes, means that the value of an attribute is a constant
- The value has to be assigned on the same line that the attribute is declared
- Normally combined with static

The keyword `this`

- Each (instance) method is applied on a concrete instance
- The keyword `this` refers to this instance!

```
public class Example {
   private String attribute;
   public static final int myconstant = 5;

   public Example( String attribute ) {
      this.attribute = attribute;
   }
}
```

Exercises

1. Define and implement a class that represents bank accounts
   - Each account has a different balance
   - However, the interest rate is shared among all accounts
2. Define and implement a class that represents students
   - Each student has a unique ID!

Theory session 3

Class members

Object modelling

Relations among classes

Object modelling

- Technique for designing programs with objects
- Design is a previous step to implementation
- Essential to implement well-structured programs
- Programming: design + implementation + maintenance

Object modelling

- An object-oriented programming language normally offers the possibility of creating new objects
- Consequently, an important task is thinking about how new objects should be defined
- This task is sufficiently important that it has generated its own field of study: object modelling

Object modelling

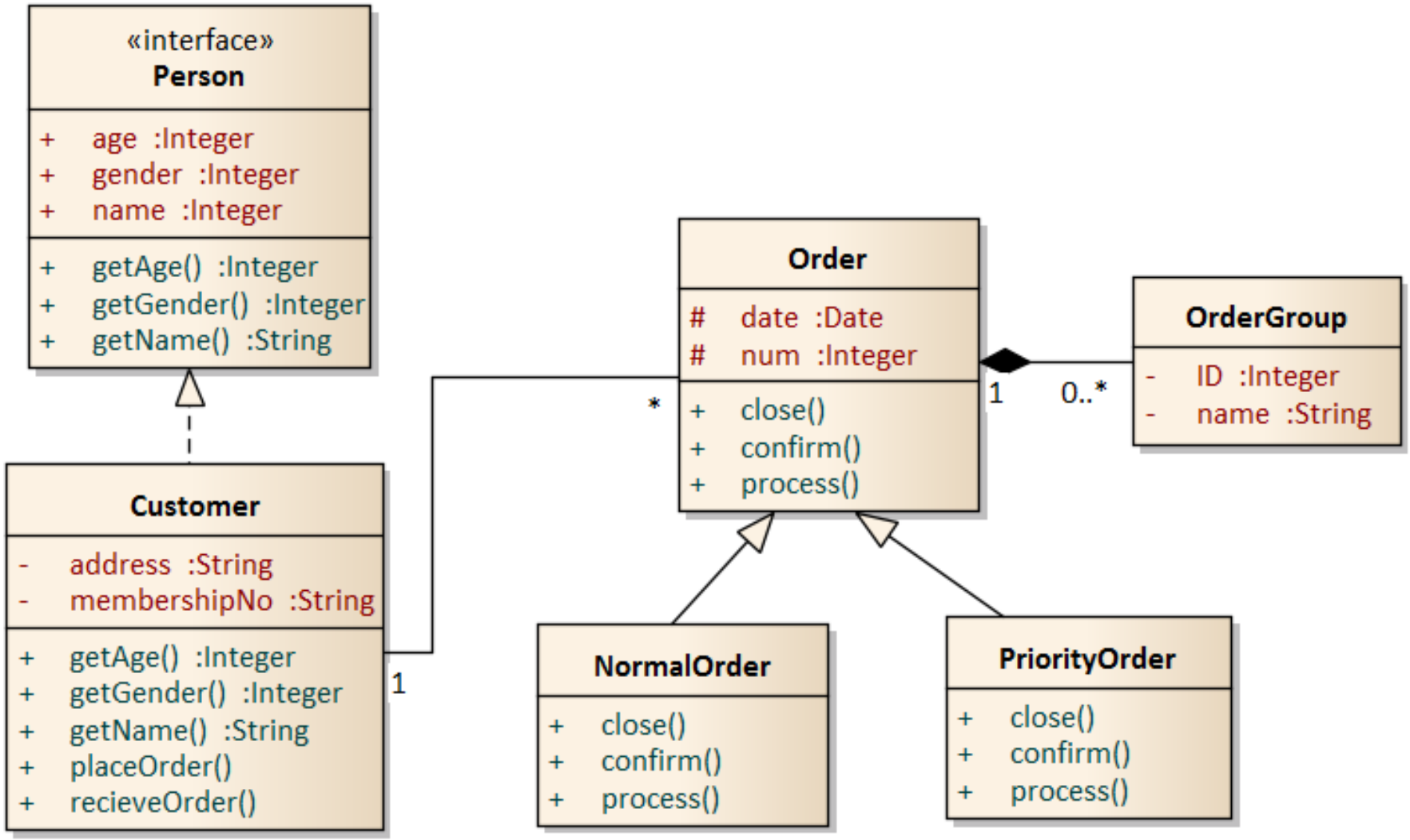
- Object modelling attempts to answer the following questions:
  - How should objects be defined?
  - How should objects be related?
  - How should objects be structured?

  The answers to these questions represent the design of the program

Object-oriented design

1. Identify the objects that will participate in the solution
2. If an object is already defined, reuse and/or modify
3. If an object is not defined, create a new definition
4. Specify the relations that exist among objects
5. Identify the type of each relation
6. Determine how the objects interact in the solution

Class diagram



Theory session 3

Class members

Object modelling

Relations among classes

Relations among classes

- Relation between two classes $\approx$ interaction of their instances
- In general, all classes of a program are related (directly or indirectly)
- If the instances of a class do not interact with instances of other classes, the class can effectively be eliminated
- The class diagram is always a connected graph!

Relations among classes

- When designing a new class it is necessary to specify:
  - Attributes: variables that describe characteristics of instances
  - Methods: tasks that the instances of the class will perform
  - The relations of the class with other classes of the program
- There is no object-oriented programming without class relations!

Relations among objects

- Definition: two or more objects are related if there exists a link between them that can be expressed by verbs
  - A car is a vehicle
  - The department is part of the company
  - An application uses the monitor and the keyboard

Identifying relations

- Two fundamental reasons why two objects are related:

1. The objects share some characteristics
   - Example: lions and tigers are both mammals (in addition, both belong to the *Panthera* genus)
2. There exists a semantic connection inside the program
   - Example: the buttons and text fields are both part of the graphical interface

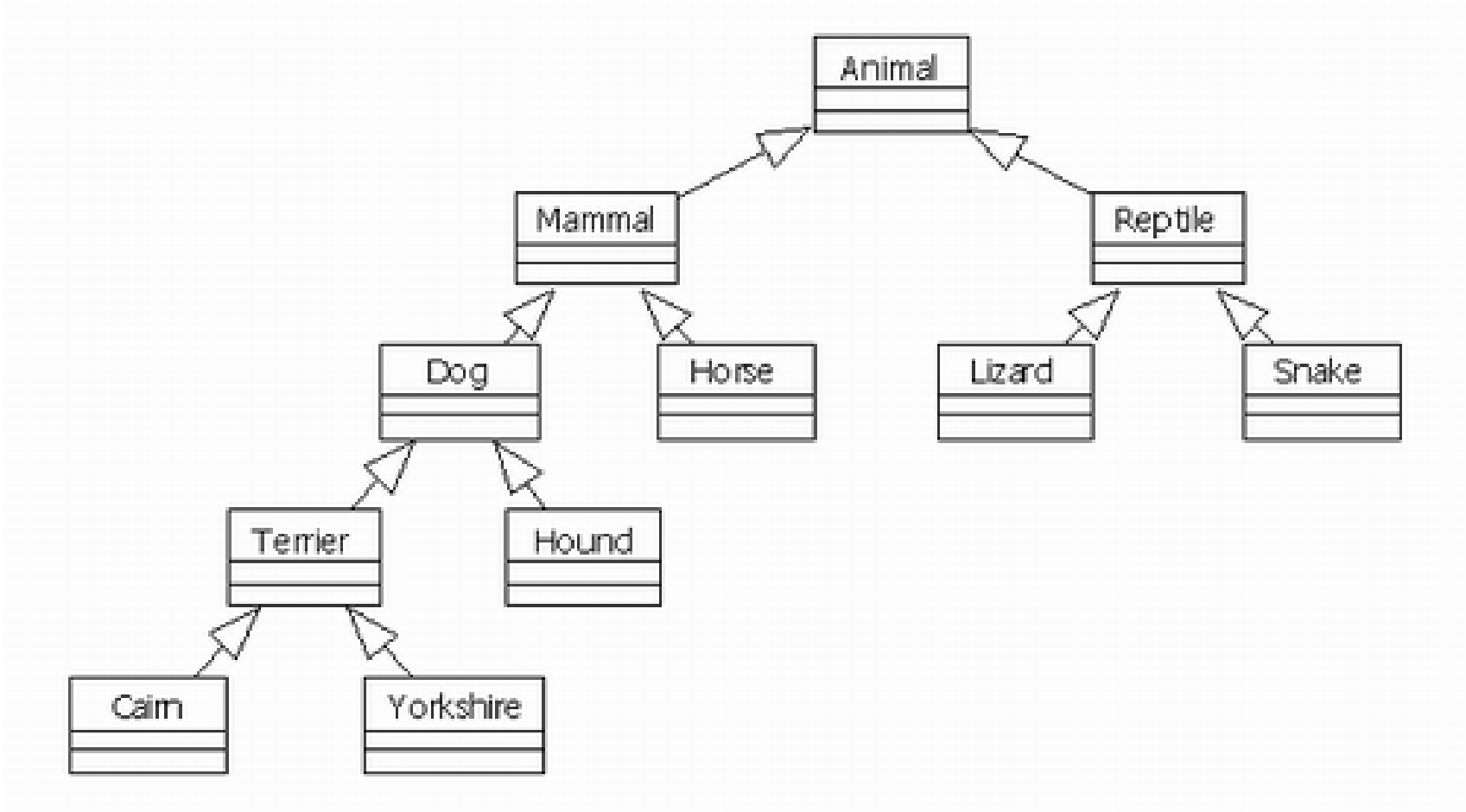
Types of relations

- There exist five fundamental types of relations:
  1. Generalization/specialization or inheritance (is a)
  2. Composition/aggregation (is part of)
  3. Use/dependency (uses)
  4. Association (general relation)
  5. Template/generics

Generalization/specialization

- Represented by the property of inheritance
- Can be expressed by the verb is a
- Transitive relation:
  - A car is a vehicle
  - A vehicle is a means of transport
    $\Rightarrow$ a car is a means of transport
- Bidirectional: A generalizes B if B specializes A

Generalization/specialization



Identification

- Inheritance is an efficient design mechanism
- Specific to object-oriented programming
- Often a good idea to establish inheritance relations before other types of relations

Exercise

- Construct a hierarchy of classes that includes all vehicles that transit on public roads

Composition/aggregation

- Represents composite objects
- Can be expressed by the verbs is part of or has a
- Bidirectional:
  - The car has an engine
  - The engine is part of the car

Composition/aggregation

- An object is composite if it is formed by other objects
- Describes models that are composed by other models
- Makes it possible to define objects composed of other, existing objects
- Aggregate or container: composite object
- Component or part: object that is part of the aggregate

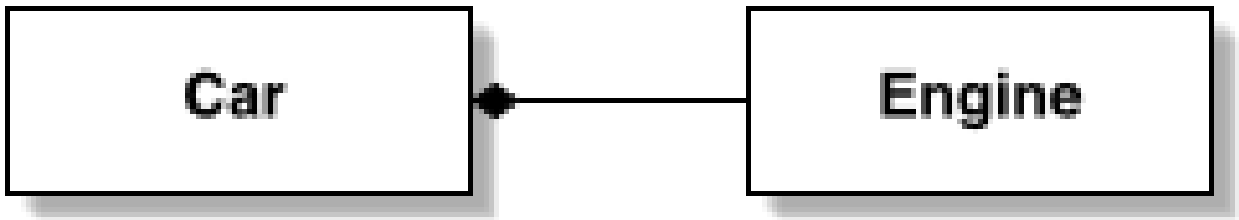
Rumbaugh's rules

- To determine whether a relation is a composition/aggregation we can apply Rumbaugh's rules:
  1. Can the relation be expressed by is part of, has a, etc.?
  2. Do the operations of the container also apply to the parts?
  3. Are the attribute values of the container propagated to some of the parts?
  4. Does there exist an intrinsic assymmetry such that one class is subordinated to another?
- If any of the four questions has an affirmative answer, the relation is a composition/aggregation

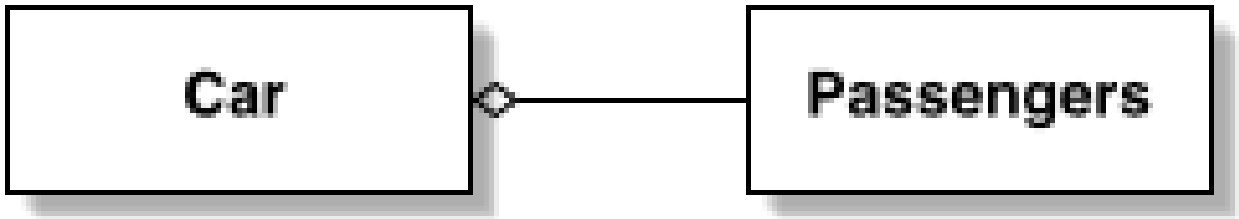
Types of composition/aggregation

- Composition (strong relation): the parts cannot exist without the container (the container owns the parts)
- Aggregation (weak relation): the parts exist independently of the container

Types of composition/aggregation



Composition: every car has an engine.

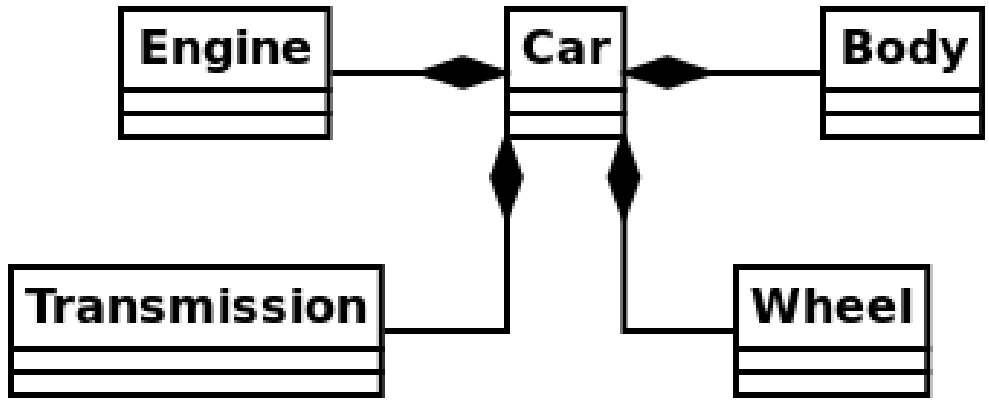


Aggregation: cars may have passengers, they come and go

Composition/aggregation

- Composition/aggregation implies that the container has attributes that represent the parts
- Optionally, the components may have an attribute that refers to the container

Example composition




```
public class Car {
    private Engine engine;
    private Transmission transmission;
    private Body body;
    private Wheel w1, w2, w3, w4;
}
```

Exercise

- Design a program that represents a music collection. Include classes for songs, artists, albums and playlists.

Use/dependency

- "Indirect" relation between A and B
- No attributes that store instances of the other class
- Two fundamental reasons for the relation:
  - Class A has a method that contains arguments of type B
  - Class A creates instances of class B

Use/dependency - Motivation

- Class A uses class B, but does not store references to B
- Models the idea of indirect access
- Temporary use: the relation only exists during the moment of calling a method or creating an instance
- Example: a class Melody could use a class Speaker to reproduce its sound

Use/dependency

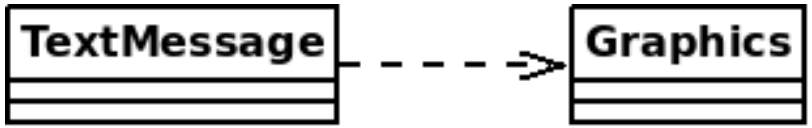
- Normally represents a unidirectional relation
  - Independent class: the class being used
  - Dependent class: uses instances of the other class
- Example: a class Car needs to know the state of a class TrafficLight, but not the other way around
- Application: method of the dependent class that contains an argument of the independent class

```
melody.play( speaker );
```

Use/dependency

- Pass a reference of the independent class to the dependent class
- By third parties, or by obtaining the reference directly
- The dependent class does not store a reference!
- When the method call finishes the reference is thrown away

Example of use/dependency



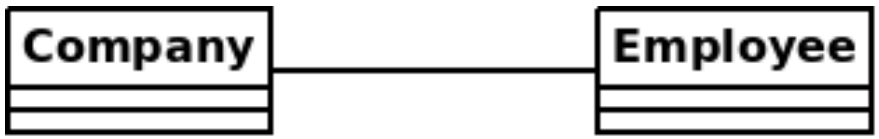

```
public class Graphics {
   // independent class
}

public class TextMessage {
   // dependent class
   private String msg;
   ...
   public void draw( java.awt.Graphics graphics ) {
      graphics.drawString( msg, 0, 0 );
   }
}
```

Association

- Relation "by default"
- Semantic connection that does not correspond to one of the other types
- Normally bidirectional, but can be unidirectional
- Defines the roles that exist among different objects
- Normally binary, but can be unary or ternary
- Also implies that a class has attributes of another class

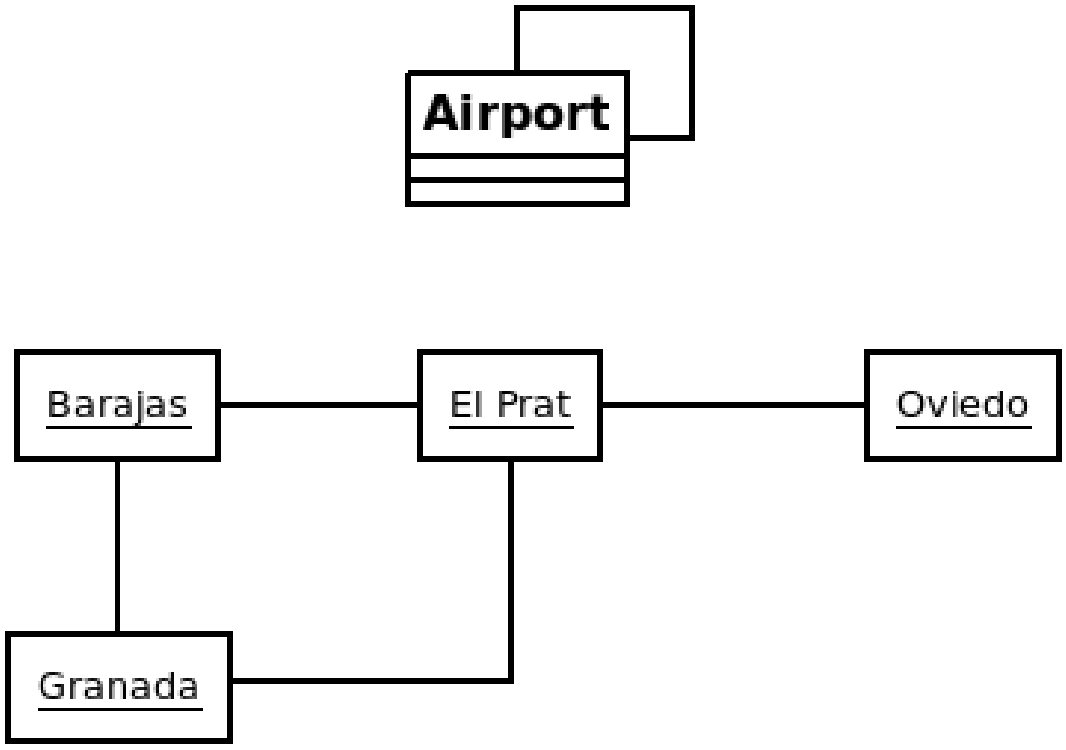
Example association




```
public class Company {
    private Employee emp;
    ...
}
```

```
public class Employee {
    private Company comp;
    ...
}
```

Example unary association




```
public class Airport {
    private Airport[] connections;
    ...
}
```

Association

- General relation that includes other types (composition/aggregation)
- If a relation is not of any other type, by elimination it is an association

Strategy for identifying the type of a relation

1. Check if it is a case of inheritance
2. Check if it is a case of use/dependency
3. Apply Rumbaugh's rules to find out if it is a case of composition/aggregation
4. If the relation type has not been identified, by default it is an association

Summary

- Class members
- All objects of a program are related
- Important task: identify relations among objects
- The relations are organized by type
- Rumbaugh's rules: used to identify relations of type aggregation/composition