CSC 2210
Object Oriented Analysis & Design

Dr. Akinul Islam Jony

Associate Professor

Department of Computer Science, FSIT

American International University - Bangladesh (AIUB)

akinul@aiub.edu

Class Diagram

>> What is Class Diagrams?
>> Class vs. Object Diagram
>> Classes
>> Associations
>> Aggregation and Composition
>> Generalization
>> Examples
>> Exercises

What is Class Diagrams?

>> A **class diagram** shows a set of classes, interfaces, and collaborations and their relationships.

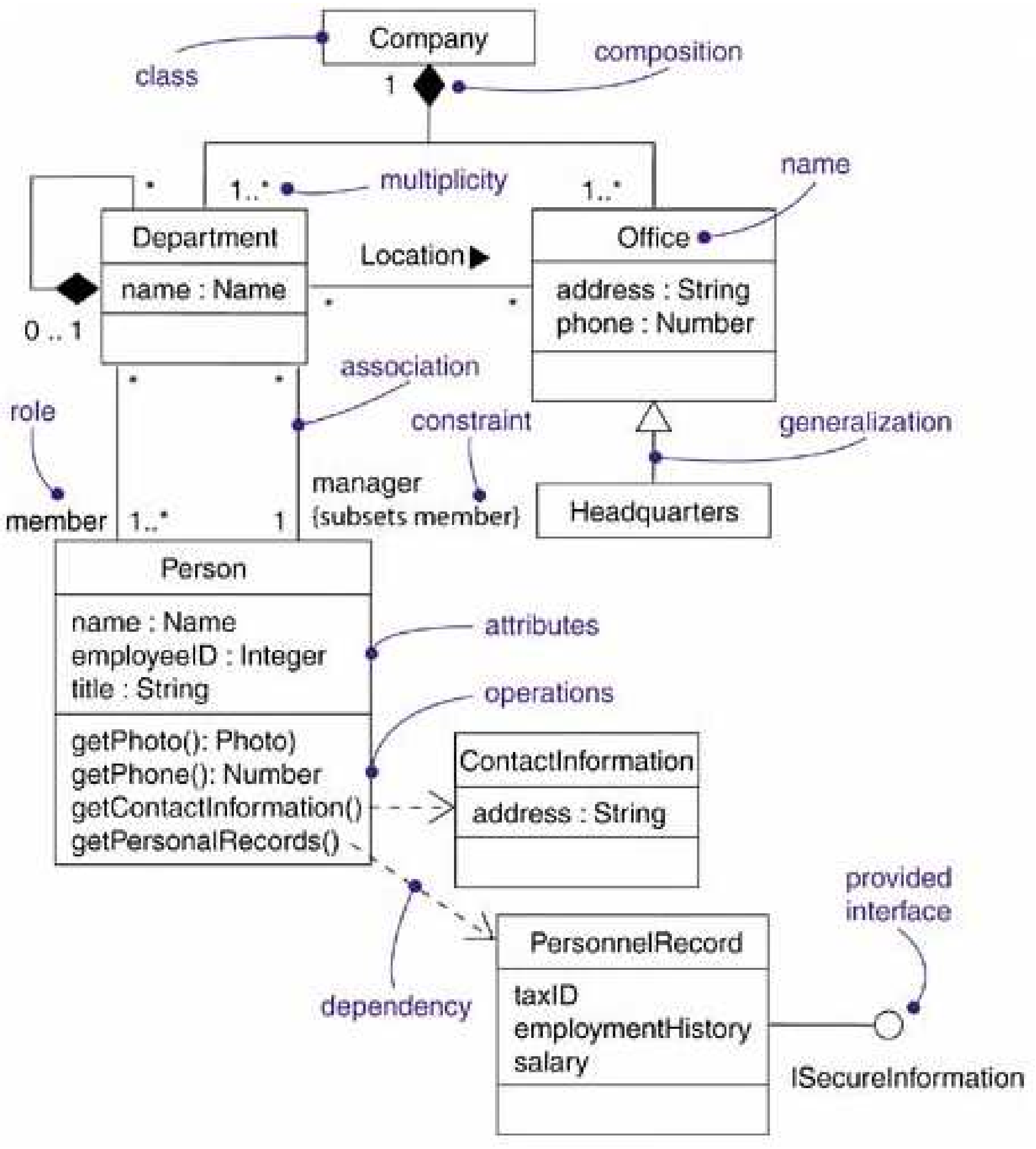
>> A class is a definition for a type of object.

>> **Class diagrams** used to **model** the **static design view** of a system.

>> Class diagrams are important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.

What is Class Diagrams?

Figure 4-1. Example of Class Diagram



What is Class Diagrams?

>> The Class diagram represents classes, their component parts, and the way in which classes of objects are related to one another.

>> Class diagram includes attributes, operations, stereotypes, properties, associations, and inheritance.

What is Class Diagrams?

>> **Attributes** describe the appearance and knowledge of a class of objects.

>> **Operations** define the behavior that a class of objects can manifest.

>> **Stereotypes** help you understand this type of object in the context of other classes of objects with similar roles within the system’s design.

>> **Properties** provide a way to track the maintenance and status of the class definition.

>> **Association** is just a formal term for a type of relationship that this type of object may participate in. Associations may come in many variations, including simple, aggregate and composite, qualified, and reflexive.

>> **Inheritance** allows you to organize the class definitions to simplify and facilitate their implementation.

What is Class Diagrams?

>> However, as the Class diagram is a static view of the elements that make up the business or software, you can only see the parts used to make it and how they are assembled, but you cannot see how the parts will behave when you set them into motion.

>> This is why we need other diagrams to model behavior and interactions over Time. Figure 4-1 shows how all the other diagrams support the Class diagram

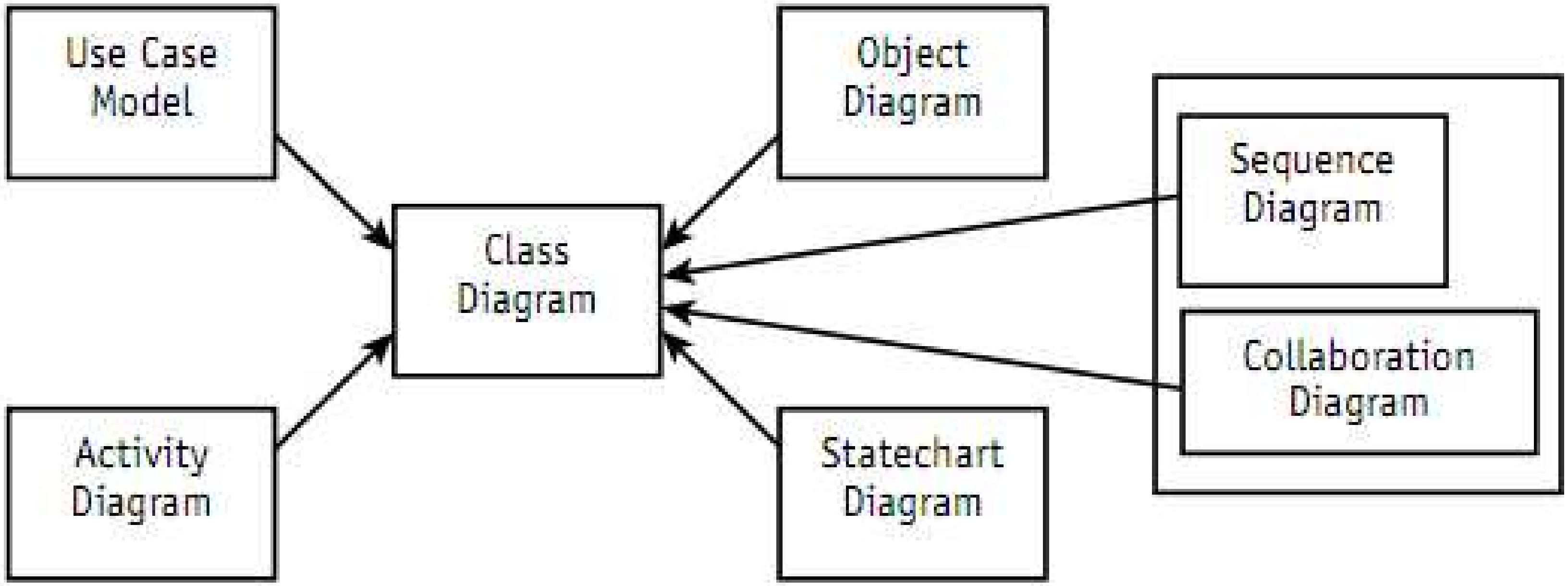


Figure 4-1. All diagrams support the Class diagram

What is Class Diagrams?

>> Although other diagrams are necessary, remember that their primary purpose is to support the construction and testing of the Class diagram.

>> Whenever another diagram reveals new or modified information about a class, the Class diagram must be updated to include the new information. If this new information is not passed on to the Class diagram, it will not be reflected in your code.

Components of Class Diagram

>> Classes

>> Associations

>> Aggregation & Composition

>> Generalization

Class vs. Object Diagram

>> The **class** defines the *rules*; *the **objects** express the facts*.

>> The **class** defines what *can be*; *the **object** describes what is*.

>> If the Class diagram says,

"This is the way things should be,"

but the Object diagram graphically demonstrates that

"it just ain't so,"

then you have a very specific problem to track down. The reverse is true, too.

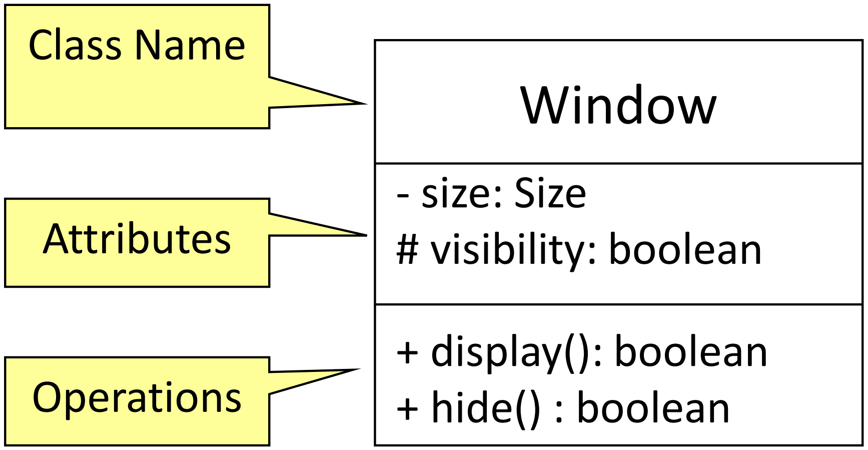
>> The **Object diagram** can confirm that everything is working as it should.

Classes

>> A class is the description of a set of objects having similar attributes, operations, relationships and behavior.

>> The class symbol is comprised of **three compartments** (rectangular spaces) that contain distinct information needed to describe the properties of a single type of object.

- The **name compartment** uniquely defines a class (a type of object) within a package. Consequently, classes may have the same name if they reside in different packages.
- The **attribute compartment** contains all the data definitions.
- The **operations compartment** contains a definition for each behavior supported by this type of object.



Class Name

Attributes

Operations

Window

- size: Size
# visibility: boolean

+ display(): boolean
+ hide() : boolean

Figure 4-6 Example of a class

Modeling an Attribute

>> An attribute describes a piece of information that an object owns or knows about itself.

>> Attributes generally require a **name**, **visibility**, and **data types**. Sometimes attribute contains a **default value** and/or **constraints** (rules) to set the attribute value. Also, sometimes an optional **slash (/)** is used to indicate a derived attribute.

>> There are also one optional attribute called **Class level attribute** (underlined attribute declaration), which denotes that all objects of the class share a single value for the attribute. (This is called a **static** value in Java.)

Modeling an Attribute

>> Attribute **visibility**:

Public (+) visibility allows access to objects of all other classes.

Private (-) visibility limits access to within the class itself. For example, only operations of the class have access to a private attribute.

Protected (#) visibility allows access by subclasses. In the case of generalizations (inheritance), subclasses must have access to the attributes and operations of the superclass or they cannot be inherited.

Package (~) visibility allows access to other objects in the same package.

>> Visibility (+, -, #, ~): *Required before code generation*. The programming language will typically specify the valid options. The *minus* sign represents the visibility “private” meaning only members of the class that defines the attribute may see the attribute.

Modeling an Attribute

>> **Slash** (/): The derived attribute indicator is *optional*. Derived values may be computed or figured out using other data and a set of rules or formulas. Consequently, there are more design decisions that need to be addressed regarding the handling of this data. Often this flag is used as a placeholder until the design decisions resolve the handling of the data.

>> **Attribute name**: *Required*. Must be *unique* within the class.

>> **Data type**: *Required*. This is a big subject. During analysis, the data type should reflect how the client sees the data. You could think of this as the external view. During design, the data type will need to represent the programming language data type for the environment in which the class will be coded. These two pieces of information can give the programmer some very specific insights for the coding of get and set methods to support access to the attribute value.

Modeling an Attribute

>> **Assignment operator and default value**: *Optional*. Default values serve two valuable purposes. First, default values can provide significant ease-of-use improvements for the client. Second and more importantly, they protect the integrity of the system from being corrupted by missing or invalid values. A common example is the tendency to let numeric attributes default to zero. If the application ever attempts to divide using this value, you will have to handle resulting errors that could have been avoided easily with the use of a default.

>> **Constraints**: Constraints express all the rules required to guarantee the integrity of this piece of information. Any time another object tries to alter the attribute value, it must pass the rules established in the constraints. The constraints are typically implemented/enforced in any method that attempts to set the attribute value.

>> **Class level attribute** (underlined attribute declaration): *Optional*. Denotes that all objects of the class share a single value for the attribute. (This is called a *static* value in Java.)

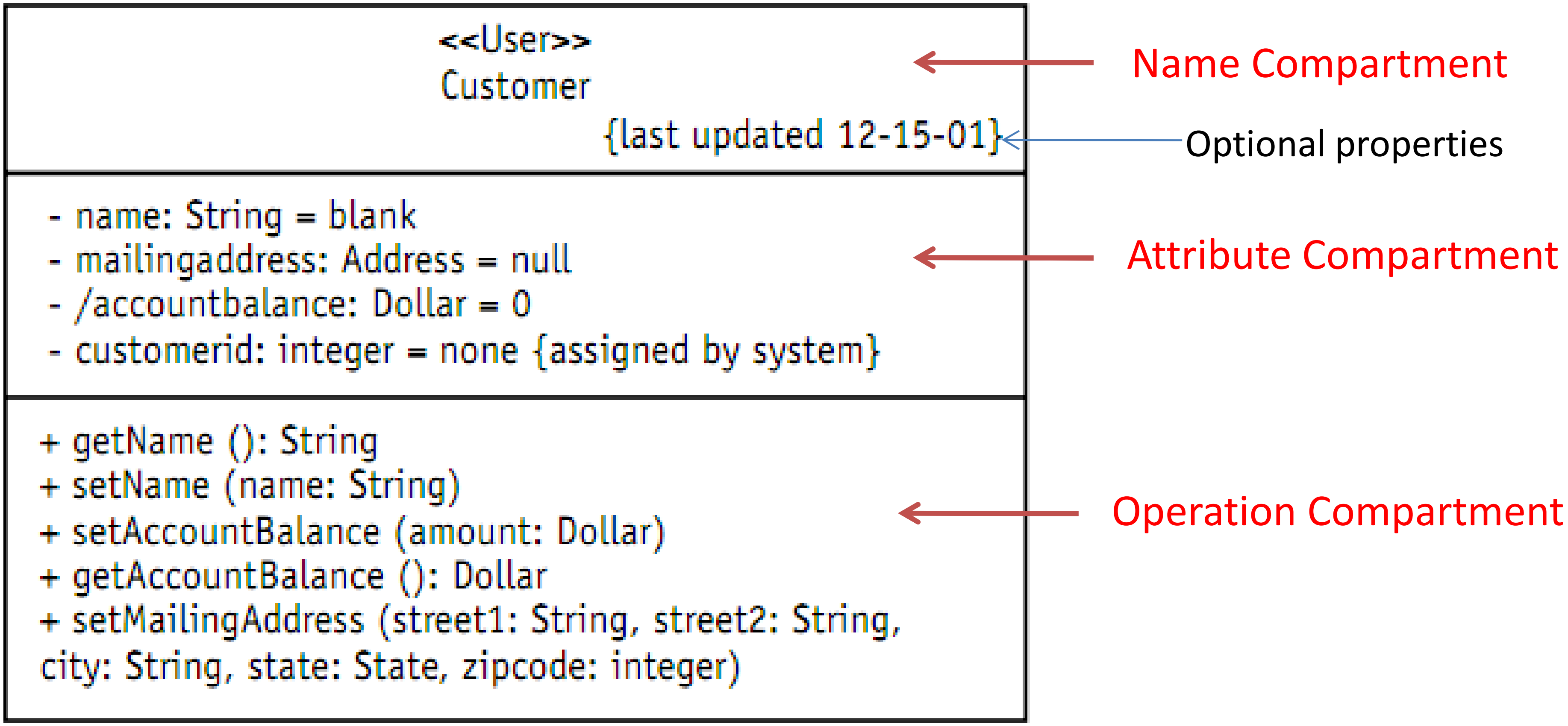
Modeling an Operation

>> Objects have behaviors, things they can do and things that can be done to them. These behaviors are modeled as **operations**.

>> Operations require a **name**, **arguments**, and sometimes a **return**.

>> Arguments, or input parameters, are simply attributes, so they are specified using the attribute notation (name, data type, constraints, and default), although it is very common to use the abbreviated form of name and data type only.

Modeling the Class Compartments



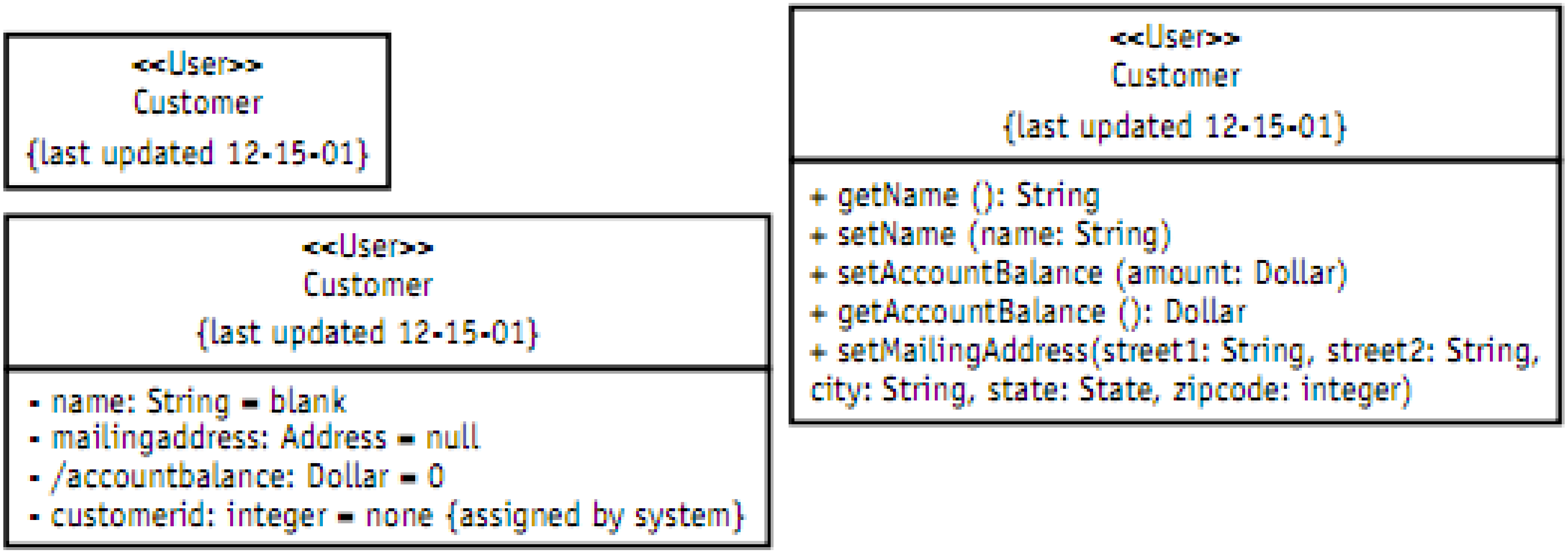
<<User>>
Customer
{last updated 12-15-01}

- name: String = blank
- mailingaddress: Address = null
- /accountbalance: Dollar = 0
- customerid: integer = none {assigned by system}

+ getName (): String
+ setName (name: String)
+ setAccountBalance (amount: Dollar)
+ getAccountBalance (): Dollar
+ setMailingAddress (street1: String, street2: String, city: String, state: State, zipcode: integer)

Figure 4-2 Complete class specification with all three compartments

Modeling the Class Compartments



<<User>>
Customer
{last updated 12-15-01}

<<User>>
Customer
{last updated 12-15-01}

- name: String = blank
- mailingaddress: Address = null
- /accountbalance: Dollar = 0
- customerid: integer = none {assigned by system}

<<User>>
Customer
{last updated 12-15-01}

+ getName (): String
+ setName (name: String)
+ setAccountBalance (amount: Dollar)
+ getAccountBalance (): Dollar
+ setMailingAddress(street1: String, street2: String, city: String, state: State, zipcode: integer)

Figure 4-8 Alternative views of a class symbol for different audiences and purposes

Associations

>> **Association Name:**

The purpose of the association can be expressed in a **name**, a verb or verb phrase that describes how objects of one type (class) relate to objects of another type (class).

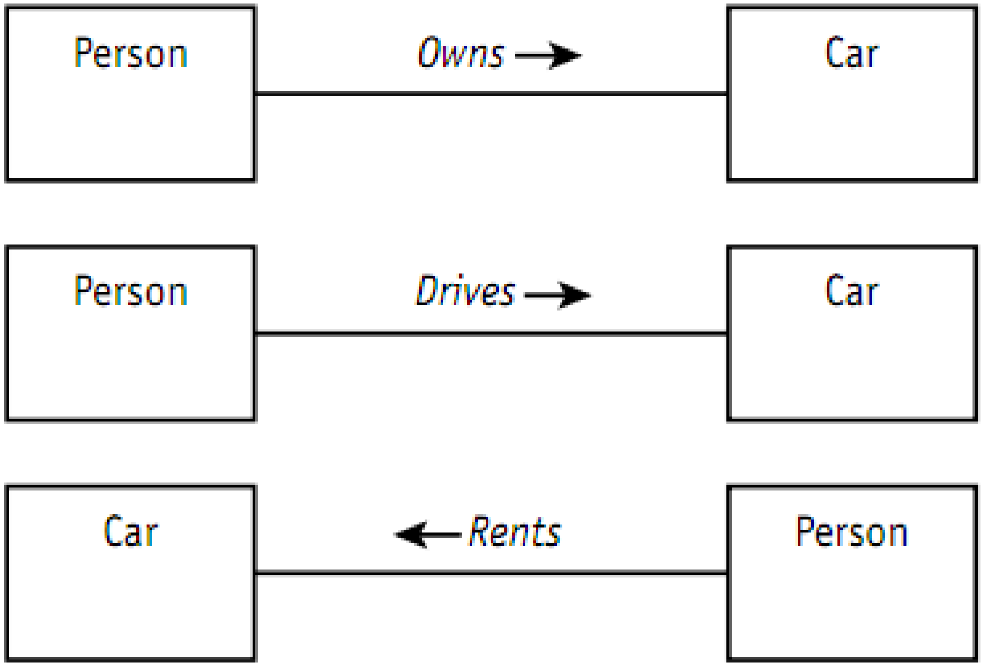


Figure 4-1 Directional notation for association names

Associations

>> **Association Multiplicity:**

- Multiplicity is the UML term for the rule that defines the number of participating objects. A multiplicity value must be assigned to each of the participating classes in an association.

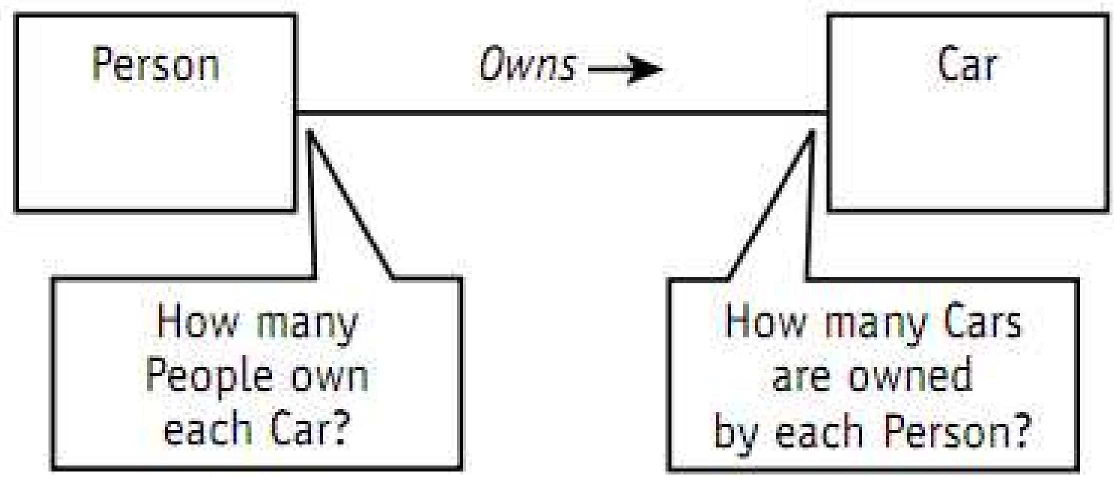

Person
Owns →
Car
How many People own each Car?
How many Cars are owned by each Person?

Figure 4-1 Assigning multiplicity to each end of an association

- The most common way to express Multiplicity is a range defining the minimum number of objects allowed and the maximum number of objects allowed.
- The format to define multiplicity: Minimum . . Maximum
- You must use integer values for the minimum and maximum.

Associations

>> **Association Roles :**

- The UML provides an alternative which is called a **role** to describe how an object participates in the association.

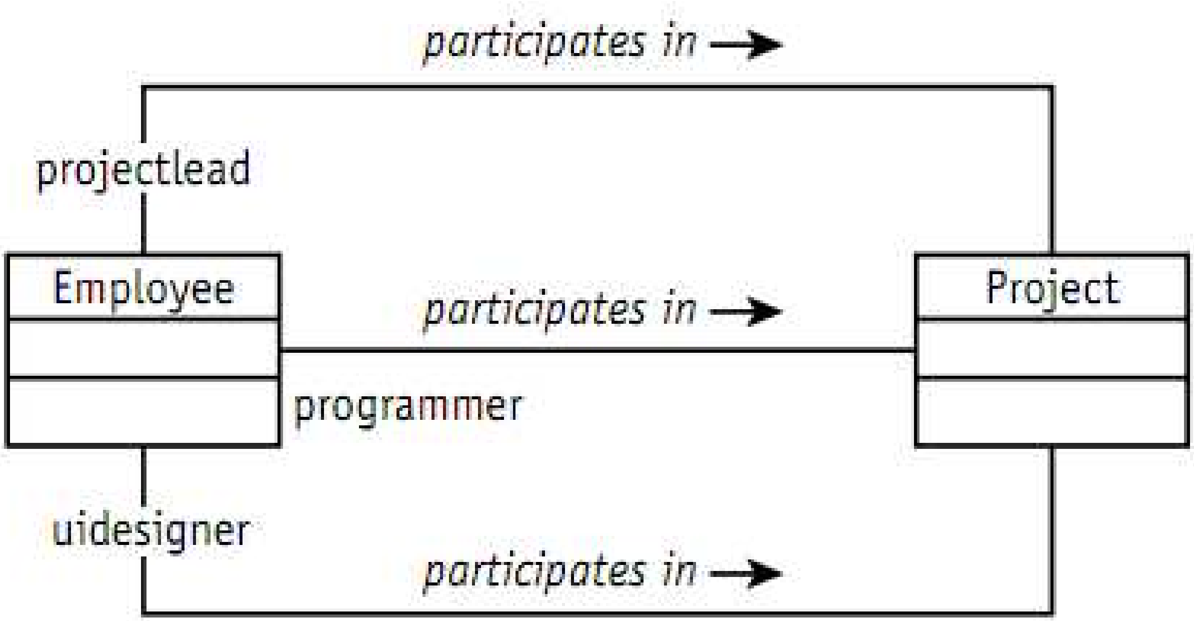


Figure 4-1 Assigning multiplicity to each end of an association

- Each role is placed at the end of the association next to the type of object that plays the role. You may use them on one, both, or neither end of each association.

Associations

>> **Association Constraints :**

- Constraints describes the rule you want to enforce with the associations.
- Constraints fulfill much the same function for associations like the functions performed with the declaration of attributes and operations.

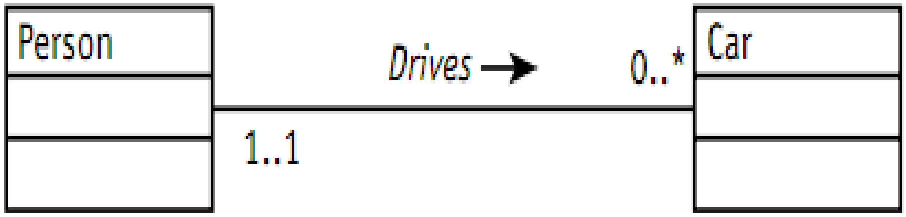


Figure 4-4 An association without constraint

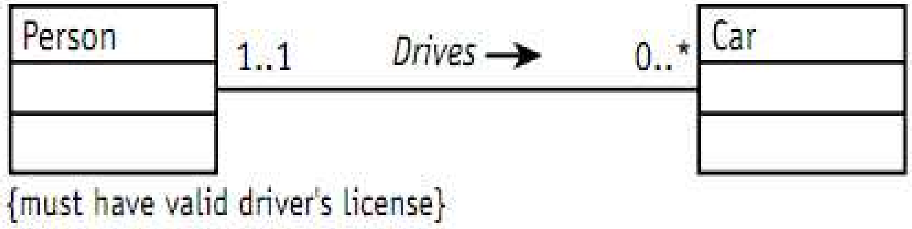


Figure 4-5 An association with a constraint on the Person objects' participation

- Constraints may appear on both ends, either ends, or neither end of the association.

Associations

>> **Association Class :**

- An association class encapsulates information about an association.
- For example, when customers order products there is usually more that you need to know, like when did they order the products? How many did they order? What were the terms of the sale? All the answers to these questions are simply data. All data in an object-oriented system must be contained in (encapsulated in) an object, as shown in Figure 4-6.

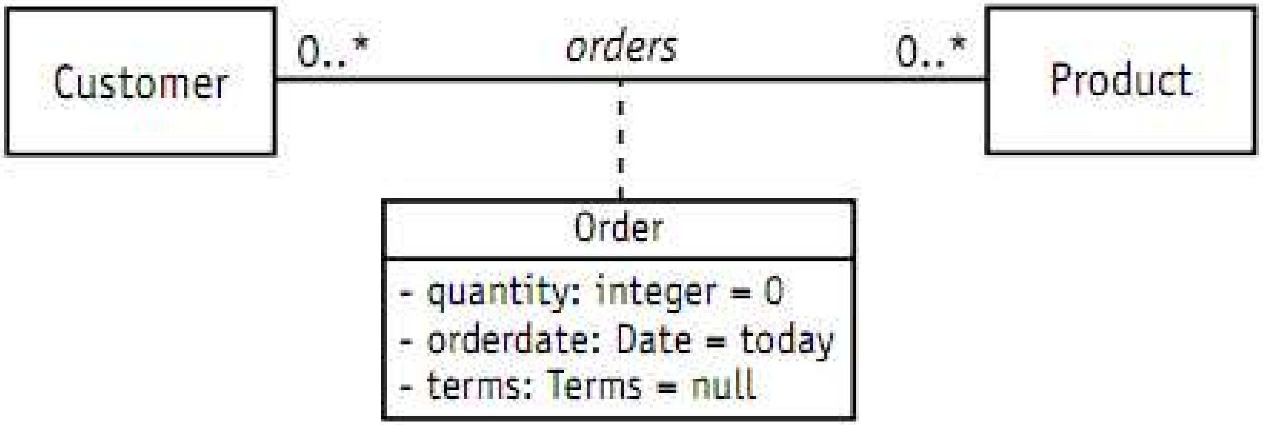


Figure 4-5 Association class notation

- So, attach the new class to the association with a dashed line

Associations

>> **<u>Reflexive Association:</u>**

- Reflexive association is a fancy expression that says objects in the same class can be related to one another.
- Booth ends of the association line point to the same class.
- The association line leaves a class and reflects back onto the same class.

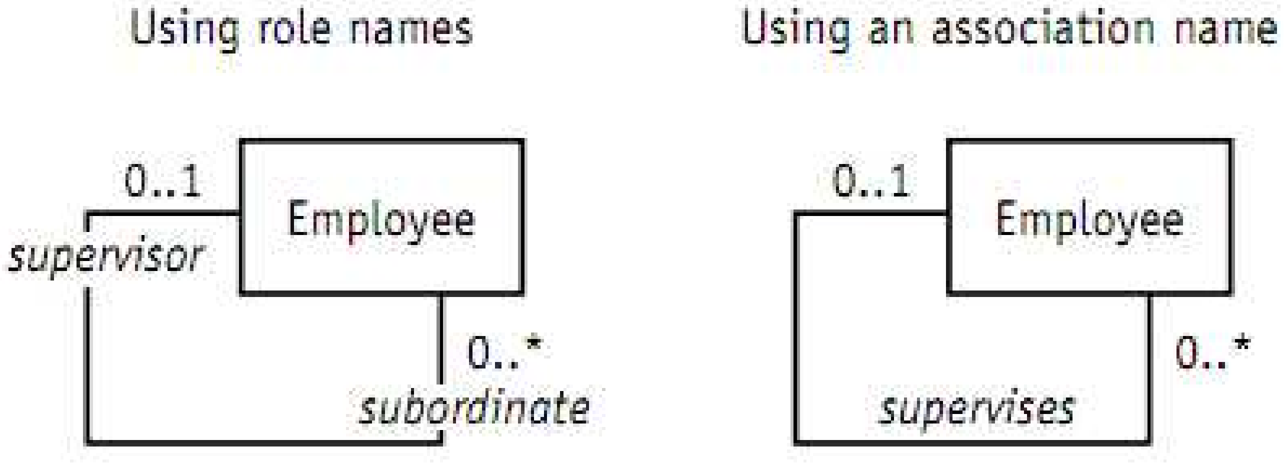


Figure 4-5 Two ways to model a reflexive association

Associations

>> **Qualified Association:**

- Qualified associations provide approximately the same functionality as indexes.
- Typically the qualifier is an attribute of the class on the opposite end of the -association, so make certain that the two names and data types agree.
- Use qualifiers to reduce the multiplicity.

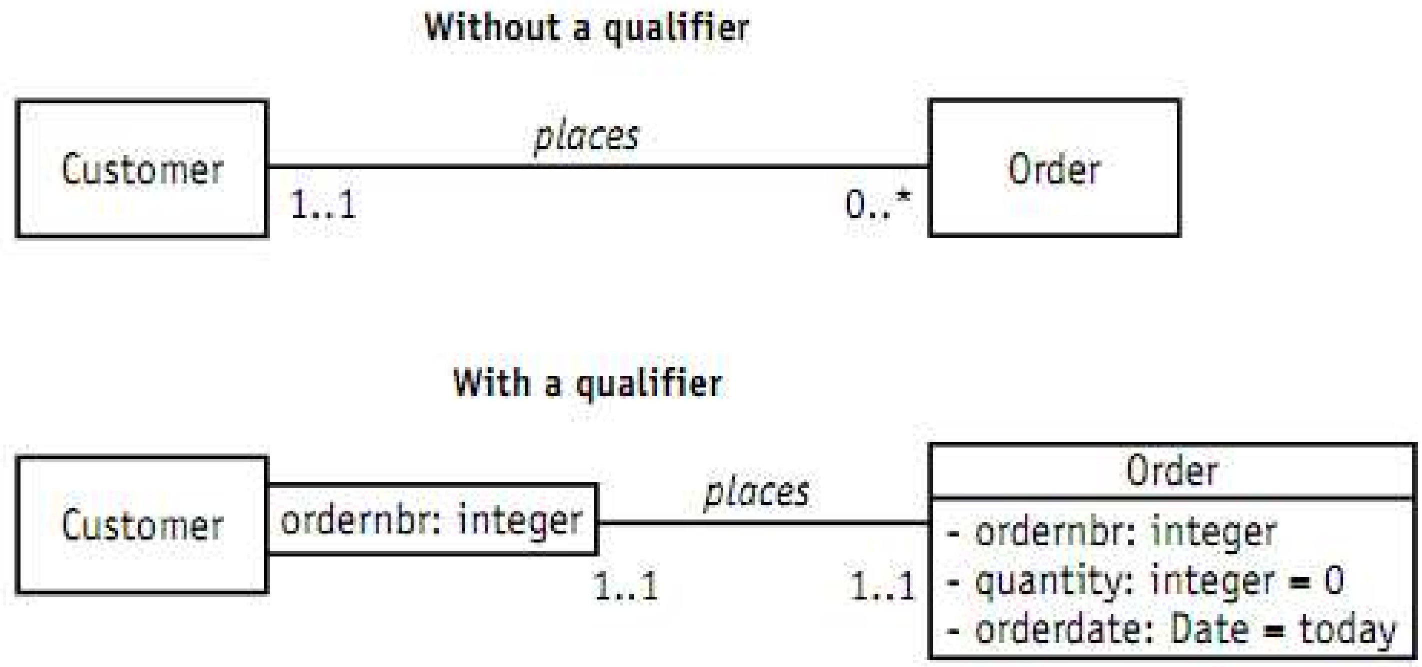
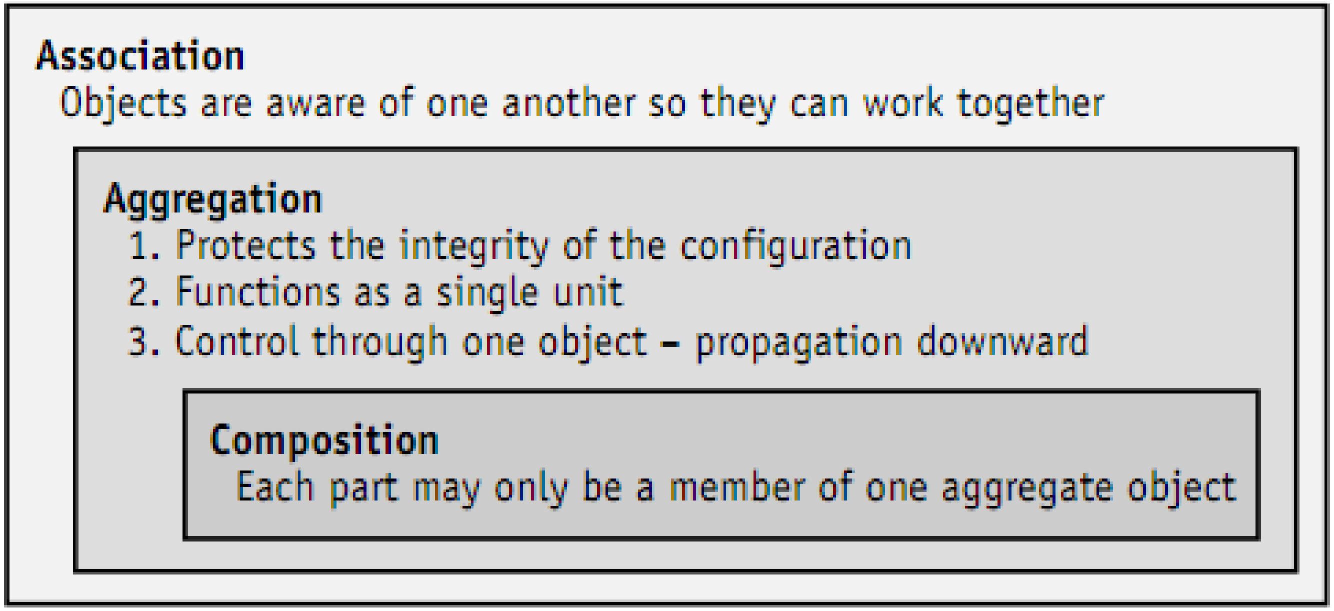


Figure 4-5 Qualified association

Aggregation & Composition

>> Every **aggregation** relationship is a type of **association**. So every **aggregation** relationship has all the properties of an **association** relationship, plus some rules of its own.

>> Every **composition** relationship is a form of **aggregation**. So every **composition** relationship has all the properties of an **aggregation**, plus some rules of its own.



Association
Objects are aware of one another so they can work together

Aggregation
1. Protects the integrity of the configuration
2. Functions as a single unit
3. Control through one object – propagation downward

Composition
Each part may only be a member of one aggregate object

Figure 4-9 The relationship between association, aggregation, and composition

Aggregation

>> **Aggregation** is a special type of association used to indicate that the participating objects are assembled or configured together to create a new, more complex object.

>> For example, a number of different parts are assembled to create a car, a boat, or a plane.

>> Draw a diamond on the end of the association that is attached to the assembly or aggregate class.

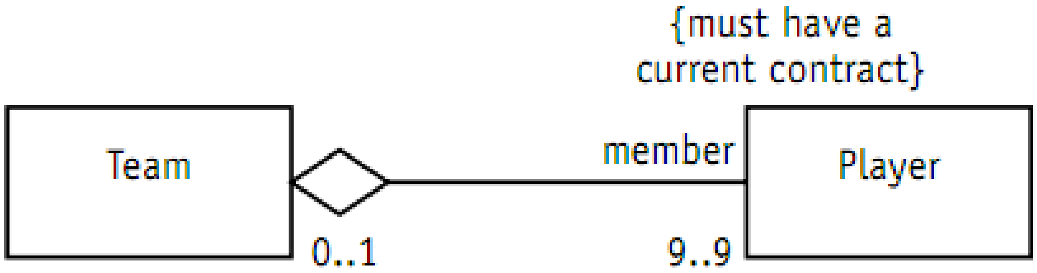


Figure 4-9 How to represent an aggregation relationship in the UML

>> Assign the appropriate multiplicities to each end of the association, and add any roles and/or constraints that may be needed to define the rules for the relationship.

Composition

>> **Composition** is used for aggregations where the life span of the part depends on the life span of the aggregate. The aggregate has control over the creation and destruction of the part. In other words, the member object cannot exist apart from the aggregation.

>> Draw this stronger form of aggregation simply by making the aggregation diamond solid (black).

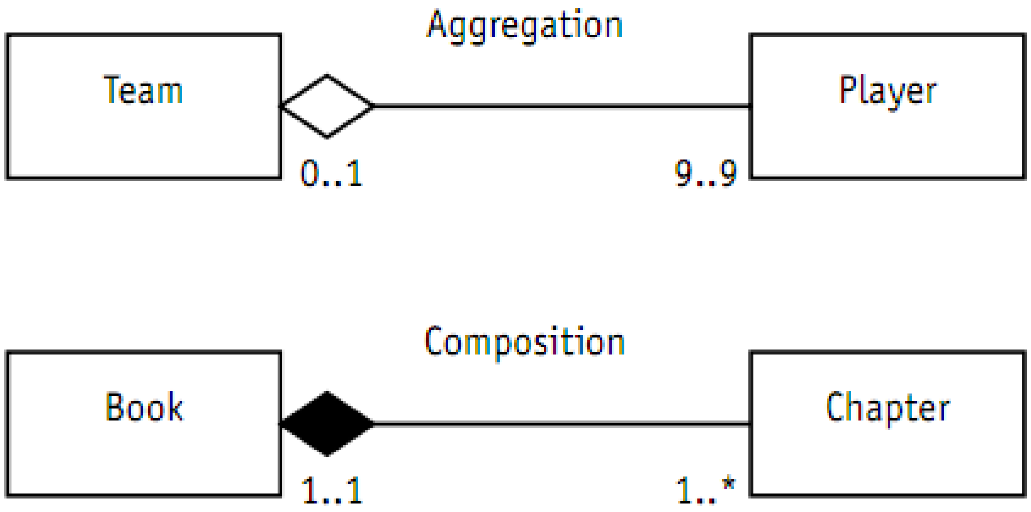


Figure 4-9 How to represent a composition relationship in the UML

>> Multiplicity provides some clues on the distinction between aggregation and composition

Aggregation and Composition

Problem statement:

“Our Company maintains a group of race cars. Our cars use some of our new 8-cylinder engines and new transmissions. Once the engines are assembled, the pistons, carburetor, and plugs cannot be swapped between engines due to changes caused by the high temperatures. We want to keep records of the performance achieved by each engine in each car and each transmission in combination with each engine. Our drivers evaluate each car to give us their

assessment of the handling. We need a system to track the configurations and the drivers’ assessments.”

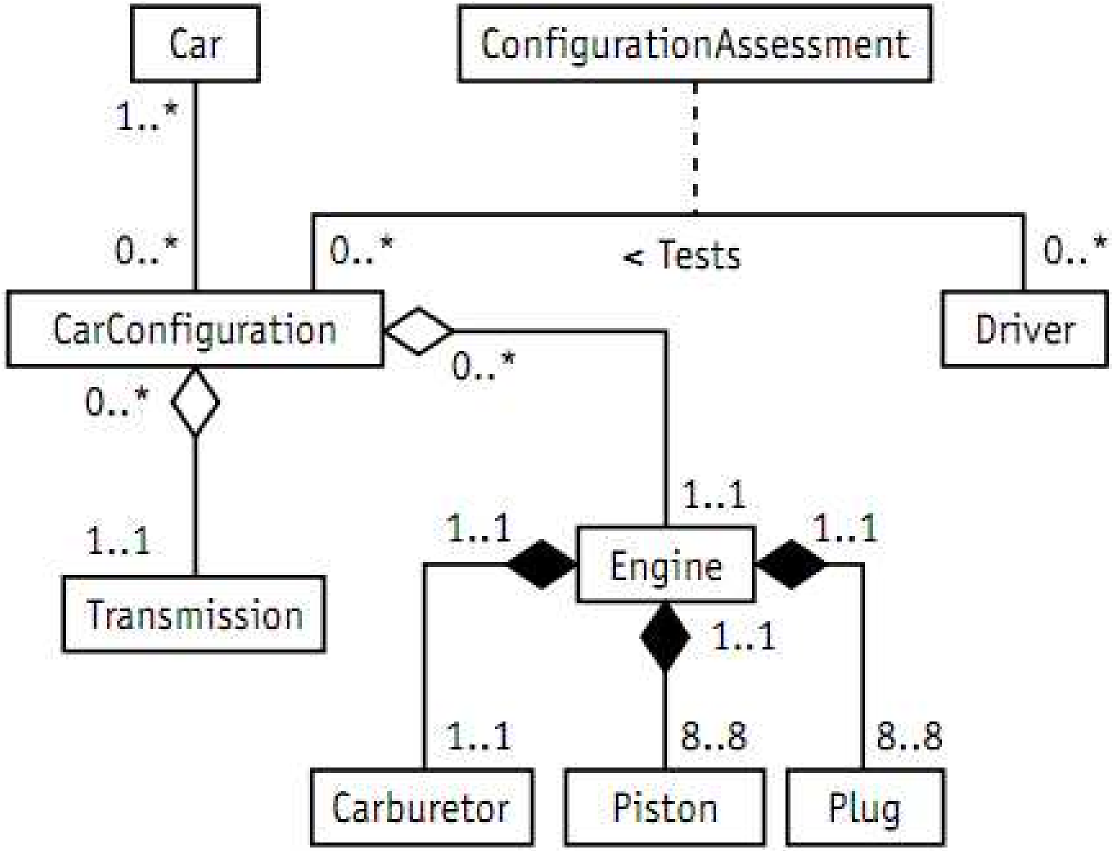


Figure 4-9 Using aggregation and composition together to model race car performance

Generalization/Specialization

>> Generalization is the process of organizing the properties of a set of objects that share the same purpose.

>> This relationship is also called inheritance.

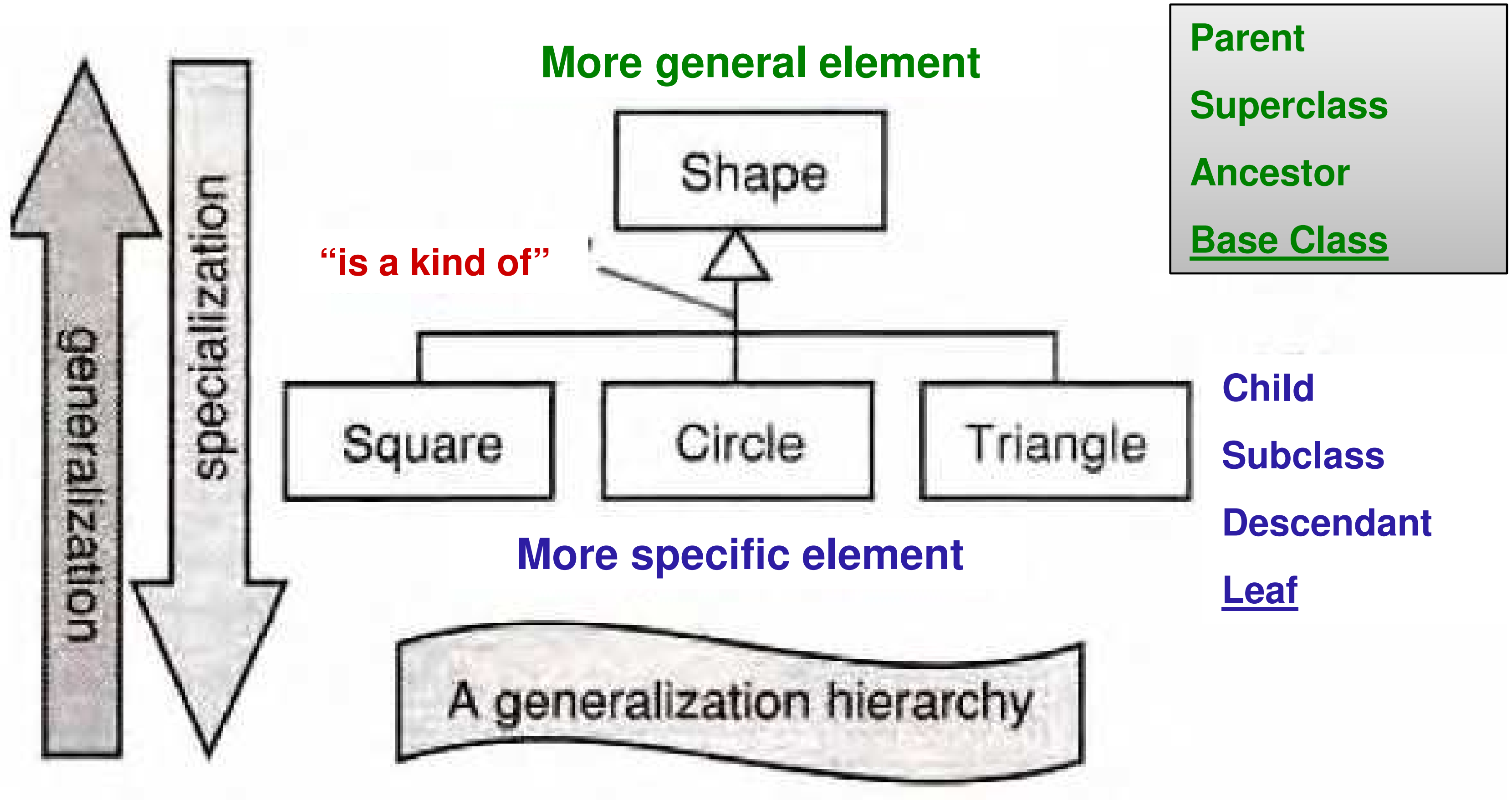
>> A generalization relationship between classes indicates that the child class inherit the properties of the parent class.

>> For example, If an apple is a kind of fruit, then it inherits all the properties of fruit. Likewise, an apple is a specialization of fruit because it inherits all the generalized properties of fruit and adds some unique properties that only apply to apples.

>> A generalization is not an association. **Associations** define the rules for how objects may relate to one another. **Generalization** relates classes together where each class contains a subset of the elements needed to define a type of object.

Generalization/Specialization

>> Generalization hierarchies may be created by generalizing from specific things or by specializing from general things.



More general element

"is a kind of"

Parent

Superclass

Ancestor

Base Class

Child

Subclass

Descendant

Leaf

More specific element

Figure 3-10 Modeling generalization

Generalization/Specialization

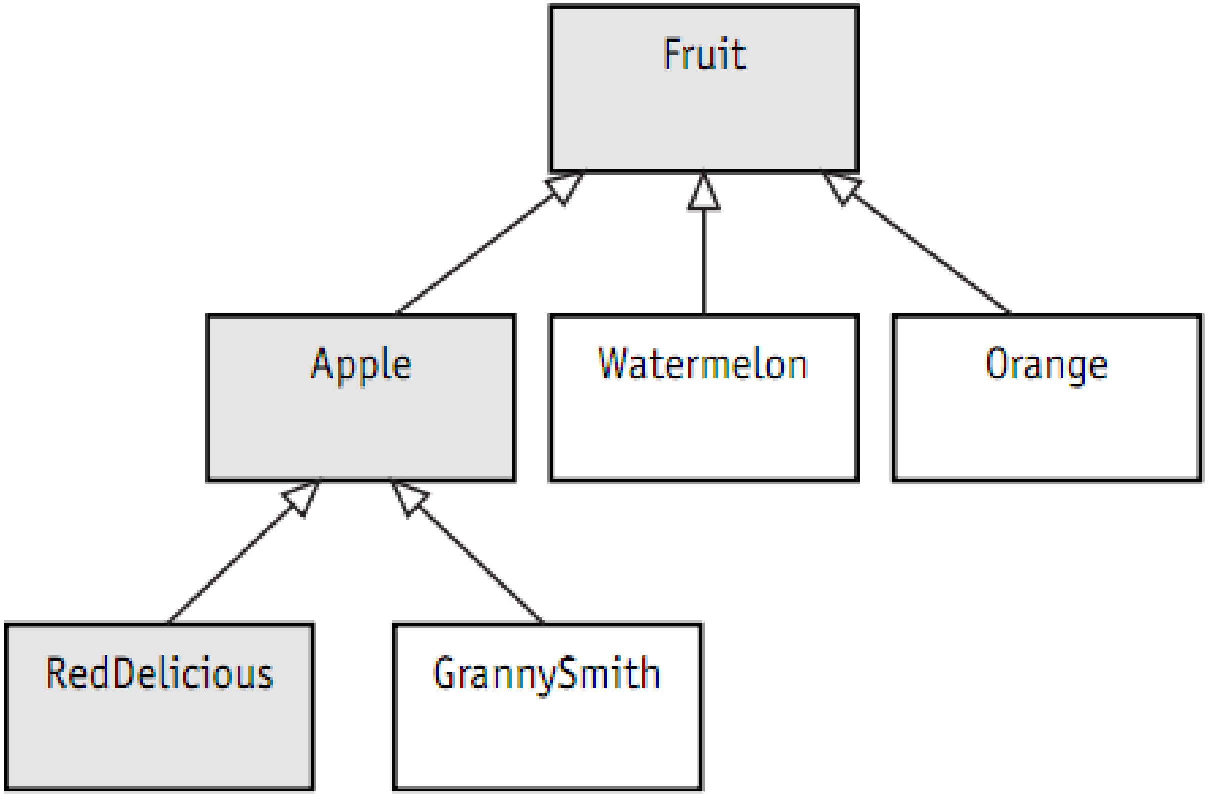


Figure 3-10 Modeling generalization

Generalization/Specialization

>> Because the **generalization** (also called an **inheritance**) relationship is not a form of association, there is no need for multiplicity, roles, and constraints. These elements are simply irrelevant.

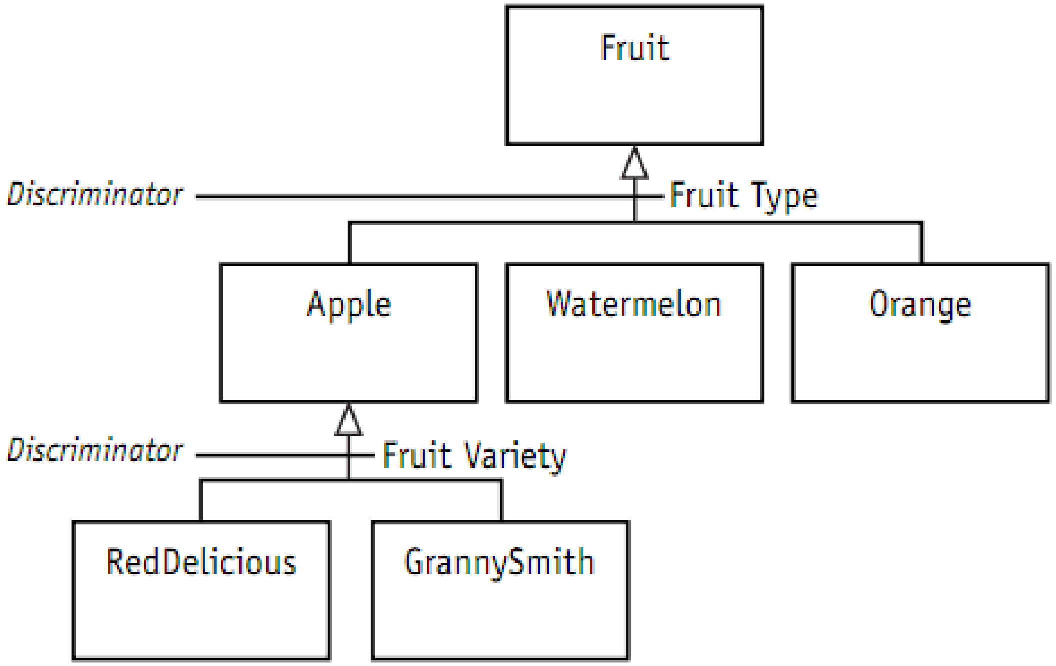


Figure 4-6 Modeling generalization with discriminators

>> A discriminator is an attribute or rule that describes how I choose to identify the set of subclasses for a superclass.

Examples

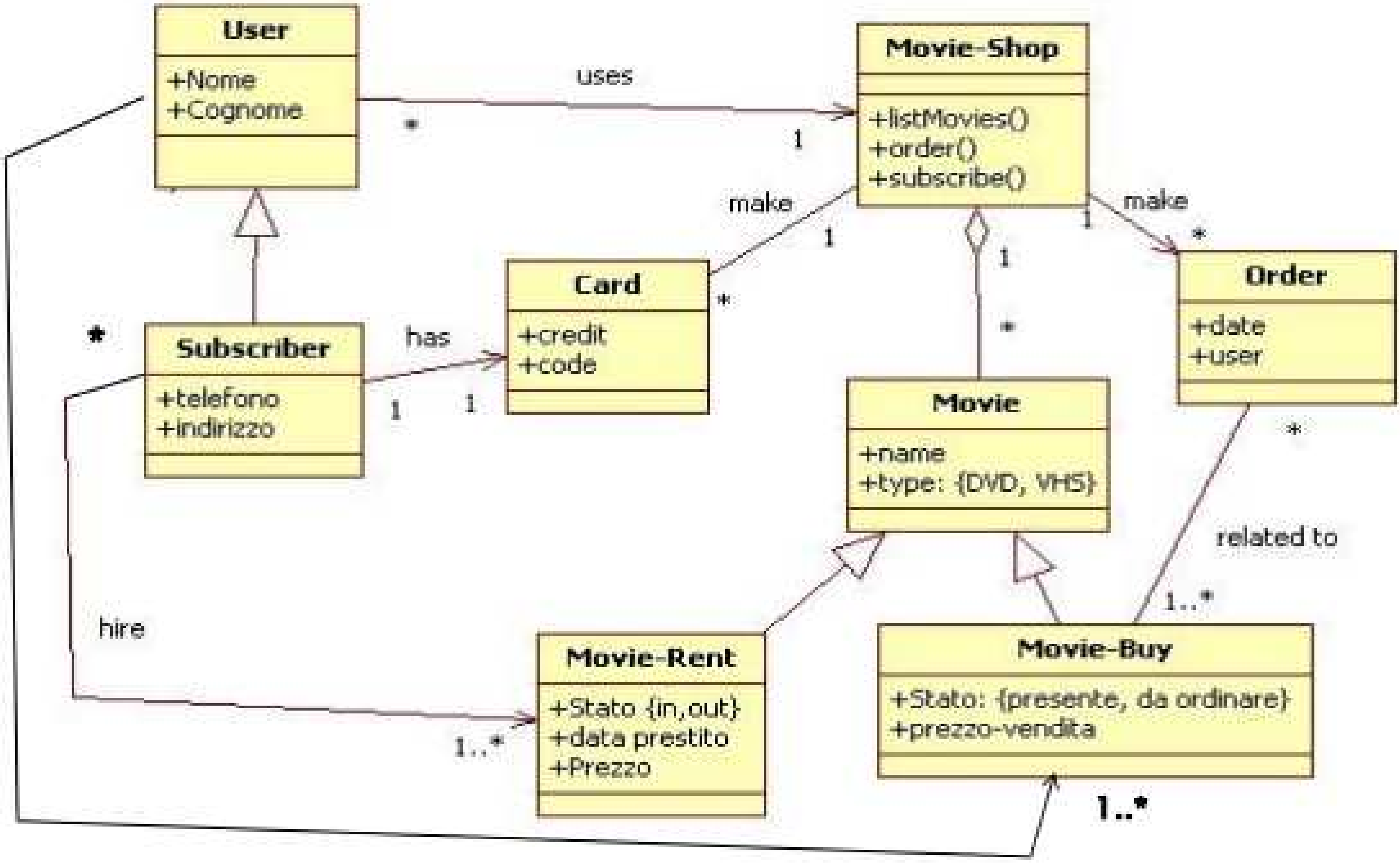
Case 1:

Design a system for a **movie-shop**, in order to handle ordering of movies and browsing of the catalogue of the store, and user subscriptions with rechargeable cards.

Only subscribers are allowed hiring movies with their own card. Credit is updated on the card during rent operations.

Both users and subscribers can buy a movie and their data are saved in the related order. When a movie is not available it is ordered .

Examples



Case 1: Movie-Shop

Examples

Case 2:

We want to model a system for management of **flights** and pilots.

An airline operates flights. Each airline has an ID and name.

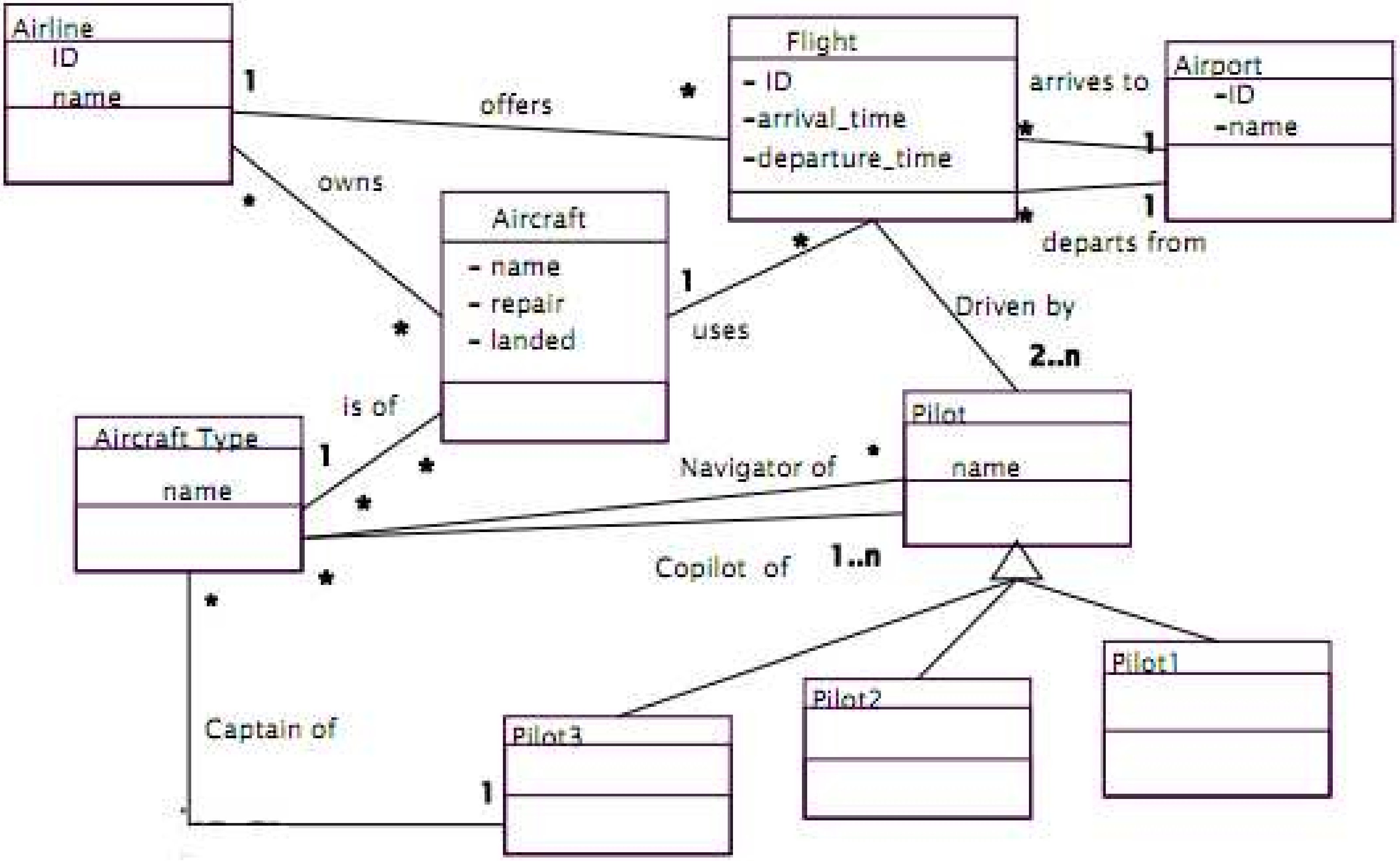
Each flight has an ID and it departs from a departure airport and arrives to an arrival airport. Each flight has a pilot and a co-pilot, and it uses an aircraft of a certain type; a flight has also a departure time and an arrival time.

An airline owns a set of aircrafts of different types. An aircraft can be in a working state or it can be under repair. In a particular moment an aircraft can be landed or airborne.

A company has a set of pilots: each pilot has an experience level: 1 is minimum, 3 is maximum.

A type of aero-plane may need a particular number of pilots, with a different role (e.g.: captain, co-pilot, navigator): there must be at least one captain and one co-pilot, and a captain must have a level 3.

Examples



Case 2: Flights

Examples

Case 3:

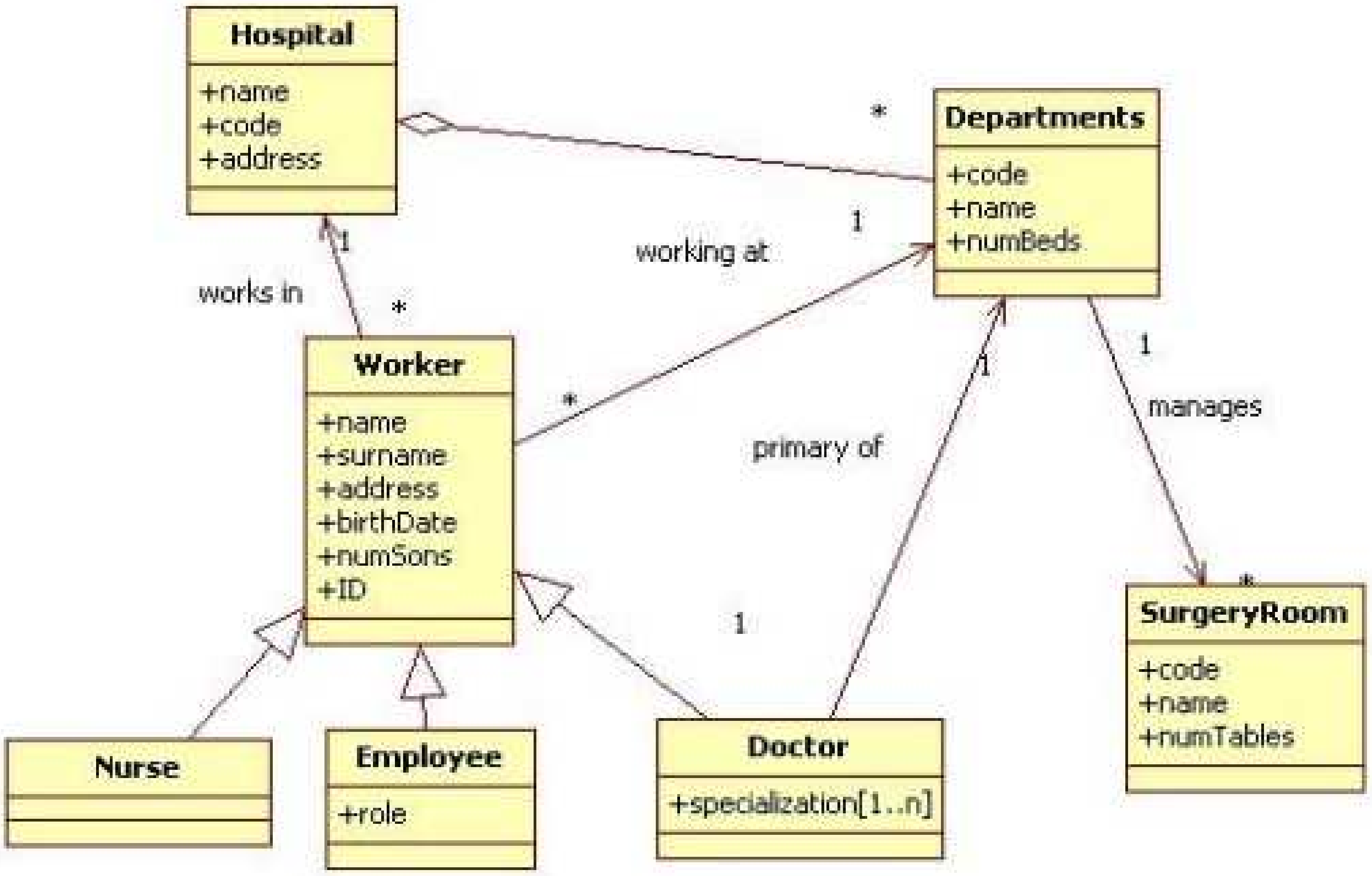
Hospitals have an ID, a name and an address.

People working in a hospital have an ID , unique in the hospital, surname, name, birth date, address, and number of sons.

Hospital workers cab be doctors (having a list of specializations), employees (having a role) and nurses.

Hospital is divided in departments, having a code, a name, and a number of beds. For each department we know the primary doctor and all the workers. Each department can manage surgery rooms identified by name, ID and number of surgery tables.

Examples



Case 3: Hospitals

Examples

Case 4:

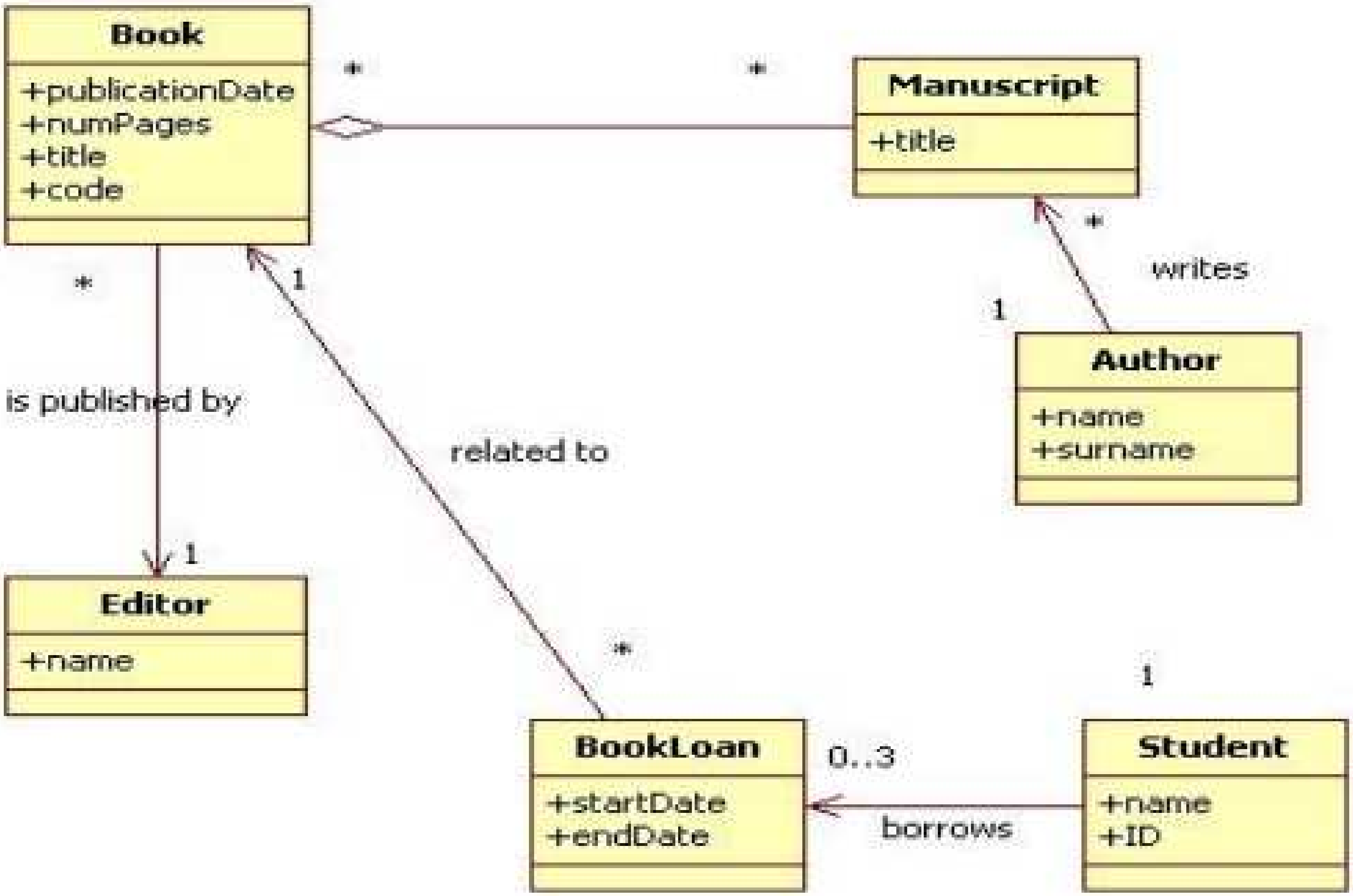
Library contains books borrowed by students identified by their ID. Books have a library code, title, one or more authors and number of pages.

Authors have name and surname, and they can have written many manuscripts published in different books.

Each book has an editor and a book may be provided by different editors in different dates.

Students cannot borrow more than 3 books, and each book has a starting date and a return date.

Examples



Case 4: Library

Exercises

Case 1:

One Student may attend any number of courses.
One course may have any number of students.
Instructors teach courses.
For every course there is at least one instructor.
Every instructor may teach zero or more courses.
A school has zero or more students.
Each student may be a registered member of one or more school.
A school has one or more departments.
Each department belongs to exactly one school.
Every instructor is assigned to one or more departments.
Each department has one or more instructors.
For every department there is exactly one instructor acting as the department chair.

Exercises

Case 2:

A building is owned by one person.
A person may own more than one building.
Each building is either a house or an apartment.
An apartment contains two or more Rental Units.
It is possible to buy a house, and rent or vacate a Rental Unit.

Exercises

<u>Case 3:</u>

A company has one or more divisions.
A division has one or more departments.
A company can have one or more persons working as employees.
A person is associated with the company as an employee.
A department can have one or more persons assigned to it.
A person is assigned to one department.

Exercises

Case 3:

A company has one or more divisions.
A division has one or more departments.
A company can have one or more persons working as employees.
A person is associated with the company as an employee.
A department can have one or more persons assigned to it.
A person is assigned to one department.

Exercises

Case 4:

A member can be of three types- potential member, past member and club member. Club members place orders. One order must include at least one product. One product can be in many orders. There are two types of products - club souvenirs and club facilities. Each member must sign an agreement. One agreement can be made for more than one member. Club facilities are courts, tables and halls. Courts are Badminton court and Tennis Court. Tables are for Tennis and Pool. Club members can also book club facilities. For each booking, date and duration of booking must be stored.

References

→ **Chapter 8 & 9**
The Unified Modeling Language User Guide
SECOND EDITION
By Grady Booch, James Rumbaugh, Ivar Jacobson

→ **Session 9, 10 & 11**
UML Weekend Crash Course
Thomas A. Pender