UML - Modelling Data

Knut Hinkelmann

Prof. Dr. Knut Hinkelmann
MSc Business Information Systems

References

- OMG Unified Modelling Language - UML, Current Standard Version 2.1.2 http://www.omg.org/spec/UML/2.1.2/
- R. Miller: Practical UML: A Hands-On Introduction for Developers. http://edn.embarcadero.com/article/31863
- Donald Bell: UML basics: The class diagram. http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

Unified Modeling Language UML

- Unified Modeling Language (UML) is a set of standardized modeling languages in the field of software engineering.
- UML includes a set of graphic notation techniques (diagrams) to create visual models of software-intensive systems, including their structure and design
- In UML, you can model
  - any type of application,
  - running on any type and combination of hardware, operating system, programming language, and network
- The UML standard is developed and managed by the Object Management Group OMG and forms a foundation of OMG's Model Driven Architecture (MDA)
  - a UML model can be either platform-independent or platform-specific,
- Using XMI (XML Metadata Interchange, another OMG standard), it is possible to transfer a UML model
  - from one tool into a repository, or
  - into another tool for refinement or the next step in your chosen development process.

Source: Introduction to OMG's Unified Modeling Language™ (UML®),
http://www.omg.org/gettingstarted/what_is_uml.htm

Types of UML Diagrams

UML contains diagrams for modelling structure (data and IT) and behavior of software systems

Structure diagrams

Data

1. Class diagram
2. Object diagram

IT systems

3. Component diagram
4. Deployment diagram
5. Composite structure diagram (*)
6. Package diagram

Behavior diagrams

7. Use-case diagram
8. State machine diagram
9. Activity diagram

Interaction diagrams

10. Sequence diagram
11. Communication diagram
12. Interaction overview diagram (*)
13. Timing diagram (*)

(*) not existing in UML 1.x, added in UML 2.0

Object Orientation

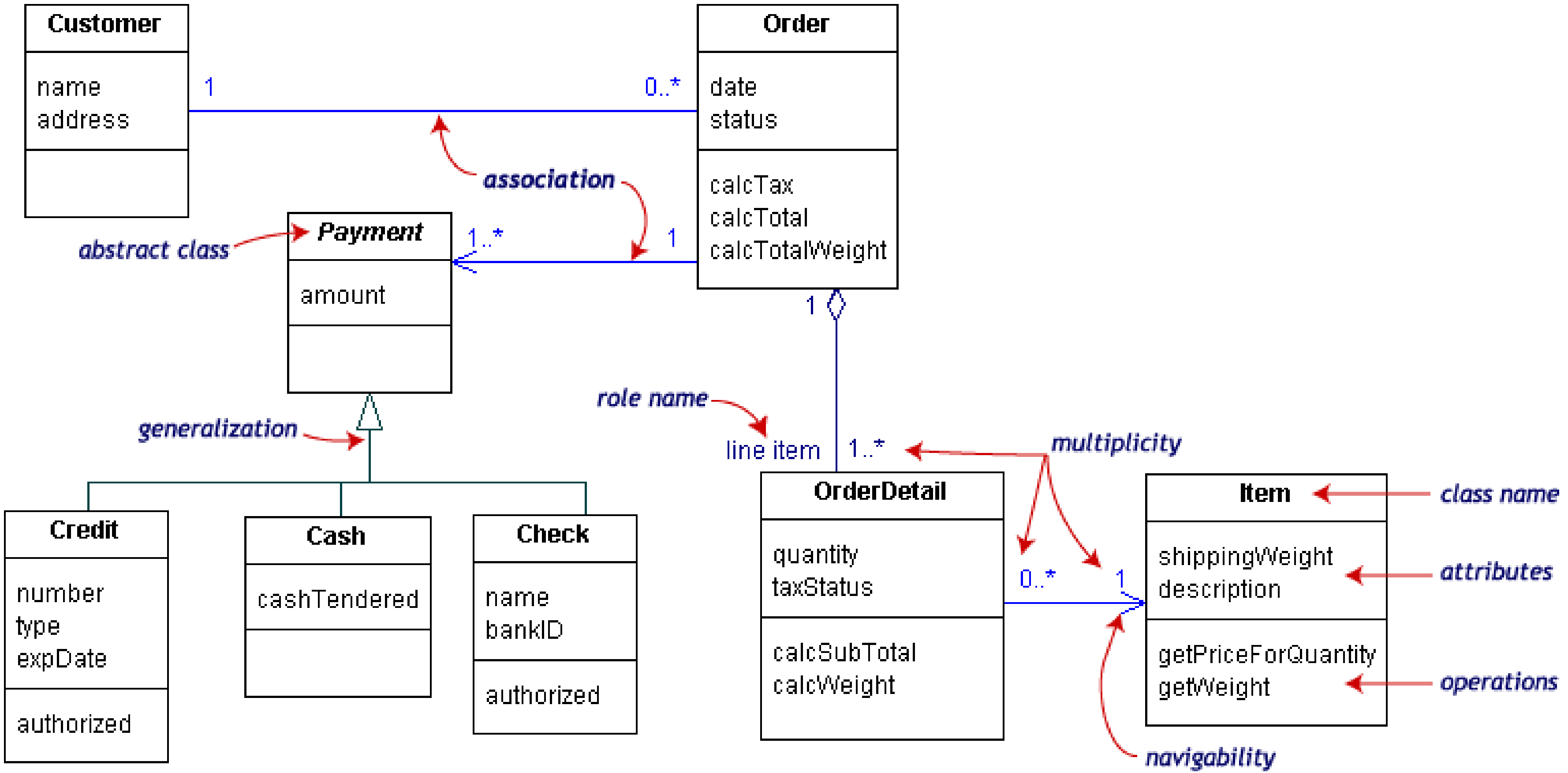
- In the first versions, UML was described as addressing the needs of modeling systems in an object-oriented manner
- Object orientation still is the inspiration for some key concepts
- Main concepts:
  - Object – individual unit capable of *receiving/sending messages*, processing data
  - Class – pattern giving an abstraction for a set of objects
  - Inheritance – technique for reusability and extendibility

Class Diagrams

- **A Class diagram** gives an overview of a system by showing its classes and the relationships among them.
- Class diagrams are static -- they display what interacts but not what happens when they do interact.
- Main concepts involved
  - Class - Object
  - Inheritance
  - (various kinds of) Associations

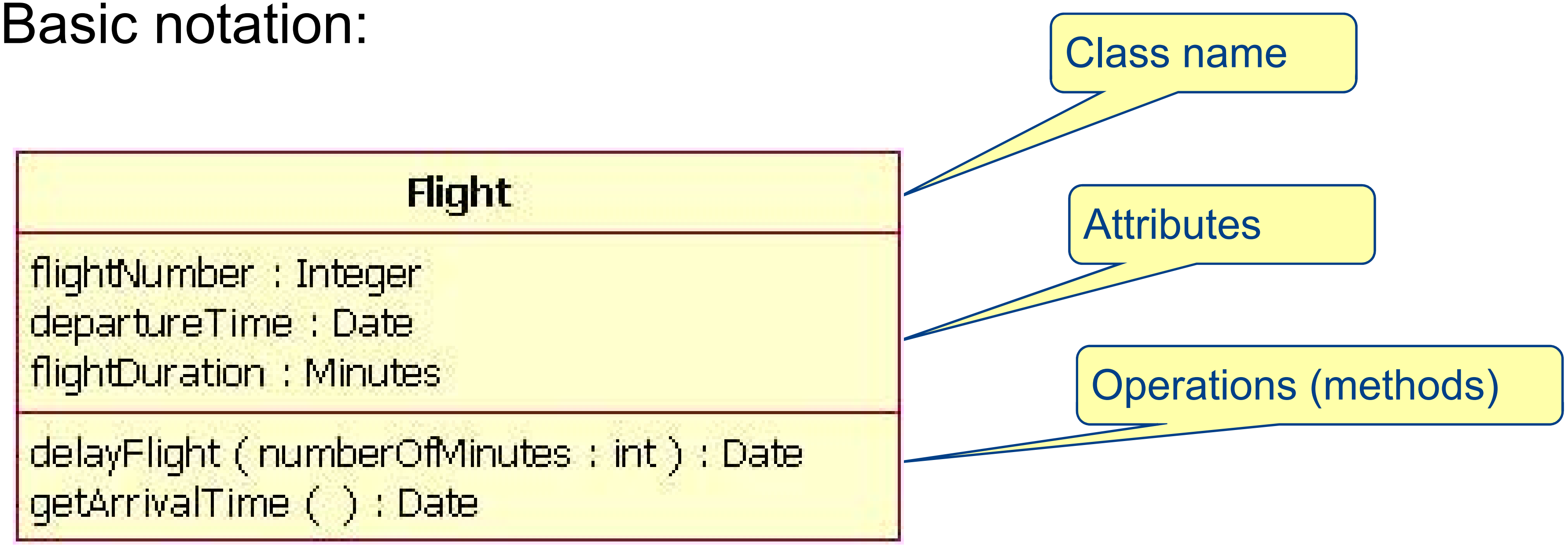
Class Diagram Example

The class diagram below models a customer order from a retail catalog. The central class is the **Order**. Associated with it are the **Customer** making the purchase and the **Payment**. A **Payment** is one of three kinds: **Cash**, **Check**, or **Credit**. The order contains **OrderDetails** (line items), each with its associated **Item**.



UML Class

- Gives the type of a set of objects existing at run-time
- Declares a collection of methods and attributes that describe the structure and behavior of its objects
- Basic notation:



Class name

Attributes

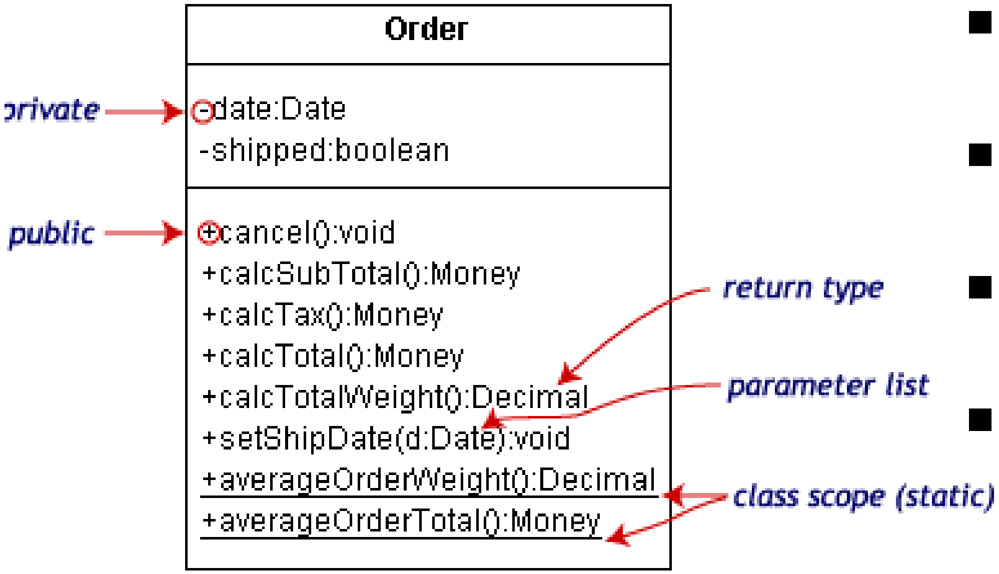
Operations (methods)

Flight

flightNumber : Integer
departureTime : Date
flightDuration : Minutes

delayFlight (numberOfMinutes : int) : Date
getArrivalTime ( ) : Date

Class Information



Access specifiers:

Symbol	Access
+	public: they are visible to all
–	private: not visible to callers outside the class
#	protected: only visible to children of the class

- UML class notation is a rectangle divided into three parts: class name, attributes, and operations.
- Names of abstract classes, such as ***Payment***, are in italics.
- Relationships between classes are the connecting links.
- Attributes and operations can be labeled according to access and scope.
- The illustration uses the following UML™ conventions.
  - Static members are underlined. Instance members are not.
  - The operations follow this form: *<access specifier> <name>* (*<parameter list>*) : *<return type>*
  - The parameter list shows each parameter type preceded by a colon.
  - Access specifiers appear in front of each member.

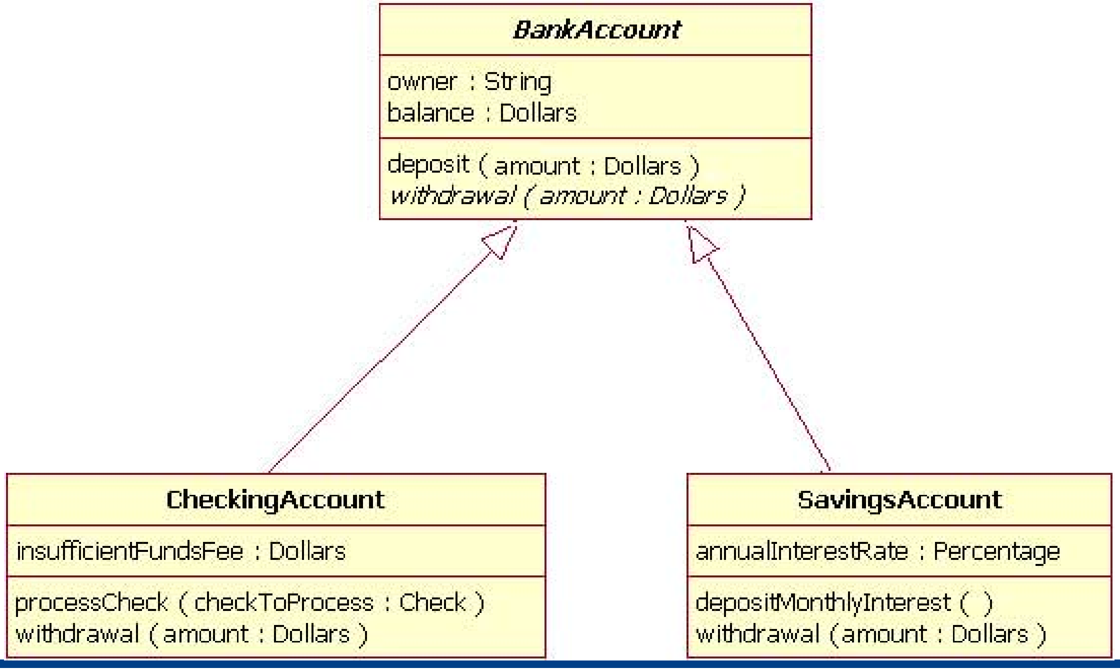
Class Diagram Elements

- **Generalization** -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass.
  - ***Payment*** is a superclass of **Cash**, **Check**, and **Credit**.
- **Association** -- a relationship between instances of the two classes. In a diagram, an association is a link connecting two classes.
- **Aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole.
  - **Order** has a collection of **OrderDetails**.
- An end of an assiciation may have a **role name** to clarify the nature of the association.
  - **OrderDetail** is a line item of each **Order**
- A **navigability** arrow on an association shows which direction the association can be traversed or queried. The arrow also indicates who "owns" the association's implementation
  - **OrderDetail** has an **Item**..
  - An **OrderDetail** can be queried about its **Item**, but not the other way around

Associations with no navigability arrows are bi-directional

Generalization - Inheritance

- Inheritance is a very important concept in object-oriented design
- *Inheritance* refers to the ability of one class (child class) to *inherit* the identical functionality of another class (super class), and then add new functionality of its own.
- Inheritance is modeled with the Generalization line from the child class to the super class.


BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)

In this example, the classes CheckingAccount and SavingsAccount inherit from the BankAccount.

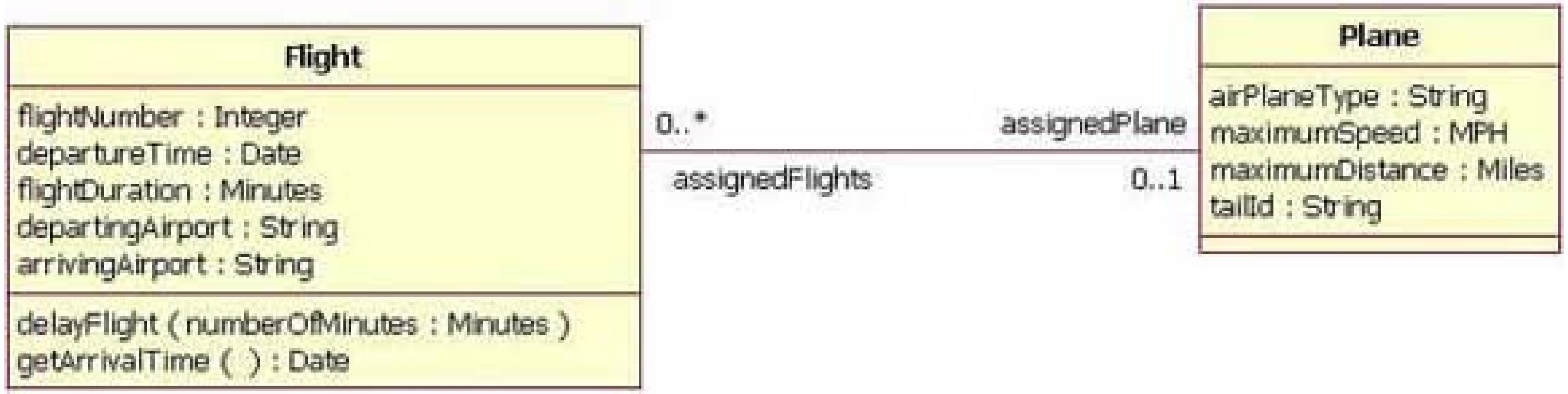
In addition to the attributes and operations explicitly mentioned, the classes CheckingAccount and SavingsAccount also have the attributes *owner* and *balance* as well as the operations *deposit()* and *withdrawal()*.

Associations

- Associations define relationships between objects
- There are five kinds of associations:
  - *Standard associations* which can be
    - bi-directional
    - uni-directional
  - *Association classes* define valuable information for associations
  - *Aggregation* is a special type of association used to model a "whole to its parts" relationship, distinguishing
    - basic aggregration
    - composition aggregration
- Associations are always assumed to be bi-directional unless you qualify the association as some other type

Bi-directional Associations

- For bi-directional associations - indicated by a solid line between two classes - both classes are aware of each other and their relationship
- At either end of the line, you place a role name and a multiplicity value.
- This example shows that a Flight is associated with a specific Plane and a Flight. The Plane takes on the role of "assignedPlane" and the Flight the rule of "assignedFlights"



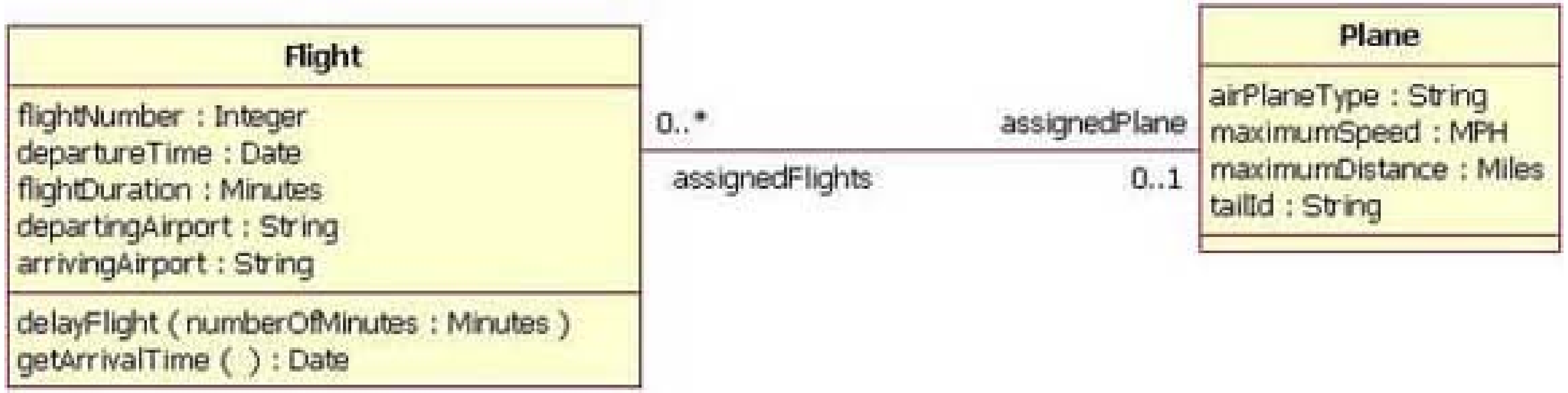
Multiplicity

- The **multiplicity** of an association end is the number of possible instances of the class associated with a single instance of the other end.
- Multiplicities are single numbers or ranges of numbers.
- This table gives the most common multiplicities.

Multiplicities	Meaning
0..1	zero or one instance.
0..* *or* *****	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance
n..m	***n*** to ***m*** instances (n and m stand for numbers, e.g. **0..4**, **3..15**)
n	exactly n instance (where n stands for a number, e.g. **3**)

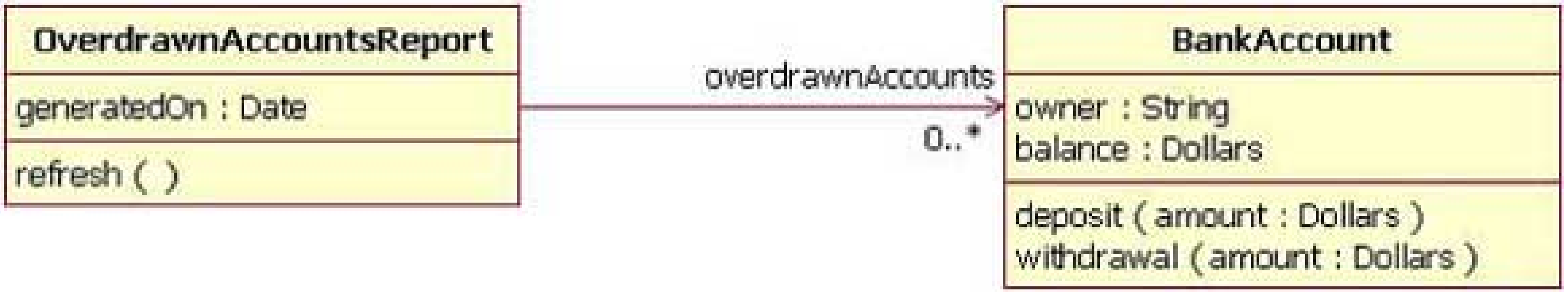
Multiplicity - Example

- The multiplicity value next to the Plane class of 0..1 means that when an instance of a Flight exists, it can either have one instance of a Plane associated with it or no Planes associated with it (i.e., maybe a plane has not yet been assigned).
- The Plane instance can be associated either with no flights or with up to an infinite number of flights.



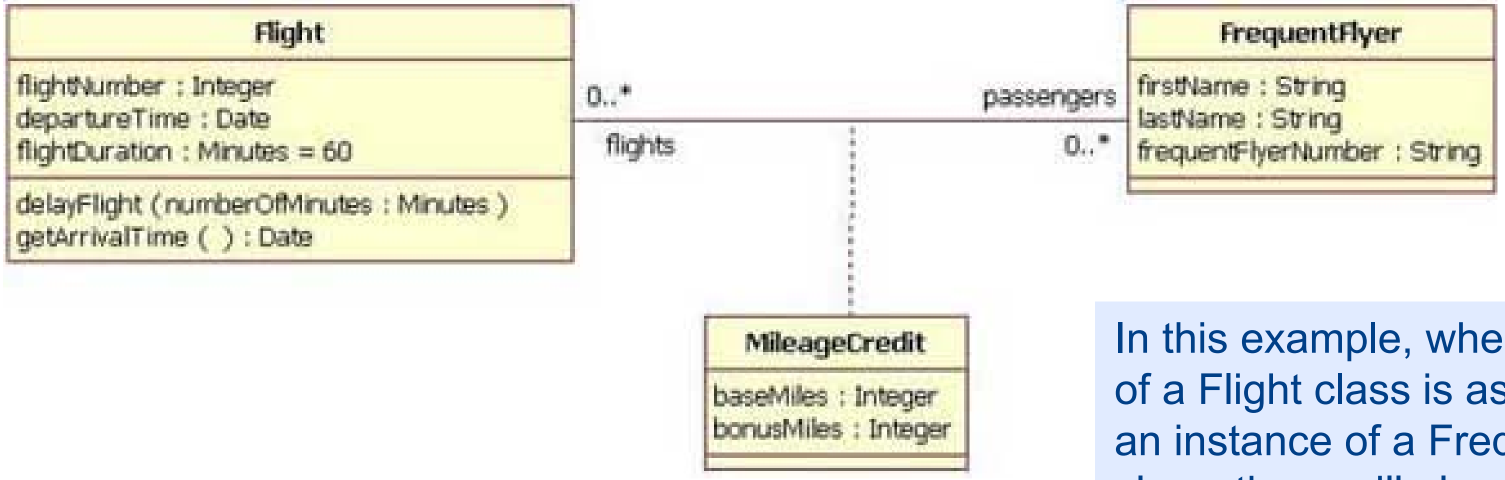
Uni-directional Association

- In a uni-directional association, two classes are related, but only one class "knows" that the relationship exists.
- A uni-directional association is drawn as a solid line with an open arrowhead pointing to the known class. Uni-directional association includes a role name and a multiplicity value, but only for the known class.



Association Classes

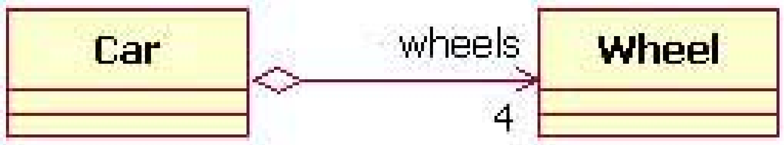
- *Association class* are tied to a primary association. It includes valuable information about the relationship.
- An association class is represented like a normal class, but it is linked to an association line with a dotted line.

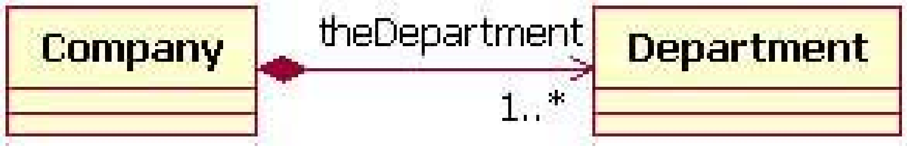


In this example, when an instance of a Flight class is associated with an instance of a FrequentFlyer class, there will also be an instance of a MileageCredit class

Aggregation and Compostion

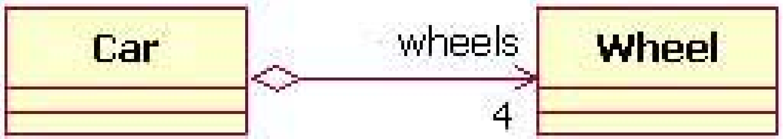
- Aggregation is a special type of association used to model a "whole to its parts" relationship.
  - **In basic aggregation** relationships, the lifecycle of a *part* class is independent from the *whole* class's lifecycle.
    - Aggregation is denoted by a empty diamond at the whole end
  - For a **composition**, the child class's instance lifecycle is dependent on the parent class's instance lifecycle.
    - Composition is denoted by a filled diamond at the whole end



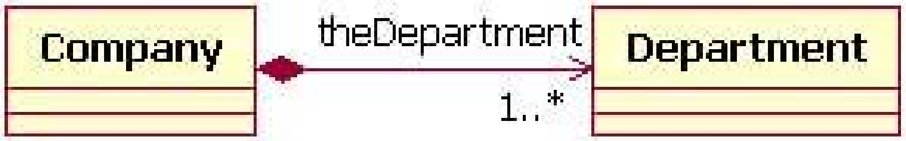


Examples of Aggregation

- Aggregation
  - The Wheel class's instance lives independently of the Car class's instance.
  - The wheel can be created before being placed on a car during assembly.
  - If the Car instance is destroyed the Wheels instance can exist further.

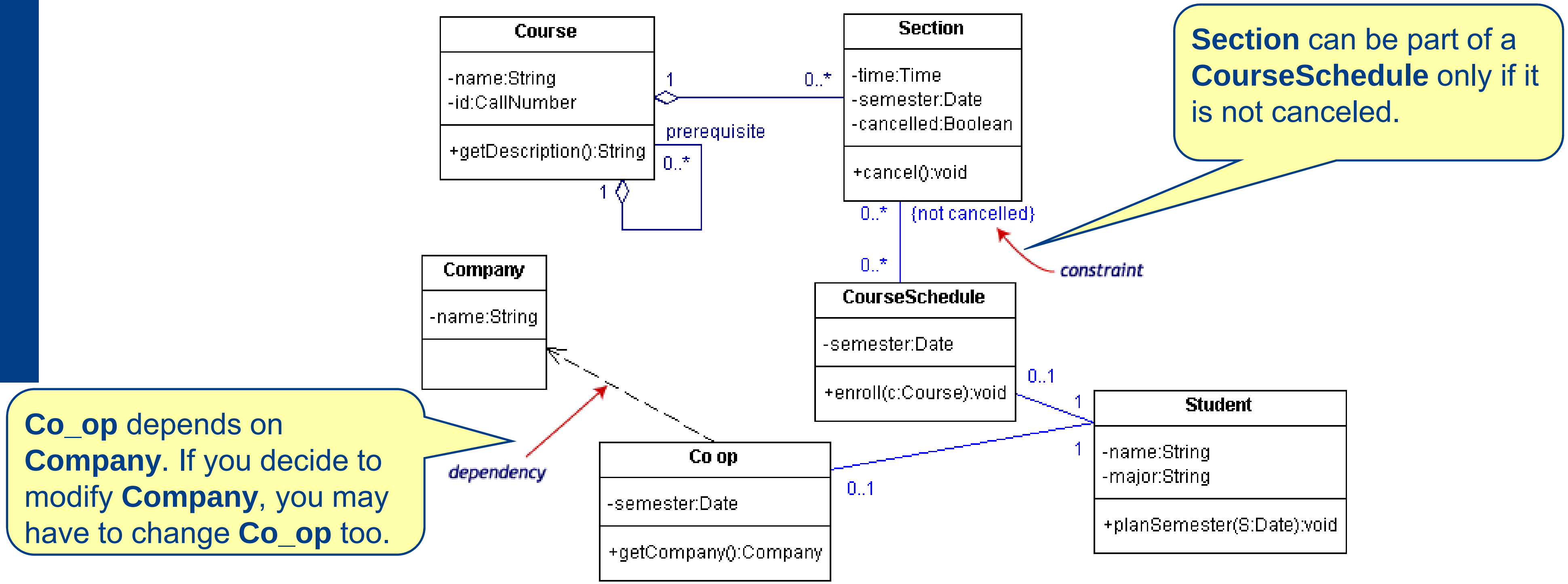


- Composition
  - Company class instance will always have at least one Department class instance.
  - A department cannot exist before a company exists.
  - When the Company instance is removed, the Department instance is automatically removed as well.



Dependencies and Constraints

- A **dependency** is a relation between two classes in which a change in one may force changes in the other. Dependencies are drawn as dotted lines.
- A **constraint** is a condition that every implementation of the design must satisfy. Constraints are written in curly braces { }.

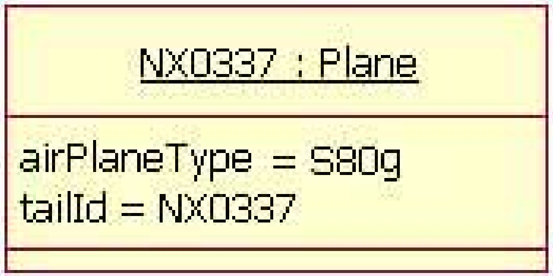


Section can be part of a **CourseSchedule** only if it is not canceled.

Co_op depends on **Company**. If you decide to modify **Company**, you may have to change **Co_op** too.

Instances - UML Objects

- Sometimes it is useful to show example instances of the classes
- The notation of an instance consists of two parts
  - The top compartment having an underlined concatenation of Instance Name and Class Name separated by a colon
  - The lower compartment having some of the attribute names and their values
- Instances can be shown in a Class and Object diagrams

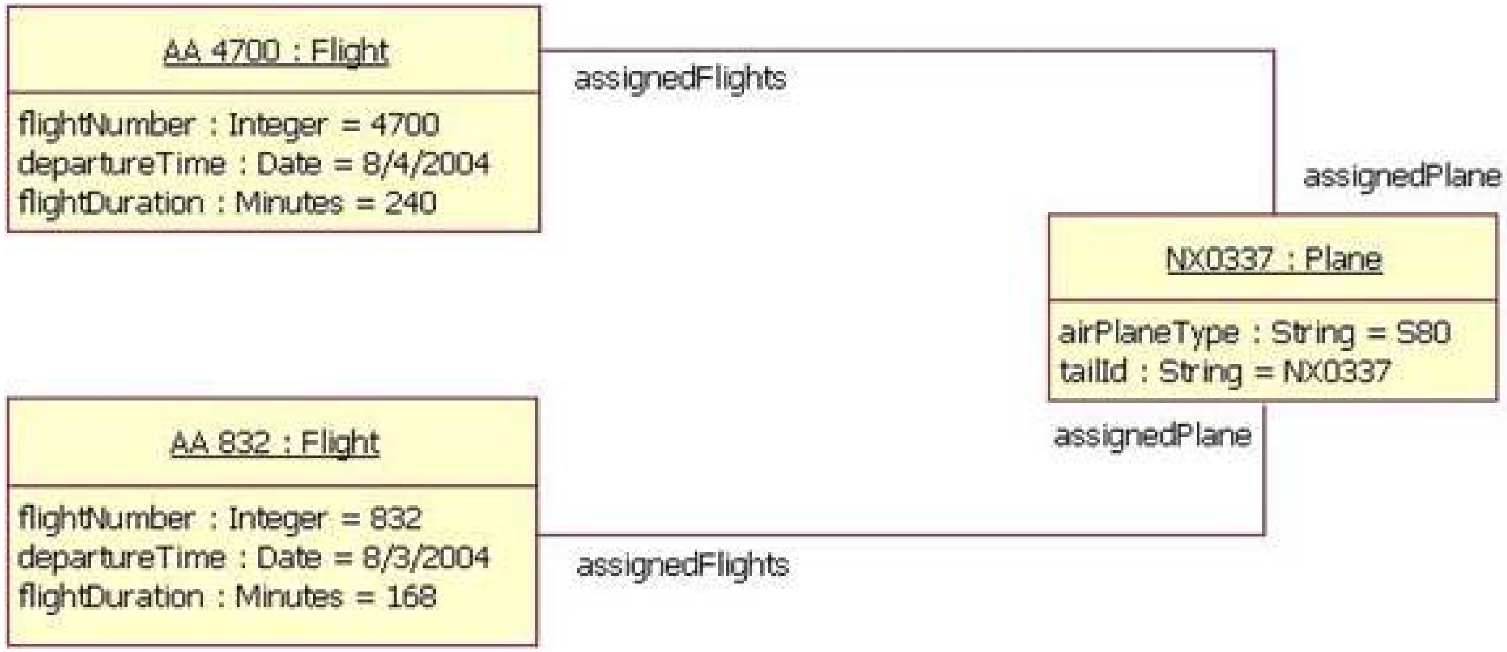


NX0337 : Plane

airPlaneType = S80g
tailId = NX0337

Instances

- Example: Object diagram with class instances and their associations.


AA 4700 : Flight
flightNumber : Integer = 4700
departureTime : Date = 8/4/2004
flightDuration : Minutes = 240
assignedFlights
assignedPlane
NX0337 : Plane
airPlaneType : String = S80
tailId : String = NX0337
assignedPlane
AA 832 : Flight
flightNumber : Integer = 832
departureTime : Date = 8/3/2004
flightDuration : Minutes = 168
assignedFlights