University of Applied Sciences Northwestern Switzerland
School of Business

UML – Unified Modelling Language

Sources:

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://dn.codegear.com/article/31863

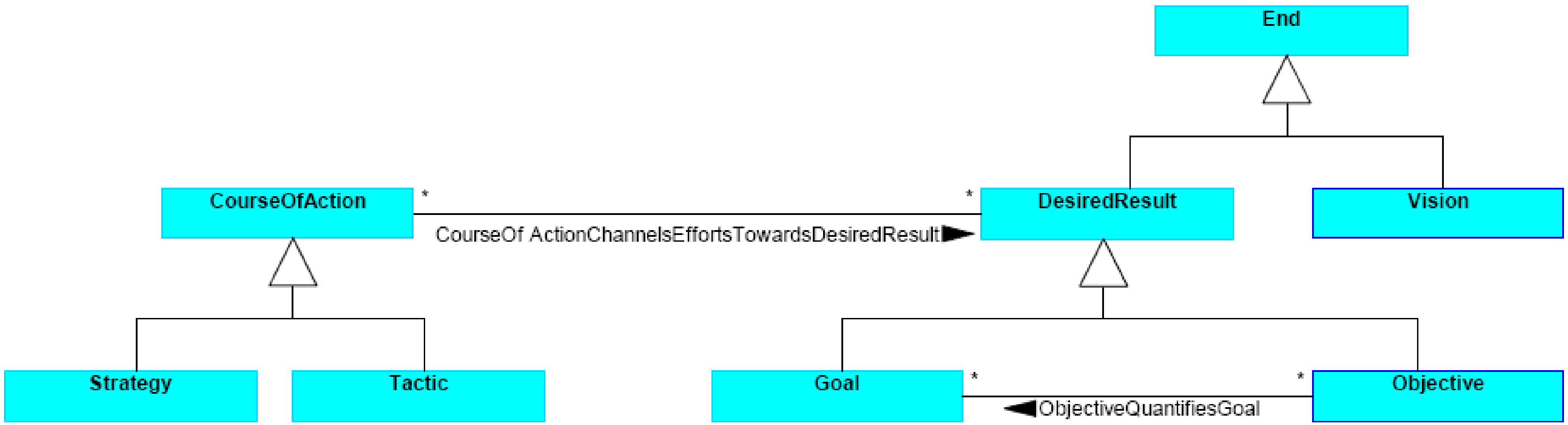
The Significance of UML

- UML helps you specify, visualize, and document models of software 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
- UML forms a foundation of OMG's Model Driven Architecture (MDA)
  - a UML model can be either platform-independent or platform-specific,
- Standardized by the OMG: Definition driven by consensus rather than innovation
- Using XMI (XML Metadata Interchange, another OMG standard), you can transfer your 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

Class Diagramm

- We already used UML class diagrams informally to describe BMM and SBVR
- Example: Class Diagramm



Types of UML Diagrams

Structure diagrams

1. Class diagram
2. Composite structure diagram (*)
3. Component diagram
4. Deployment diagram
5. Object 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

Overview of this Section

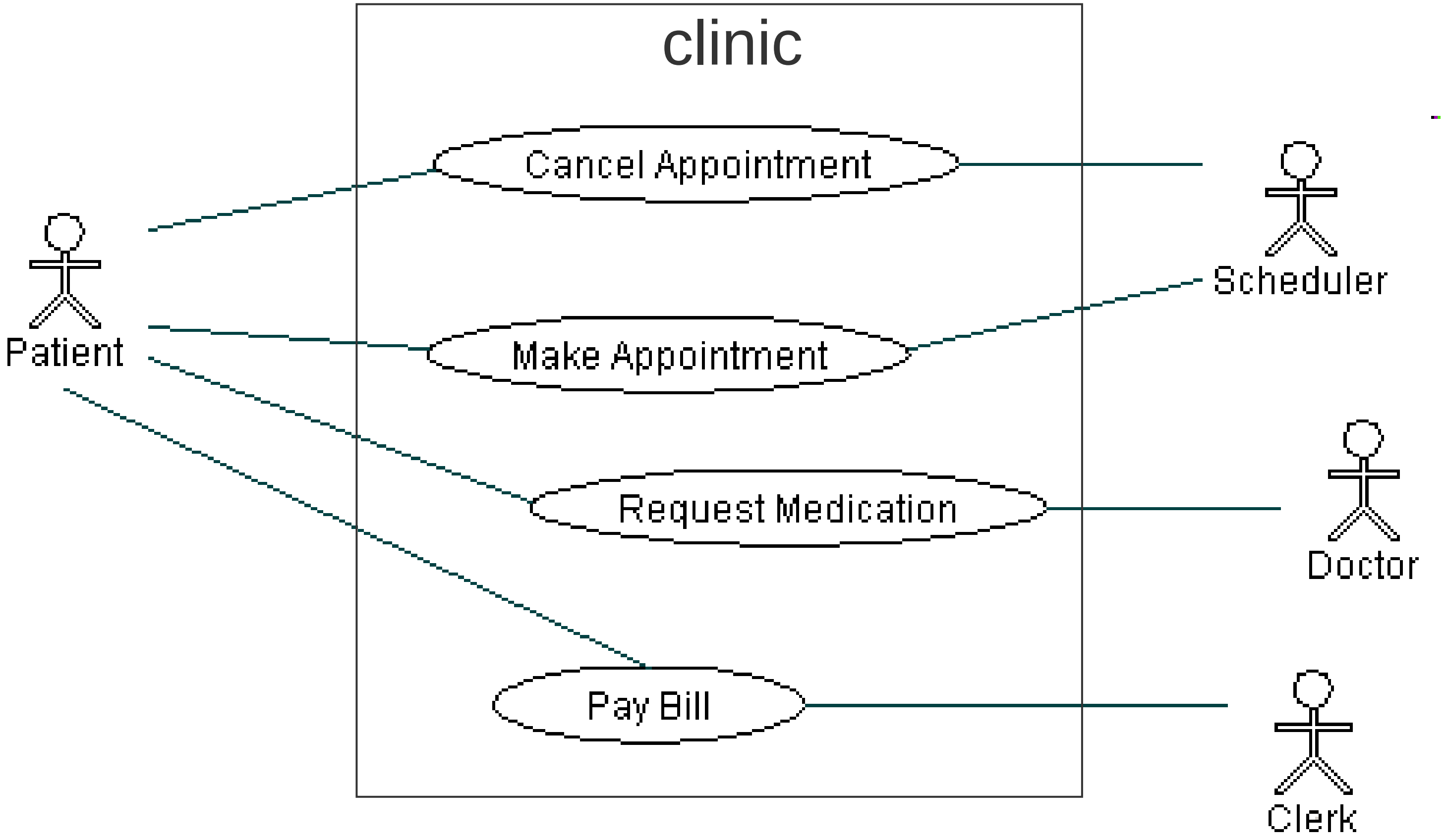
- A closer look at …
  - Use case diagram
  - Class diagram
- A short look at …
  - Object diagram
  - Deployment diagram
  - State machine diagram
  - Activity diagram

  and the
  - OCL Object Constraint Language

Use Case Diagrams

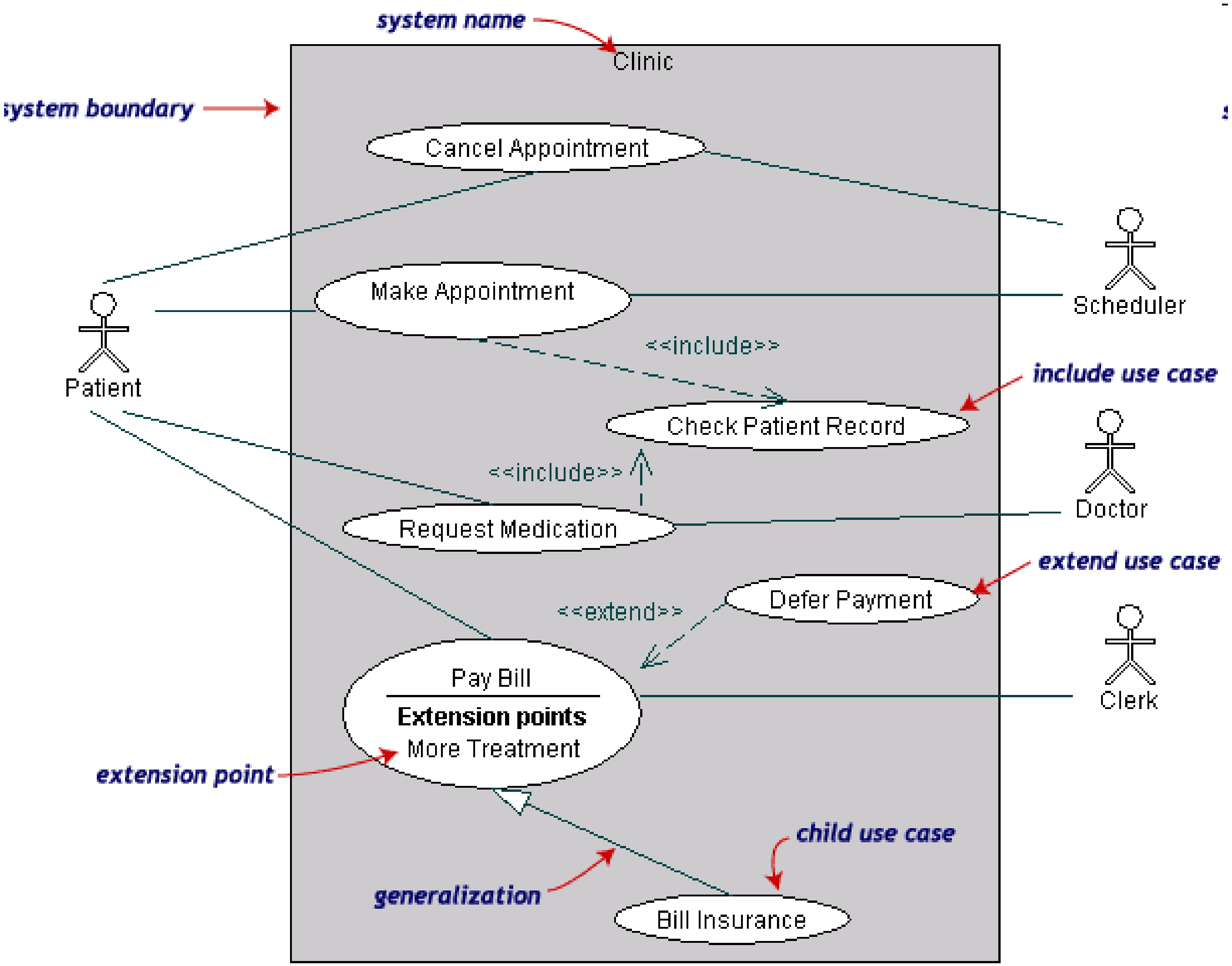
- **Use case diagrams** describe what a system does from the standpoint of an external observer. The emphasis is on *what* a system does rather than *how.*
- Main concepts:
  - System: the system under modeling
  - Actor: external "user" of the system: who or what initiates the events involved in that task. Actors are simply roles that people or objects play.
  - Use case: execution scenario, observable by an actor:
  - Communication: The connection between actor and use case is a communication association (or communication for short).
- Use Case diagrams are widely used in real-life projects, e.g. for
  - Exposing requirements
  - Communicate with clients
  - Planning the project
- Additional textual notes are often used/required

Use Case Diagram Example



A **system boundary** rectangle separates the clinic system from the external actors.

Use Case Diagram extended



Use Case Example - Explanations

- A use case **generalization** shows that one use case is simply a special kind of another.
  - **Pay Bill** is a parent use case and **Bill Insurance** is the child.

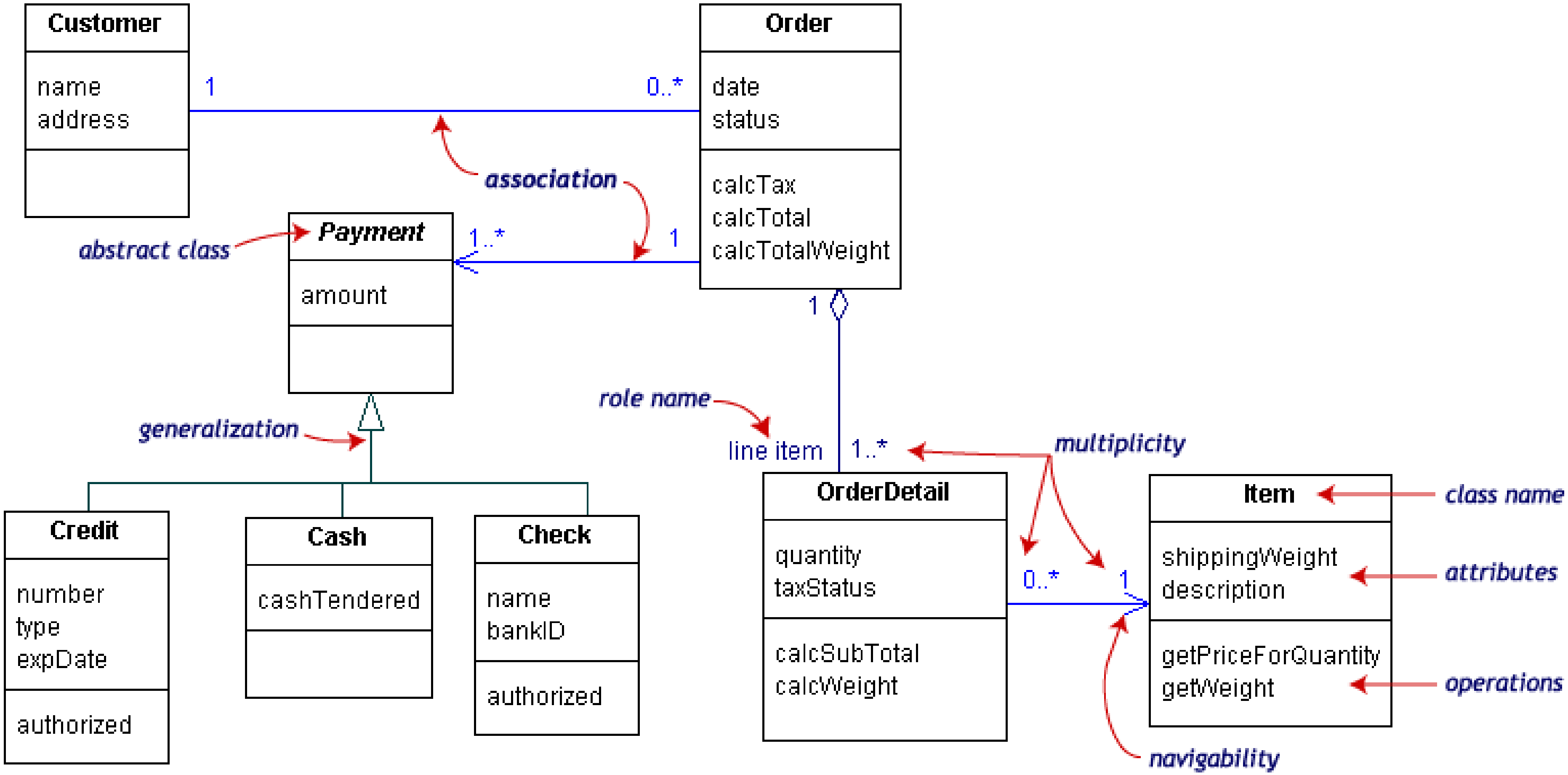
  A child can be substituted for its parent whenever necessary. Generalization appears as a line with a triangular arrow head toward the parent use case.
- **Include** relationships factor use cases into additional ones. Includes are especially helpful when the same use case can be factored out of two different use cases.
  - **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask.

  In the diagram, include notation is a dotted line beginning at base use case ending with an arrows pointing to the include use case. The dotted line is labeled <<include>>.
- An **extend** relationship indicates that one use case is a variation of another. Extend notation is a dotted line, labeled <<extend>>, and with an arrow toward the base case. The **extension point**, which determines when the extended case is appropriate, is written inside the base case.

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

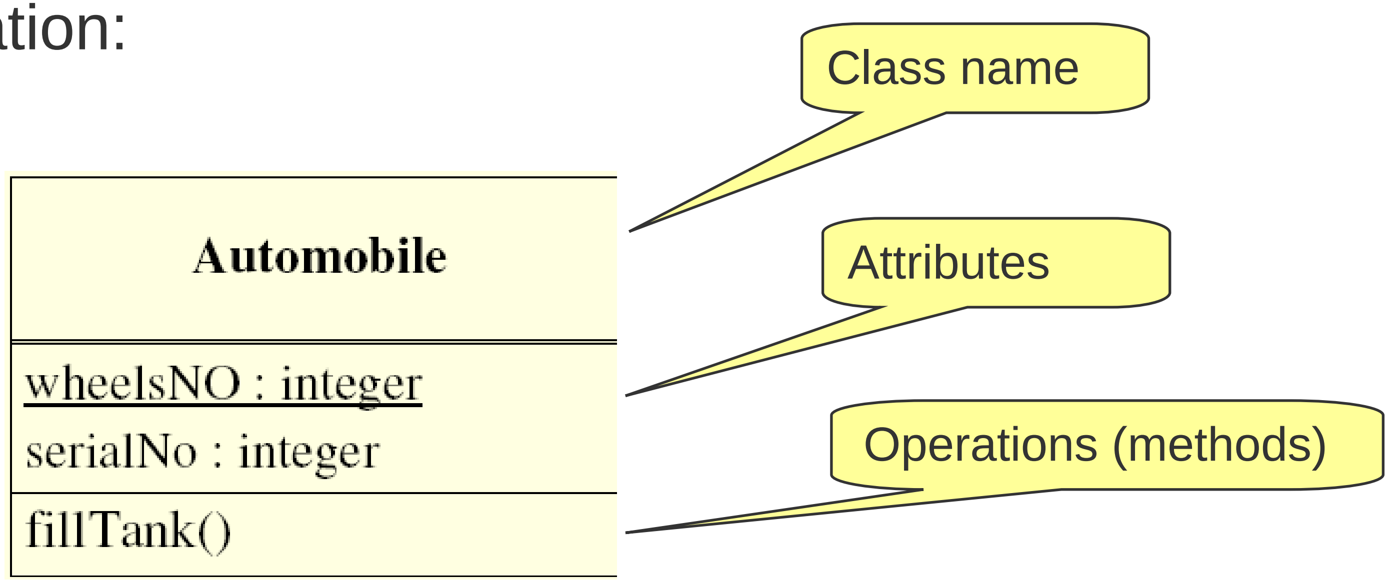


Object Orientation

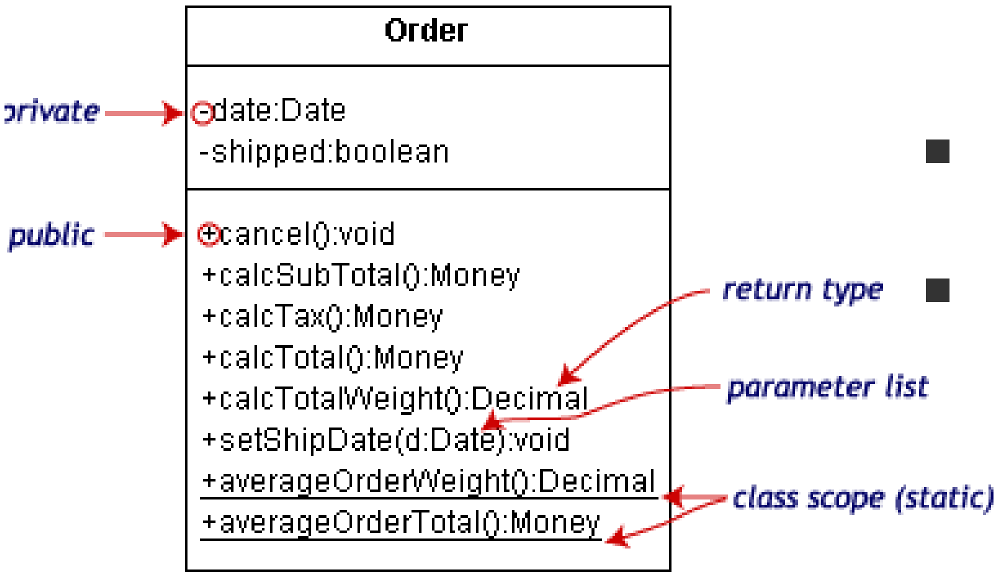
- In the first versions, UML was described as addressing the needs of modeling systems in a OO manner
- Object orientation still is the inspiration for some key concepts
- Main concepts:
  - Object – individual unit capable of *receiving/sending messages*, processing data
  - Class – pattern giving an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

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 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

- The class notation is a 3-piece rectangle with the class name, attributes, and operations.
- 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

- **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**.
- **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**.
- 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

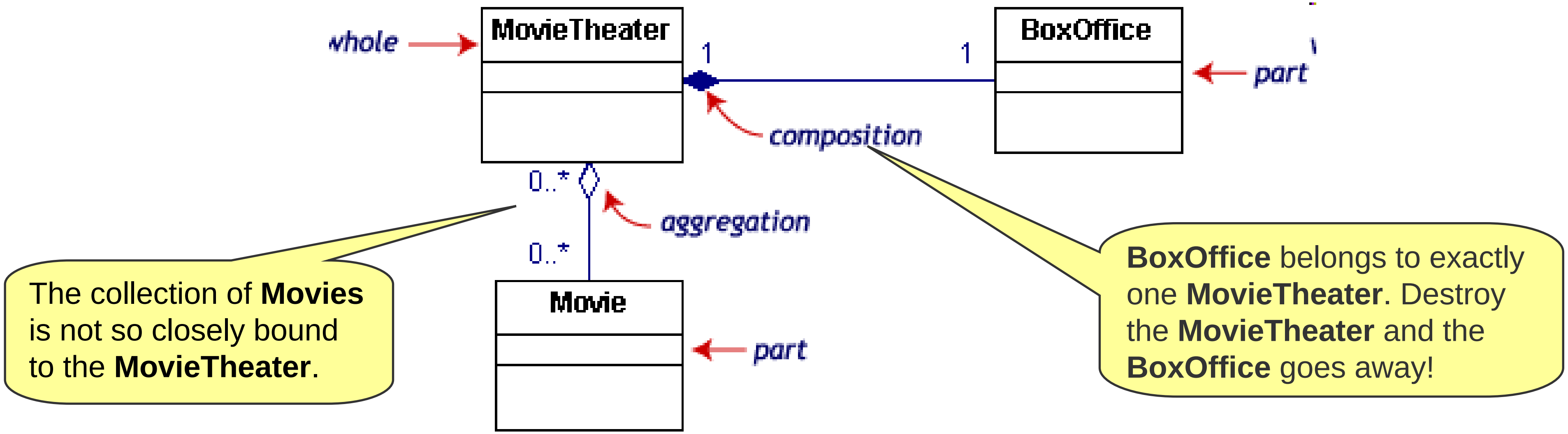
Class Diagram Elements (cont.)

- 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.
  - In our example, there can be only one **Customer** for each **Order**, but a **Customer** can have any number of **Orders**.
- This table gives the most common multiplicities.

Multiplicities	Meaning
0..1	zero or one instance. The notation ***n . . m*** indicates ***n*** to ***m*** instances.
0..* *or* *****	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

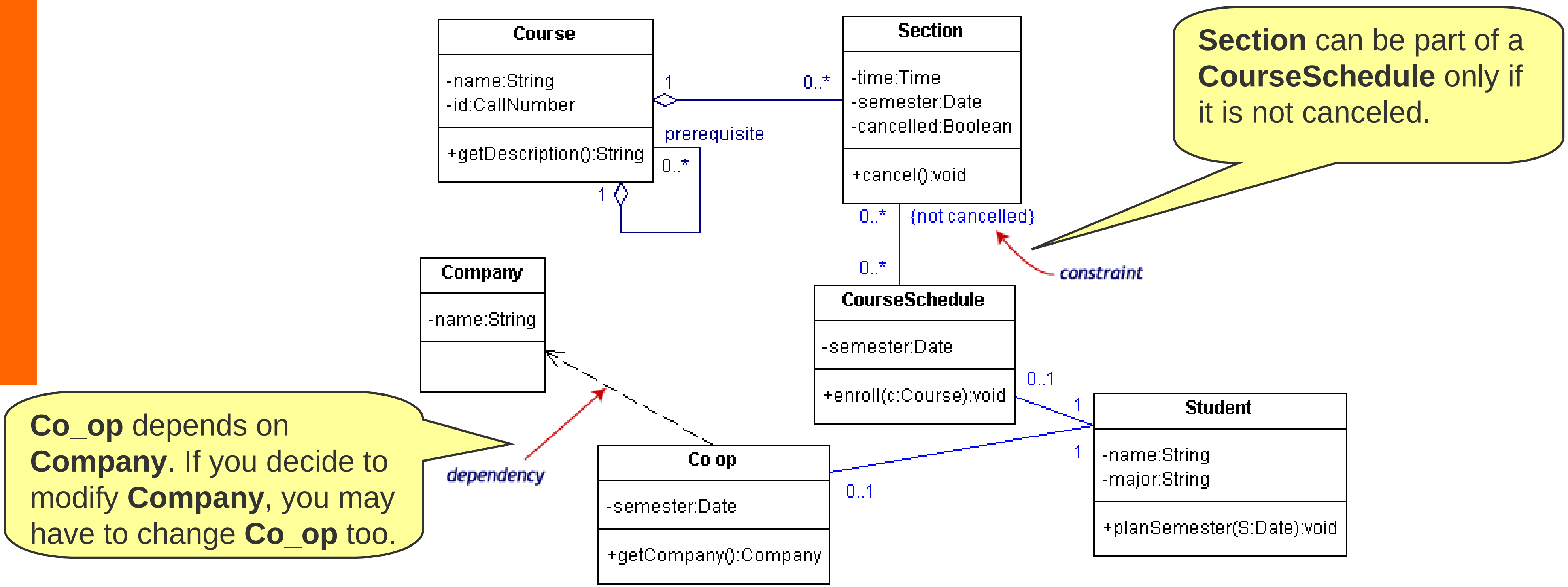
Composition and Aggregation

- **Composition** is a strong association in which the part can belong to only one whole -- the part cannot exist without the whole.
  - Composition is denoted by a filled diamond at the whole end.
- **Aggregation** is a kind of "light" composition (semantics open, to be accommodated to user needs)
  - Aggregation is denoted by a empty diamond at the whole end.



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 { }.



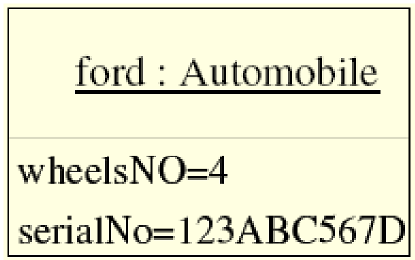
Other Elements of Class Diagrams

There are other elements of class diagrams

- Association Classes
- Interfaces
- Stereotypes
- Templates
- Comments

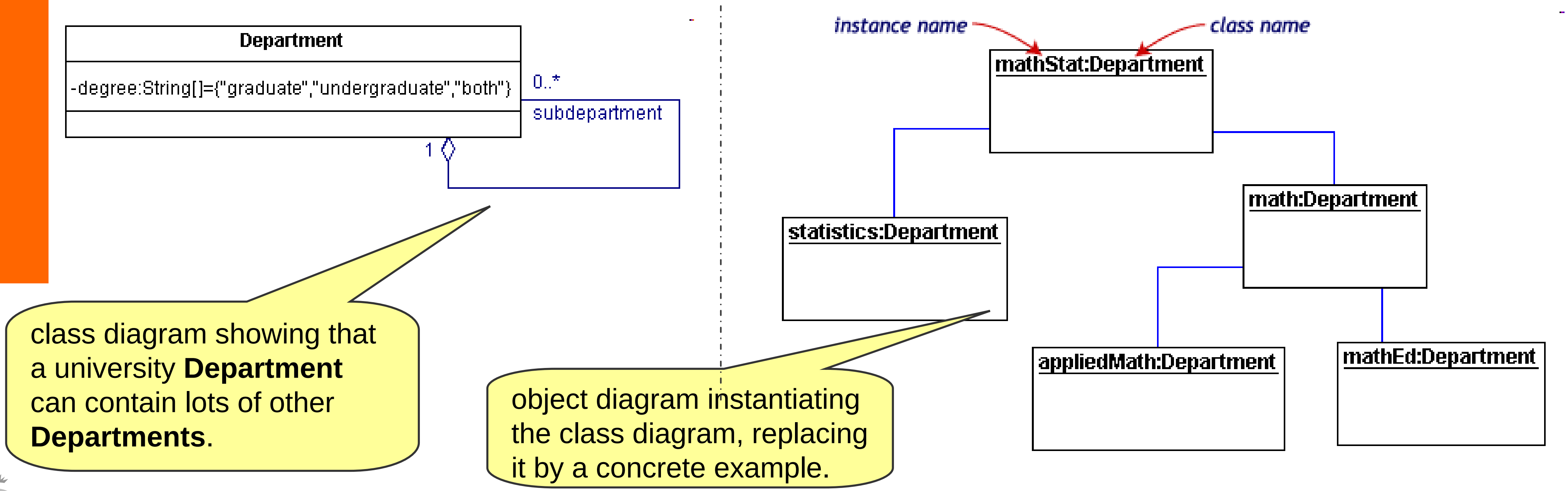
UML Object

- Instance of a class
- Can be shown in a class and object diagram
- Notation

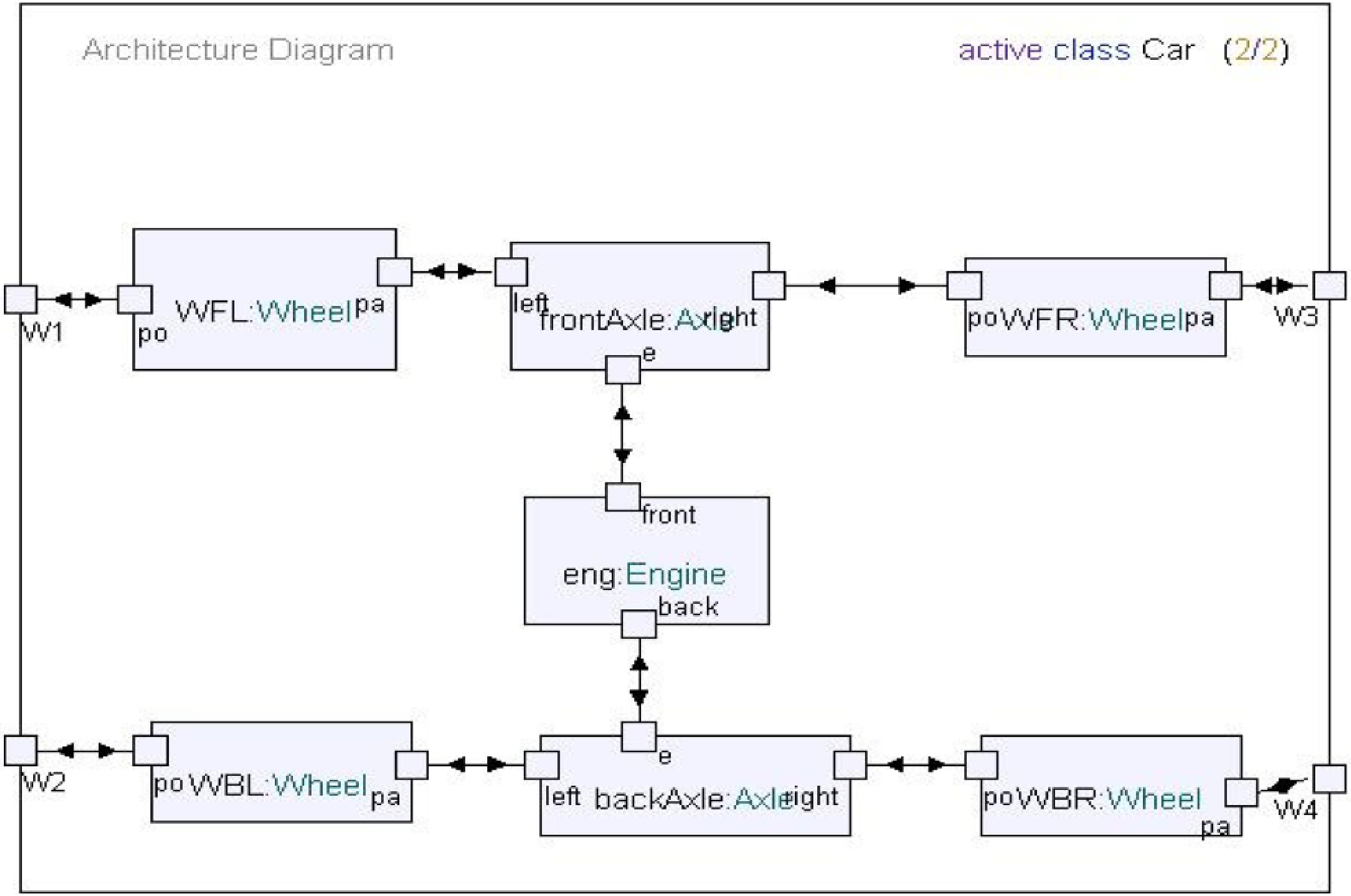


Object Diagram

- **Object diagrams** show instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships.
- Each rectangle in the object diagram corresponds to a single instance.
- Instance names are underlined in UML diagrams.
- Class or instance names may be omitted from object diagrams as long as the diagram meaning is still clear.

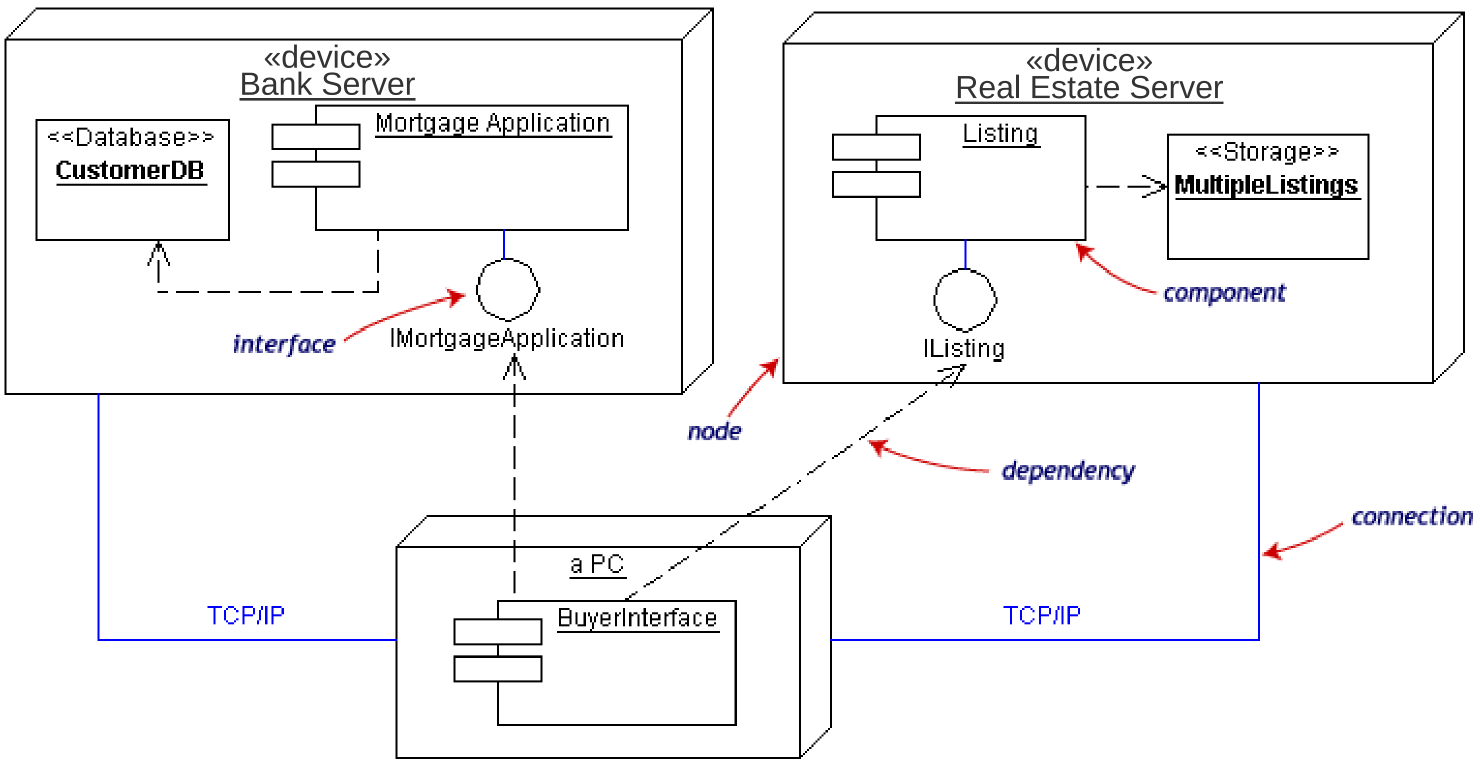


Composite Structure Diagram



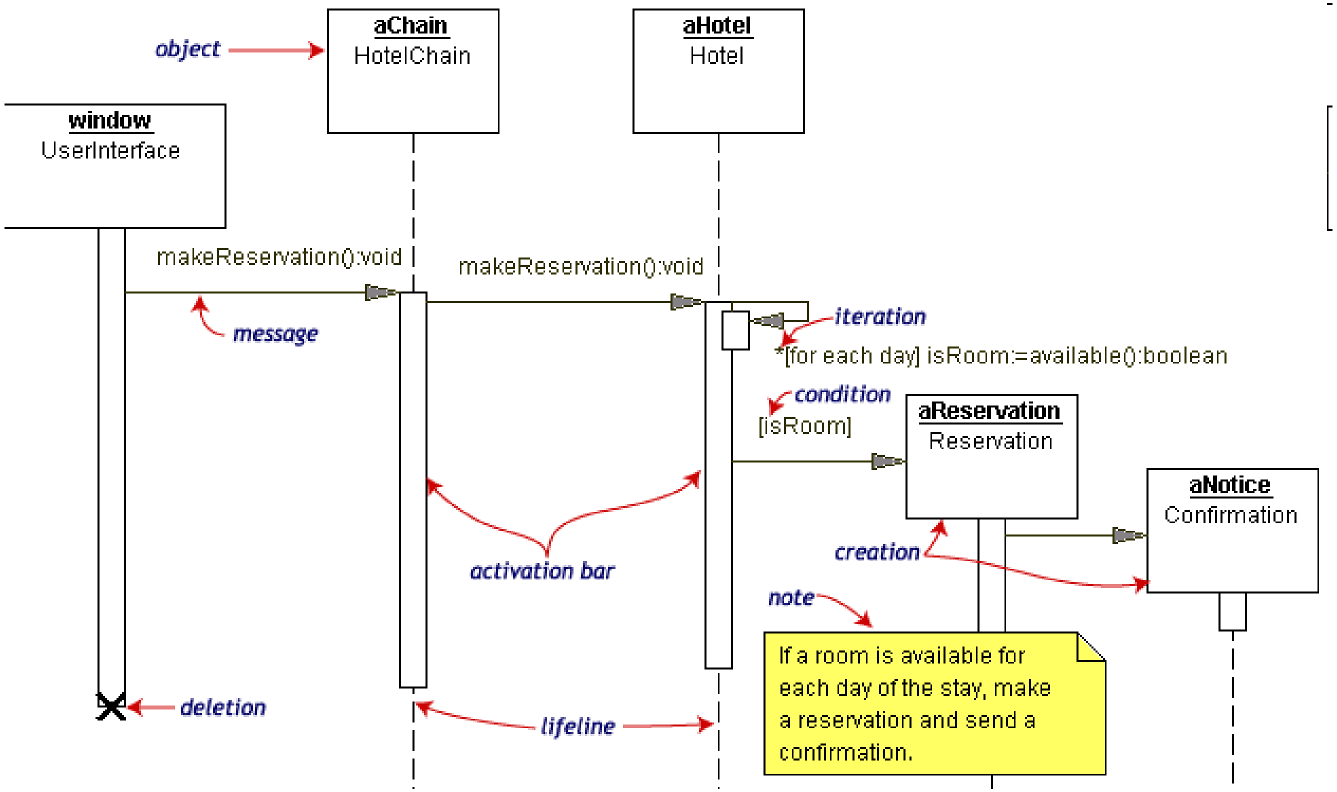
Deployment Diagrams

- **Deployment diagrams** show the physical configurations of software and hardware.
  - **Nodes** represent either physical hardware (keyword «device») or software (<<executionEnvironment>>)
  - Nodes are connected by communication relations
  - A **component** is a code module. Components are shown as rectangles with two tabs at the upper left. Each component belongs on a node.



Sequence Diagram

- A **sequence diagram** is an interaction diagram that details how operations are carried out -- what messages are sent and when.
  - Sequence diagrams are organized according to time. The time progresses as you go down the page.
  - The objects involved in the operation are listed from left to right according to when they take part in the message sequence.

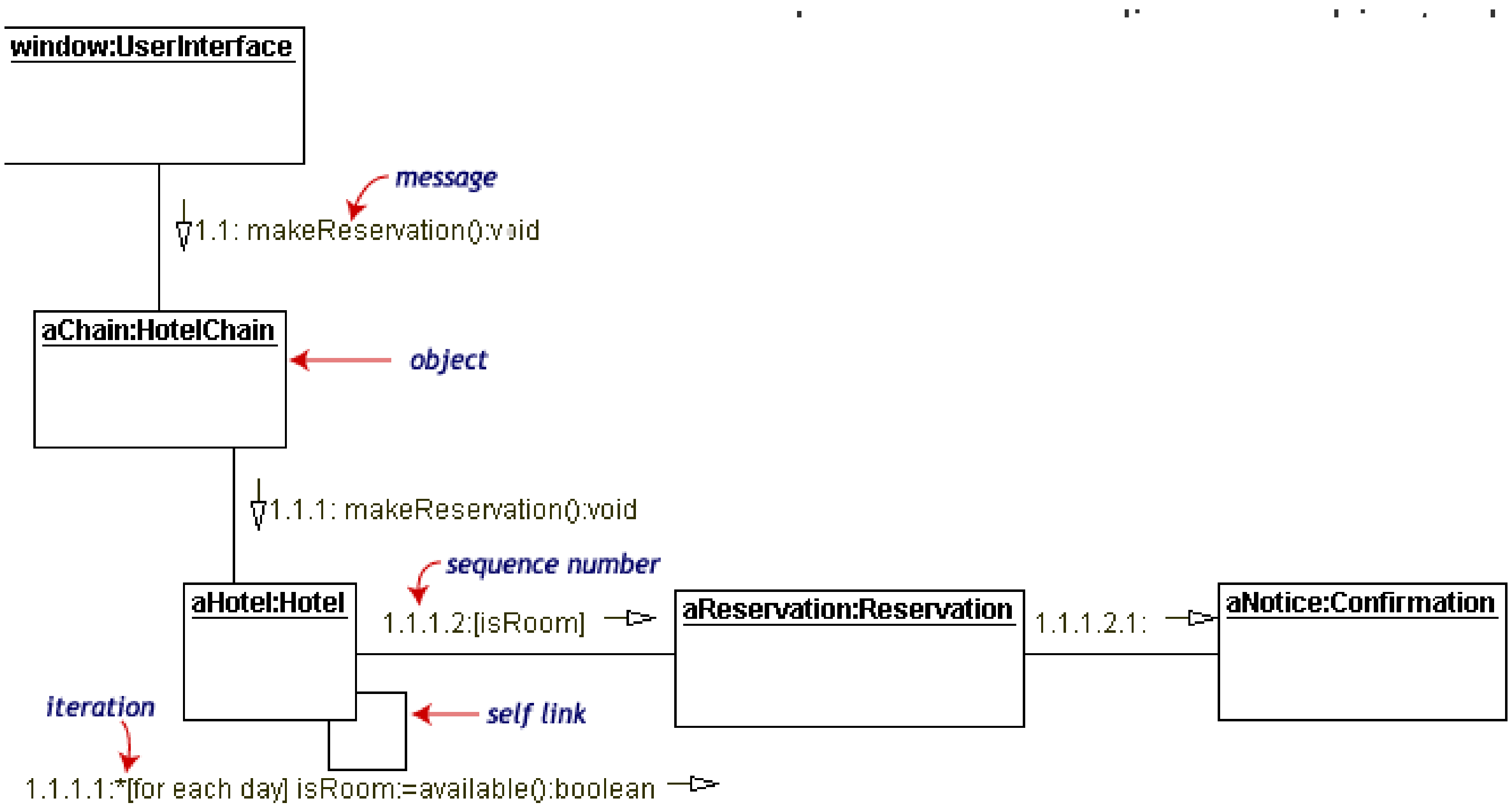


Collaboration Diagram

- **Collaboration diagrams** are also interaction diagrams. They convey the same information as sequence diagrams, but they focus on object roles instead of the times that messages are sent.

es are the
nnecting links.

diagram has a
message is
e level (sent
ne decimal prefix
to when they

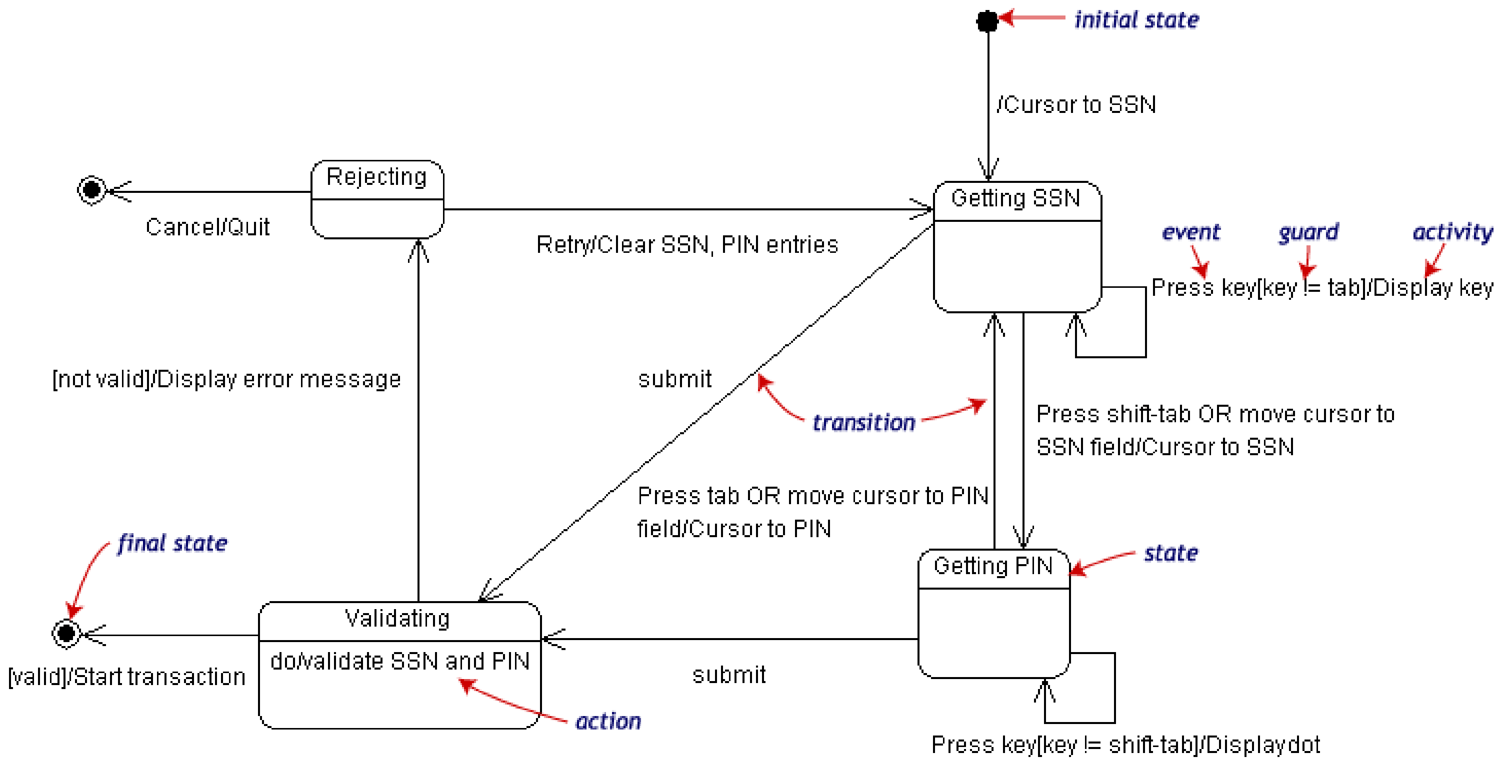


Activity Diagram

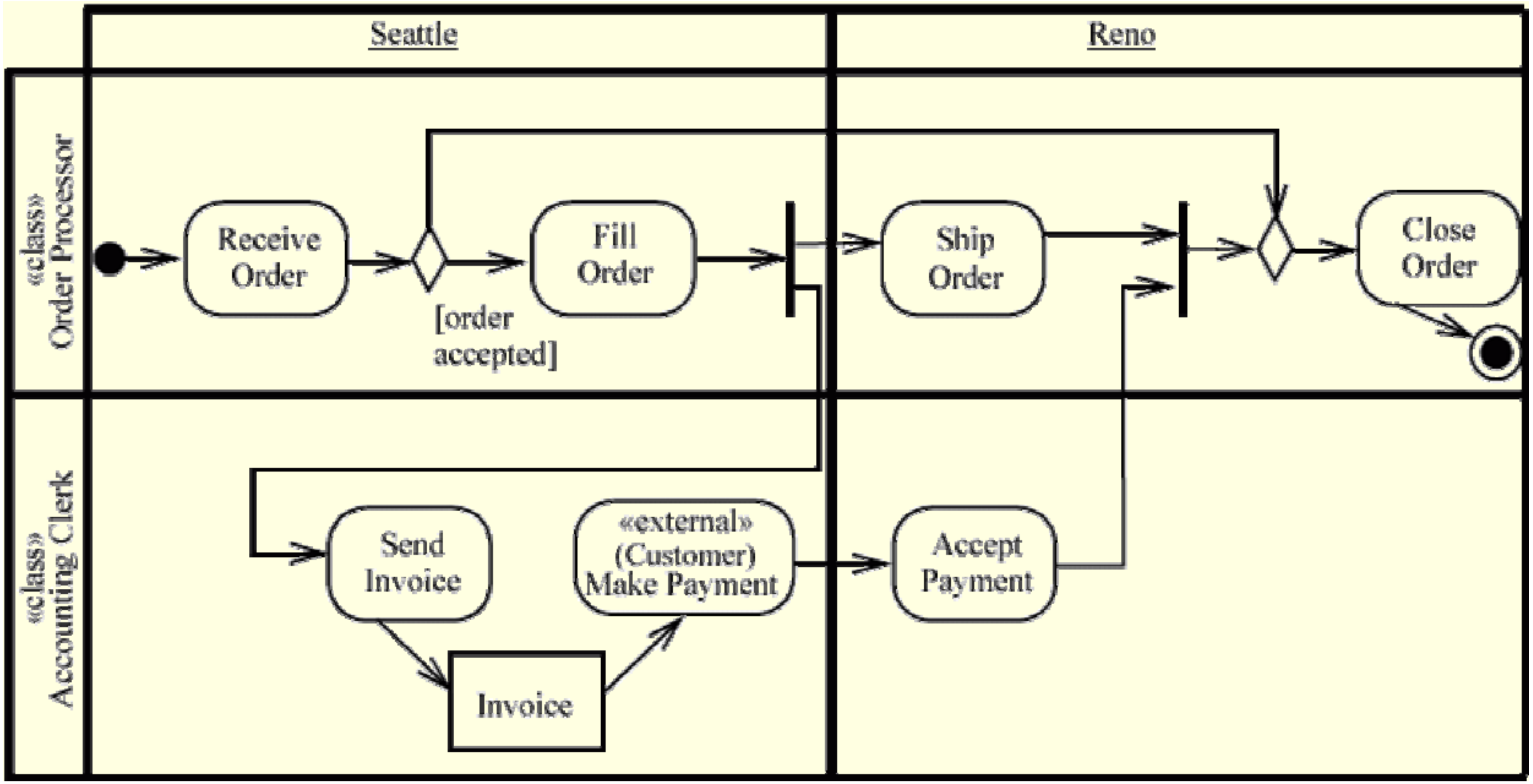
- An **activity diagram** is essentially a fancy flowchart. Activity diagrams can be divided into object **swimlanes** that determine which object is responsible for which activity.
- A single **transition** comes out of each activity, connecting it to the next activity.
- A transition may **branch** into two or more mutually exclusive transitions. **Guard expressions** (inside [ ]) label the transitions coming out of a branch.
- A branch and its subsequent **merge** marking the end of the branch appear in the diagram as hollow diamonds.
- A transition may **fork** into two or more parallel activities. The fork and the subsequent **join** of the threads coming out of the fork appear in the diagram as solid bars.

State Chart Diagram

- A **statechart diagram** shows the possible states of the object and the transitions that cause a change in state.
  - States are rounded rectangles.
  - Transitions are arrows from one state to another.
  - Events or conditions that trigger transitions are written beside the arrows.



Activity Diagram - Example



OCL – Object Constraint Language

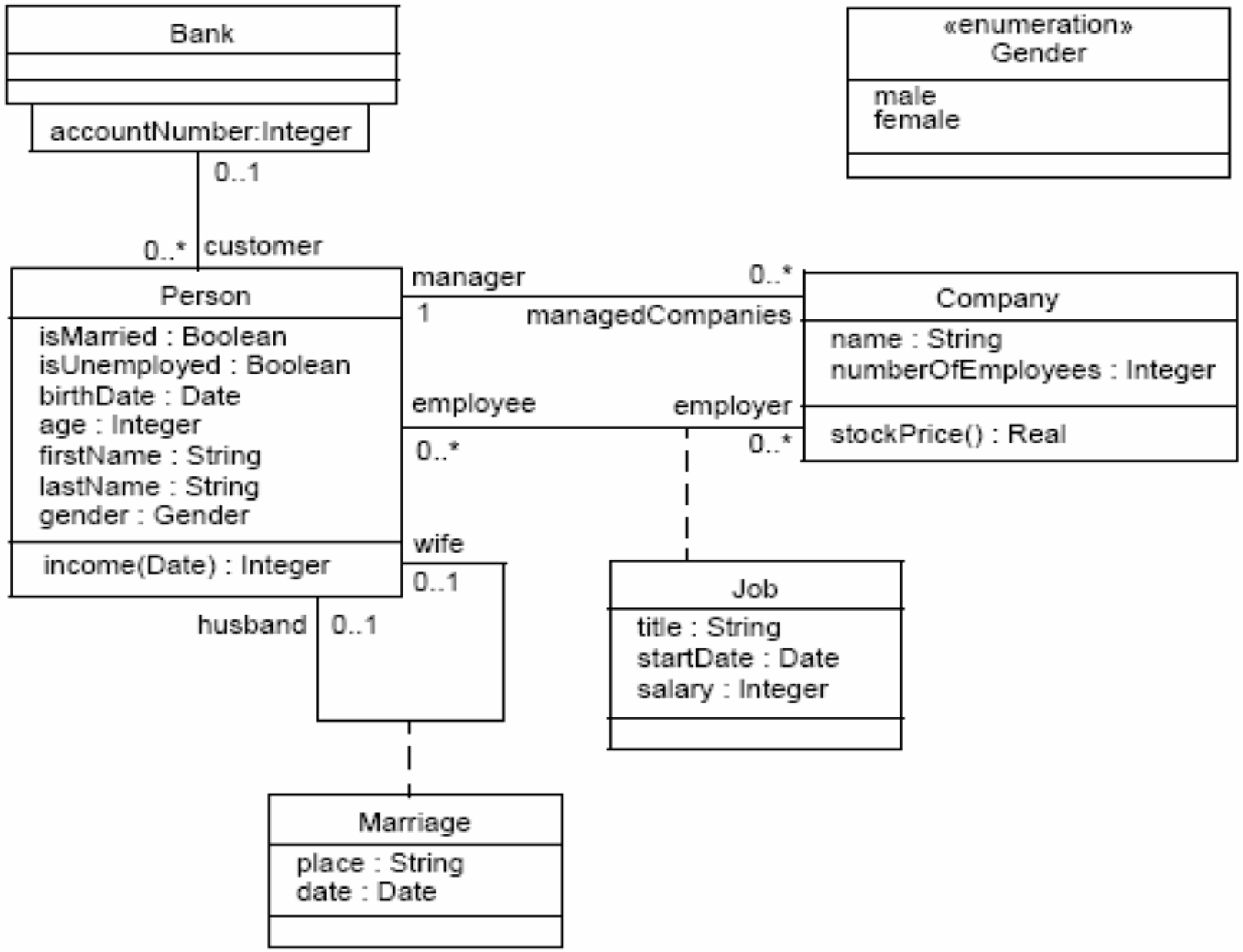
- OCL is a constraint language integrated in the UML standard
- OCL aims to fill the gap between mathematical rigor and business modeling
  - formal language with precise semantics for expression but
  - easy to read and write
- It is recommended in UML for:
  - Constraints: pre and post conditions, invariants
  - Boolean expressions: guards, query body specification
  - Defining initial and derived values of features

  and
  - to specify queries on the UML model, which are completely programming language independent.
- OCL is a pure specification/modeling language; therefore, an OCL expression is guaranteed to be without side effects – it simply returns a value
- OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL.

Where to use OCL

- OCL can be used for a number of different purposes:
  - As a query language
  - To specify invariants on classes and types in the class model
  - To specify type invariant for Stereotypes
  - To describe pre- and post conditions on Operations and Methods
  - To describe Guards
  - To specify target (sets) for messages and actions
  - To specify constraints on operations
  - To specify derivation rules for attributes for any expression over a UML model.

Example Diagram



Relation to the UML Metamodel

- Context
  - Each OCL expression is written in the context of an instance of a specific type.
  - In an OCL expression, the reserved word *self* is used to refer to the contextual instance
  - The context of an OCL expression within a UML model can be specified through a so-called context declaration at the beginning of an OCL expression
- Invariant
  - The OCL expression can be part of an Invariant, a condition that must be true for all instances at any time

context Company **inv**:

self.numberOfEmployees > 50

Relation to the UML Metamodel

- Preconditions and Postconditions
  - The stereotype of constraint is shown by putting the labels ‘pre:’ and ‘post:’ before the actual Preconditions and Postconditions. For example:
  - The OCL expression can be part of an Invariant, a condition that must be true for all instances at any time

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
    pre : param1 > ...
    post: result = ...
```

- In an example diagram, we can write:

```
context Person::income(d : Date) : Integer
    post: result = 5000
```