

# Unified Modeling Language

UML 2.0 fundamentals

*Gheorghe Aurel Pacurar*

# What is UML?

1. **Unified Modeling Language**
2. **Domain** - the World of the Problem
3. **Model** - the Solution for Problem - consists of:
4. **Objects** (Attributes, Behaviors => State)
5. **Messages** (Interactions between Objects)
6. **UML** - a set of static and dynamic Diagrams for a model graphical representation:

Static Diagrams	Dynamic Diagrams
No time represented	Time represented
Use case	Sequence
Class - package, objects	Collaboration
Component	Statechart

## Why UML ?

Unified Modeling  
Language:

Is the software  
blueprint language for:

- Analysts
- Designers
- Programmers

# 1. Use Case Diagrams

---



A **Use Case** is a summary of scenarios for a single task or goal.



A **Scenario** is an example of what happens when an actor (someone) plays (interacts) in a show (with the system).

# Example



**Scenario:** „A patient calls the Clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot“

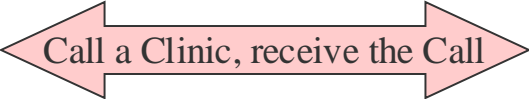
After analysis, we can find the following:

**Actors:** Patient, Clinic, Receptionist, Appointment Book, Scheduler

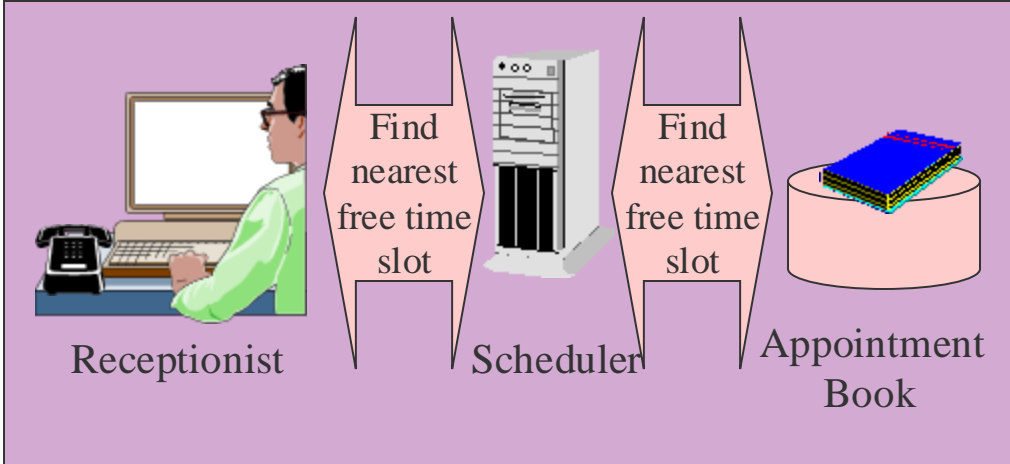
**Interactions:** The patient calls the Clinic to make an appointment. The Receptionist receives the Patient’s call, finds the nearest free time slot in the Appointment Book, and schedules the Appointment in the Appointment Book.



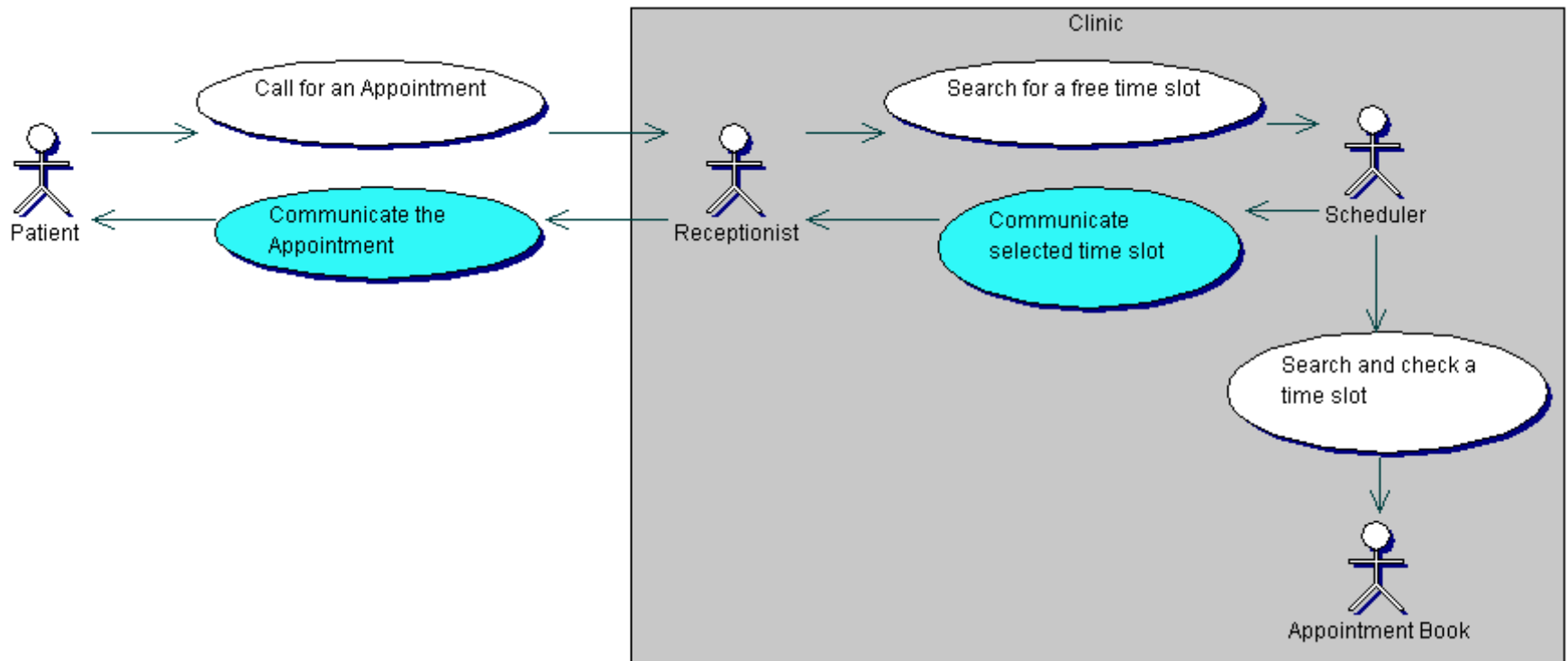
Patient



Clinic



# Use Case Diagram for Clinical Appointment





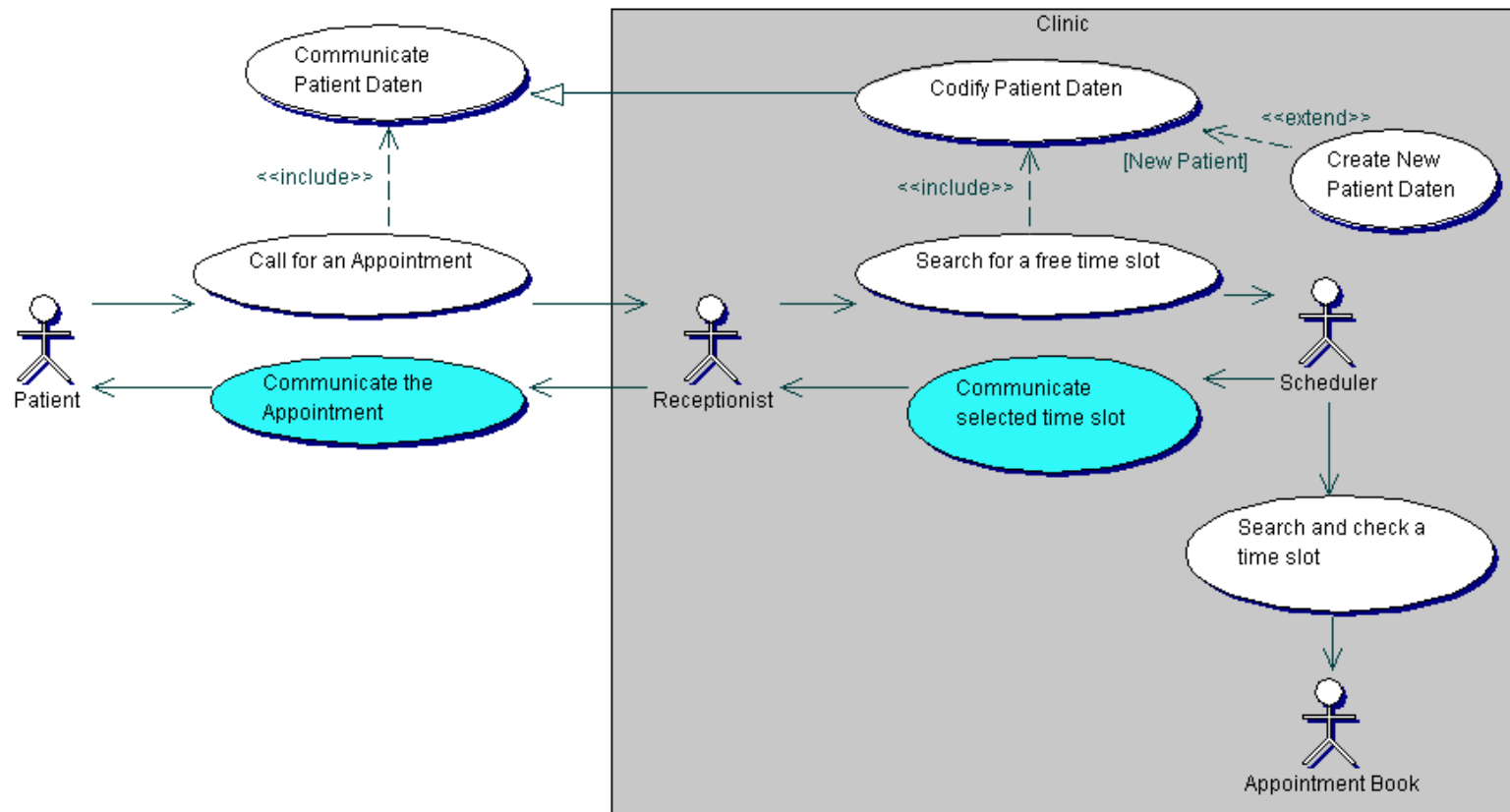
# Use Case Diagrams Extensions

Generalization : one Child Use Case is only a special kind of another Parent Use Case

Include relationship factor use cases into additional ones.

The extended relationship indicates that an Extended Use Case is a variation of another Base Use Case. The Base Use Case must define an Extension Point that determines when the Extended Use Case is appropriate.

# Use Case Diagrams Extensions





# 2. Class Diagrams

---



It gives an overview of a system by showing its classes and their relationships.

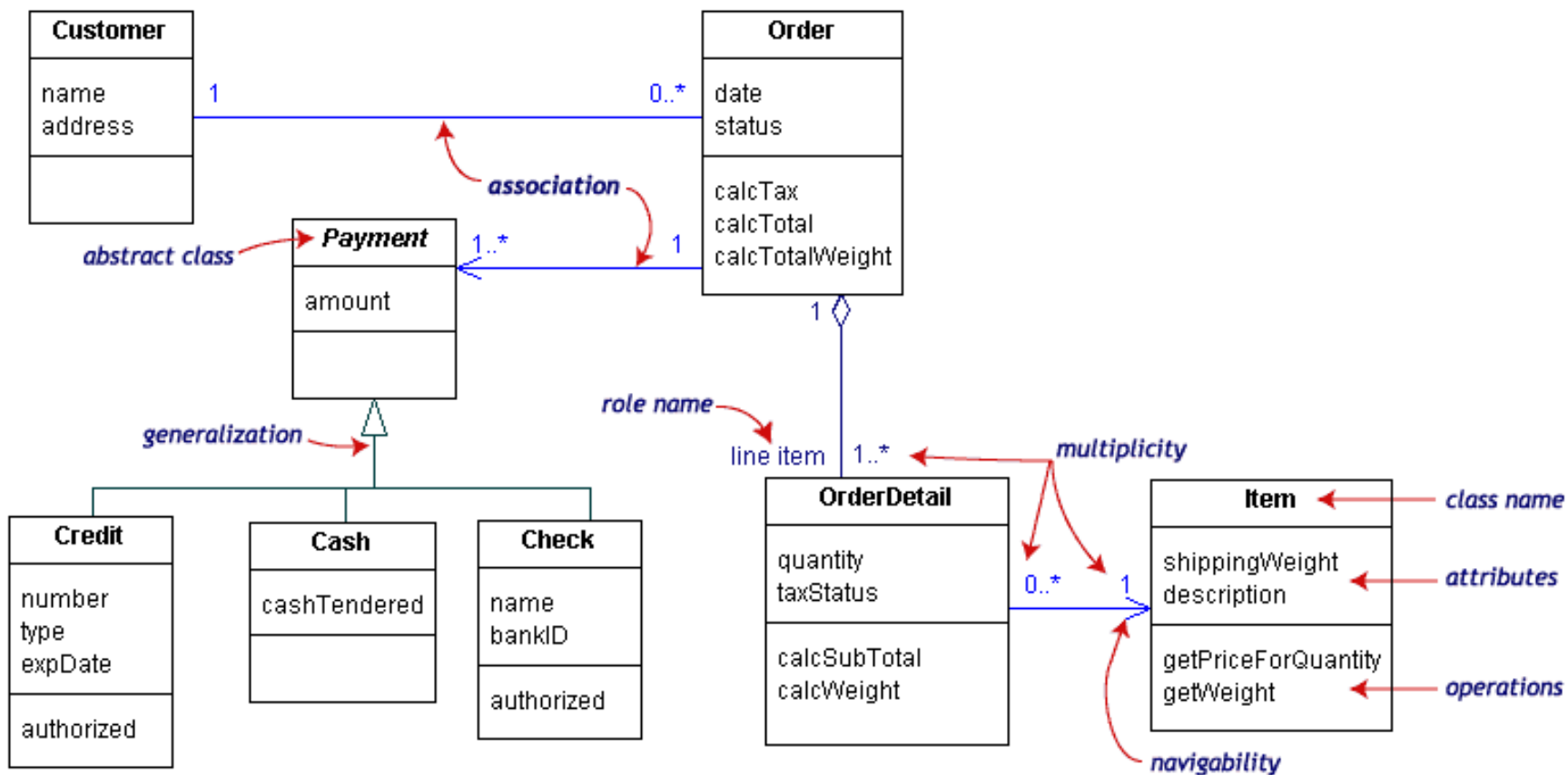


Class diagrams are static -- they display what interacts but not what happens when they do interact.



UML class notation is a rectangle divided into three parts: class name, attributes, and operations. Names of abstract classes are in italics. Relationships between classes are the connecting links.

# Example



# Class Diagrams - Relationships



A class diagram has three kinds of relationships:



**Association** -- a relationship between instances of the two classes. There is an association between two classes if an example of one class must know about the other to perform its work. In a diagram, an association is a link connecting two classes. An association has two ends. An end may have a role name to clarify the nature of the association (for example, an **OrderDetail** is a line item of each **Order**).



**Aggregation** -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole (In our diagram, 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).

# Class Diagrams - Relationships

- A **navigability** arrow on an association :
  - Shows *which direction the association can be traversed or queried* (An **OrderDetail** can be queried about its **Item**, but not vice versa).
  - The arrow also *lets you know who "owns" the association's implementation*; (in this case, **OrderDetail** has an **Item**).
  - **Associations with no navigability arrows are bi-directional.**
- The **multiplicity** of an association end :
  - Is the number of possible class instances associated with a single instance of the other end.
  - 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**).

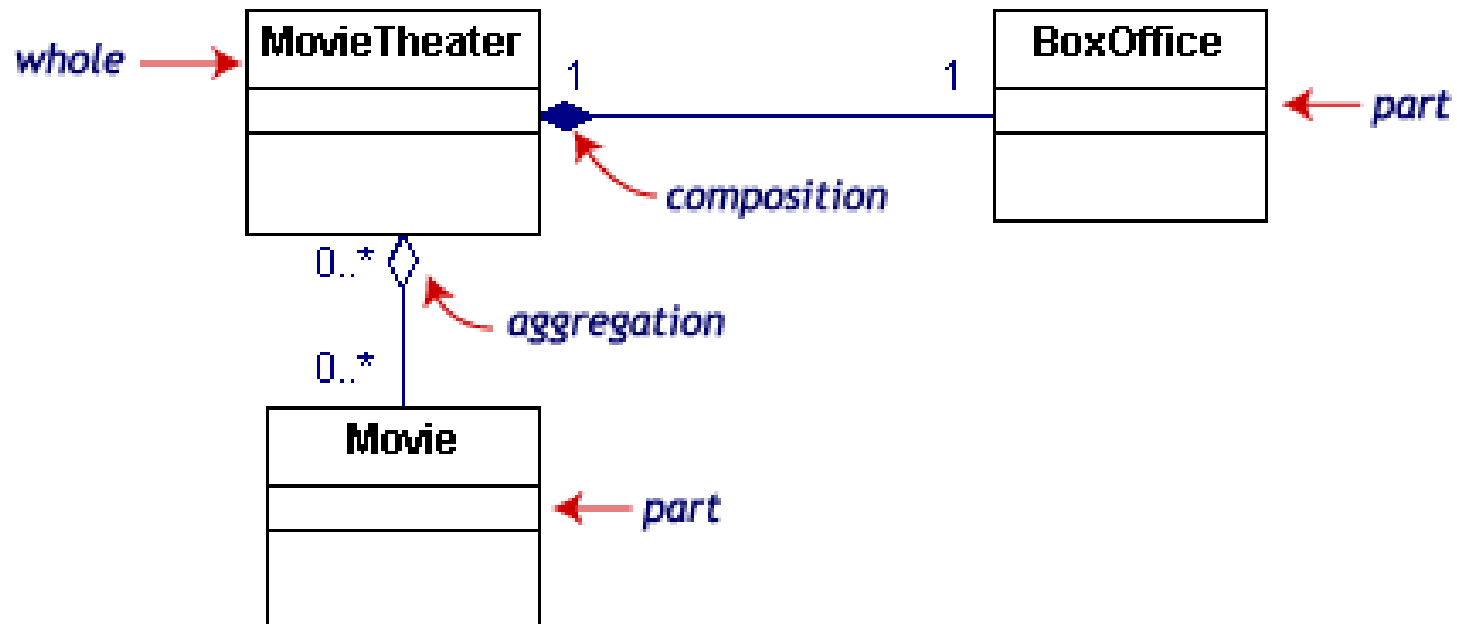
Multiplicities	Meaning
0..1	zero or one instance. The notation <i>n..m</i> indicates <i>n</i> to <i>m</i> instances. <b>0</b>
..* or *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

# Class Diagrams - Relationships

- All class diagrams have classes, links, and multiplicities. But a class diagram can show even more information. We've already looked at:
  - generalization,
  - aggregation, and
  - navigability.
- On this page, we will look at these additional items as well.
  - compositions
  - class member visibility and scope
  - dependencies and constraints
  - interfaces

# Class Diagrams – Relationships - Composition

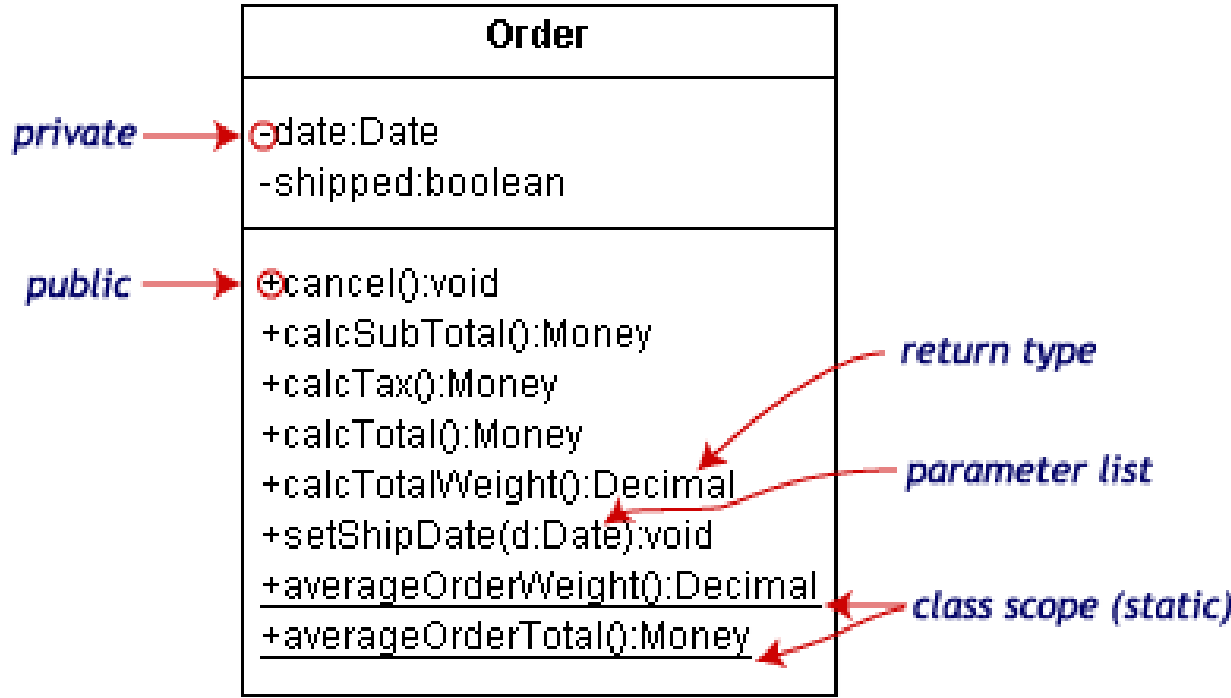
**Composition** is a strong association in which the part can belong to only one whole -- the part cannot exist without the whole.



This diagram shows that a **BoxOffice** belongs to exactly one **MovieTheater**. Destroy the **MovieTheater** and the **BoxOffice** goes away! The collection of **Movies** is not so closely bound to the **MovieTheater**

# Class Diagrams – Relationships - accessibility and scope

- 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**.



## Access specifiers\*

Symbol	Access
+	public
-	private
#	protected

\* appear in front of each member.

### UML Scope and Visibility 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.

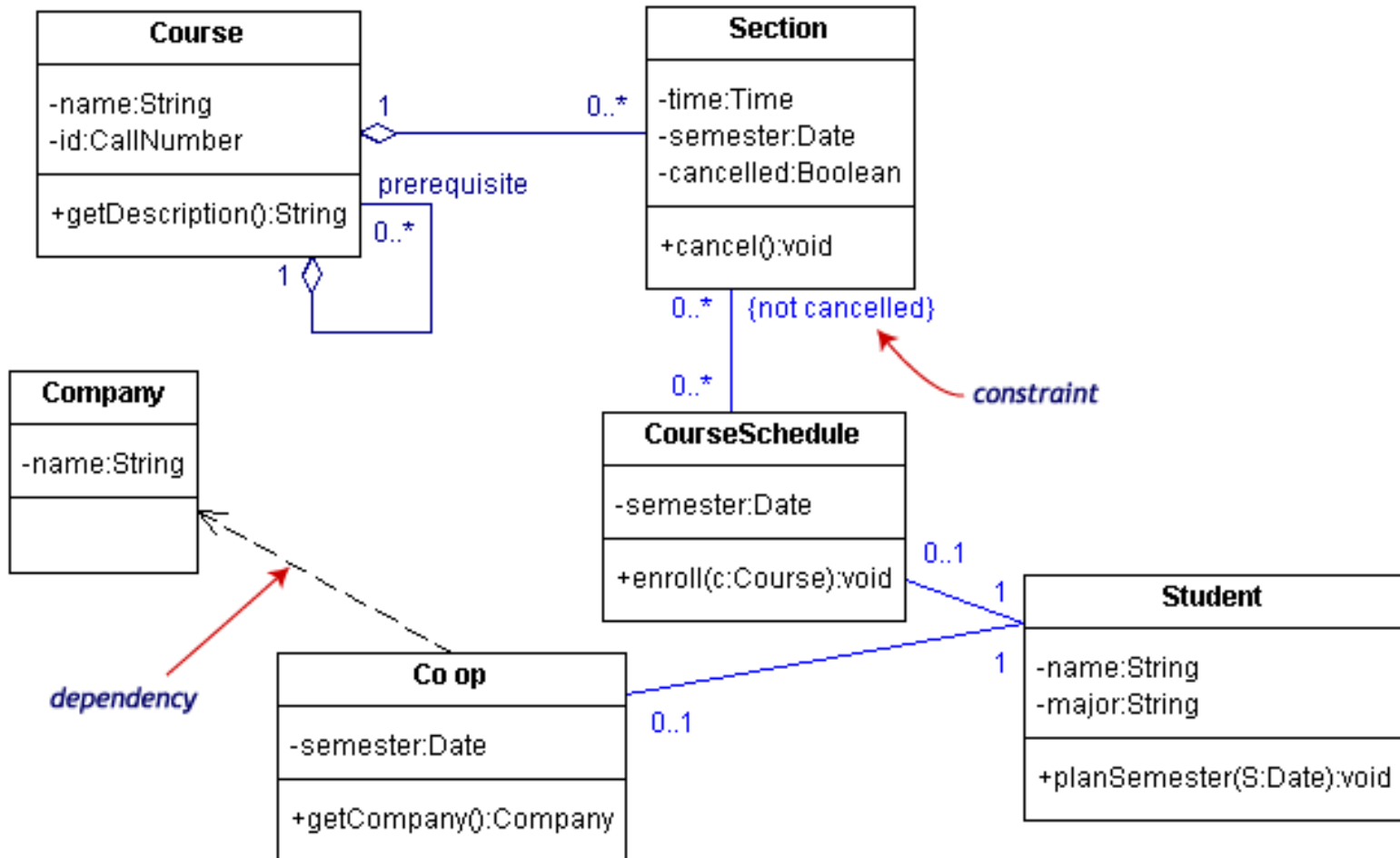
# Class Diagrams – Dependencies and constraints



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 (In the class diagram below, **Co\_op** depends on **Company**. If you decide to modify **Company**, you may have to change **Co\_op** too).



A **constraint** is a condition that every implementation of the design must satisfy. Constraints are written in curly braces { } (The constraint on our diagram indicates that a **Section** can be part of a **CourseSchedule** only if it is not canceled).





# Class Diagrams – Interfaces and stereotypes



An **interface** is a set of operation signatures and a communication protocol.

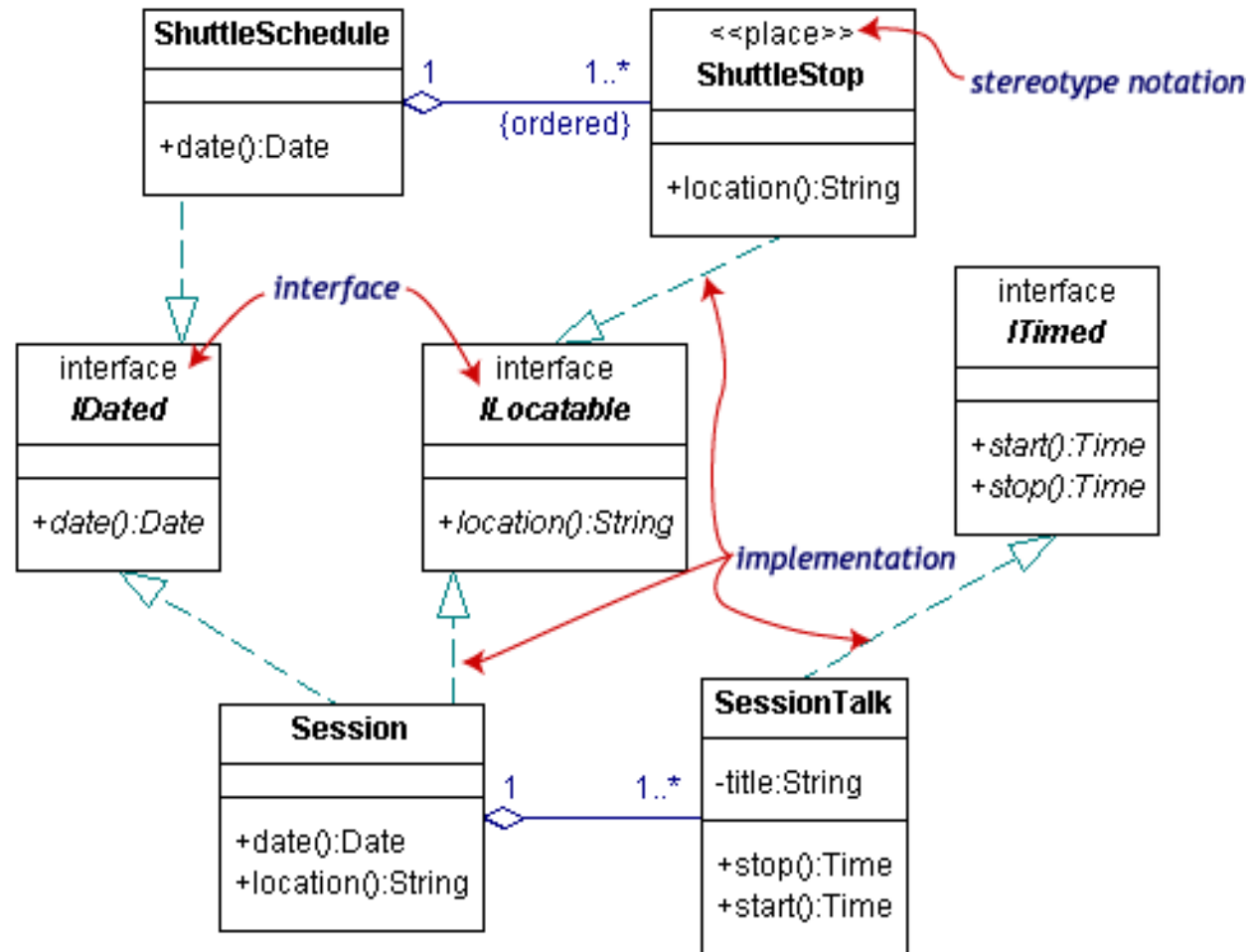


A class with operations matching those in an interface is an **implementation** (or **realization**) of the interface.



**Stereotypes**, which provide a way of extending UML, are new kinds of model elements created from existing kinds. A stereotype name is written above the class name. Ordinary stereotype names are enclosed in guillemots, which look like pairs of angle braces. An interface is a special kind of stereotype.

The class diagram is a model of a professional conference. The classes of interest to the conference are **SessionTalk**, a single presentation, and **Session**, a one-day collection of related **SessionTalks**. The **ShuttleSchedule**, with its list of **ShuttleStops**, is essential to the attendees staying at remote hotels. The diagram has one constraint: the **ShuttleStops** are ordered.



# Class Diagrams – Interfaces and stereotypes



An **interface** is a set of operation signatures and a communication protocol.



A class with operations matching those in an interface is an **implementation** (or **realization**) of the interface.

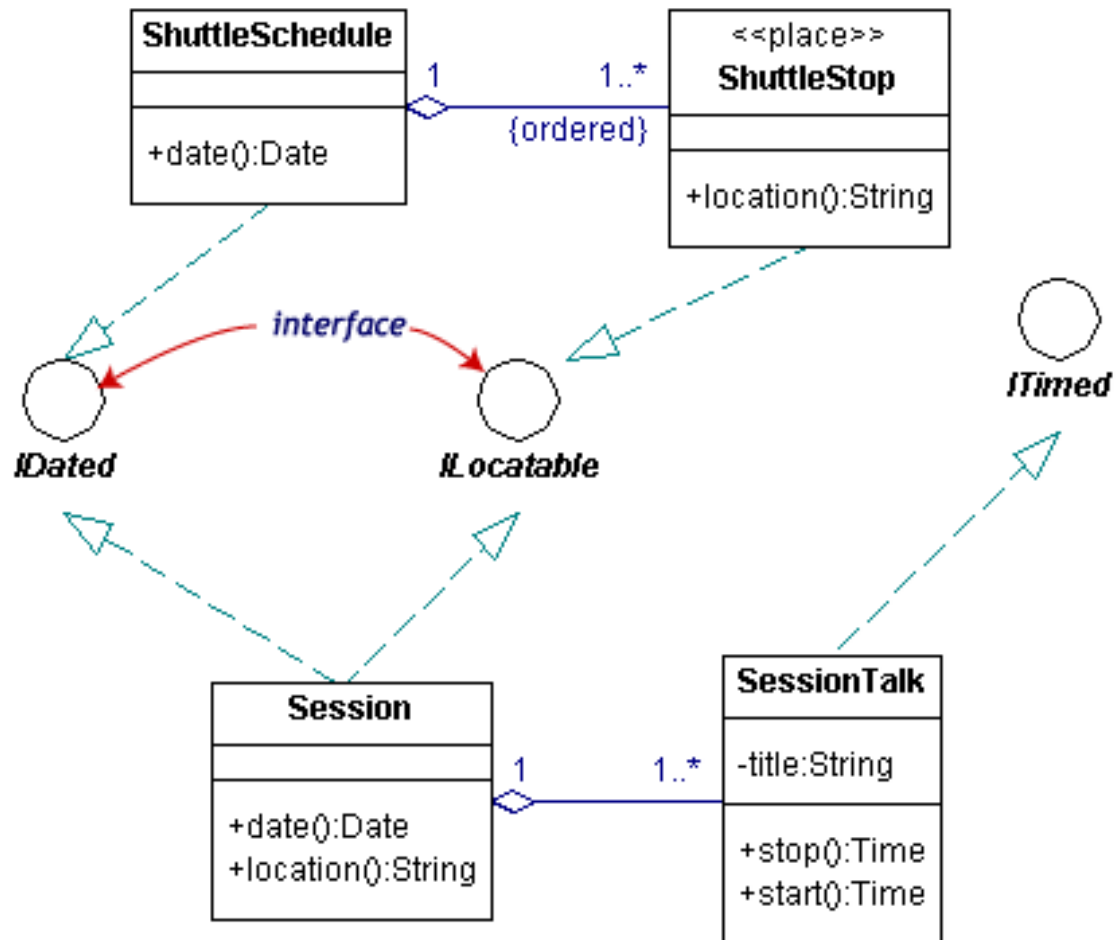


**Stereotypes**, which provide a way of extending UML, are new kinds of model elements created from existing kinds. A stereotype name is written above the class name. Ordinary stereotype names are enclosed in guillemets, which look like pairs of angle braces. An interface is a special kind of stereotype.

There are two acceptable notations for interfaces in the UML.

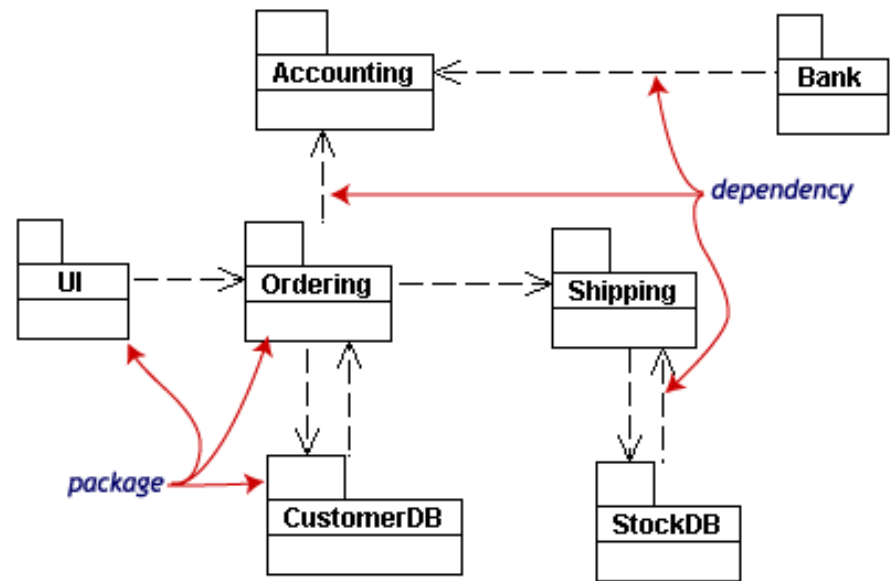
The first was illustrated in the previous slide.

The second uses the lollipop or circle notation.



# 3. Packages and Object Diagram

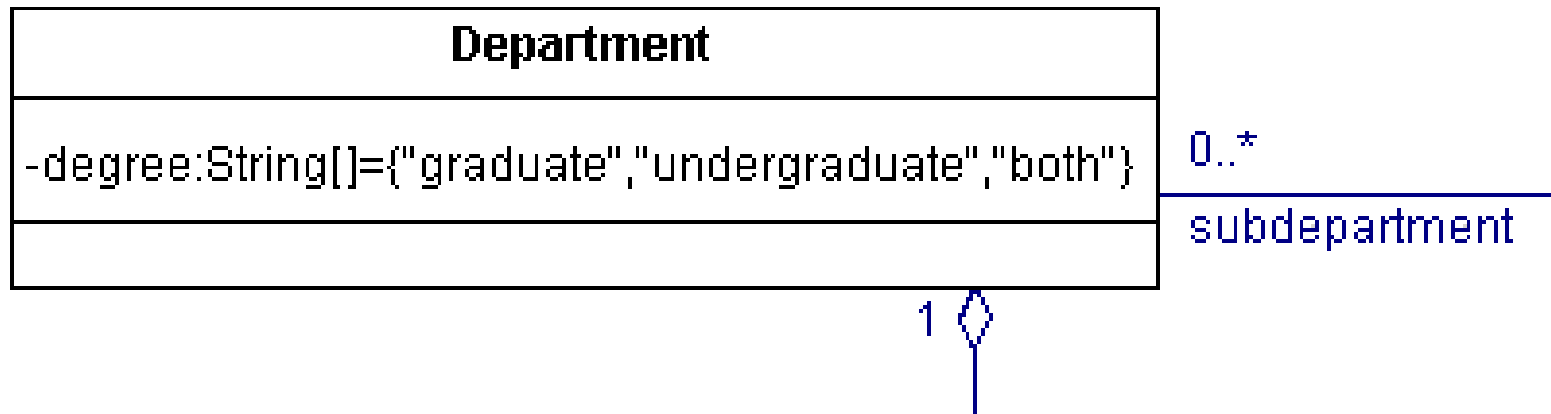
- To simplify complex class diagrams, you can group classes into **packages**.
- A **package** is a collection of logically related UML elements.
- The dotted arrows are **dependencies**. One package depends on another if changes in the other could force changes in the first.



# Example

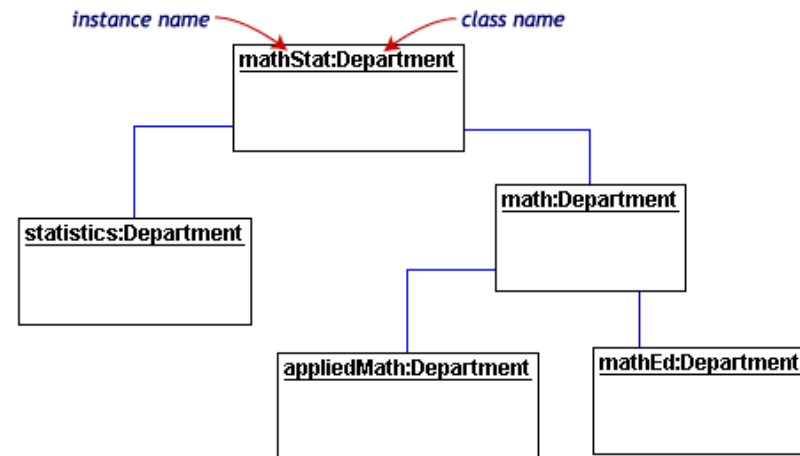
---

This small class diagram shows that a university **Department** can contain lots of other **Departments**.



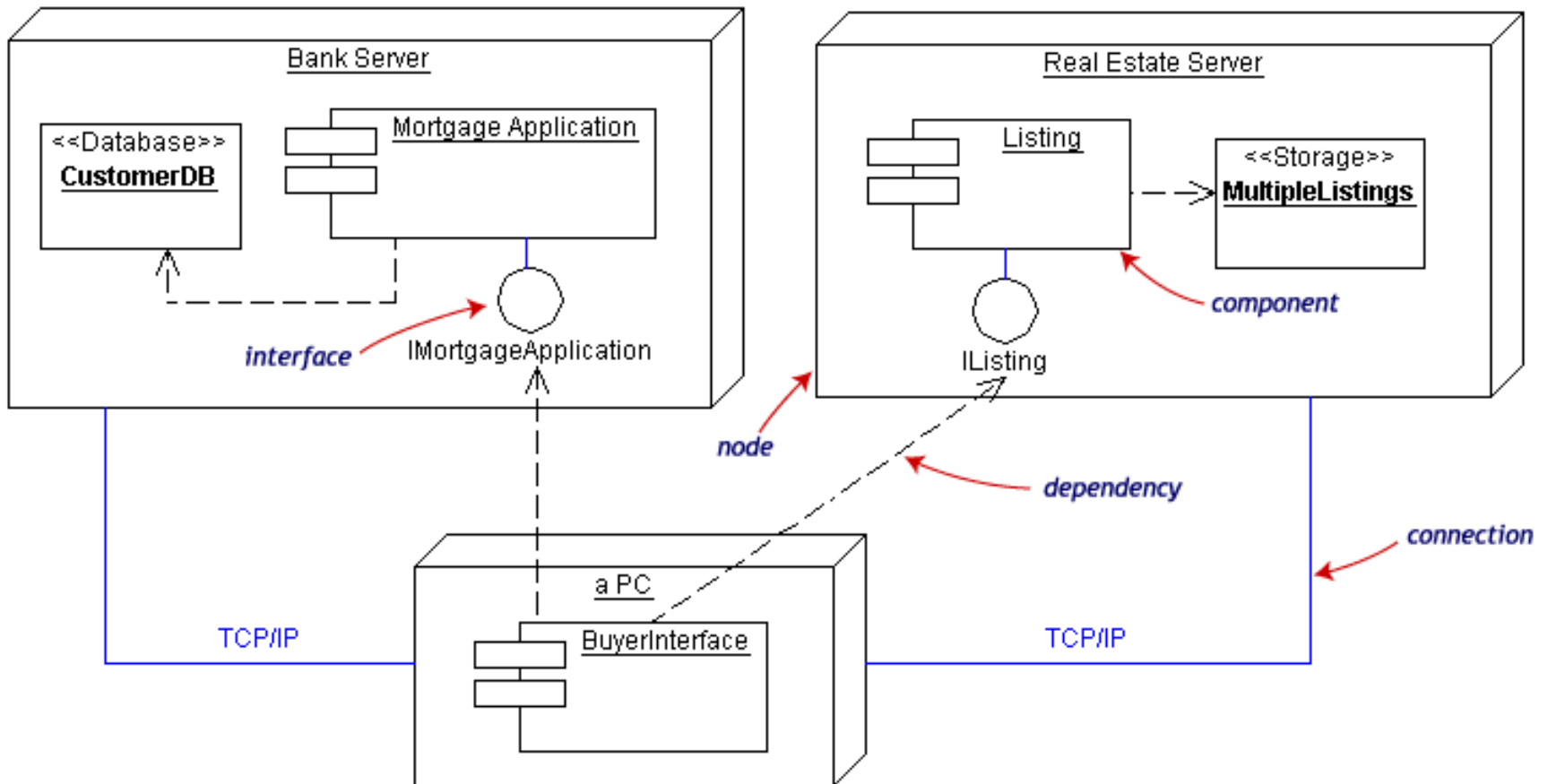
# Packages and Object Diagram

- The object diagram below instantiates the class diagram, replacing it by a concrete example.
- 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.



# 4. Component and Deployment Diagram

- A **component** is a code module. Component diagrams are physical analogs of class diagram.
- **Deployment diagrams** show the physical configurations of software and hardware.
- The physical hardware is made up of **nodes**. Each component belongs on a node.



# 5. Sequence Diagrams

---



**Interaction diagrams** are dynamic diagrams that describe how objects collaborate.



A **sequence diagram** is an interaction diagram that details how operations are carried out, such as **what** messages are sent and **when**.



Sequence diagrams are organized according to **time**.



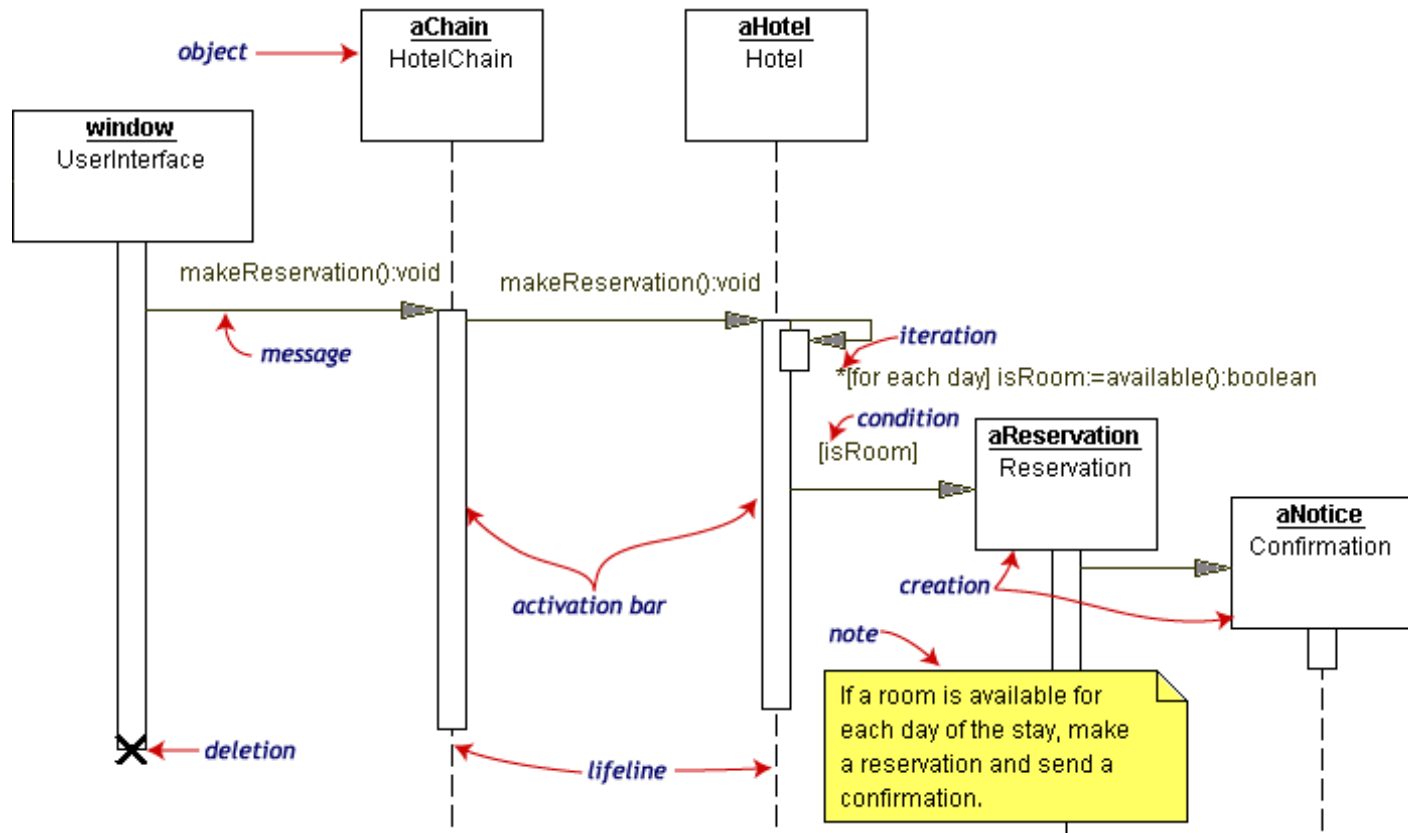
The time progresses as you go down the page.



The objects involved in the operation are listed from left to right according to when they take part in the message sequence.

# Sequence Diagrams - Example

- The example is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a **Reservation window**.





# Sequence Diagrams

---

- The **Reservation window** sends a `makeReservation()` message to a **HotelChain**. The **HotelChain** then sends a `makeReservation()` message to a **Hotel**. If the **Hotel** has available rooms, it makes a **Reservation** and a **Confirmation**.
- Each vertical dotted line is a **lifeline**, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the **activation bar** of the message on the receiver's lifeline.
- The **activation bar** represents the duration of execution of the message.
- In our diagram, the **Hotel** issues a **self-call** to determine if a room is available. If so, the Hotel creates a Reservation and a Confirmation. The asterisk on the self-call means iteration (to ensure the room is available for each day of the stay in the hotel). The expression in square brackets, [ ], is a **condition**.
- The diagram has a clarifying **note**, text inside a dog-eared rectangle. Notes can be put into any UML diagram.

# Sequence diagrams with asynchronous messages

- A **message** is **asynchronous** if it allows its sender to send additional messages while processing the original. The timing of an asynchronous message is independent of the timing of the intervening messages.

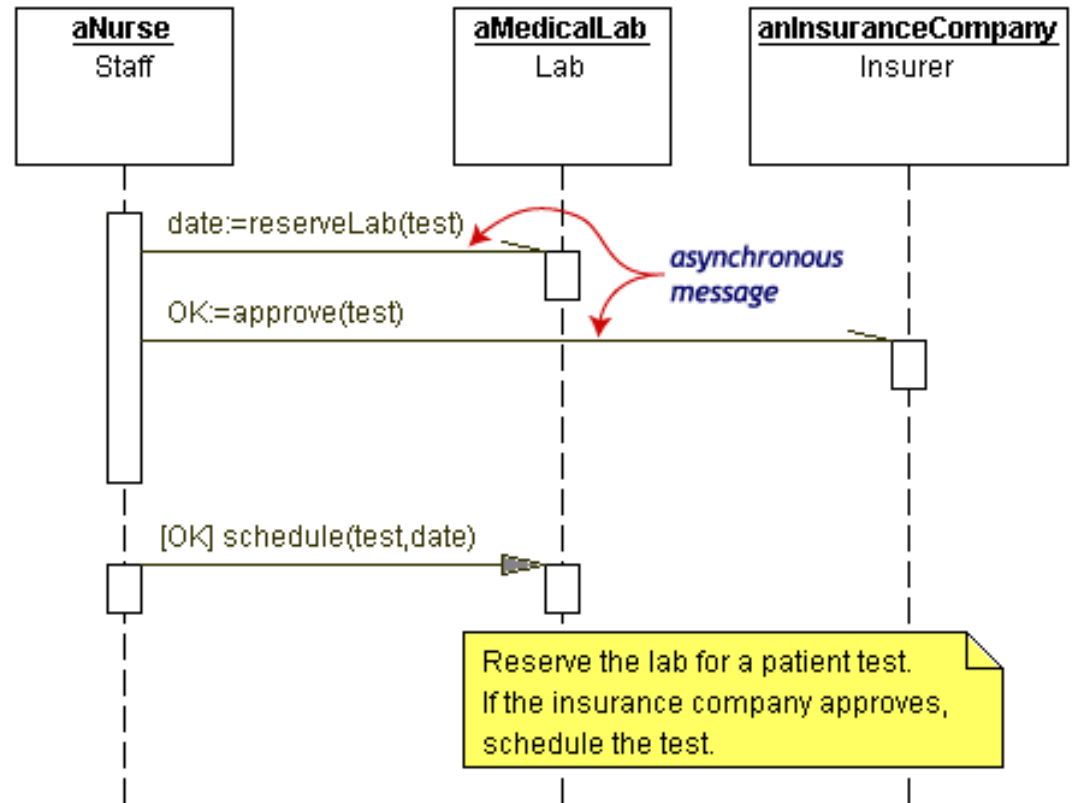
The following sequence diagram illustrates the action of a nurse requesting a diagnostic test at a medical lab. There are two asynchronous messages from the **Nurse**:

1) Ask the **MedicalLab** to reserve a date for the test and

2) Ask the **InsuranceCompany** to approve the test.

The order in which these messages are sent or completed is irrelevant. If the **InsuranceCompany** approves the test, then the **Nurse** will schedule the test on the date supplied by the **MedicalLab**.




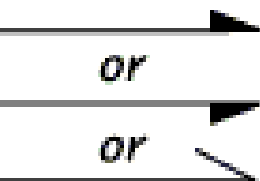
# Sequence diagrams with asynchronous messages - Example



# Sequence diagrams

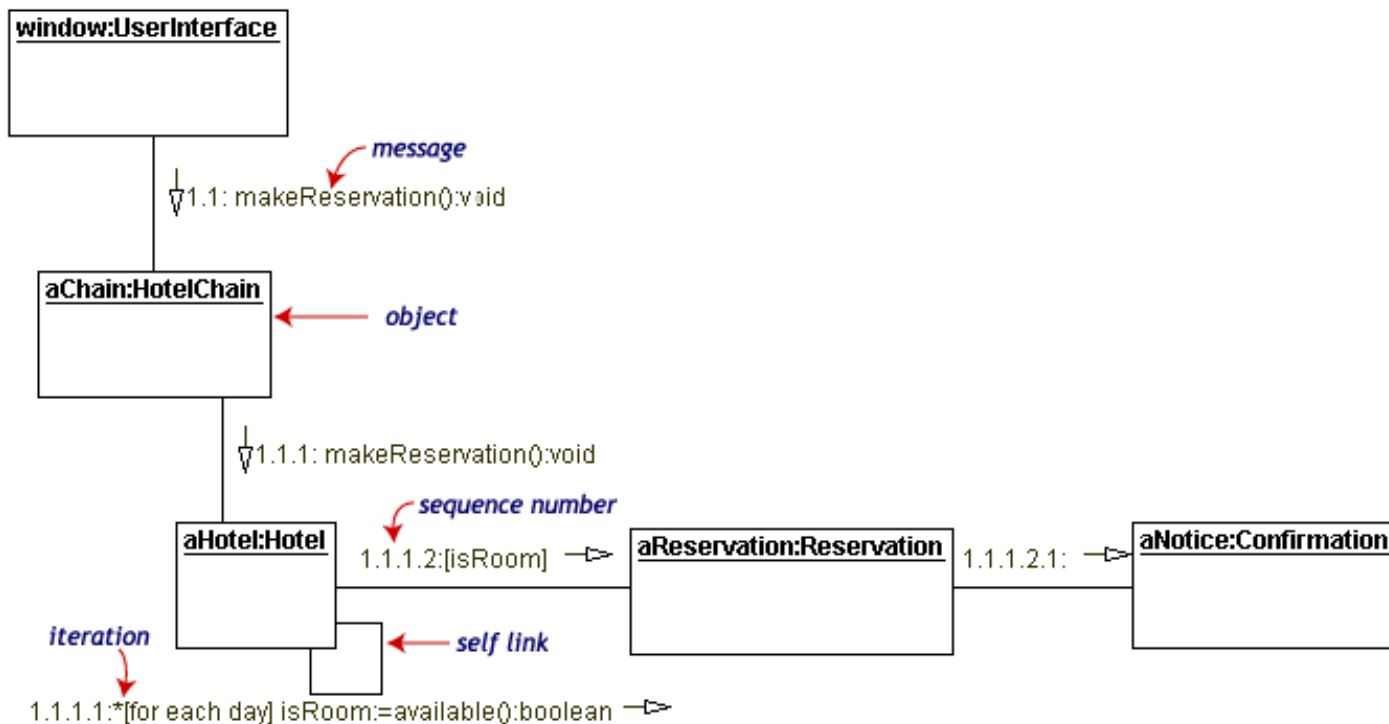
## UML message conventions.



Symbol	Meaning
	simple message which may be synchronous or asynchronous
	simple message return (optional)
	a synchronous message
	an asynchronous message

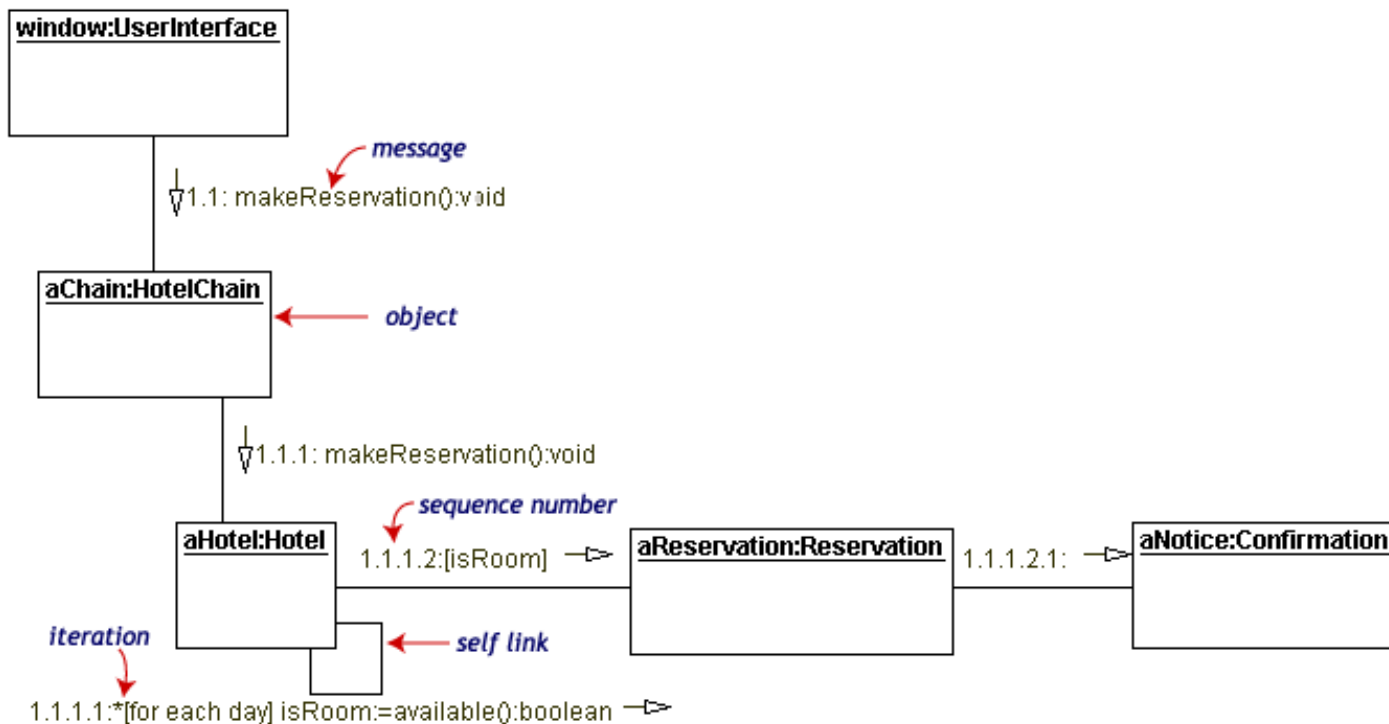
## 6. Collaboration Diagrams

- **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.
- In a sequence diagram, object roles are the vertices, and messages are the connecting links.



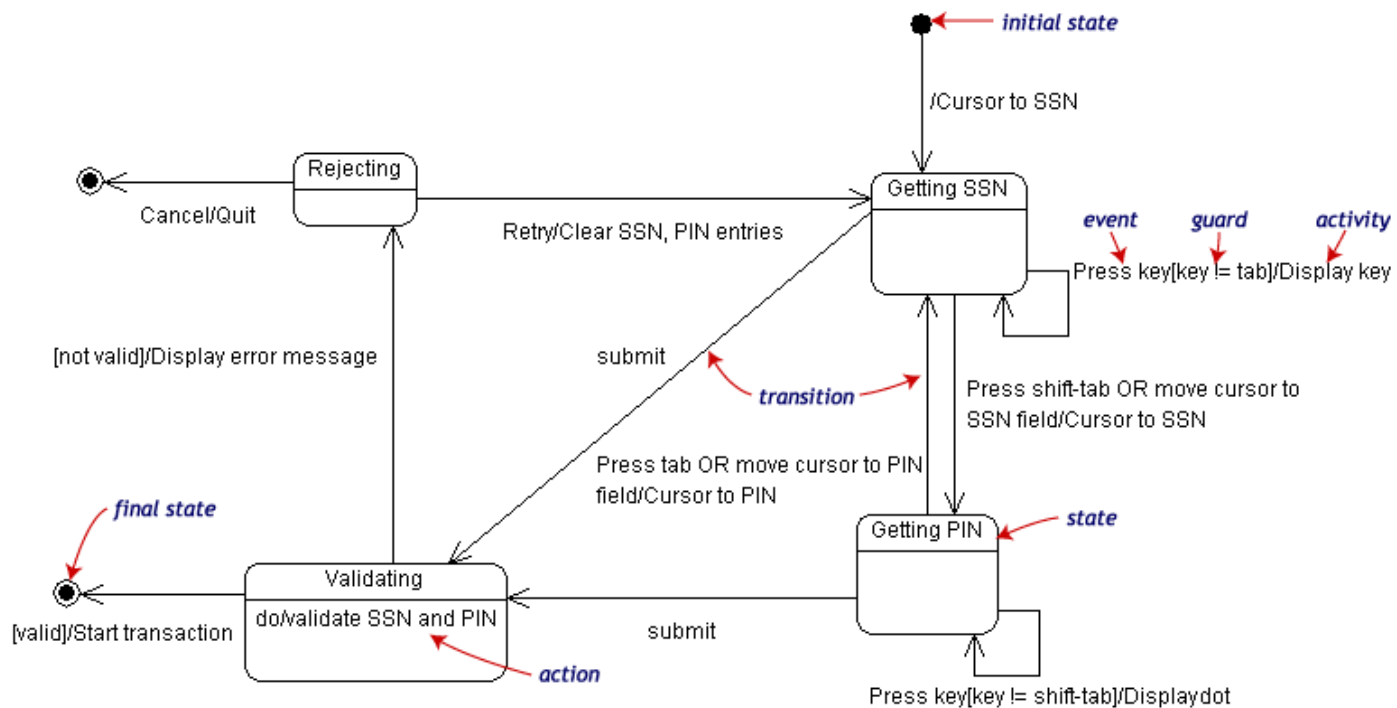
# Collaboration Diagrams

- The **object-role rectangles** are labeled with either class or object names (or both). Class names are preceded by colons (:).  
• Each **message** in a collaboration diagram has a **sequence number**. The top-level message is numbered 1. Messages at the same level (sent during the same call) have the same decimal prefix but suffixes of 1, 2, etc., according to when they occur.



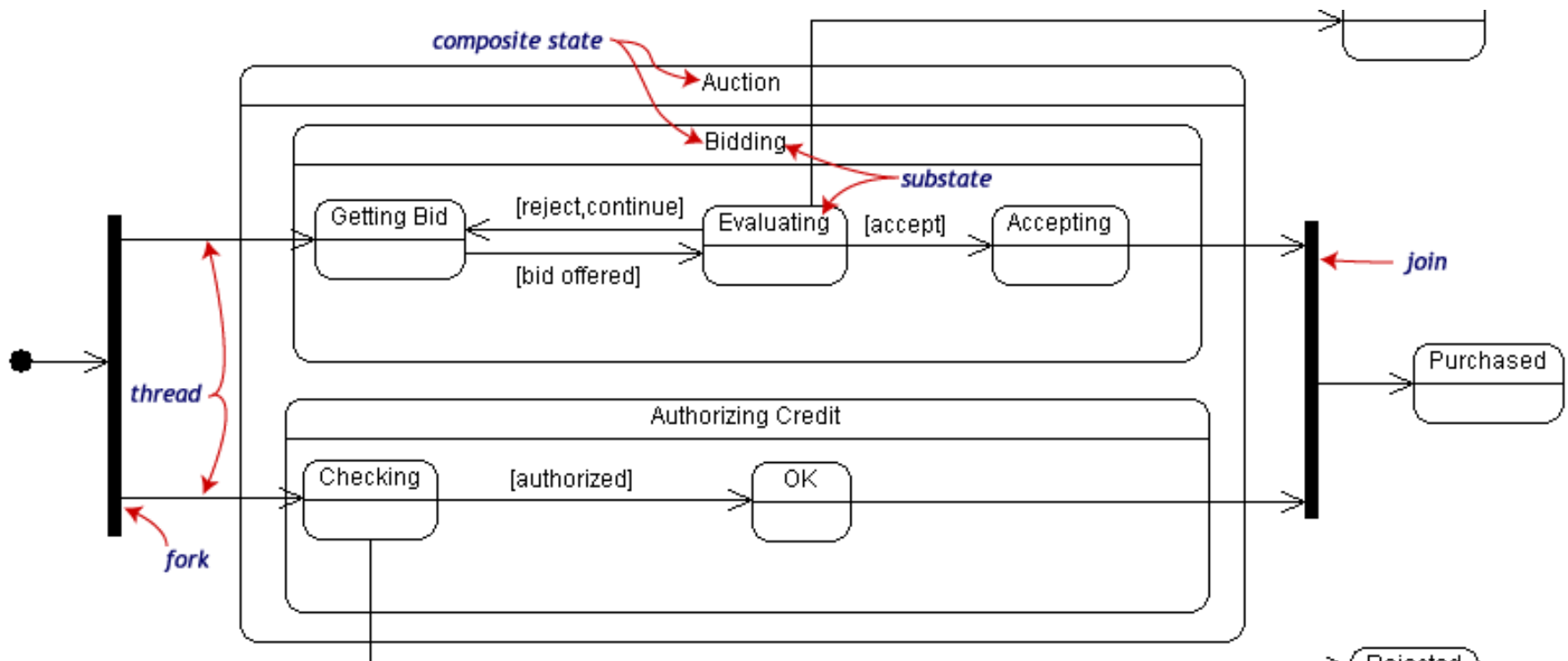
# 7. State chart Diagrams

- Objects have behaviors and states.
- The state of an object depends on its current activity or condition.
- A **state chart diagram** shows the possible states of the object and the transitions that cause a state change.
- From each state comes a complete set of **transitions** that determine the subsequent state.



# State chart Diagrams – concurrency and synchronization

- States in a state chart diagram can be nested. Related states can be grouped into a single **composite state**. Nesting states is necessary when an activity involves concurrent or asynchronous sub-activities.





# 8. Activity Diagrams

- Activity diagrams can be divided into object **swim lanes** that determine which object is responsible for which activity. A single **transition** connects each activity to the next activity.
- A transition may **branch** into two or more mutually exclusive transitions. **Guard expressions** (inside [ ]) label the transitions from 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.

