SOFTWARE ENGINEERING

Introduction to Unified Modeling Language (UML)

WHAT IS UML AND WHY WE USE UML?

- UML → “Unified Modeling Language”

- Modeling: Describing a software system at a high level of abstraction

- Unified: UML has become a world standard

  Object Management Group (OMG): www.omg.org

- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems

- The UML uses mostly graphical notations to express the OO analysis and design of software projects.

- Simplifies the complex process of software design

UML: UNIFIED MODELING LANGUAGE


James Rumbaugh


Grady Booch


Ivar Jacobson

Developed by the "Three Amigos": Grady Booch, Jim Rumbaugh, Ivar Jacobson in 1994-85 at Rational Software

- Each had their own development methodology
- More or less emphasis on notation and process

UML is a notation and a process

- Diagrams and notation from UML 1.3 Definition (http://www.rational.com)

DIAGRAMS

Class diagrams: Represent static structure

Use case diagrams: Sequence of actions a system performs to yield an observable result to an actor

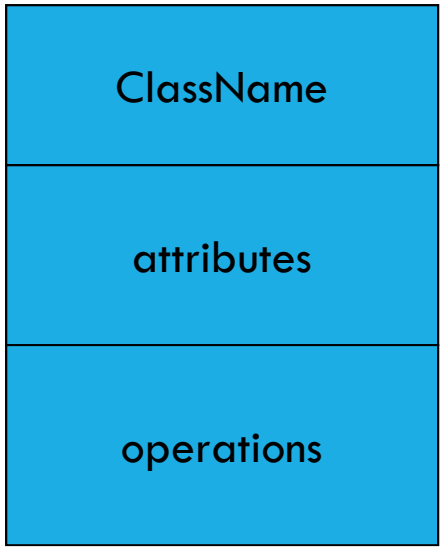
Sequence diagrams: Show how groups of objects interact in some behavior

State diagrams: Describe behavior of system by describing states of an object

Activity diagrams: Activity diagram is a flowchart to represent the flow from one activity to another activity

Collaboration diagrams: Show the message flow between objects in an OO application, and also imply the basic associations (relationships) between classes

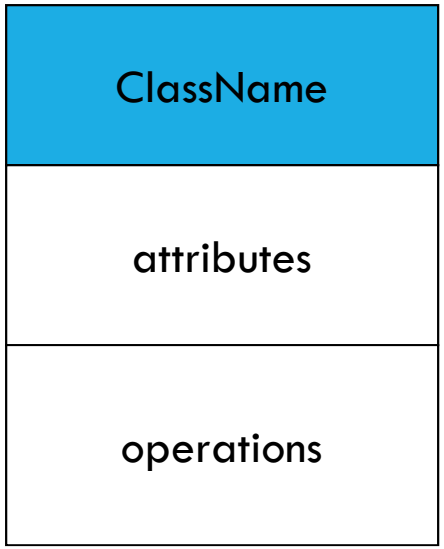
CLASSES



A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

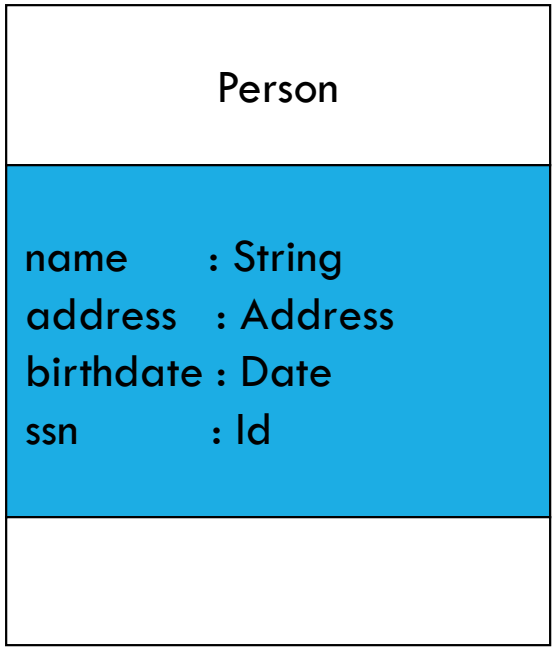
Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

CLASS NAMES



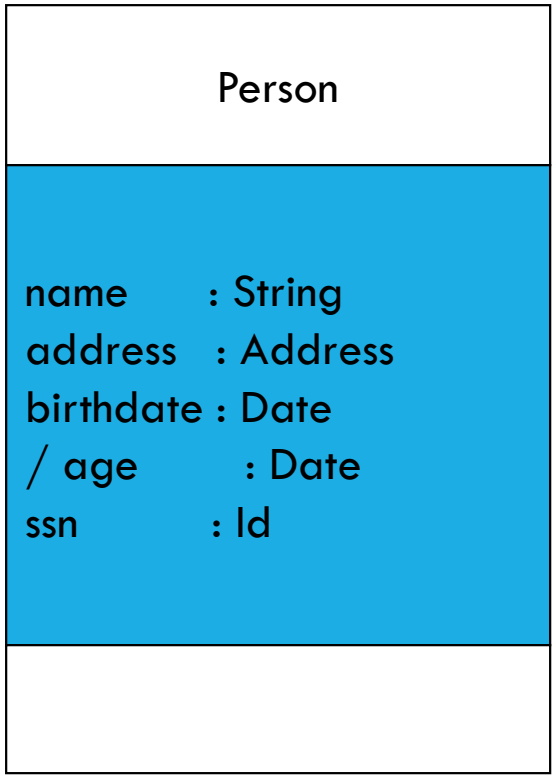
The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

CLASS ATTRIBUTES



An *attribute* is a named property of a class that describes the object being modeled. In the class diagram, attributes appear in the second compartment just below the name-compartment.

CLASS ATTRIBUTES (CONT'D)



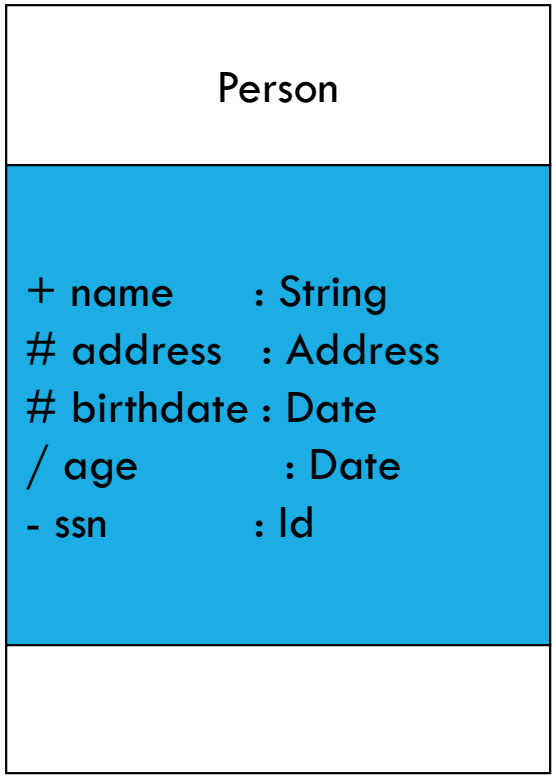
Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:

/ age : Date

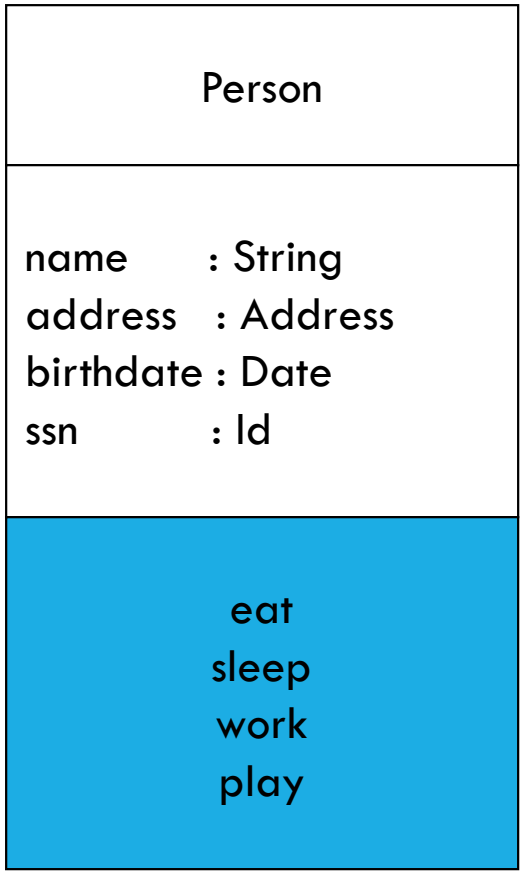
CLASS ATTRIBUTES (CONT'D)



Attributes can be:

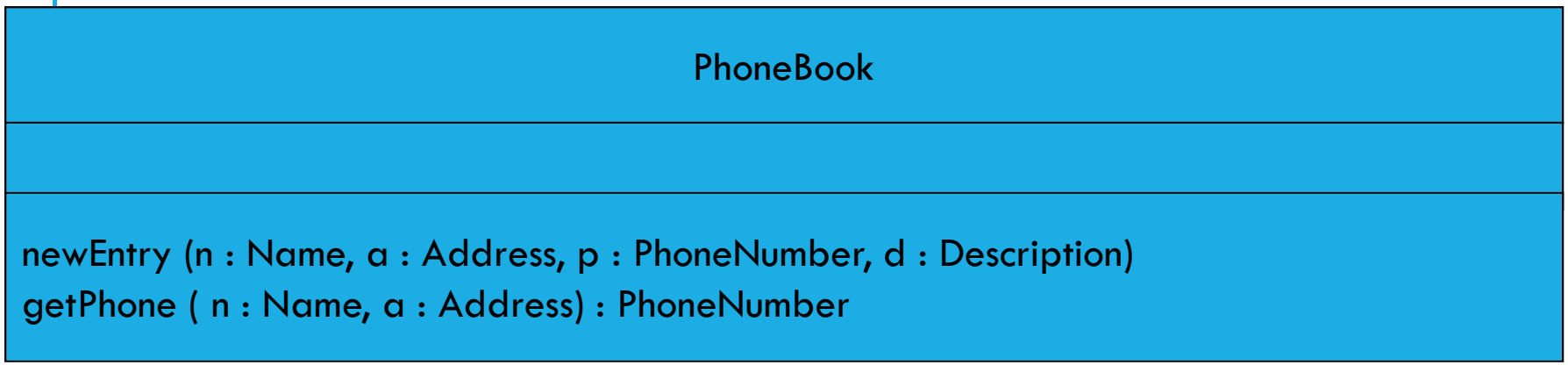
+ public
# protected
- private
/ derived

CLASS OPERATIONS



Operations describe the class behavior and appear in the third compartment.

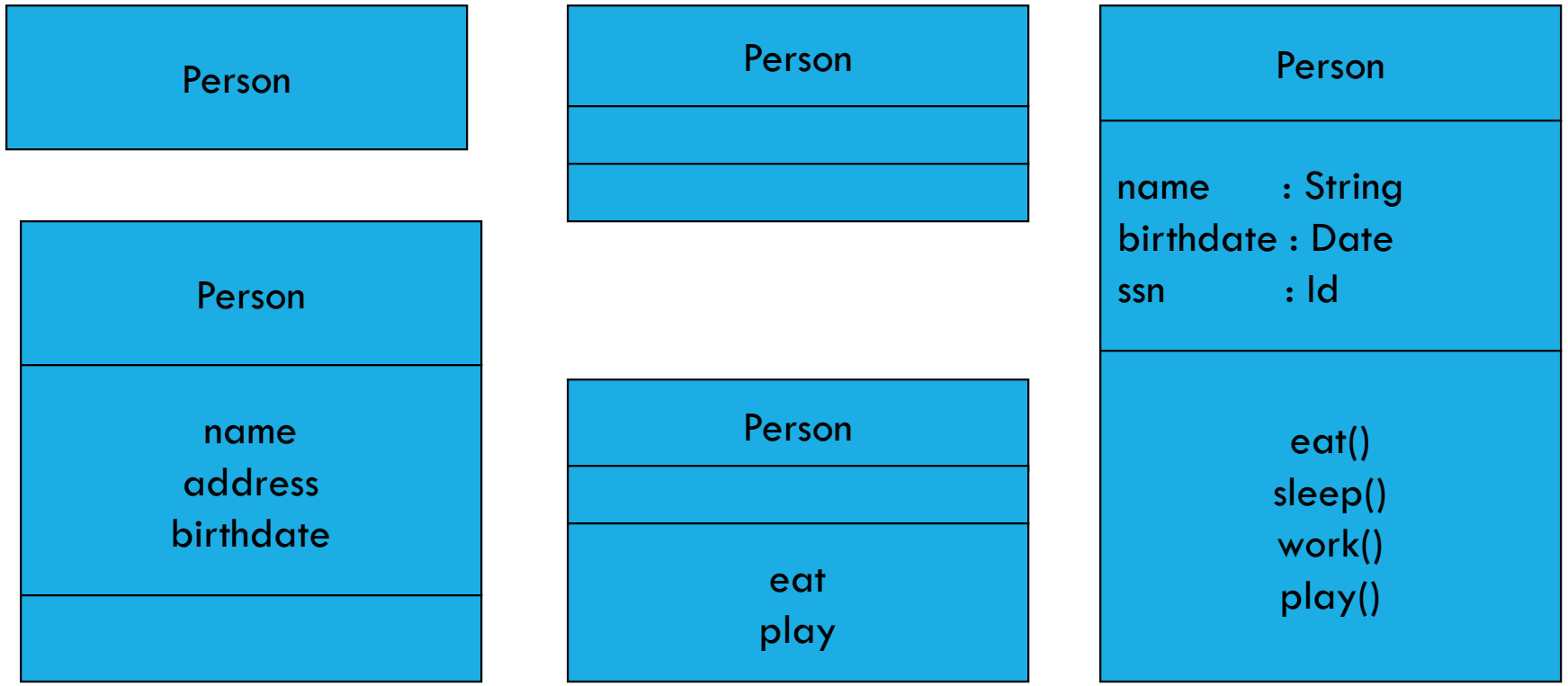
CLASS OPERATIONS (CONT'D)



You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

DEPICTING CLASSES

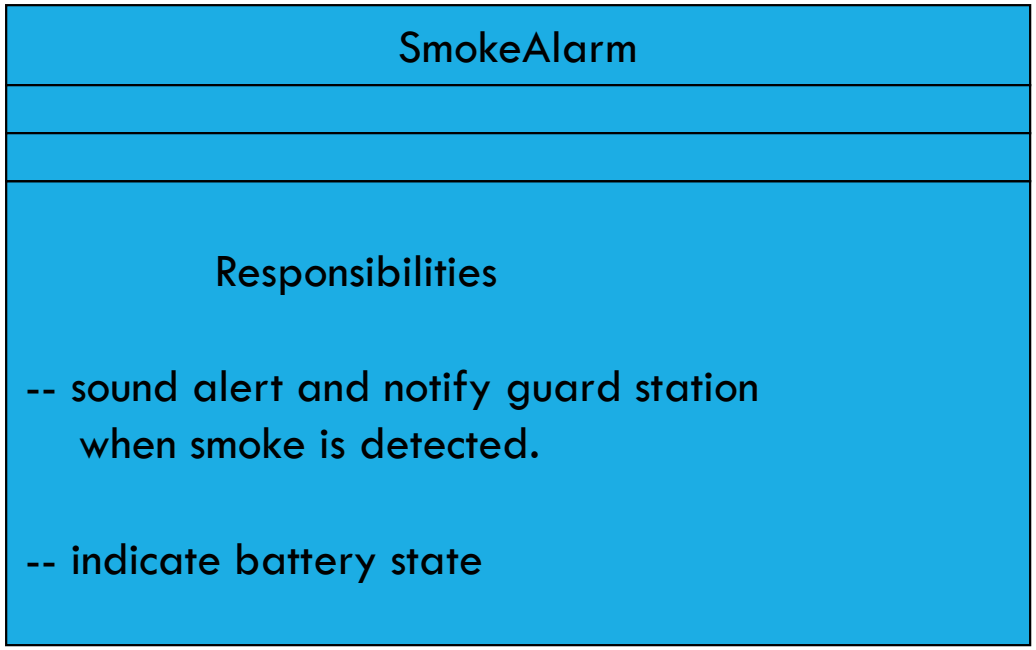
When drawing a class, you needn't show attributes and operation in every diagram.



CLASS RESPONSIBILITIES

A class may also include its responsibilities in a class diagram.

A responsibility is a contract or obligation of a class to perform a particular service.



RELATIONSHIPS

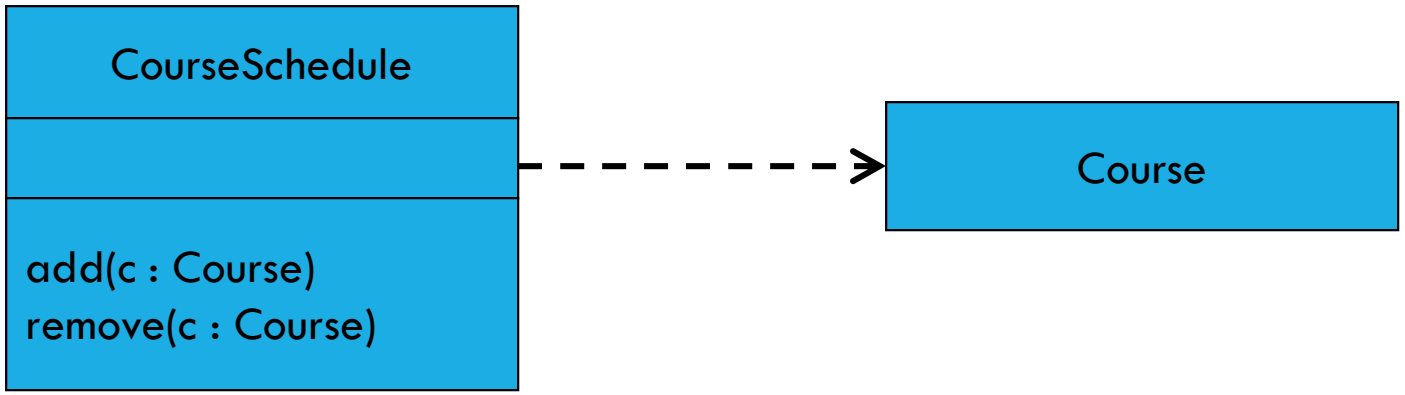
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

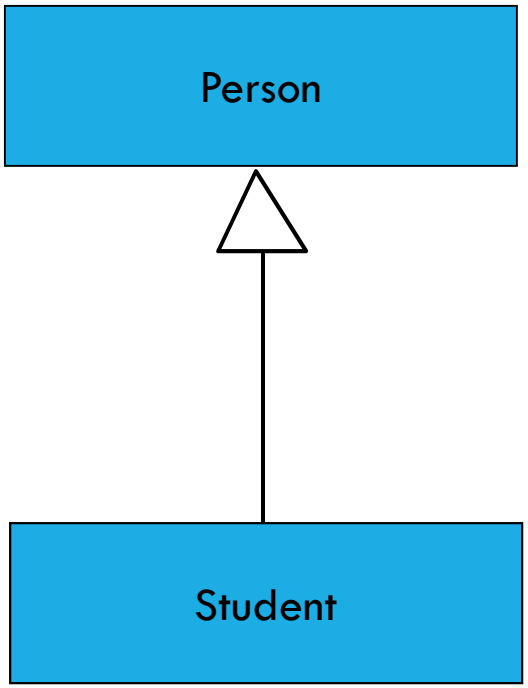
- dependencies
- generalizations
- associations

DEPENDENCY RELATIONSHIPS

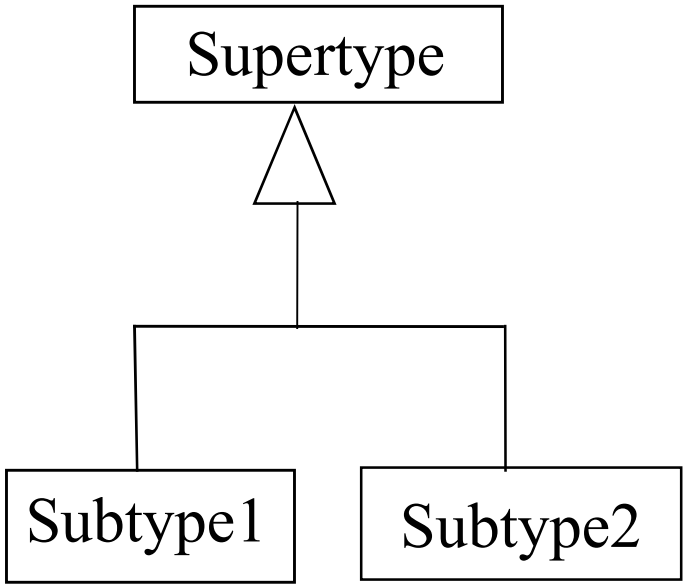
A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



GENERALIZATION RELATIONSHIPS

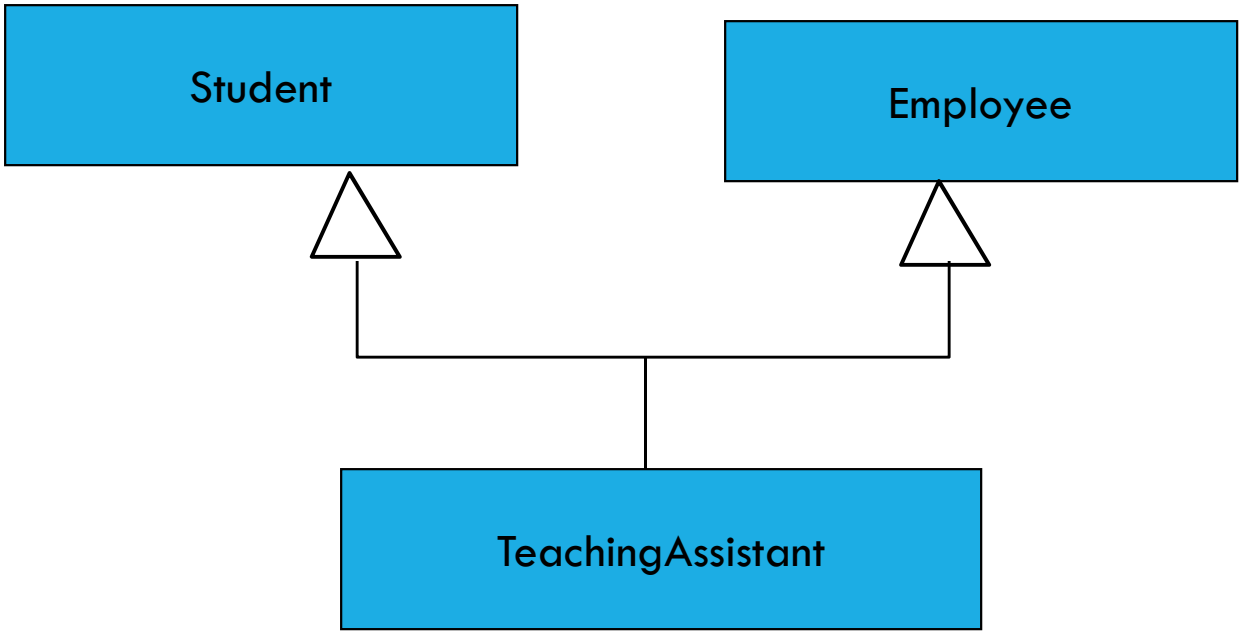


A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.



GENERALIZATION RELATIONSHIPS (CONT'D)

UML permits a class to inherit from multiple superclasses, although some programming languages (e.g., Java) do not permit multiple inheritance.



ASSOCIATION RELATIONSHIPS

If two classes in a model need to communicate with each other, there must be link between them.

An *association* denotes that link.

ASSOCIATION RELATIONSHIPS (CONT'D)

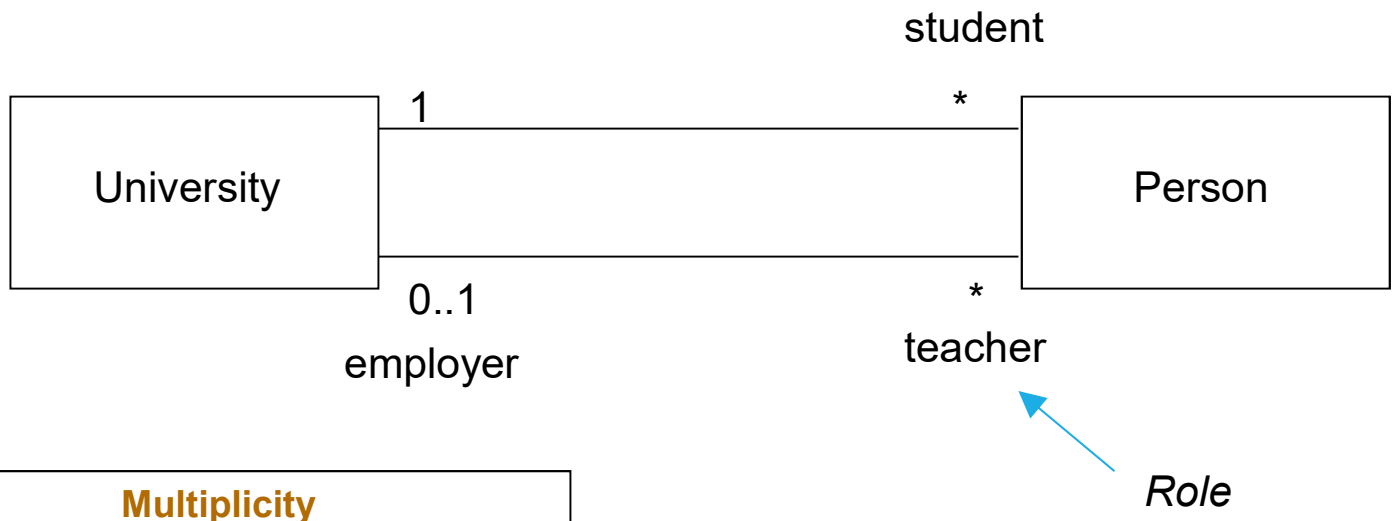
We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

The example indicates that a *Student* has one or more *Instructors*:

ASSOCIATION RELATIONSHIPS (CONT'D)

The example indicates that every *Instructor* has one or more *Students*:

ASSOCIATION: MULTIPLICITY AND ROLES



Multiplicity

Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

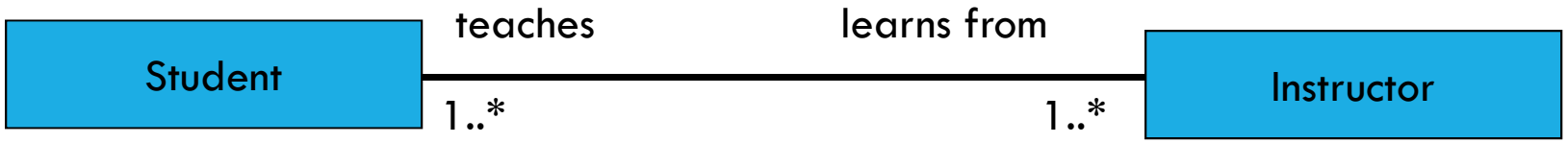
ASSOCIATION: MODEL TO IMPLEMENTATION


```
Class Student {
    Course enrolls[4];
}

Class Course {
 Student have[];
}
```

ASSOCIATION RELATIONSHIPS (CONT'D)

We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.

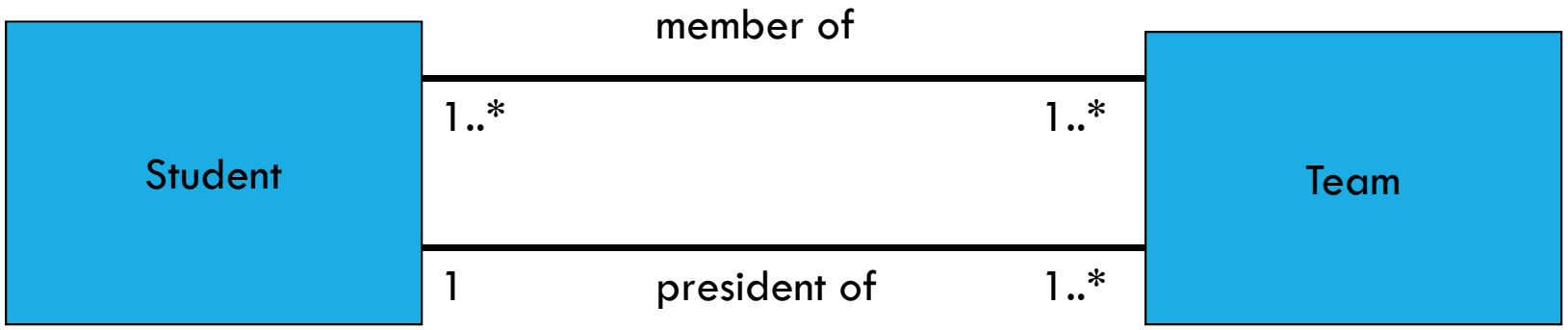


ASSOCIATION RELATIONSHIPS (CONT'D)

We can also name the association.

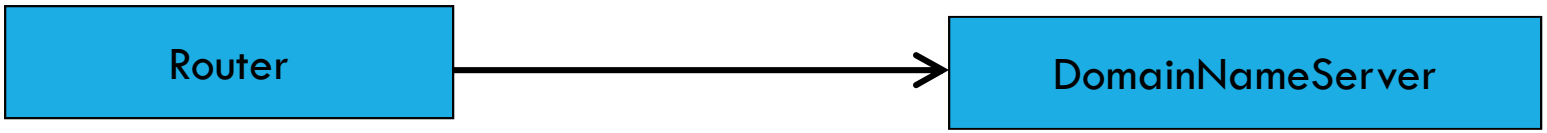
ASSOCIATION RELATIONSHIPS (CONT'D)

We can specify dual associations.



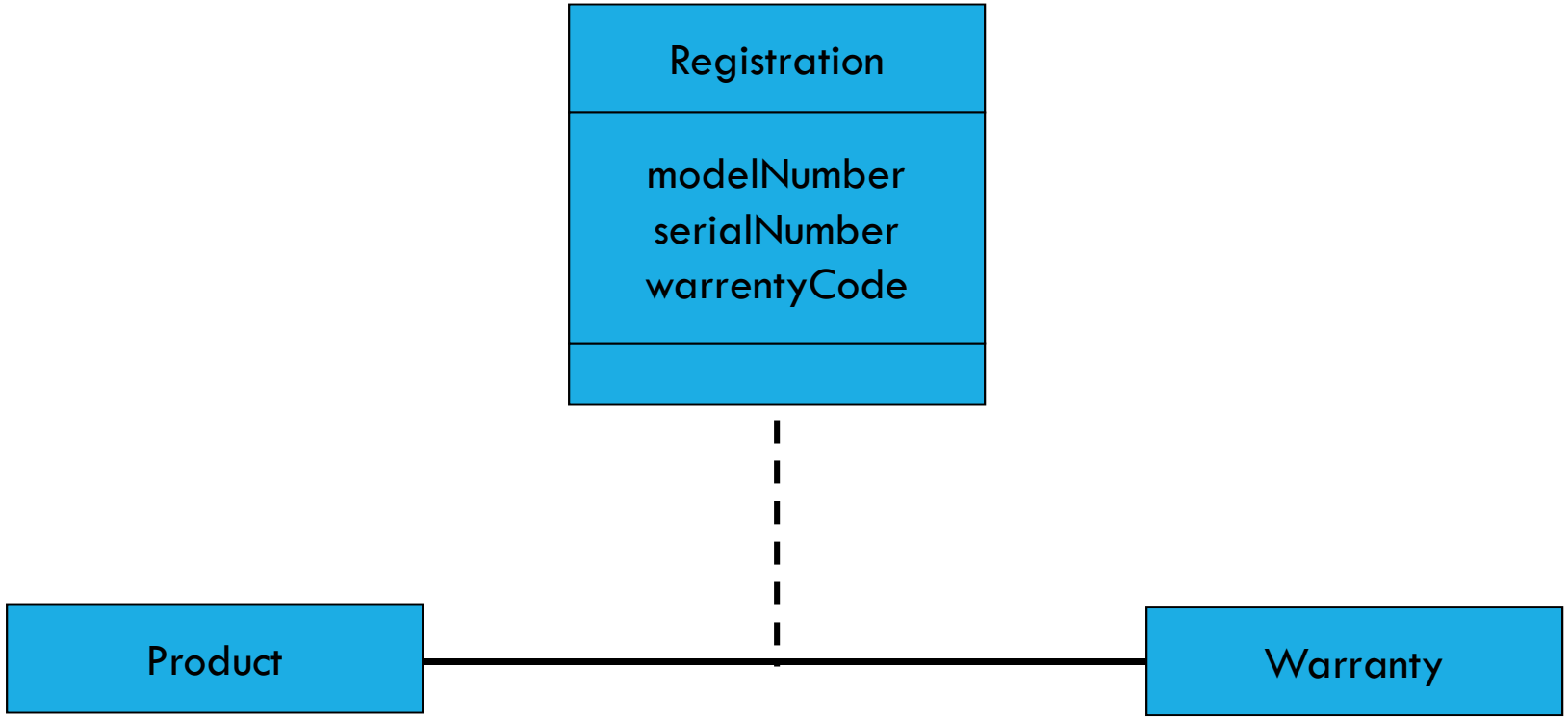
ASSOCIATION RELATIONSHIPS (CONT'D)

We can constrain the association relationship by defining the *navigability* of the association. Here, a *Router* object requests services from a *DNS* object by sending messages to (invoking the operations of) the server. The direction of the association indicates that the server has no knowledge of the *Router*.



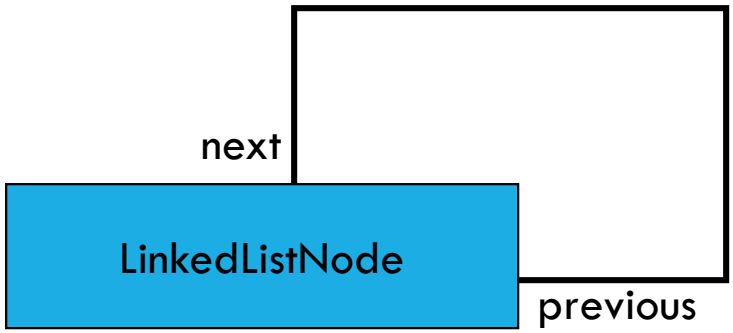
ASSOCIATION RELATIONSHIPS (CONT'D)

Associations can also be objects themselves, called *link classes* or an *association classes*.



ASSOCIATION RELATIONSHIPS (CONT'D)

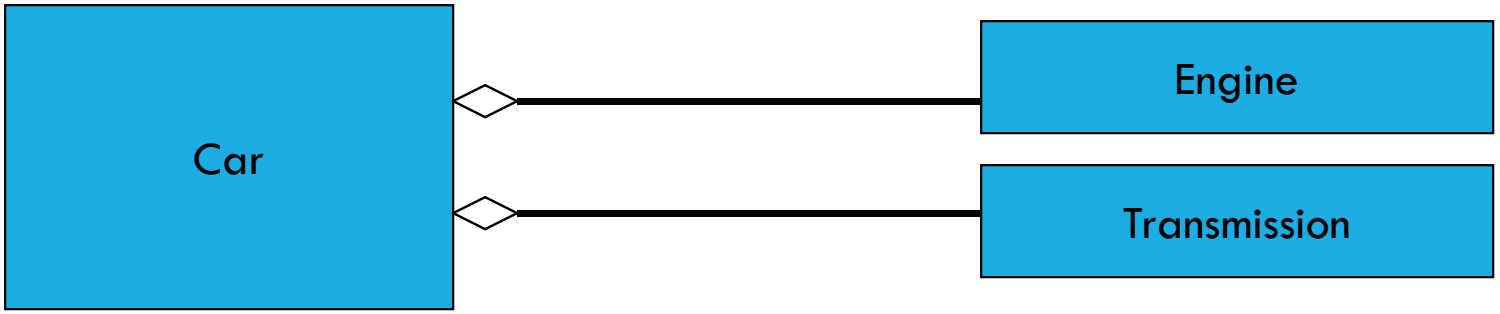
A class can have a *self association*.



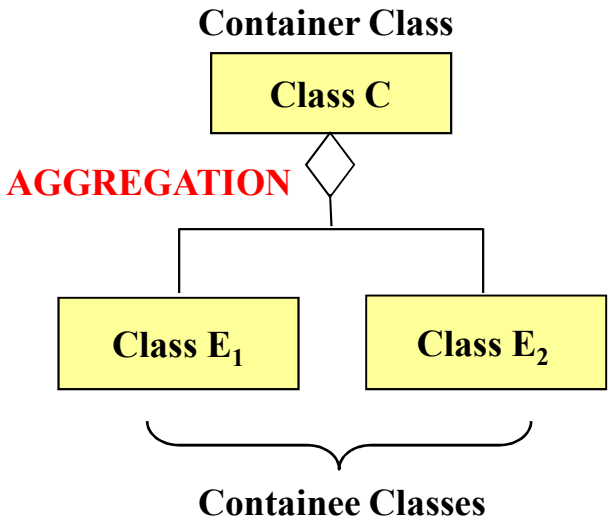
ASSOCIATION RELATIONSHIPS (CONT'D)

We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

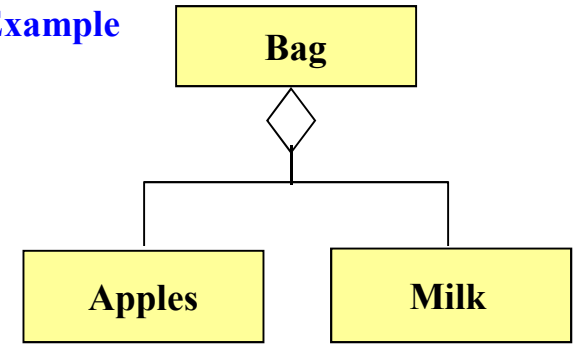
An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.



OO RELATIONSHIPS: **AGGREGATION**



Example



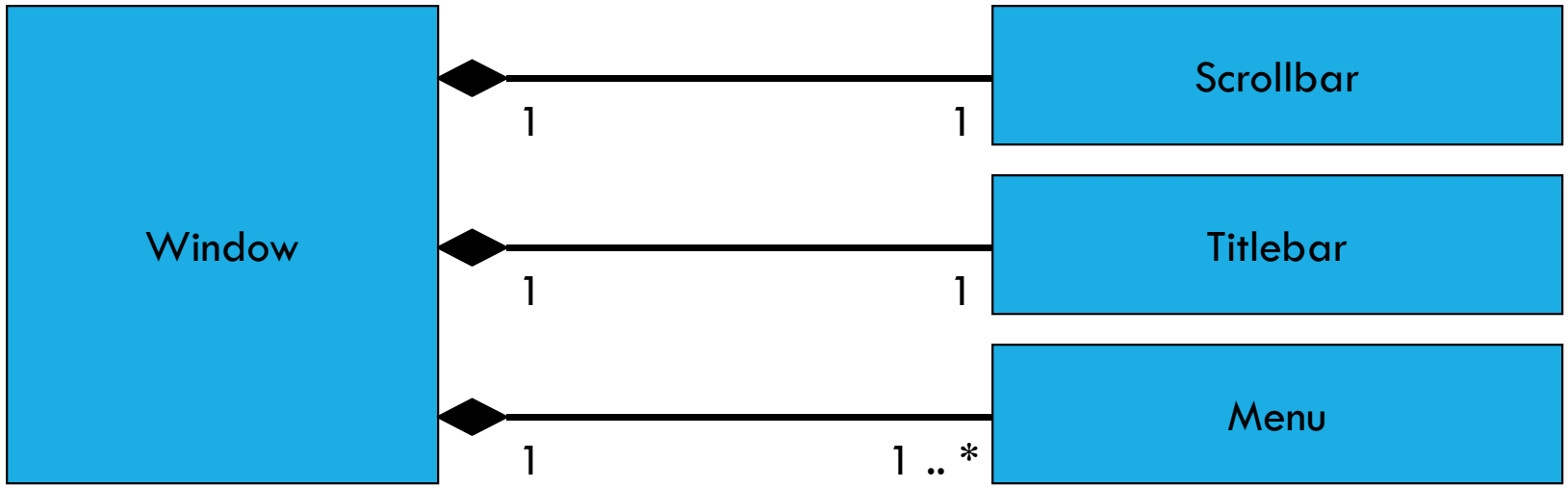
Aggregation:
expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

express a more informal relationship than composition expresses.

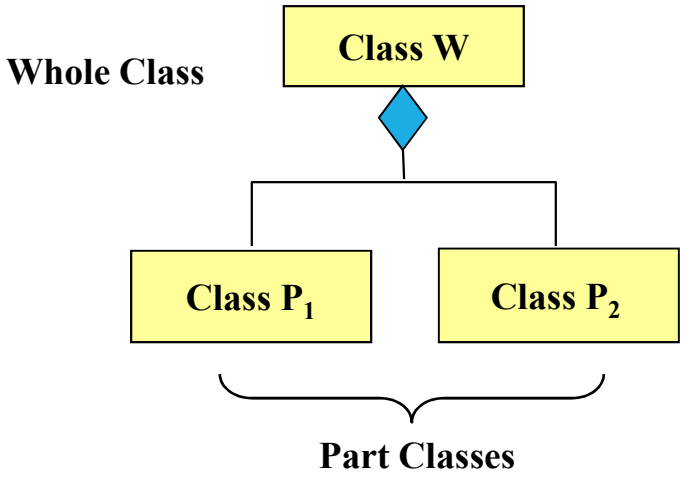
Aggregation is appropriate when Container and Containees have no special access privileges to each other.

ASSOCIATION RELATIONSHIPS (CONT'D)

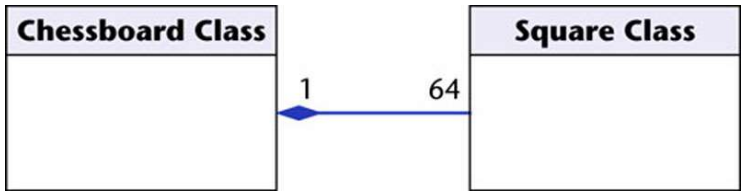
A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



OO Relationships: **Composition**



Example



Association

Models the part–whole relationship

Composition

Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

Example:

A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

Figure 16.7

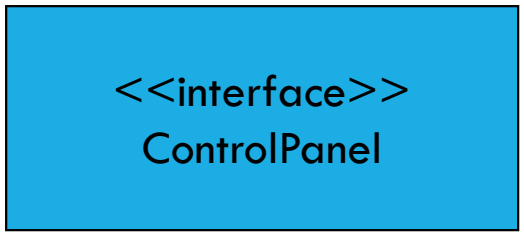
AGGREGATION VS. COMPOSITION

- **Composition** is really a strong form of **association**
  - components have only one owner
  - components cannot exist independent of their owner
  - components live or die with their owner
  - e.g. Each car has an engine that can not be shared with other cars.

- **Aggregations**

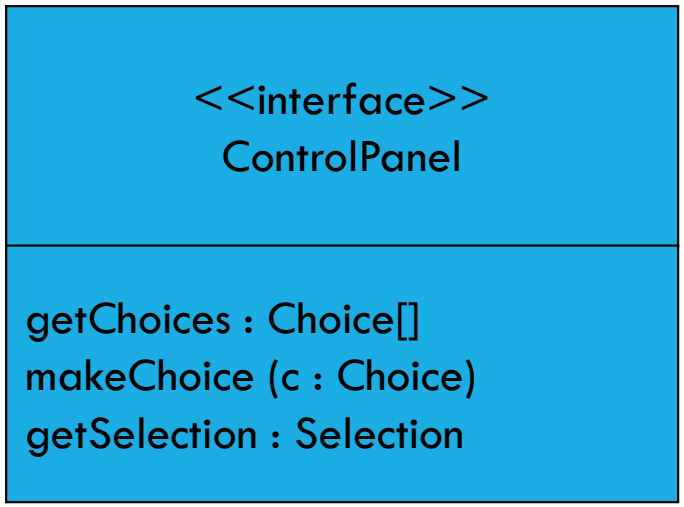
may form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate. e.g. Apples may exist independent of the bag.

INTERFACES



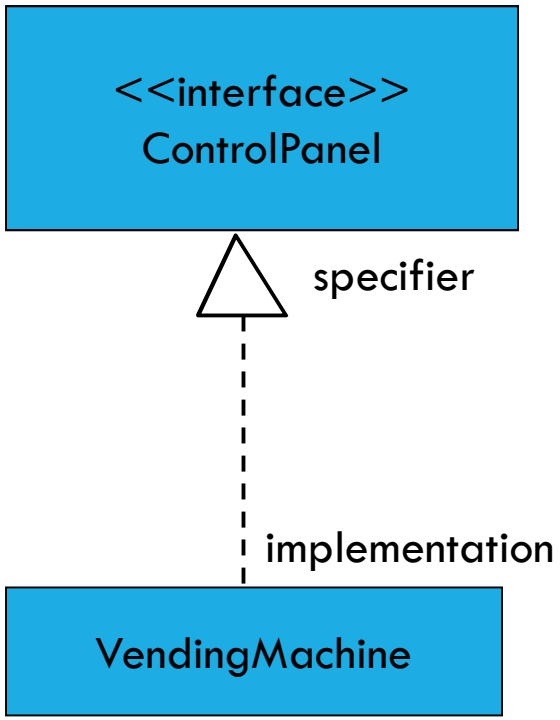
An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure. It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name.

INTERFACE SERVICES



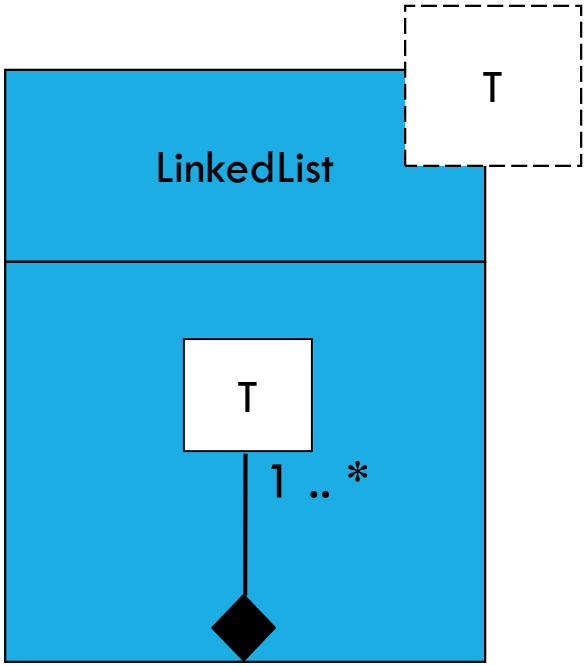
Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.

INTERFACE REALIZATION RELATIONSHIP



A *realization* relationship connects a class with an interface that supplies its behavioral specification. It is rendered by a dashed line with a hollow triangle towards the specifier.

PARAMETERIZED CLASS

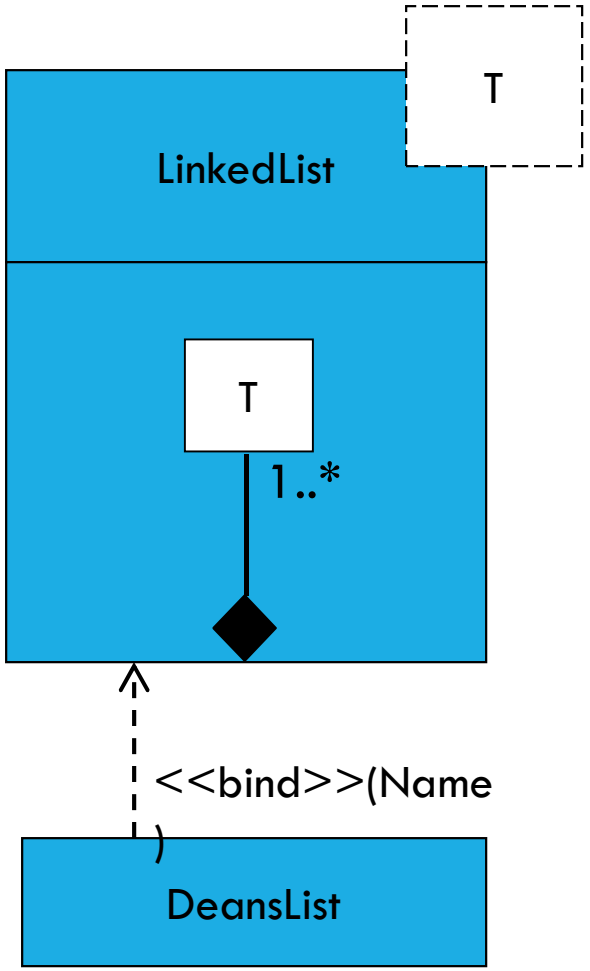


A *parameterized class* or *template* defines a family of potential elements.

To use it, the parameter must be bound.

A *template* is rendered by a small dashed rectangle superimposed on the upper-right corner of the class rectangle. The dashed rectangle contains a list of formal parameters for the class.

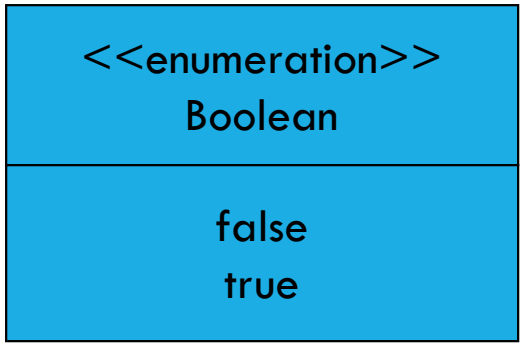
PARAMETERIZED CLASS (CONT'D)



Binding is done with the <<bind>> stereotype and a parameter to supply to the template. These are adornments to the dashed arrow denoting the realization relationship.

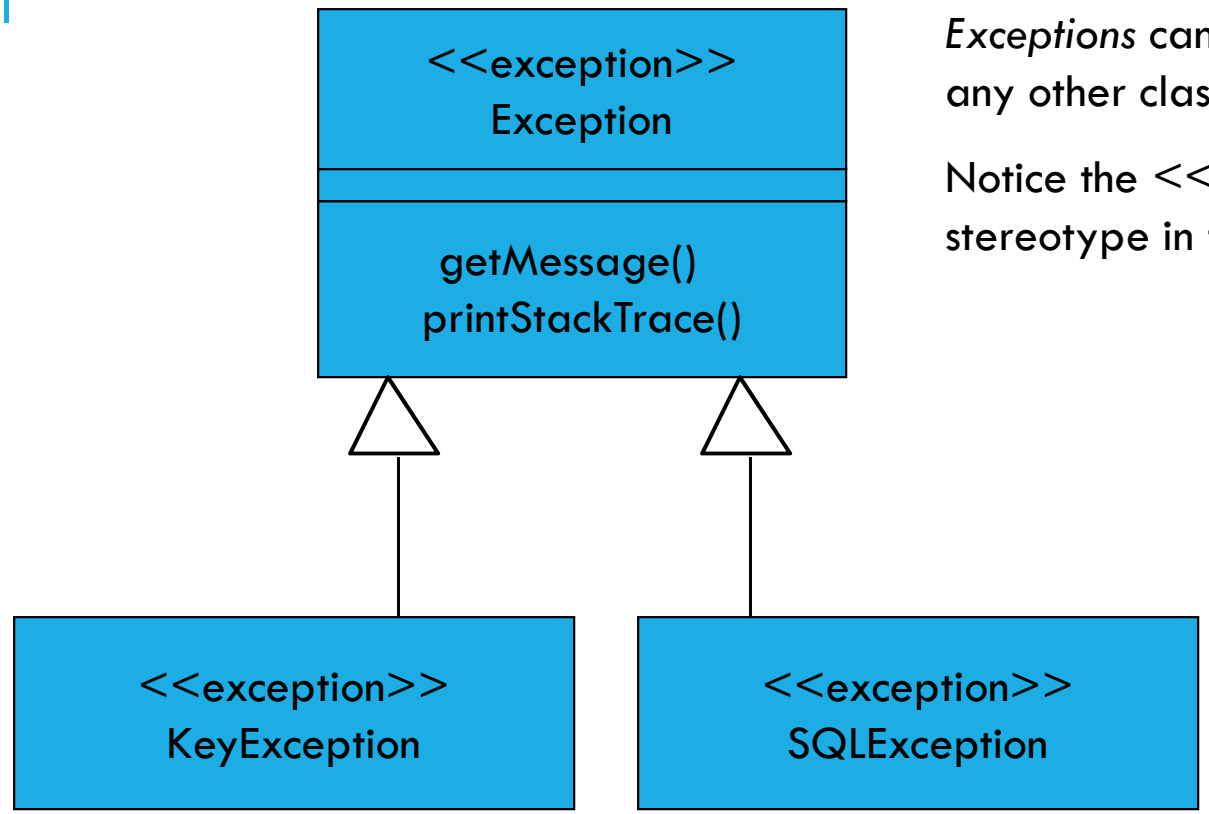
Here we create a linked-list of names for the Dean's List.

ENUMERATION



An *enumeration* is a user-defined data type that consists of a name and an ordered list of enumeration literals.

EXCEPTIONS



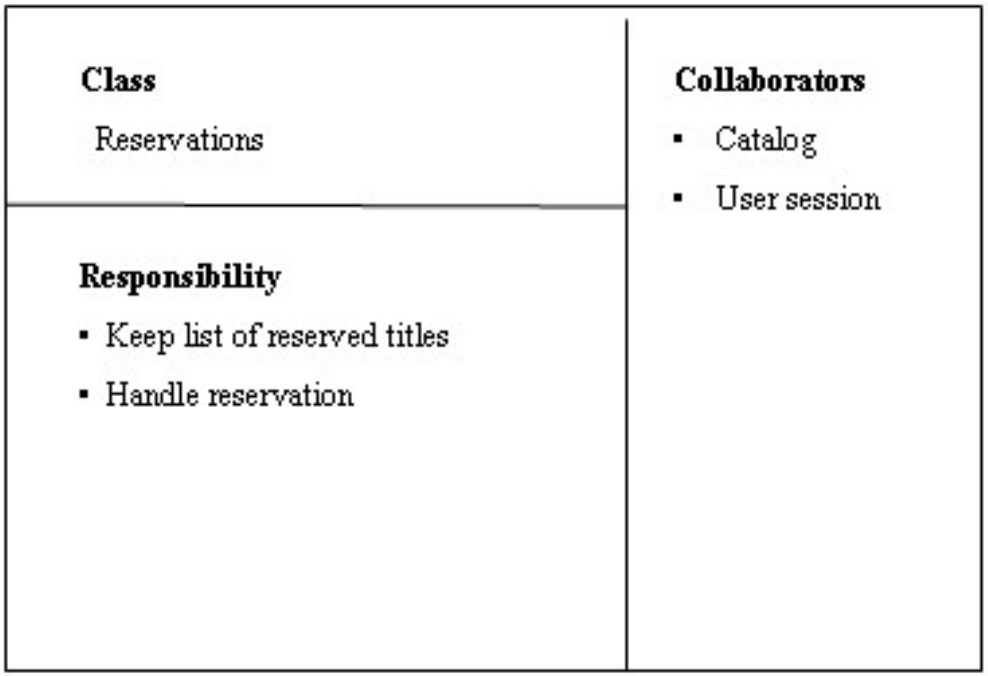
Exceptions can be modeled just like any other class.

Notice the <<exception>> stereotype in the name compartment.

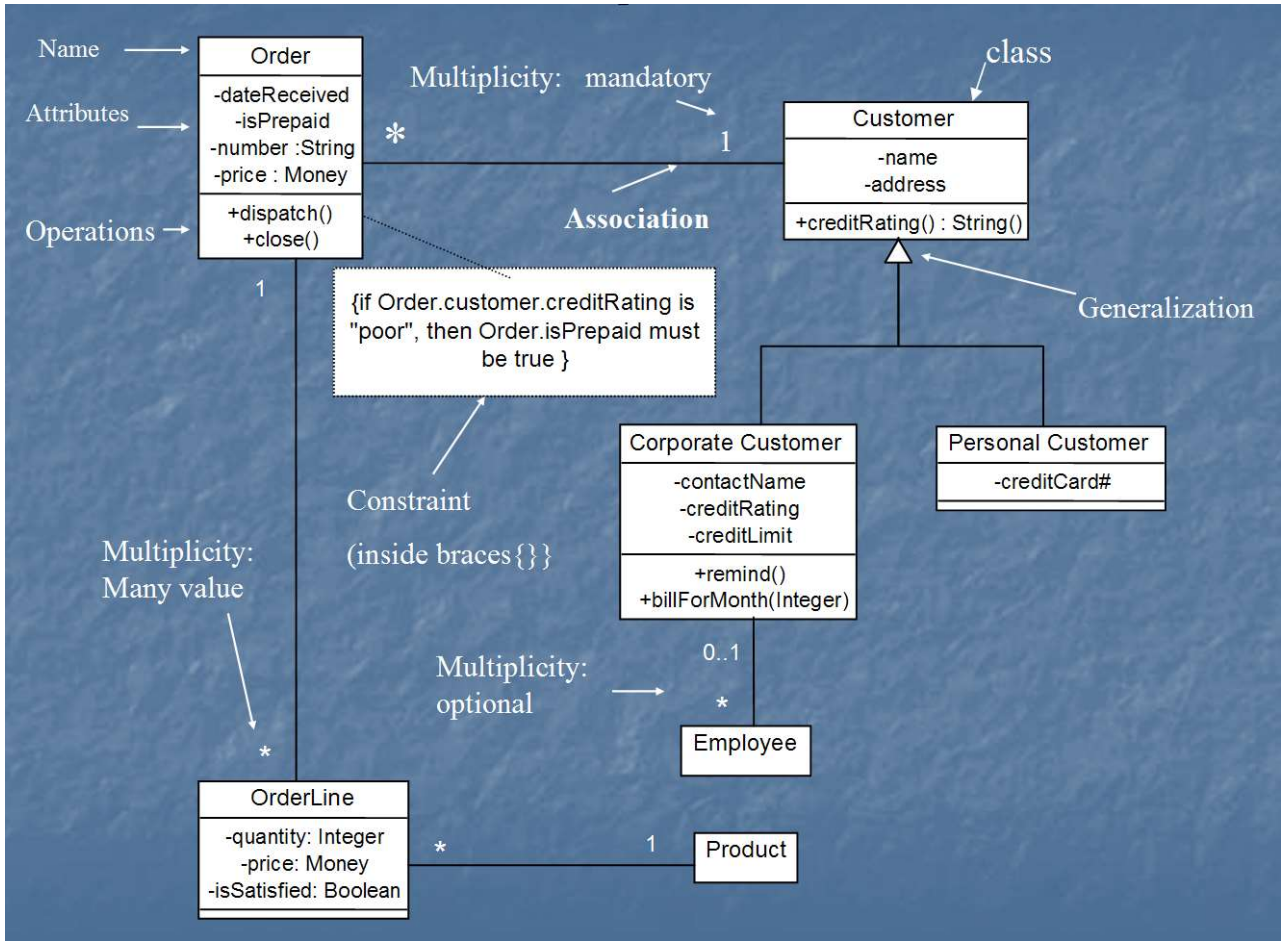
GOOD PRACTICE: CRC CARD

Class Responsibility Collaborator

easy to describe how classes work by moving cards around; allows to quickly consider alternatives.

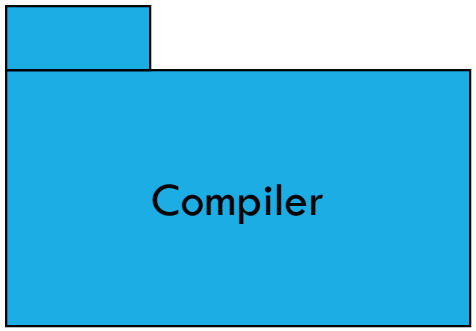


CLASS DIAGRAM



[from *UML Distilled Third Edition*]

PACKAGES



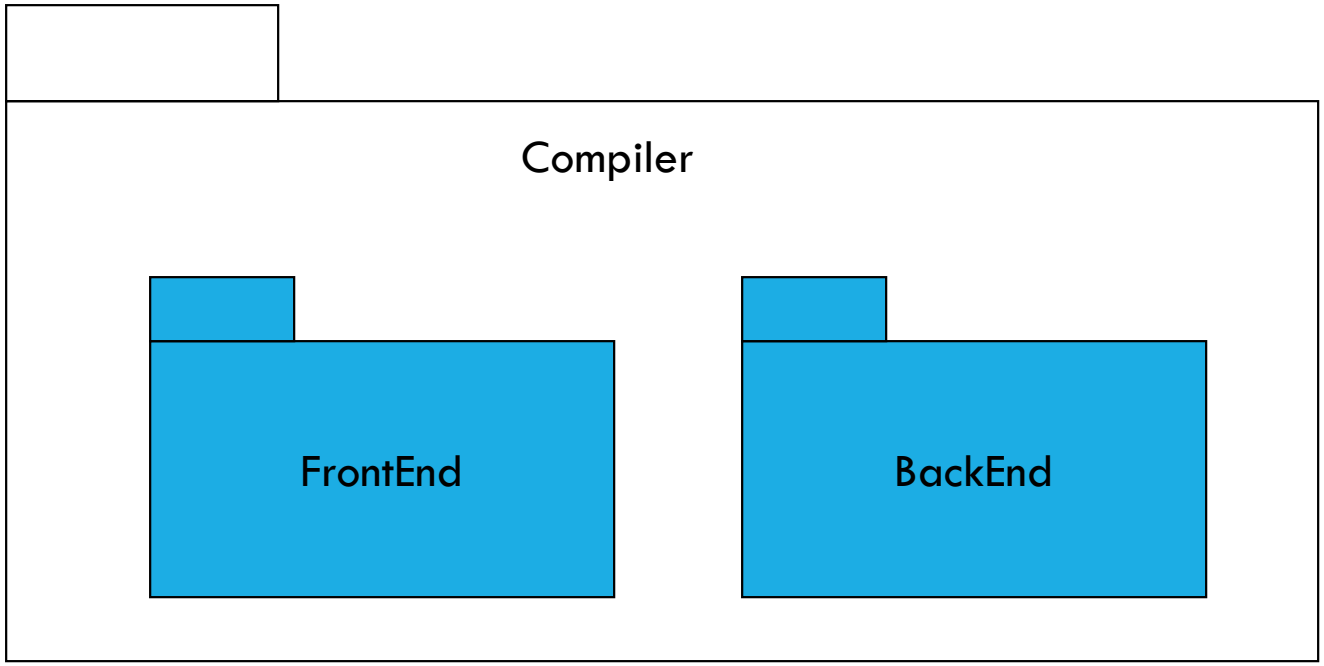
A *package* is a container-like element for organizing other elements into groups.

A package can contain classes and other packages and diagrams.

Packages can be used to provide controlled access between classes in different packages.

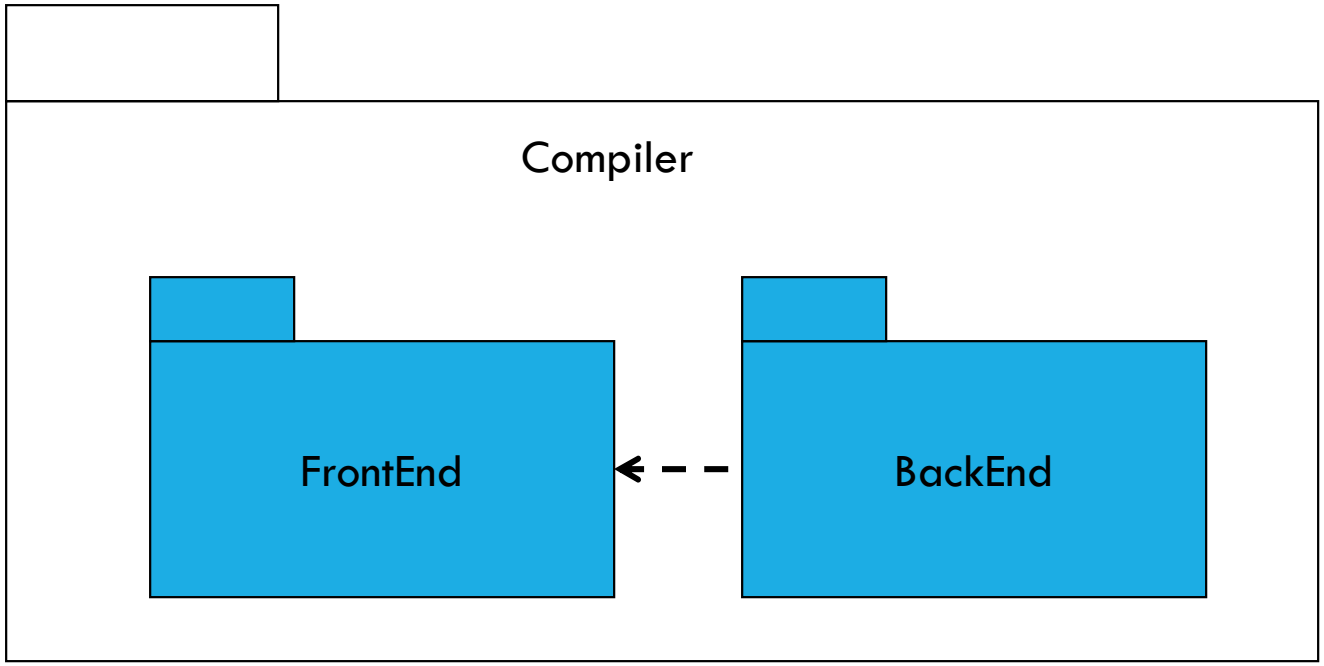
PACKAGES (CONT'D)

Classes in the *FrontEnd* package and classes in the *BackEnd* package cannot access each other in this diagram.

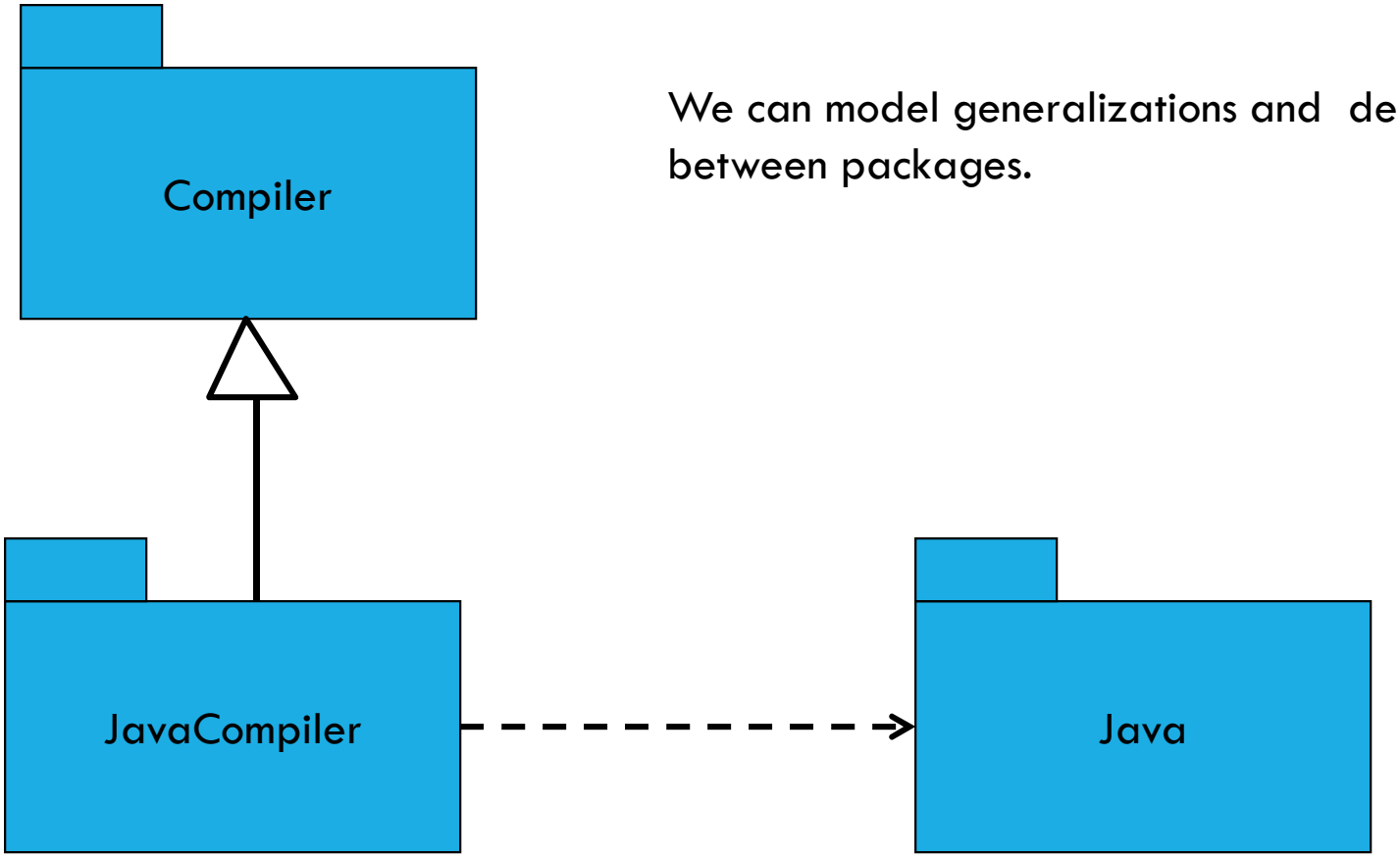


PACKAGES (CONT'D)

Classes in the *BackEnd* package now have access to the classes in the *FrontEnd* package.



PACKAGES (CONT'D)



We can model generalizations and dependencies between packages.

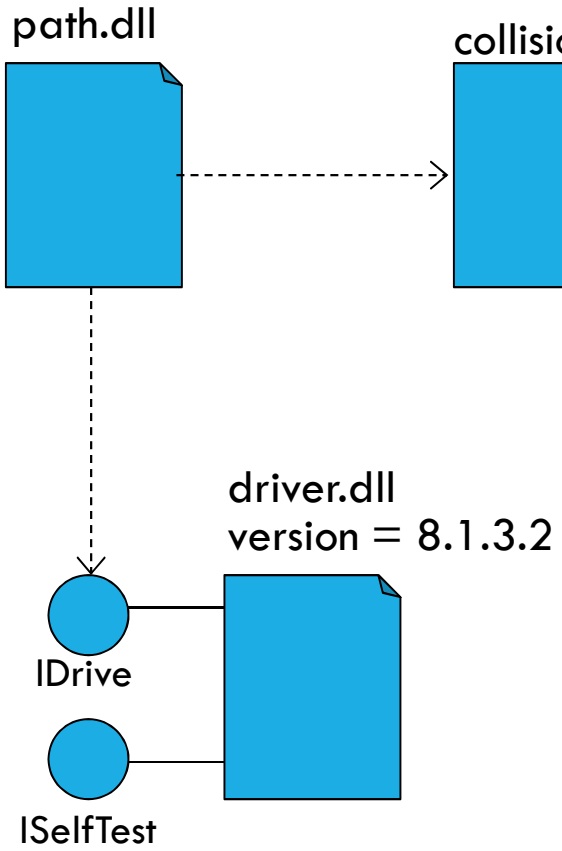
COMPONENT DIAGRAM

Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of an object-oriented system. They show the organization and dependencies between a set of components.

Use component diagrams to model the ***static implementation view*** of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents.

- The UML User Guide, Booch et. al., 1999

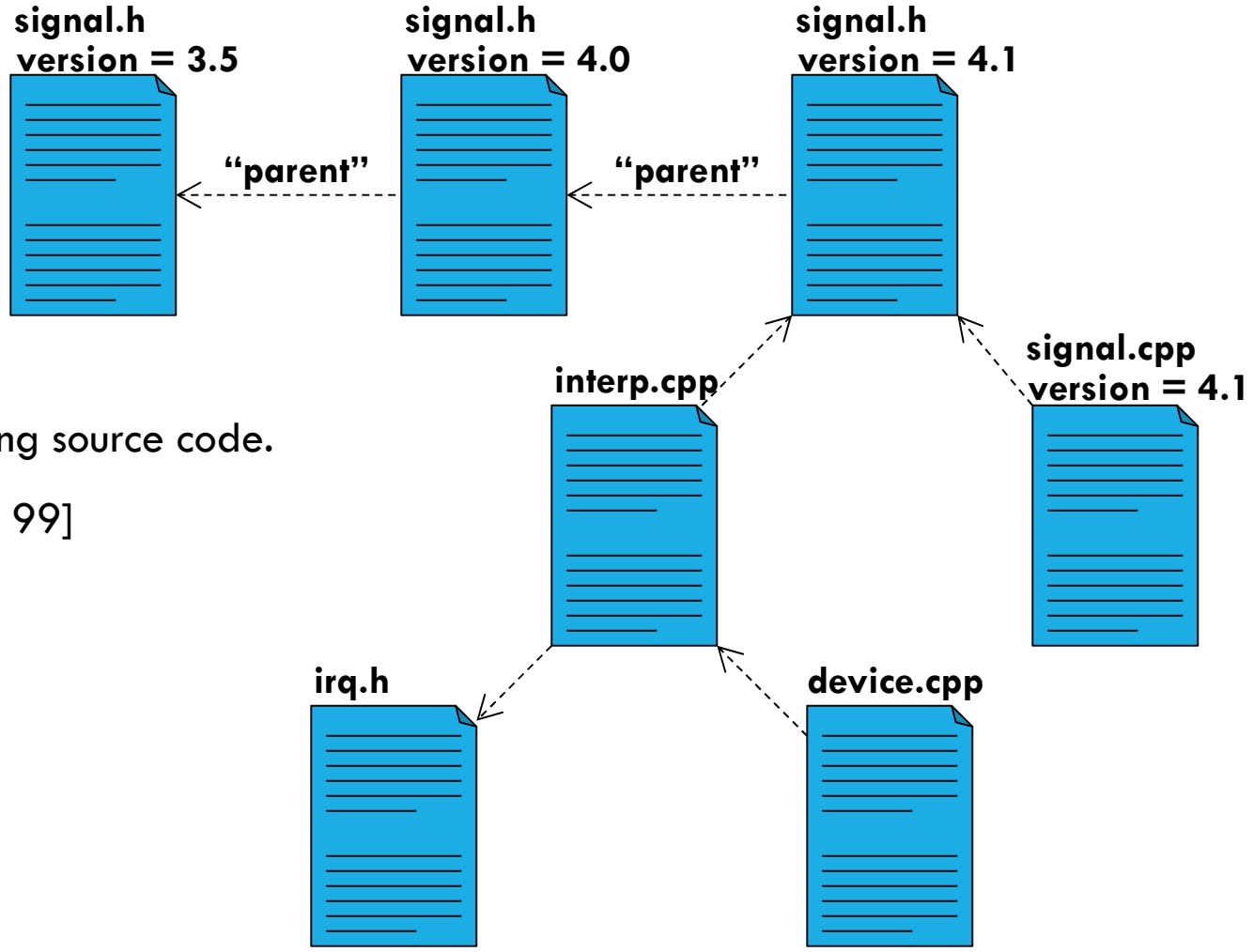
COMPONENT DIAGRAM



Here's an example of a component model of an executable release.

[Booch,99]

COMPONENT DIAGRAM



Modeling source code.

[Booch, 99]

DEPLOYMENT DIAGRAM

Deployment diagrams are one of the two kinds of diagrams found in modeling the physical aspects of an object-oriented system. They show the configuration of ***run-time processing*** nodes and the components that live on them.

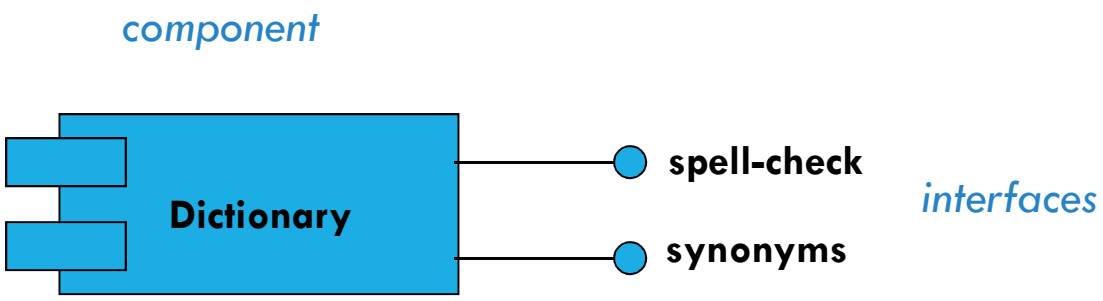
Use deployment diagrams to model the ***static deployment view*** of a system. This involves modeling the topology of the hardware on which the system executes.

- The UML User Guide, [Booch,99]

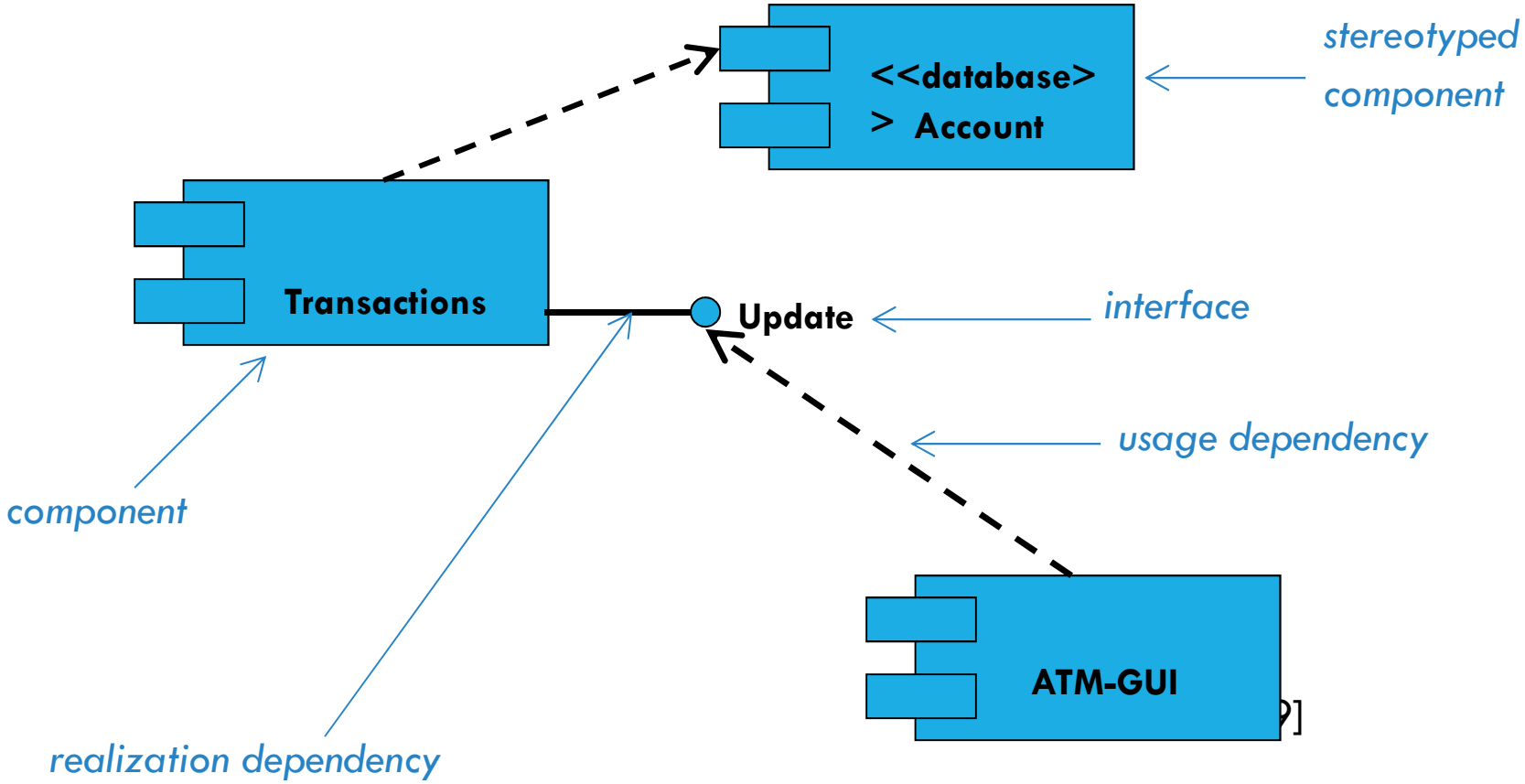
DEPLOYMENT DIAGRAM

A component is a physical unit of implementation with well-defined interfaces that is intended to be used as a replaceable part of a system. Well designed components do not depend directly on other components, but rather on interfaces that components support.

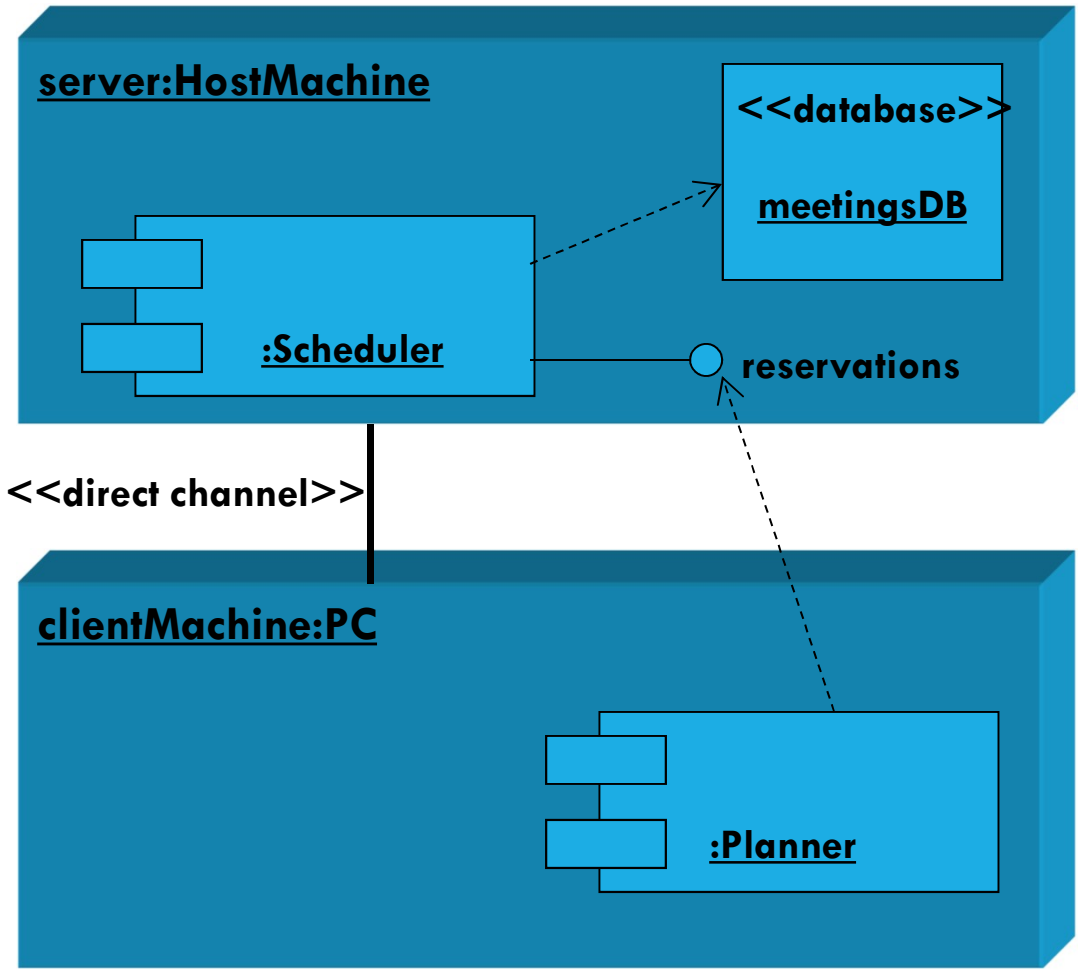
- *The UML Reference Manual, [Rumbaugh,99]*



DEPLOYMENT DIAGRAM



DEPLOYMENT DIAGRAM



Deployment diagram of a client-server system.

[Rumbaugh,99]

USE CASE

"A *use case* specifies the behavior of a system or a part of a system, and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor."

- *The UML User Guide, [Booch,99]*

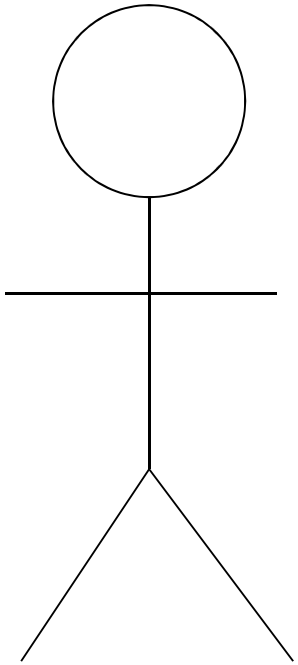
"An *actor* is an idealization of an external person, process, or thing interacting with a system, subsystem, or class. An actor characterizes the interactions that outside users may have with the system."

- *The UML Reference Manual, [Rumbaugh,99]*

USE CASE (CONT'D)

A use case is rendered as an ellipse in a use case diagram. A use case is always labeled with its name.

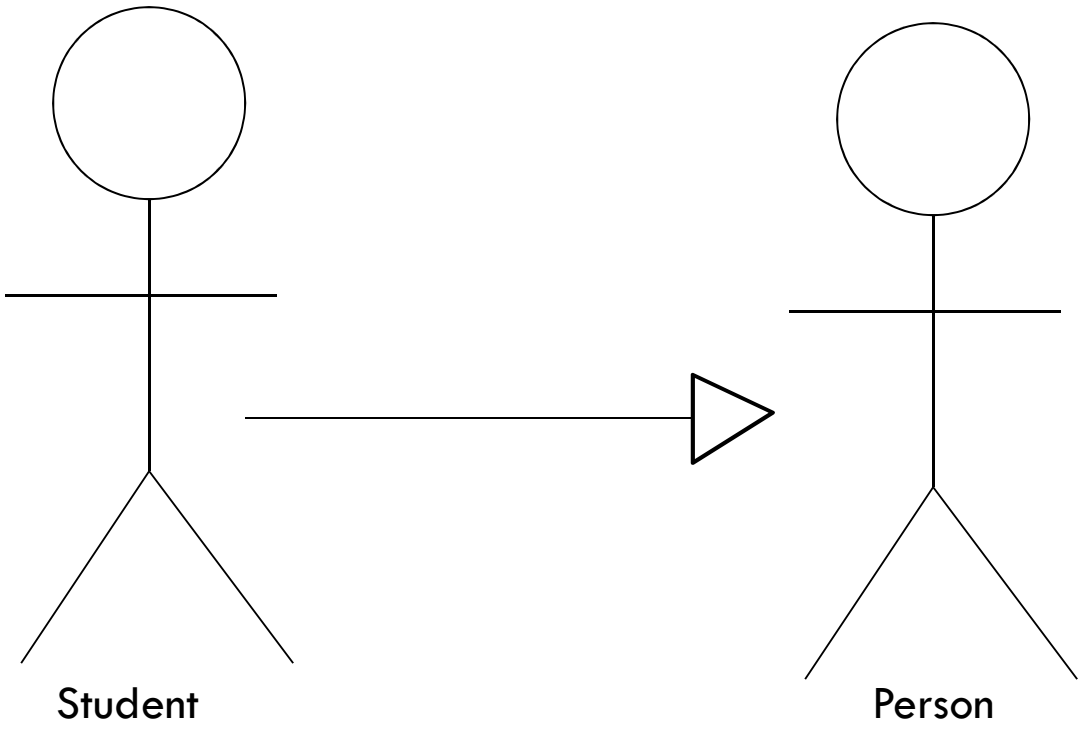
USE CASE (CONT'D)



An actor is rendered as a stick figure in a use case diagram. Each actor participates in one or more use cases.

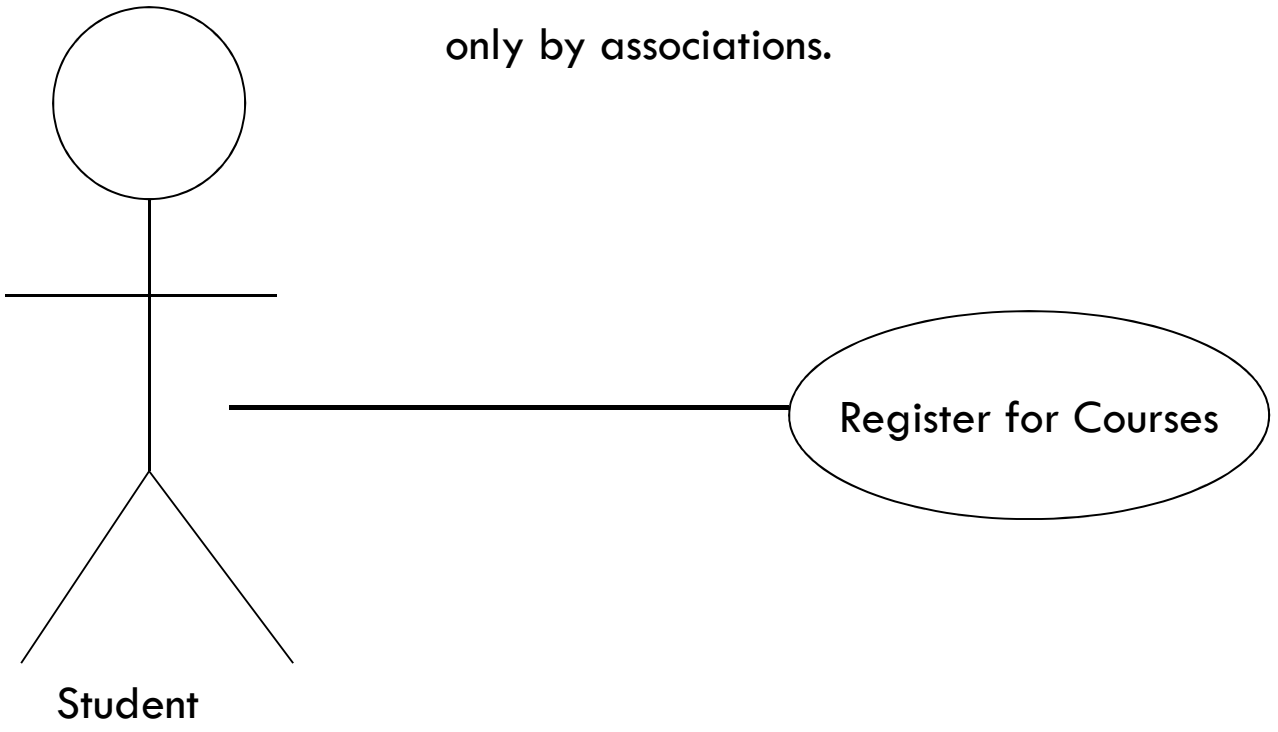
USE CASE (CONT'D)

Actors can participate in a generalization relation with other actors.



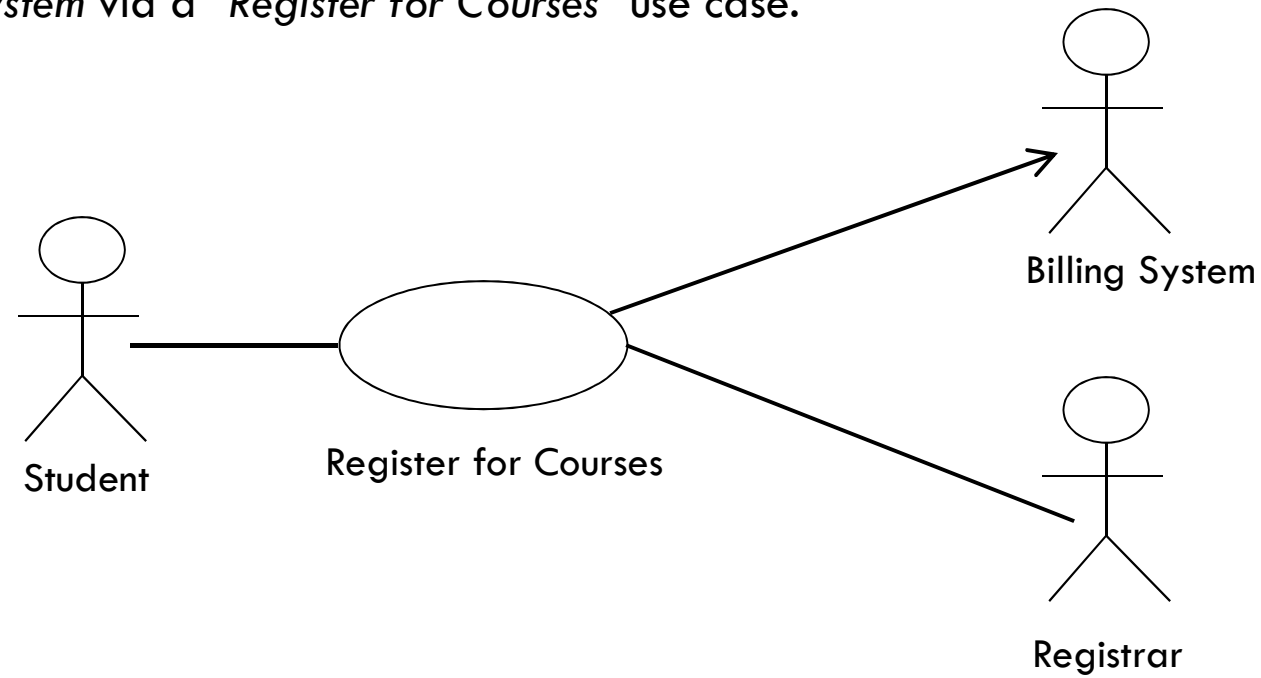
USE CASE (CONT'D)

Actors may be connected to use cases only by associations.



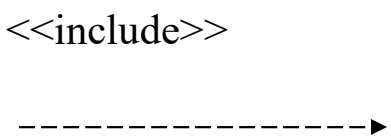
USE CASE (CONT'D)

Here we have a *Student* interacting with the *Registrar* and the *Billing System* via a "*Register for Courses*" use case.

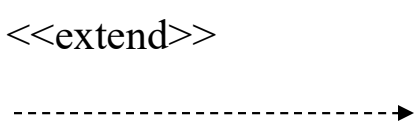


USE-CASE DIAGRAMS

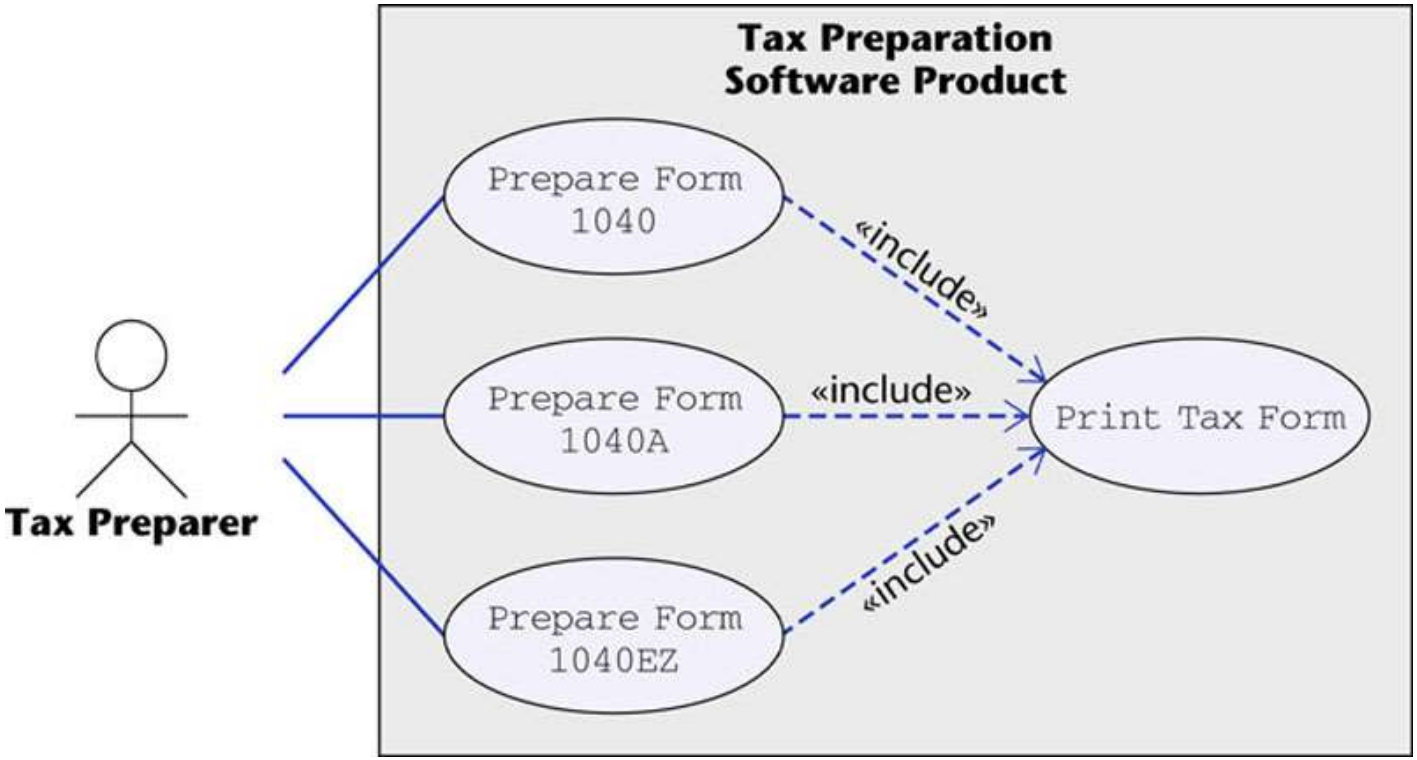
Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use "include" instead of copying the description of that behavior.



Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares "extension points".



USE-CASE DIAGRAMS

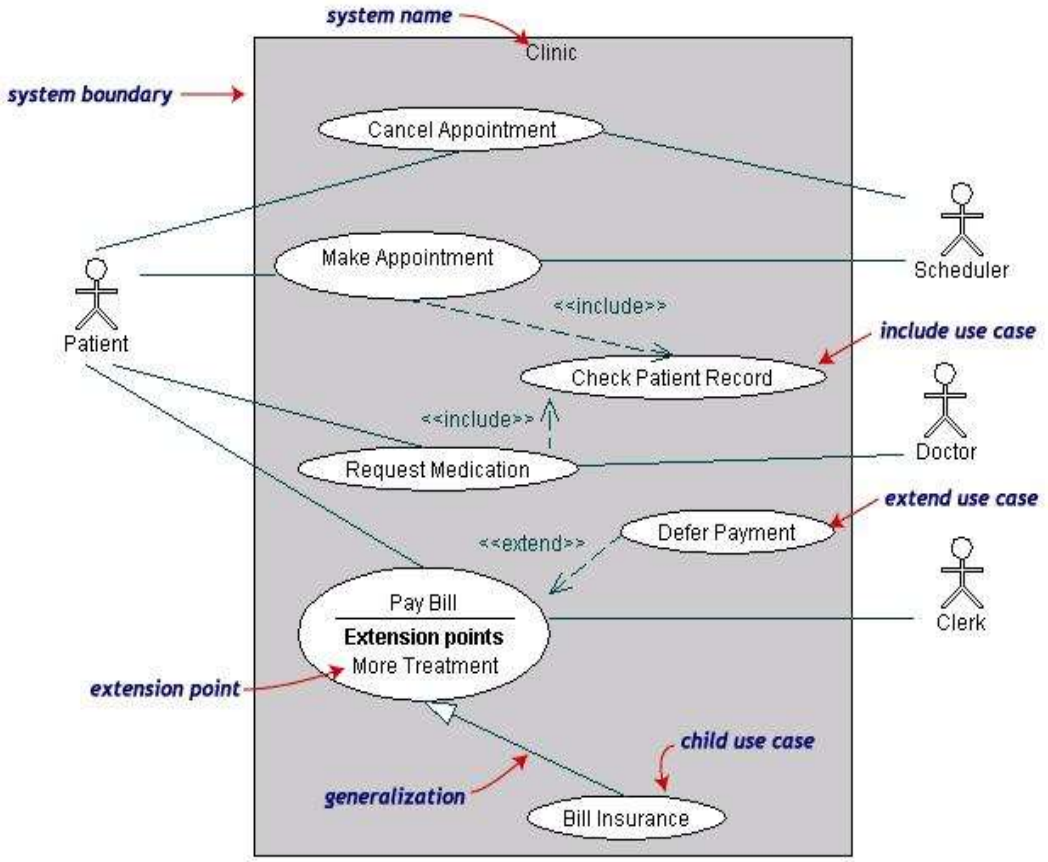


USE-CASE DIAGRAMS

Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask (include)

The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)

Pay Bill is a parent use case and **Bill Insurance** is the child use case. (generalization)



STATE MACHINE

"The state machine view describes the dynamic behavior of objects over time by modeling the lifecycles of objects of each class. Each object is treated as an isolated entity that communicates with the rest of the world by detecting events and responding to them. Events represent the kinds of changes that objects can detect... Anything that can affect an object can be characterized as an event."

- *The UML Reference Manual, [Rumbaugh,99]*

STATE MACHINE

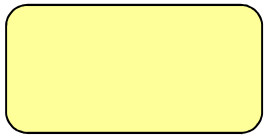
An object must be in some specific state at any given time during its lifecycle. An object transitions from one state to another as the result of some event that affects it. You may create a state diagram for any class, collaboration, operation, or use case in a UML model .

There can be only one start state in a state diagram, but there may be many intermediate and final states.

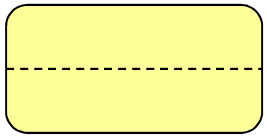
STATE MACHINE

start state

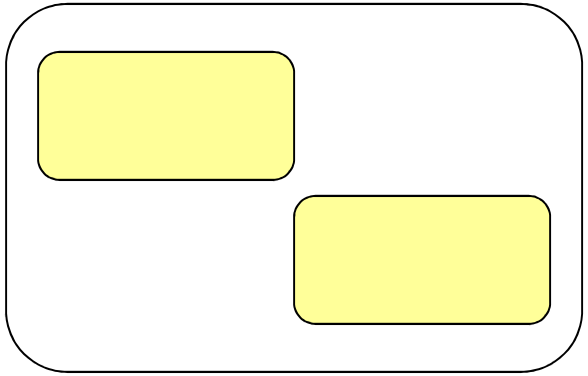
final state



simple state



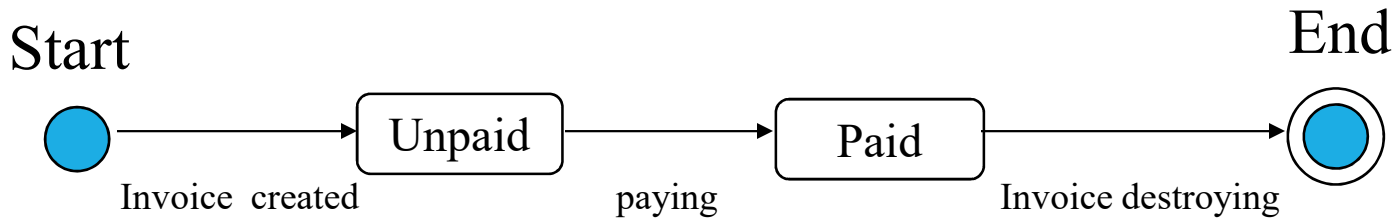
concurrent composite state



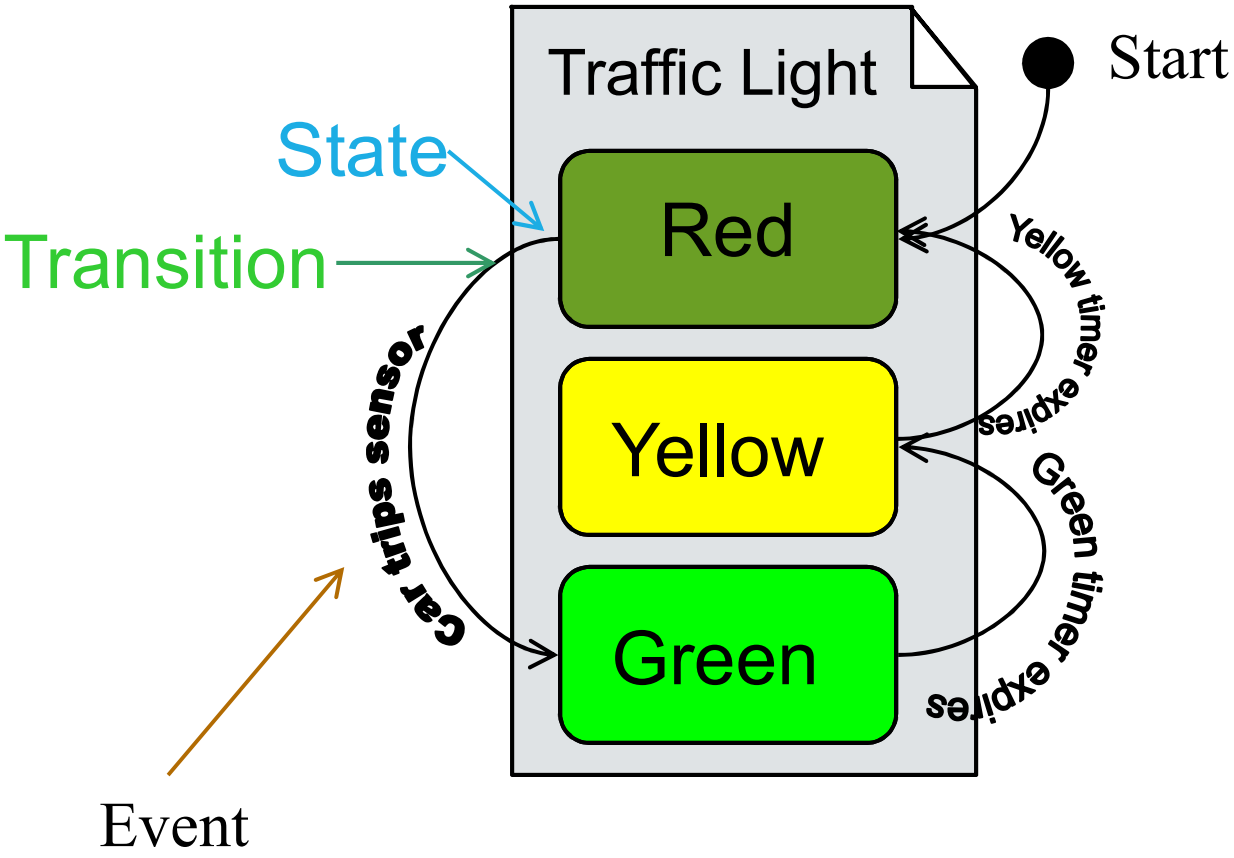
sequential composite state

STATE DIAGRAMS (BILLING EXAMPLE)

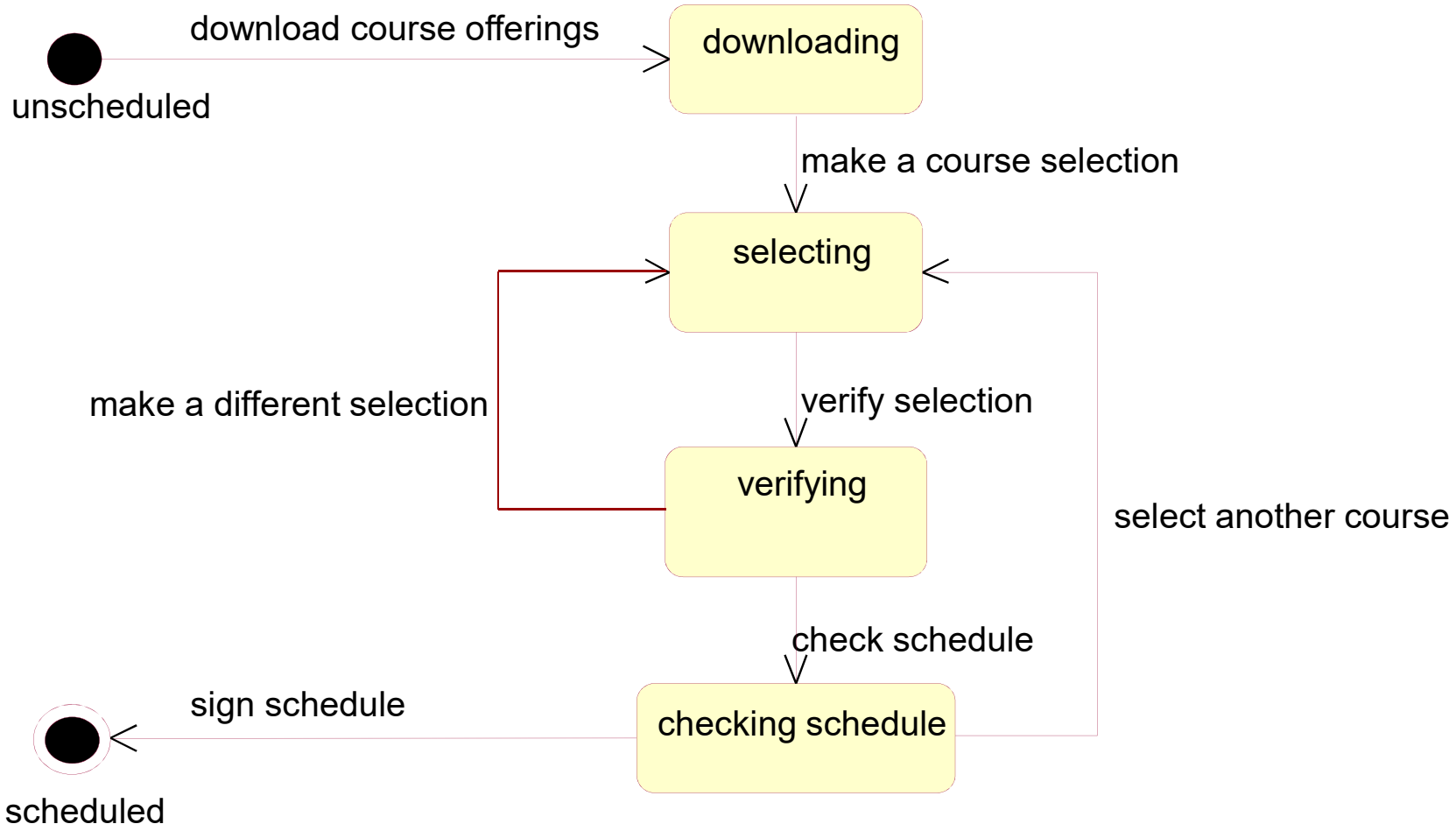
State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.



STATE DIAGRAMS (TRAFFIC LIGHT EXAMPLE)



STATE MACHINE – COURSE SCHEDULING



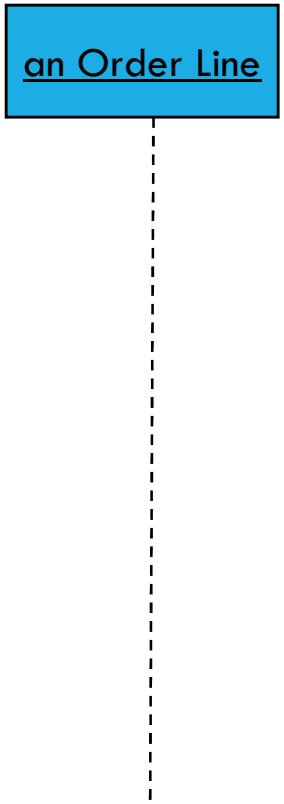
SEQUENCE DIAGRAM

A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. It shows a set of objects and the messages sent and received by those objects.

Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.

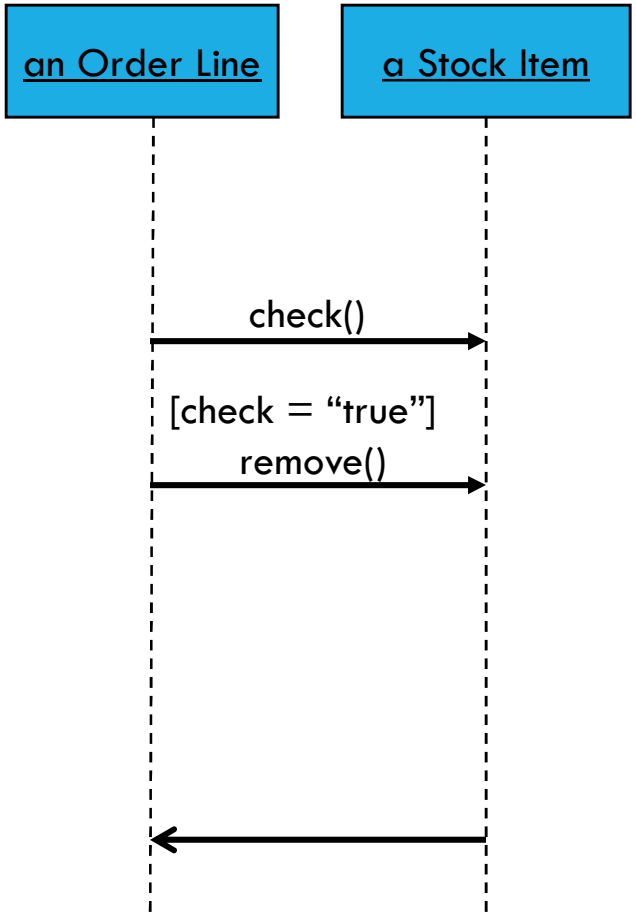
- *The UML User Guide, [Booch,99]*

SEQUENCE DIAGRAM



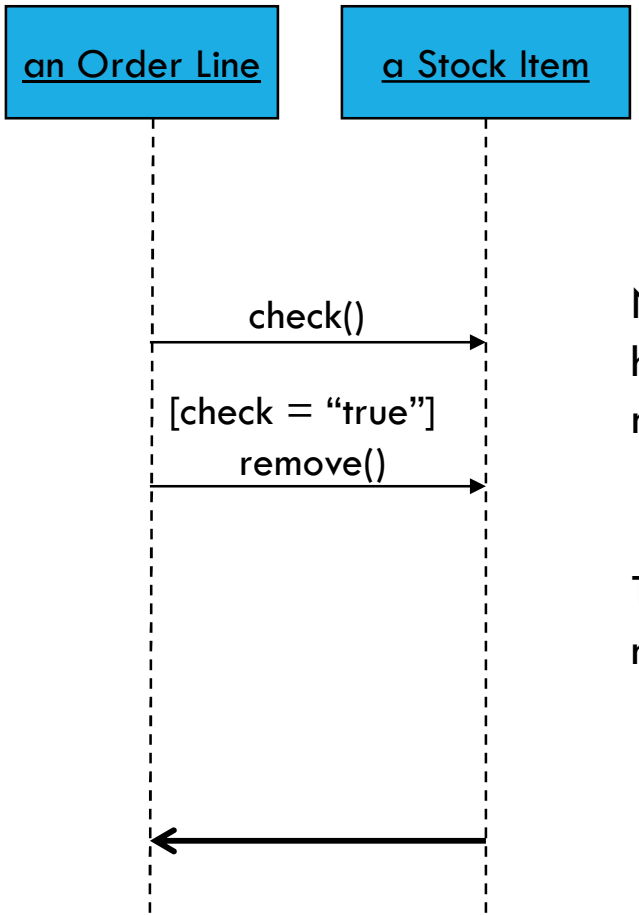
An object in a sequence diagram is rendered as a box with a dashed line descending from it. The line is called the *object lifeline*, and it represents the existence of an object over a period of time.

SEQUENCE DIAGRAM



Messages are rendered as horizontal arrows being passed from object to object as time advances down the object lifelines. Conditions (such as [check = "true"]) indicate when a message gets passed.

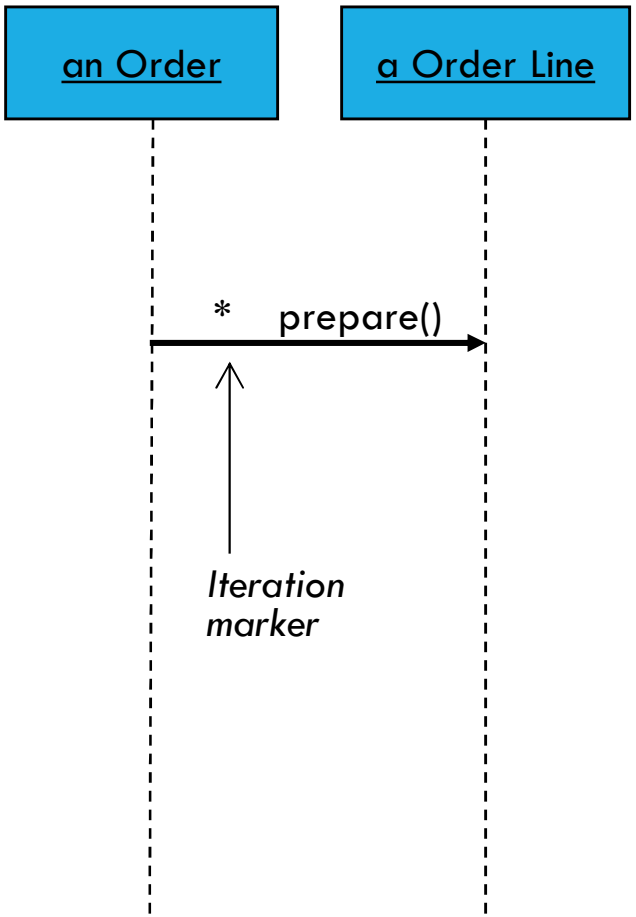
SEQUENCE DIAGRAM



Notice that the bottom arrow is different. The arrow head is not solid, and there is no accompanying message.

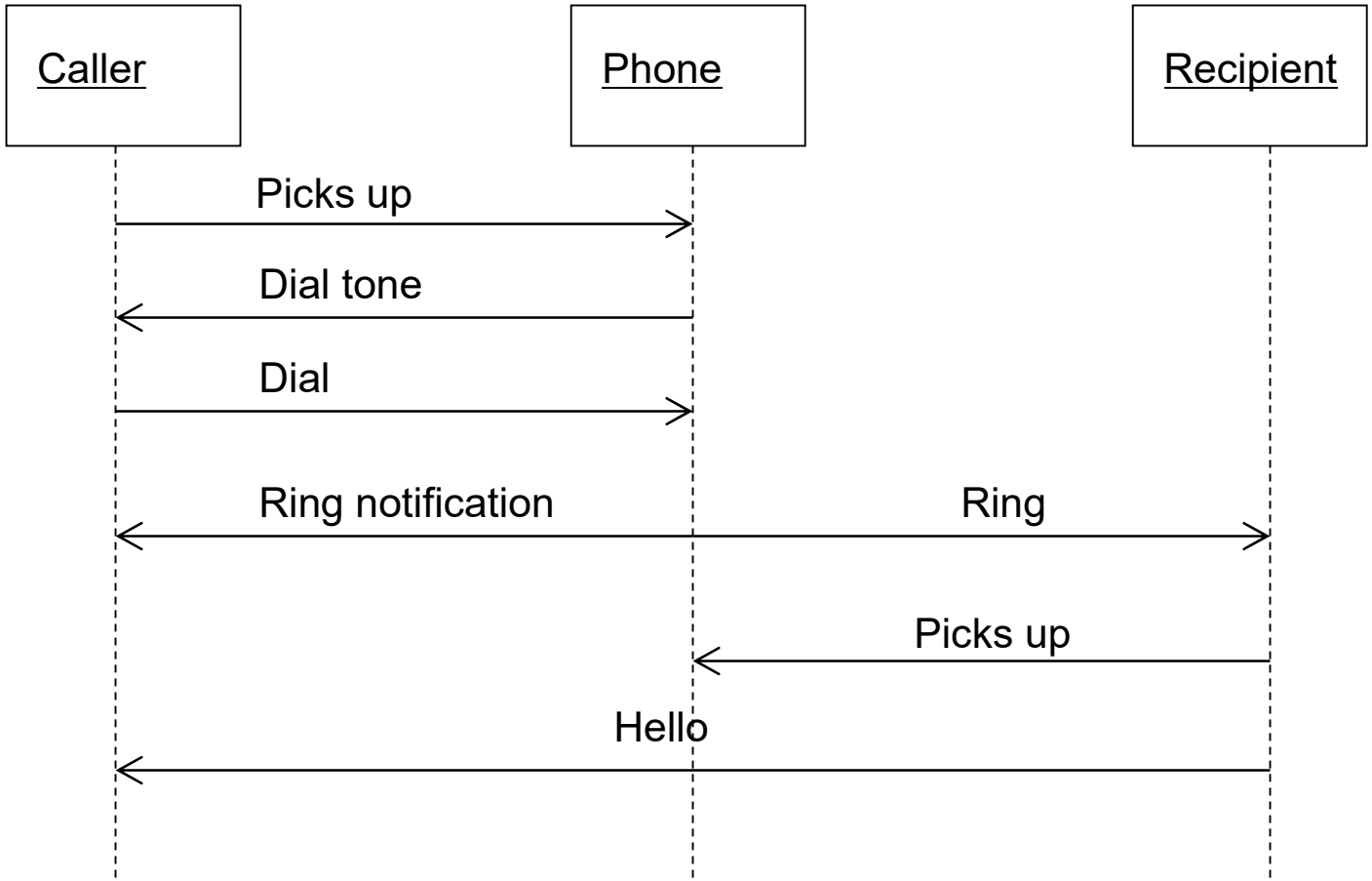
This arrow indicates a **return** from a previous message, not a new message.

SEQUENCE DIAGRAM



An iteration marker, such as * (as shown), or *[i = 1..n] , indicates that a message will be repeated as indicated.

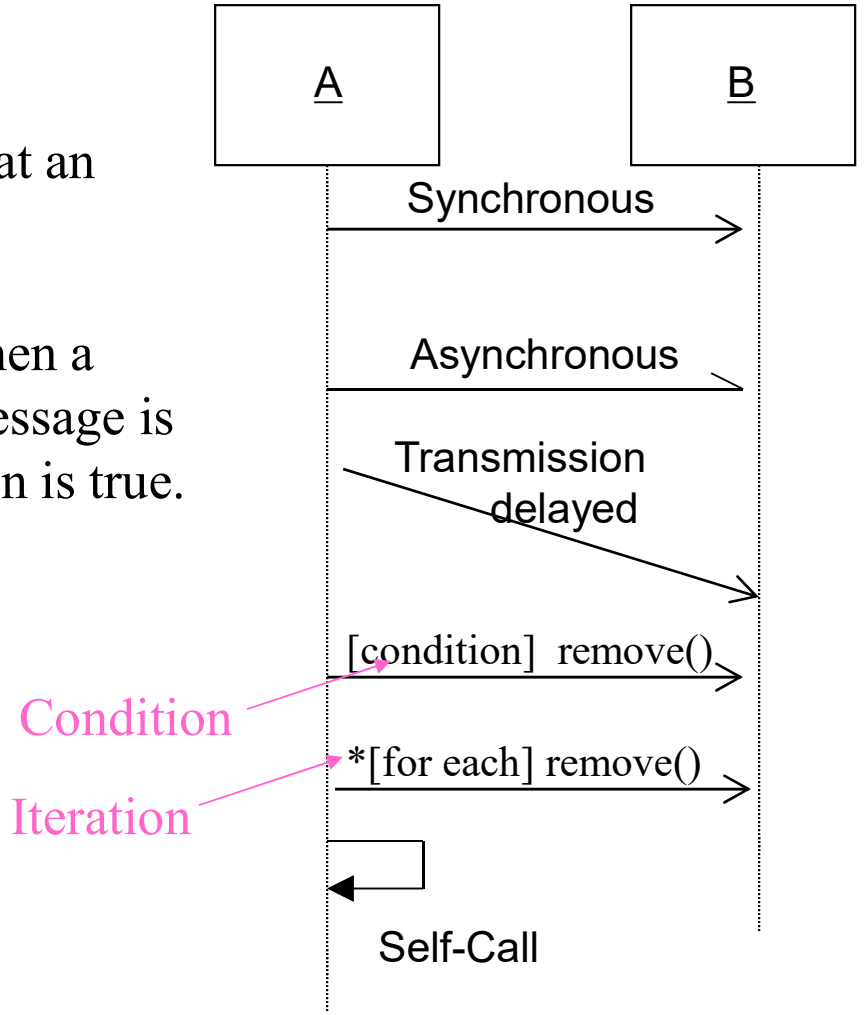
SEQUENCE DIAGRAM(MAKE A PHONE CALL)



SEQUENCE DIAGRAM:OBJECT INTERACTION

Self-Call: A message that an Object sends to itself.

Condition: indicates when a message is sent. The message is sent only if the condition is true.



SEQUENCE DIAGRAMS – OBJECT LIFE SPANS

Creation

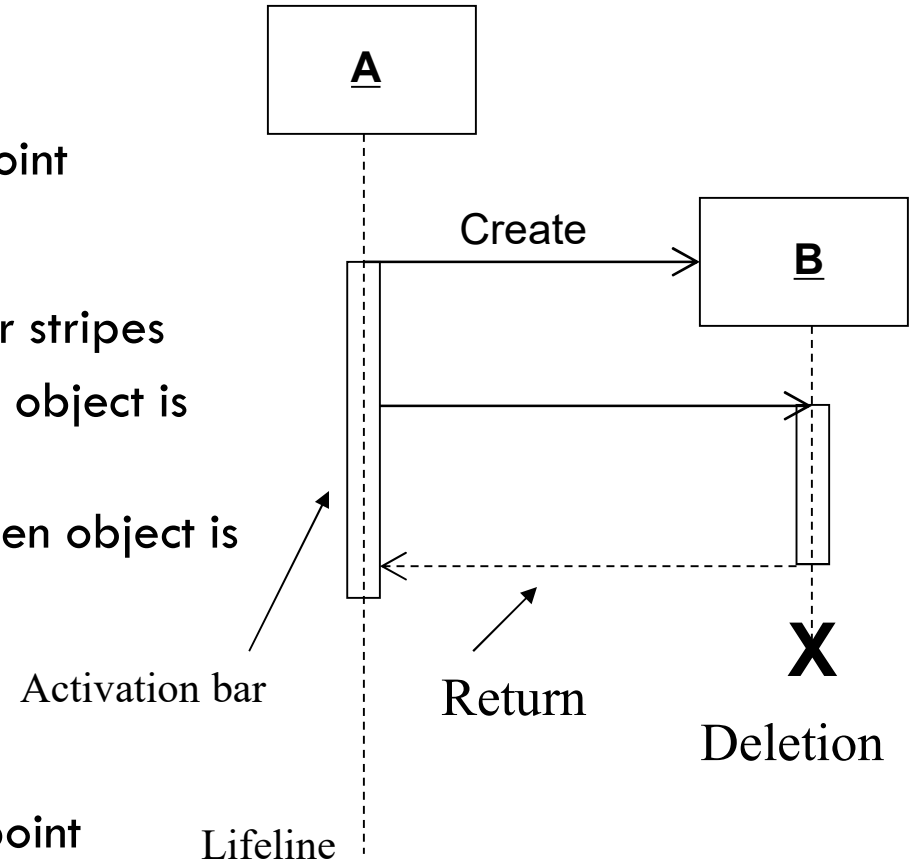
- Create message
- Object life starts at that point

Activation

- Symbolized by rectangular stripes
- Place on the lifeline where object is activated.
- Rectangle also denotes when object is deactivated.

Deletion

- Placing an 'X' on lifeline
- Object's life ends at that point



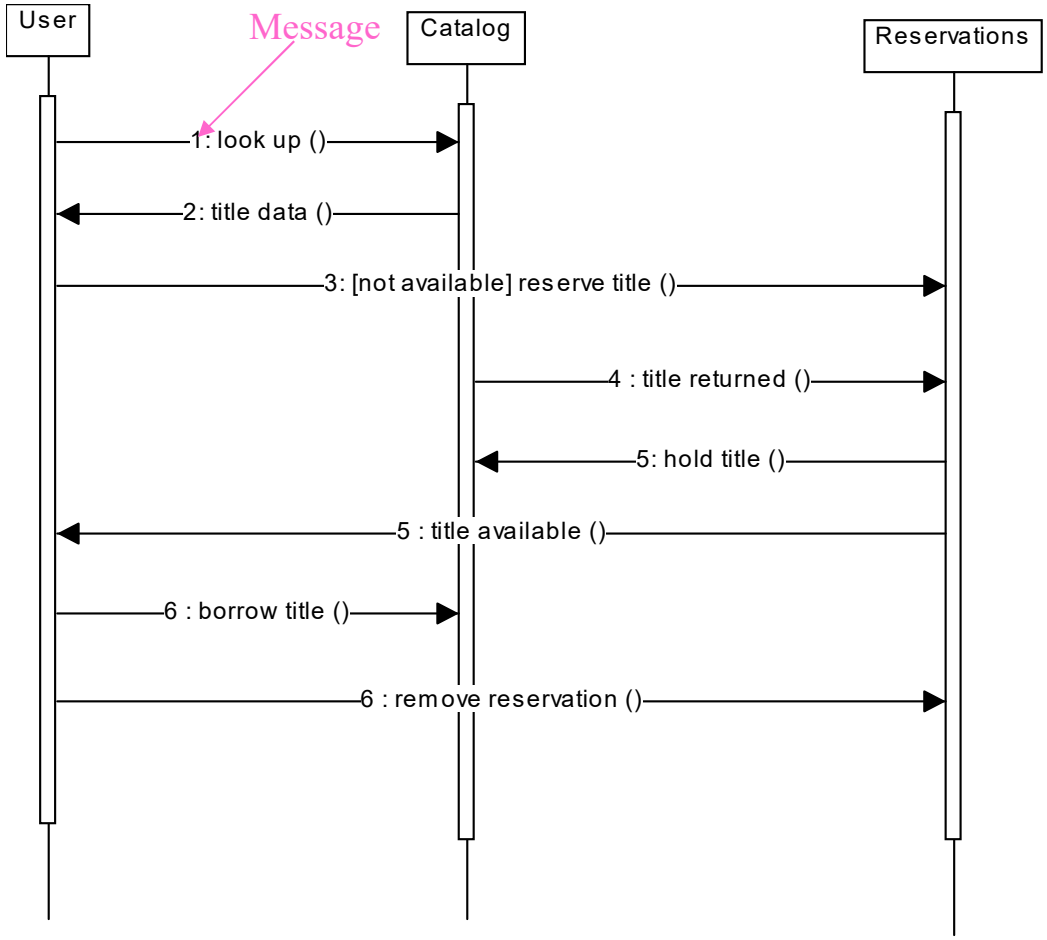
SEQUENCE DIAGRAM

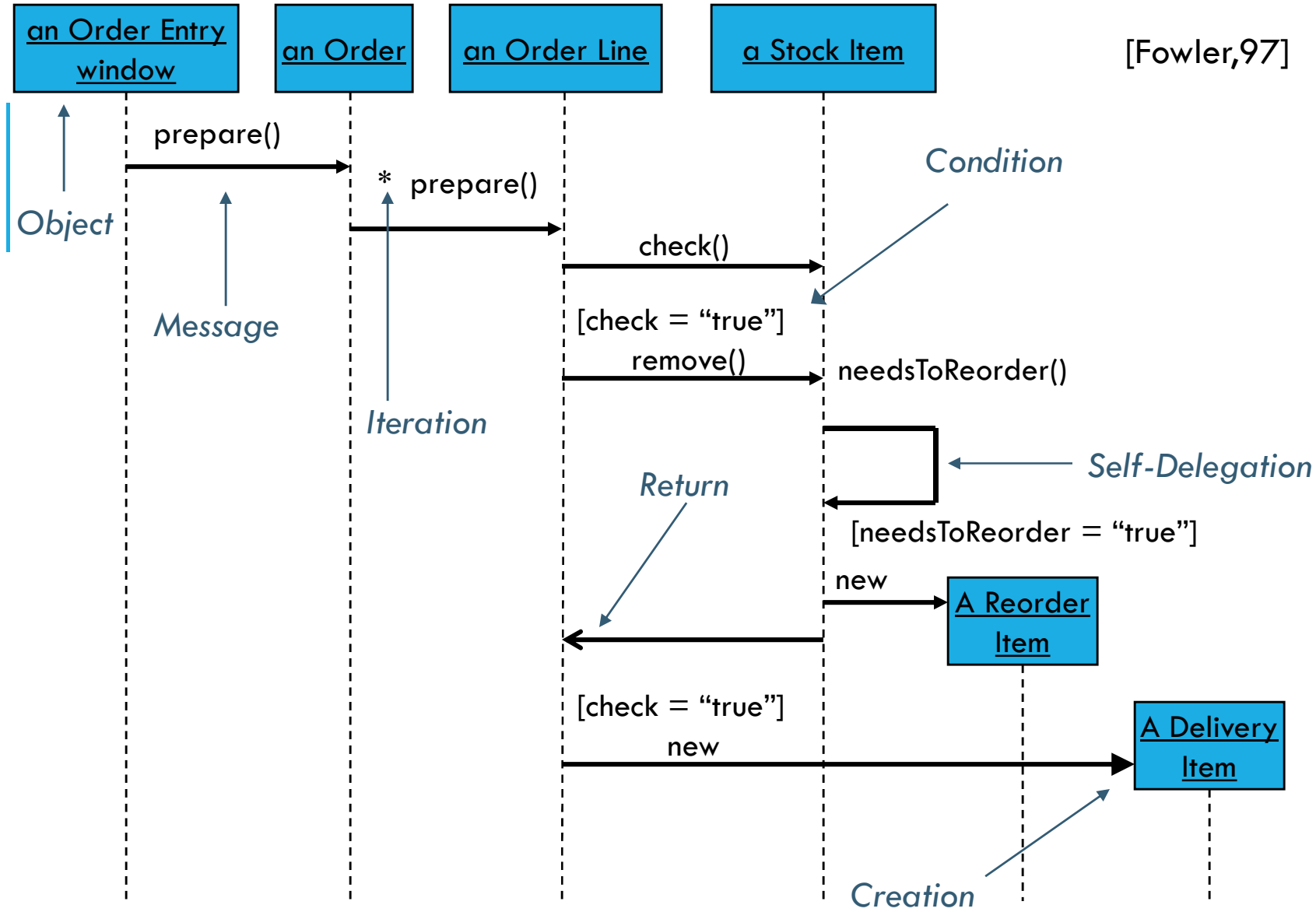
•Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.

•The horizontal dimension shows the objects participating in the interaction.

•The vertical arrangement of messages indicates their order.

•The labels may contain the seq. # to indicate concurrency.



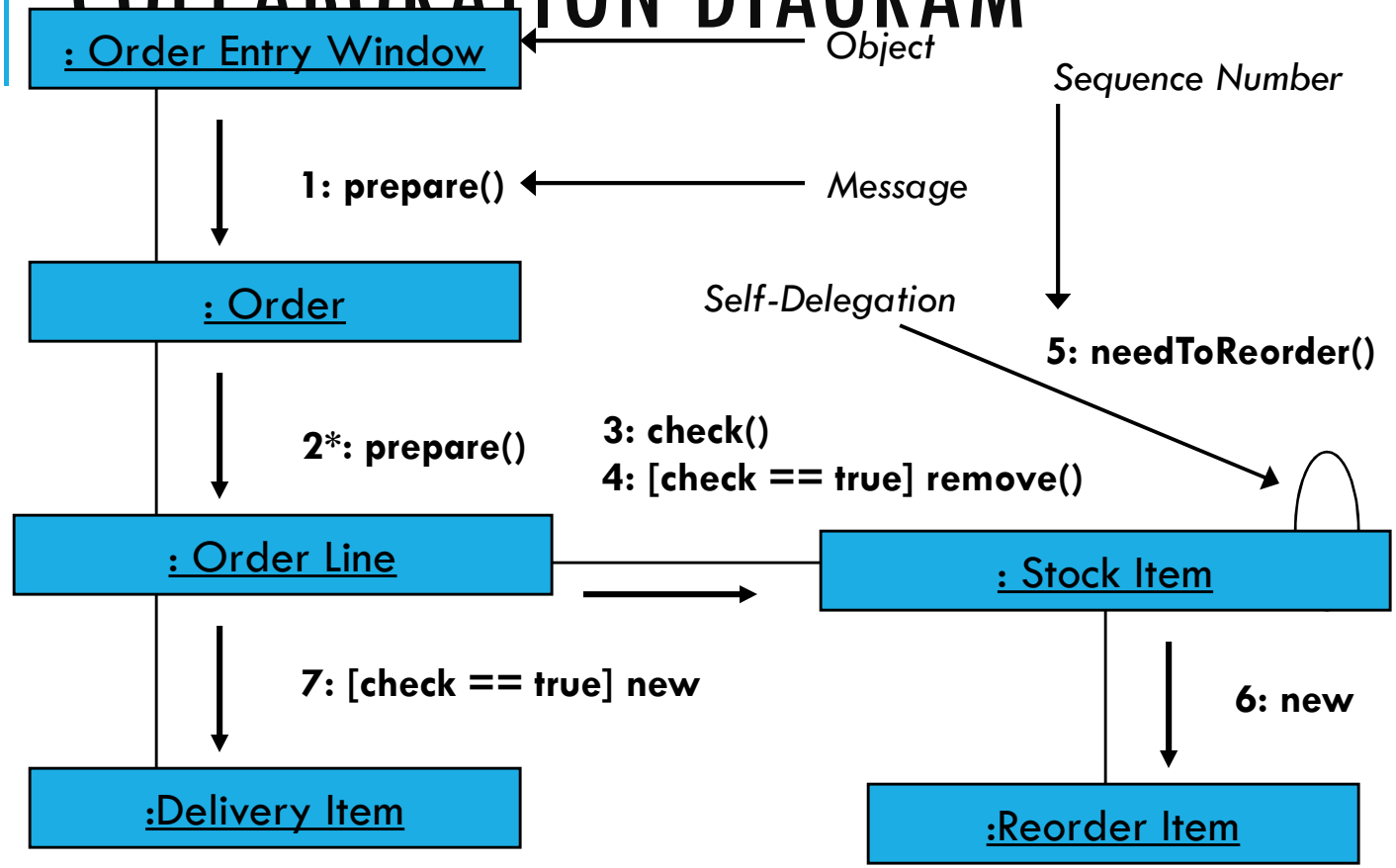

an Order Entry window
an Order
an Order Line
a Stock Item
[Fowler,97]
Object
prepare()
Message
* prepare()
Iteration
check()
Condition
[check = "true"]
remove()
needsToReorder()
Self-Delegation
Return
[needsToReorder = "true"]
new
A Reorder Item
[check = "true"]
new
A Delivery Item
Creation

COLLABORATION DIAGRAM

A collaboration diagram emphasizes the relationship of the objects that participate in an interaction. Unlike a sequence diagram, you don't have to show the lifeline of an object explicitly in a collaboration diagram. The sequence of events are indicated by sequence numbers preceding messages.

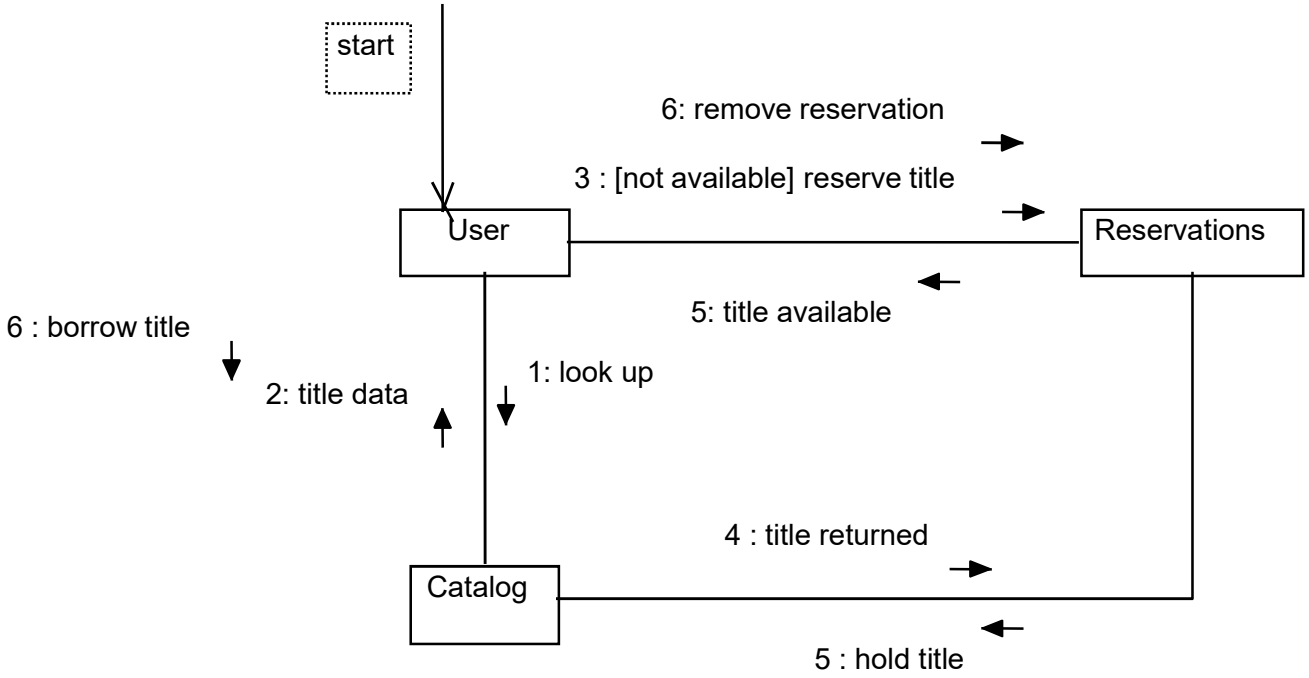
Object identifiers are of the form *objectName : className,* and either the objectName or the className can be omitted, and the placement of the colon indicates either an objectName: , or a :className.

COLLABORATION DIAGRAM



[Fowler,97]

INTERACTION DIAGRAMS: COLLABORATION DIAGRAMS



- Both a collaboration diagram and a sequence diagram derive from the same information in the UML's metamodel, so you can take a diagram in one form and convert it into the other. They are semantically equivalent.

- Use a sequence diagram when the transfer of information is the focus of attention

- Use a collaboration diagram when concentrating on the classes

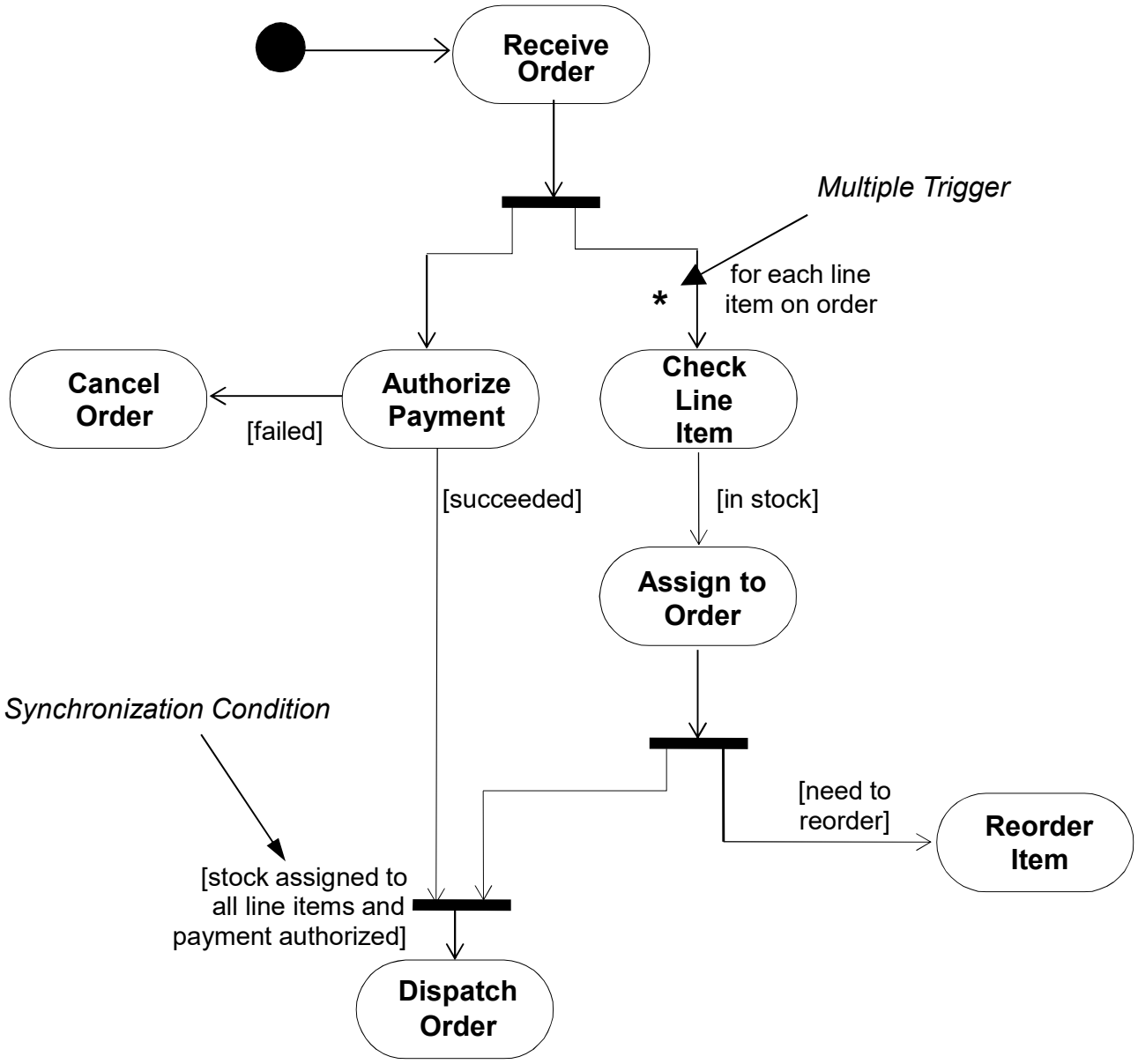
ACTIVITY DIAGRAM

An activity diagram is essentially a flowchart, showing the flow of control from activity to activity.

Use activity diagrams to specify, construct, and document the dynamics of a society of objects, or to model the flow of control of an operation. Whereas interaction diagrams emphasize the flow of control from object to object, activity diagrams emphasize the flow of control from activity to activity. ***An activity is an ongoing non-atomic execution within a state machine.***

- *The UML User Guide, [Booch,99]*

[Fowler,97]


Receive Order
Multiple Trigger
for each line item on order
*
Cancel Order
[failed]
Authorize Payment
Check Line Item
[succeeded]
[in stock]
Assign to Order
Synchronization Condition
[need to reorder]
Reorder Item
[stock assigned to all line items and payment authorized]
Dispatch Order

SOME REFERENCES

https://www.cs.drexel.edu/~spiros/teaching/CS575/slides/uml.ppt

https://www.cs.ucf.edu/~turgut/COURSES/EEL5881_SEI_Fall07/UML_Lecture.ppt

Booch, Grady, James Rumbaugh, Ivar Jacobson,

The Unified Modeling Language User Guide, Addison Wesley, 1999

Rumbaugh, James, Ivar Jacobson, Grady Booch, The Unified

Modeling Language Reference Manual, Addison Wesley, 1999

Jacobson, Ivar, Grady Booch, James Rumbaugh, The Unified

Software Development Process, Addison Wesley, 1999

Fowler, Martin, Kendall Scott, UML Distilled (Applying the Standard Object Modeling Language), Addison Wesley, 1997.