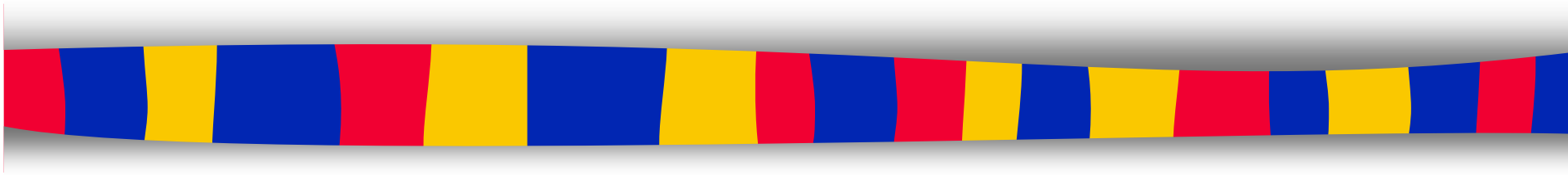
EECS 4314
Advanced Software Engineering



Topic 03:
UML Overview

Zhen Ming (Jack) Jiang

Acknowledgement

- Some slides are from Ahmed E. Hassan, Richard Paige, and Spiros Mancoridis

Unified Modelling Language (UML)

- UML is a set of *notations*, not a methodology or process.
  - Version 2.x is the latest standard
    - There are now ~12 kinds of diagrams!
  - UML *does* have an official standard, backed by OMG
  - Rational (now owned by IBM) is the big mover behind UML, but they don't "own" it.
  - Lots of history and politics behind it.
- Many expensive tools, seminars, books, hype, *etc.* … but
  - *"UML is just a bunch of notations."*
  - UML doesn't solve your problems for you, it gives you a way of writing them down.
    - *"Guaranteed cockroach killer"* [Gause/Weinberg]

A short history of analysis and design notations

1970s:

- Procedural languages (COBOL, FORTRAN, PL/I, C)
- Systems are structured as TDFD
  - (TDFD == top-down functional decomposition)
- Data is mostly global and passive
- Notations and tools:
  - ER diagrams (for DB design)
  - DFDs, CFGs, flowcharts
  - STDs (for state-oriented engineering applications)
  - Data dictionaries
- Methodologies: Structured analysis

1980s:

- Some OO languages start to creep in (C++, OO-Cobol)
- Systems structured as modules, use info-hiding & interfaces.
- Data is encapsulated; must use interfaces.
- Notations and tools:
  - Class/object diagrams (ER++) for analysis modelling
  - Statecharts (formal STDs for engineering appls.)
  - Message sequence charts (aka scenario diags)
  - Use cases (Jacobson)
- Methodologies:
  - OMT (Rumbaugh)
  - Booch notation
  - many others

1990s:

- Most of the software industry is tired of tool/notation wars.
  - They want some form of agreement on a notation without an accompanying state religion.
- The “three amigos” gather at Rational; they confer, then announce “war is over (if you want it).”
- UML takes a kitchen-sink approach to language design.
  - It contains many kinds of diagrams(!), and makes few restrictions on how to use them.
  - UML diagrams are used to model various requirements views as well as architecture, design, implementation, and run-time views

UML in a Nutshell

- Early reqs views:
  - *Use cases:*
    - Identify actors, SUD boundaries/scope
    - Map out basic SUD functionality at coarsely-grained level; consider variations
- Elaborate analysis model:
  - *Class diagrams:*
    - Shows static, structural *domain model*
    - Model abstract relationships between problem-space entities
  - *Sequence / communication (a.k.a., collaboration) diagrams:*
    - Expand use cases into a set of scenarios
    - Model interactions and flow of information between objects
    - Show inter-object dynamic properties (can be coarse or fine)

UML in a Nutshell

- (Very) detailed reqs view:
  - *State diagrams (Statecharts) + (detailed) sequence diagrams:*
    - Show detailed view of how objects "work".
      - **Statecharts model:**
        - » *intra*-object dynamic behaviour
      - **Sequence diagrams** model:
        - » *inter*-object dynamic behaviour
    - Usually models "reactive" objects that respond to external stimuli/events, *e.g.,* embedded systems.
    - Objects basically rest in a state until they are informed of an event; they then perform some action and then go rest in another state.
    - Mainly used for real-time, embedded, and other engineering applications; less useful for other types of systems.

1. Fowler on UML “religions”

1. UML as “religion”:
   - How rigorously should you create and maintain the UML diagrams?
   - How much detail to show?
   - Is XXX legal UML? Does it matter?
2. Different “perspectives” for UML diagrams
   - Using the same kinds of diagrams for different purposes (analysis *vs.* design)

1. Fowler on UML “religions”

- *UML as blueprint*
  - Goal is rigorous, *complete* specification of analysis and/or design of a software system:
    - Analysis UML models are kept consistent with design UML models
      - There is traceability between analysis and design models
    - UML design artifacts are kept consistent with code
      - Usually, this means extracting design info from code (*reverse engineering*) and verifying current reality matches the specified design model.
  - UML diagrams express *partial* semantics of system

    e.g., structure, communication paths, control / data / other dependencies
  - UML diagrams do *not* completely specify low-level semantics

    e.g., full details of what happens inside a method body

1. Fowler on UML “religions”

- *UML as blueprint*
  - Tool support is key: *“round trip engineering”*
    - *Code generation* from UML models
      - Typically of interfaces / class skeletons, not method bodies
    - *Reverse engineering* of UML models from source code
      - Class (design) diagrams extracted from static source structure
      - Sequence / communication diagrams extracted from system execution traces
  - Typically, we choose a desired level of detail; the models are then *complete* with respect to that level of detail

    e.g., static class structure and some set of relations (instantiates, calls, inherits, package membership, …)

1. Fowler on UML “religions”

- *UML as programming language*
  - Tool support is even more important!
    - Unfortunately, we are not quite there yet
    - *“Practical UML MDA tools are on the horizon.”*
  - The UML diagrams *are* the system
    - They are the “maintenance artifacts” of the system, not the code!
    - The code is auto-generated from detailed state models (and class diagrams)
  - Rarely done
    - But it’s the grand goal of the *MDA* movement (model-driven architecture)
      - See http://www.omg.org/mda

Grady Booch
@Grady_Booch

Follow

I designed the UML as a way to reason about complex software-intensive systems (and not as a programming language).

> **Architect Thinkery** @ArchiThinkery
> We model to think -- model out loud, to think together. To collaborate -- now, and with our future selves and others
> twitter.com/agile_geek/sta…

RETWEETS 79 LIKES 93

4:05 pm - 1 Oct 2016

79 93

Kevin Trethewey @KevinTrethewey · Oct 2
@Grady_Booch if you could go back and have a complete “do-over” is there anything drastic you’d do or design differently? / @joshilewis

1

Grady Booch @Grady_Booch · Oct 2
@KevinTrethewey @joshilewis yes.

Kevin Trethewey @KevinTrethewey · Oct 2
@Grady_Booch as someone interested in making it easier for people to reason about complex human-intensive systems, any advice? /@joshilewis

2. Fowler on UML perspectives

- *Conceptual (domain / reqs) perspective*
  - Can involve use case diagrams & use cases, class diagrams, scenarios with actors & SUD, …
  - The conceptual class diagram also called the *domain model*
    - Entities are things in the domain, and actors
      - Classes in this model do not (usually) correspond to programming language classes; that's what design is!
    - Associations are *abstract* relationships between classes/objects (including inheritance and aggregation).

2. Fowler on UML perspectives

- *Software (design) perspective*
  - Can include class diagrams, scenarios, …
  - The key difference is that the “things” modeled here correspond to source code entities
    - The class diagram shows Java classes, their interrelationships that can be seen in the code
      - Attributes are fields, operations are methods
      - Associations model *responsibilities*

        e.g., updates, manages
    - Scenarios show sequences of real method calls

Additional UML References

- *UML Distilled - Applying the Standard Object Modeling Language* by Martin Fowler and Kendall Scott
- *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (3rd Edition)* by Craig Larman

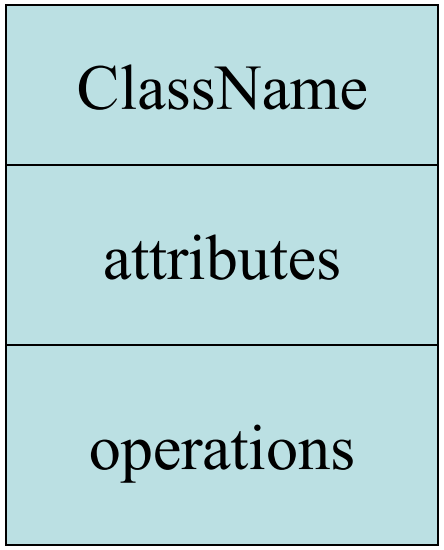
UML Tools

- Anything you can find to use
  - ArgoUML, MagicDraw, Rational, Microsoft Visio, etc.
  - Different tools produce slightly different diagrams
    - Don’t get stuck in the details
    - Make sure the notations in the diagrams are consistent

Software Design

Static Modeling using the Unified Modeling Language (UML)

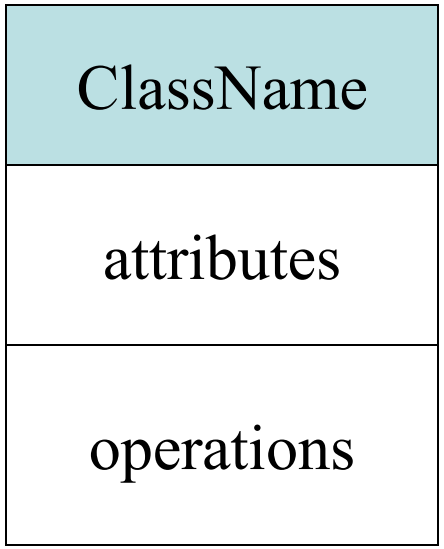
Classes



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

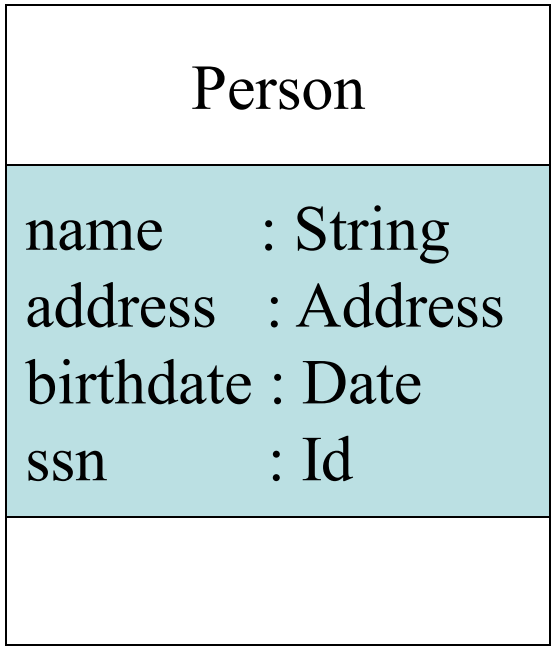
Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

Class Names



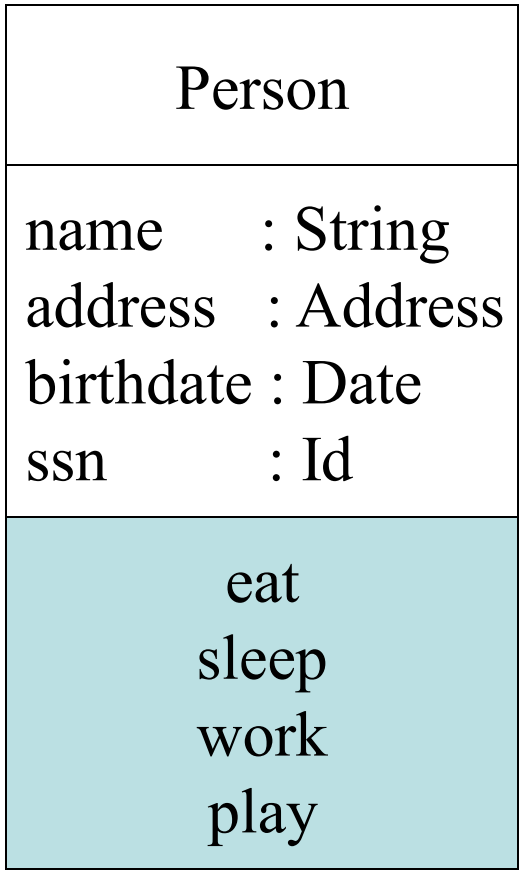
The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

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 Operations



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

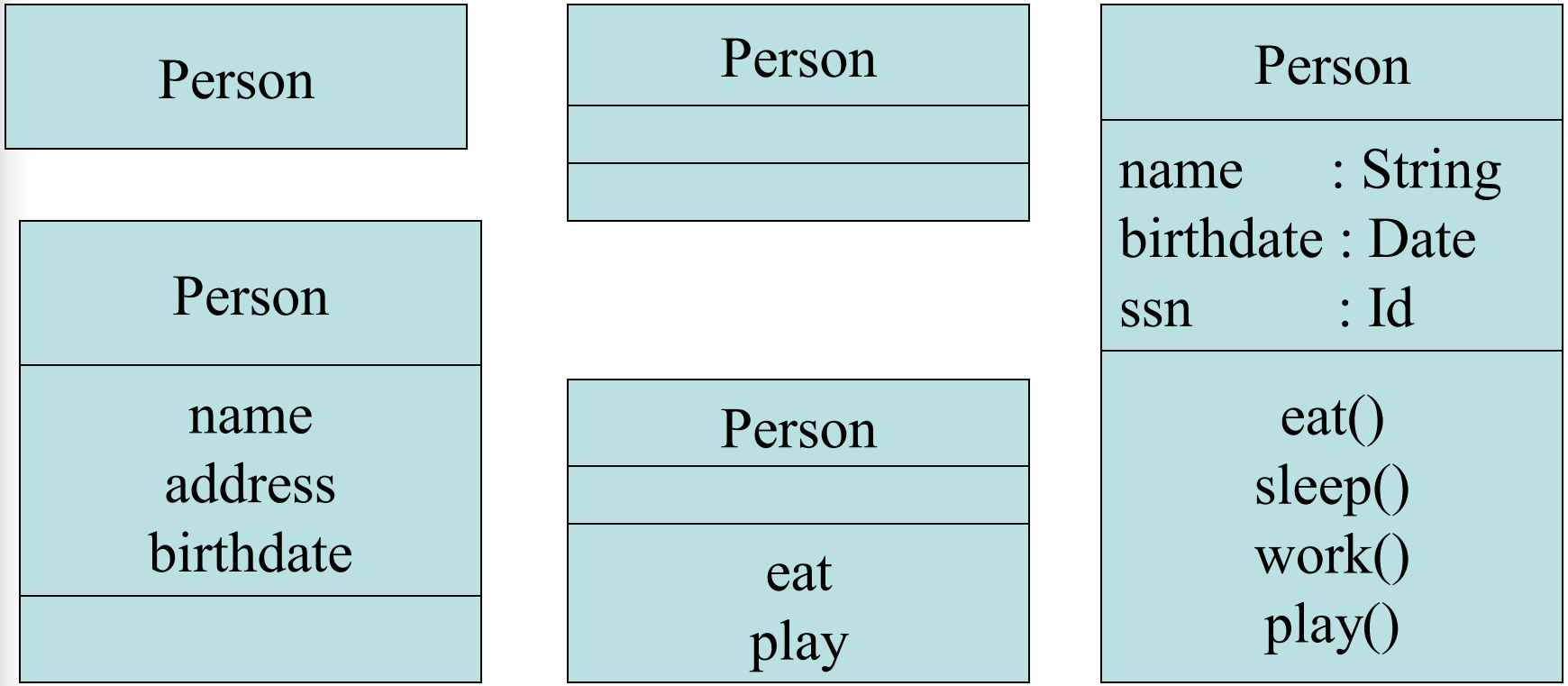
Class Operations (Cont'd)

PhoneBook
newEntry (n : Name, a : Address, p : PhoneNumber, d : Description) getPhone (n : Name, a : Address) : PhoneNumber

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.



Relationships

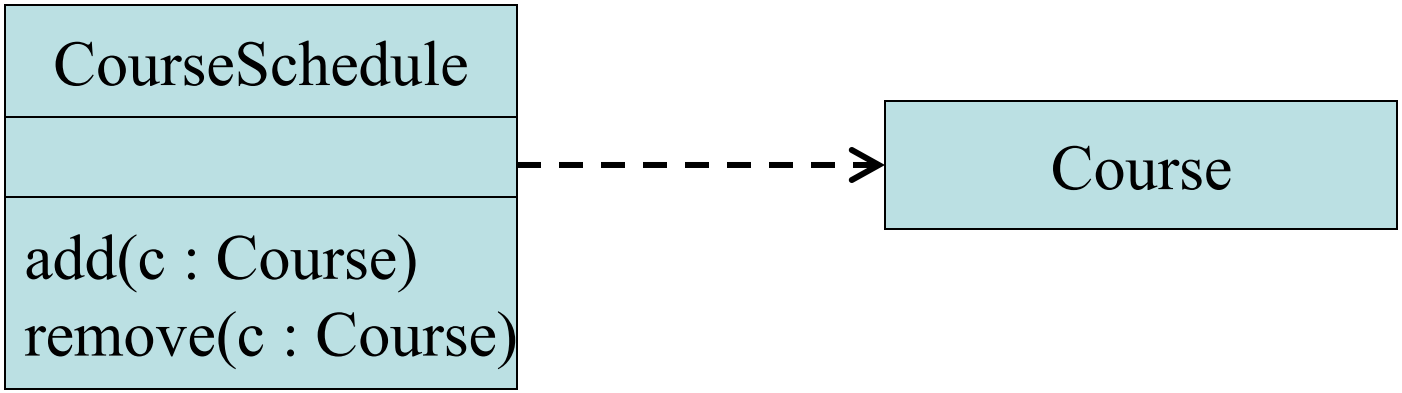
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

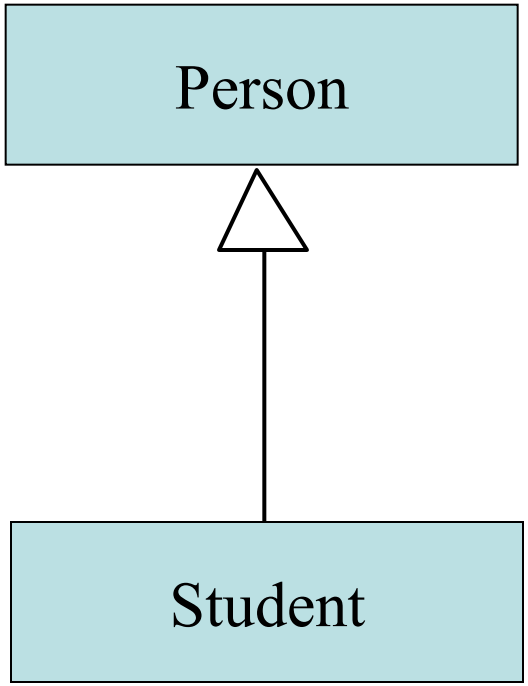
- dependencies
- generalizations
- associations

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*.



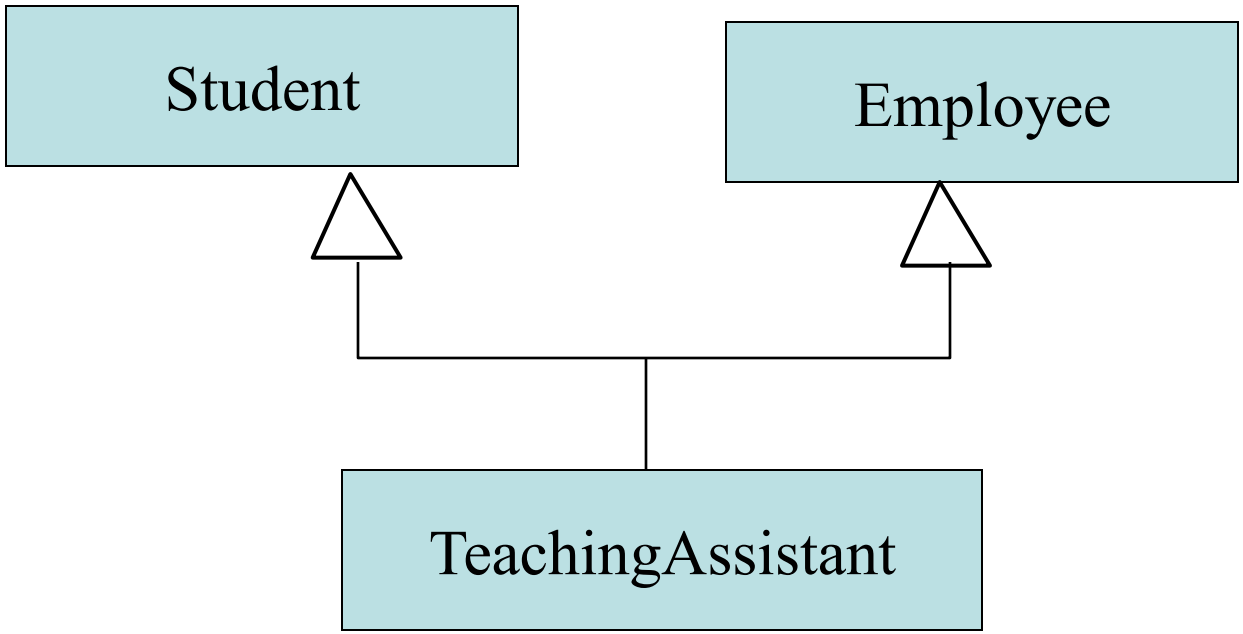
Generalization Relationships



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

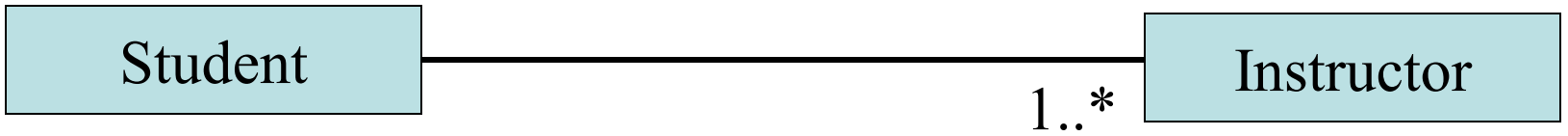
If two classes in a model need to communicate with each other, there must be link between them.

An *association* denotes that link.

Association Relationships (Cont'd)

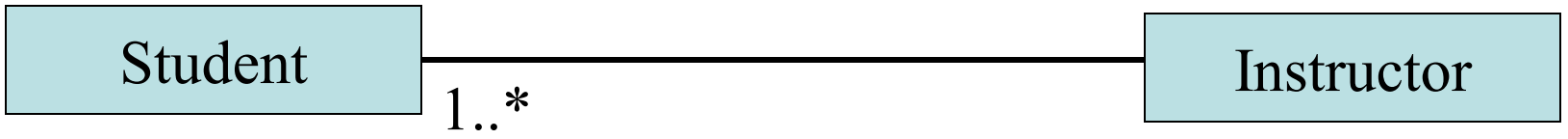
We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

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



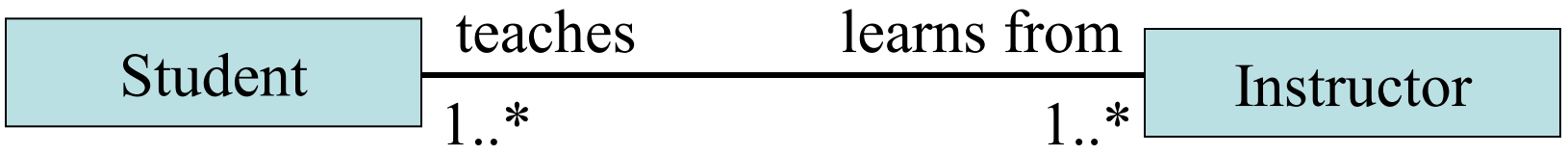
Association Relationships (Cont'd)

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



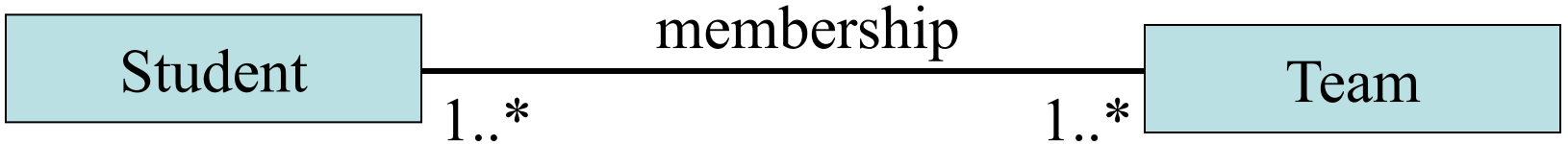
Association Relationships (Cont'd)

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



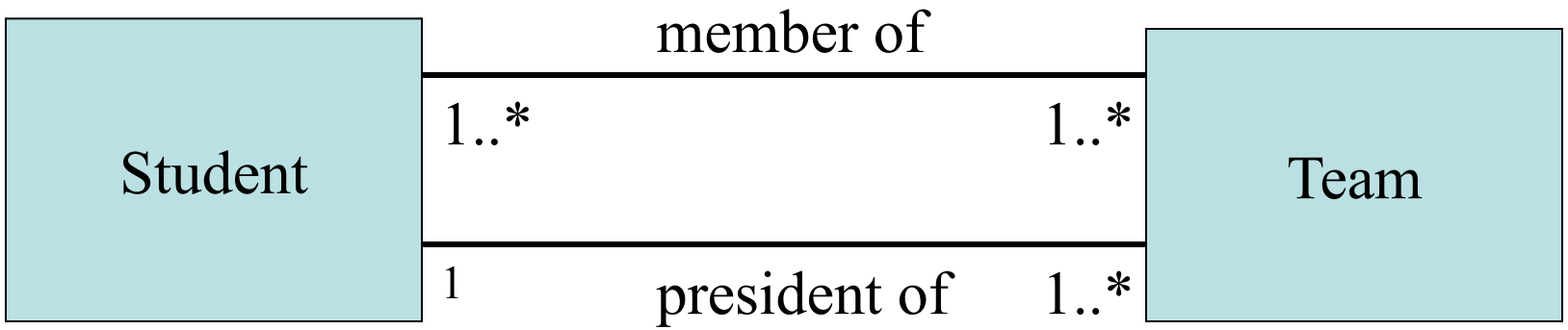
Association Relationships (Cont'd)

We can also name the association.



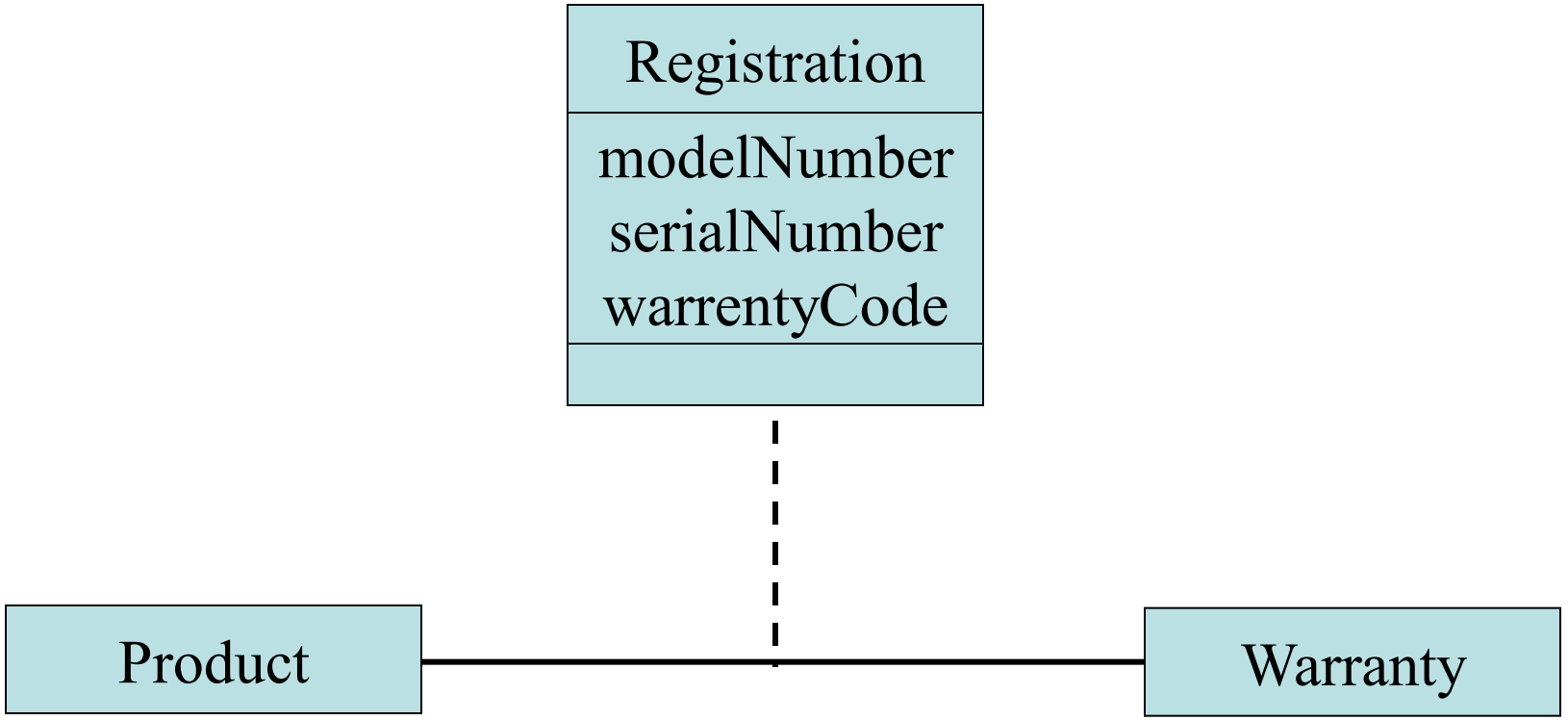
Association Relationships (Cont'd)

We can specify dual associations.



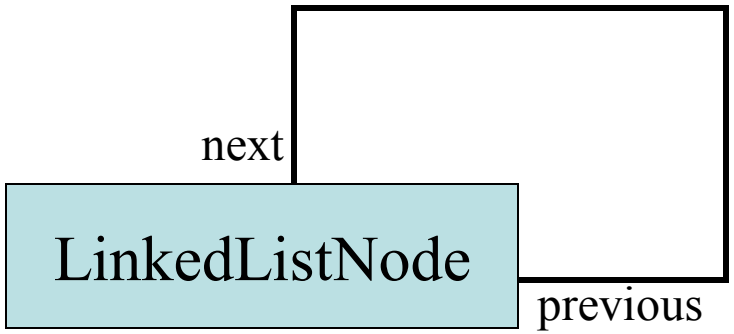
Association Relationships (Cont'd)

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



Association Relationships (Cont'd)

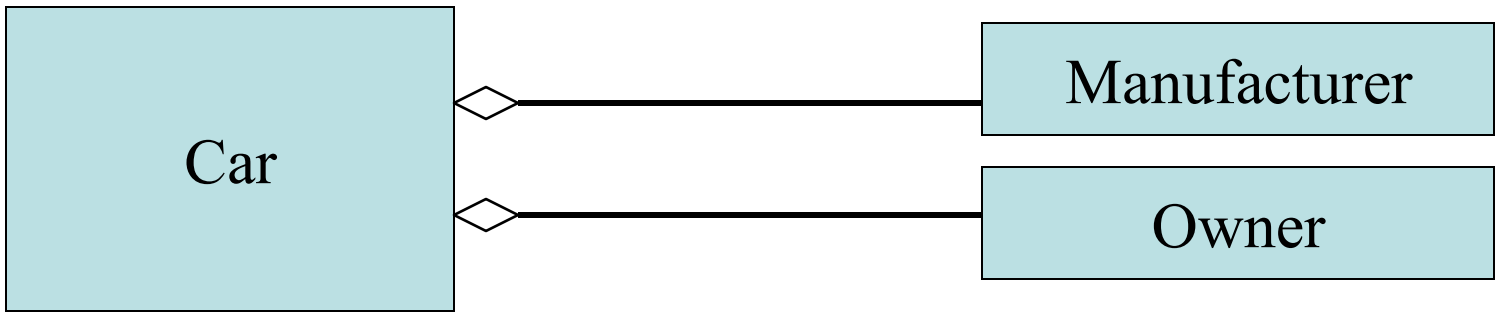
A class can have a *self association*.



Association Relationships (Cont'd)

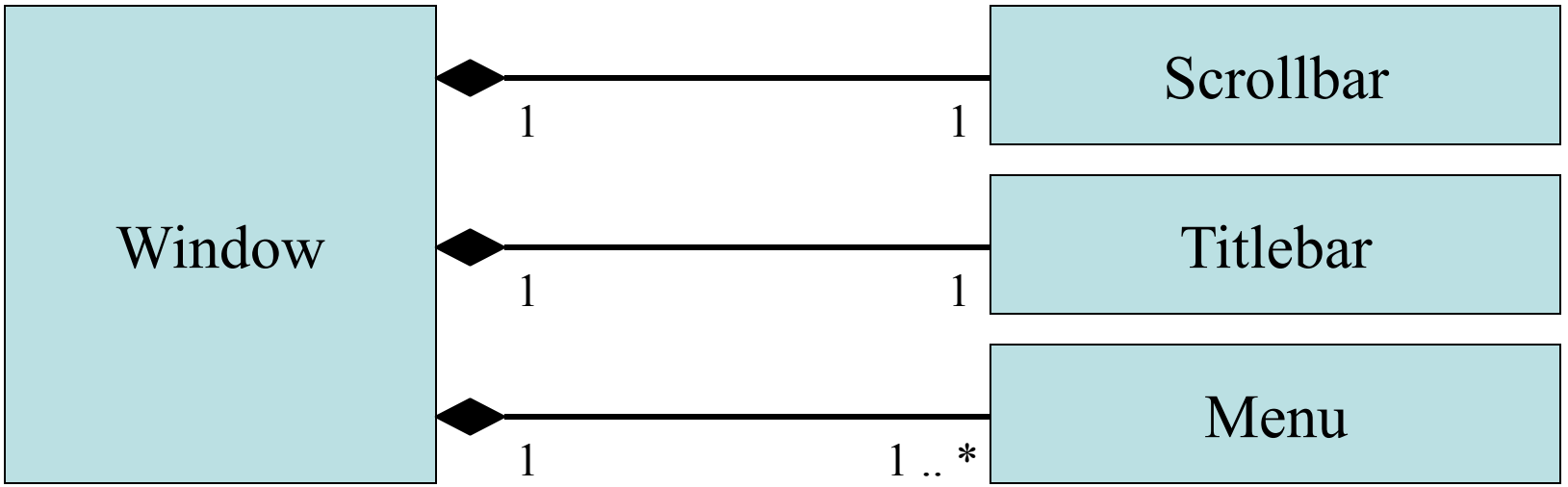
We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

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.



Association Relationships (Cont'd)

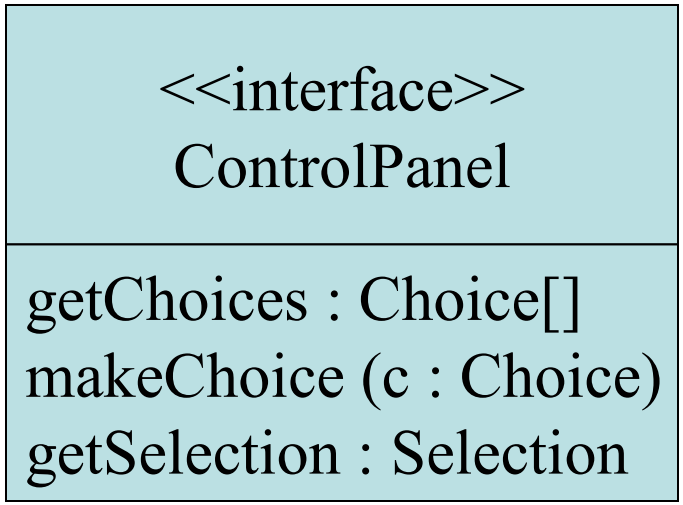
A *composition* (aggregation in Eiffel's term) indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



Interfaces

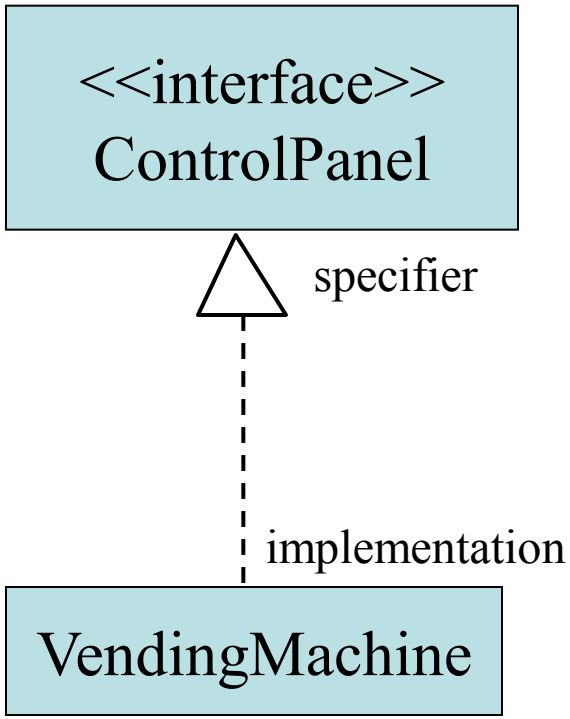
An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure. It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name.

Interface Services



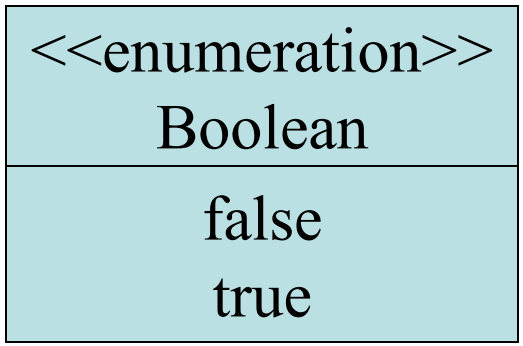
Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.

Interface Realization Relationship



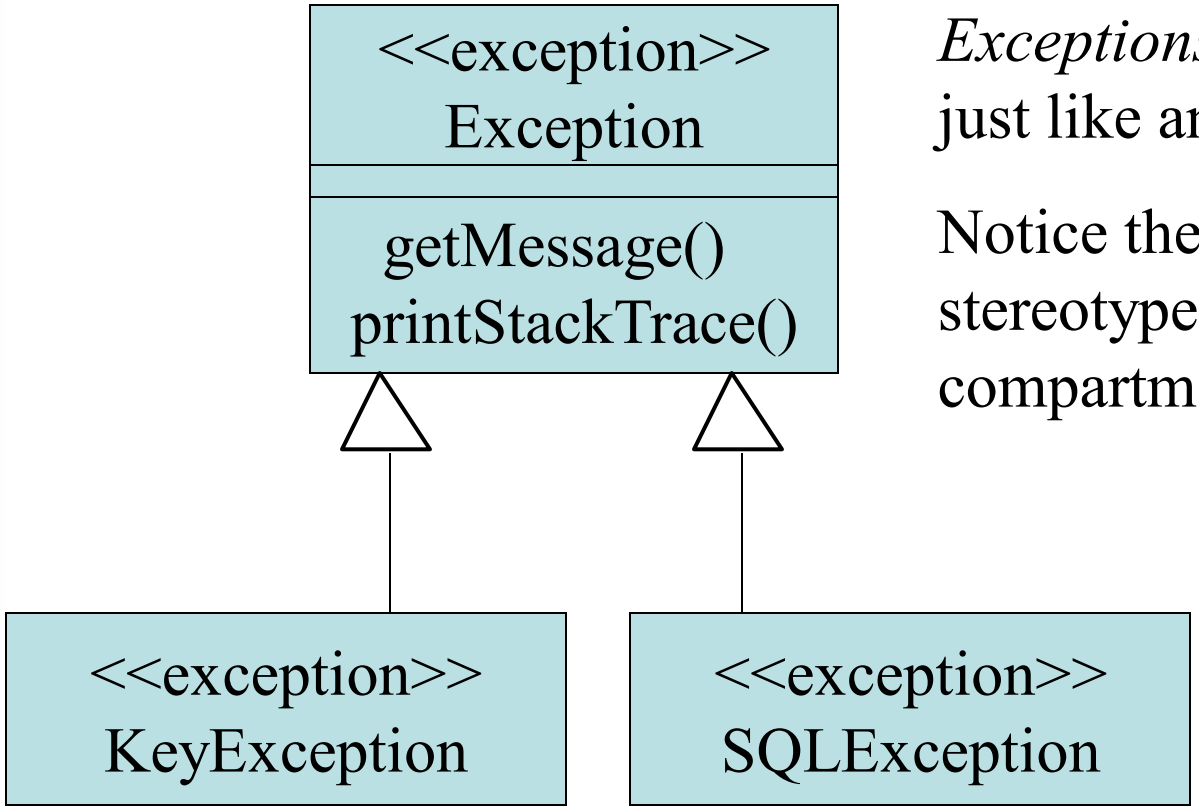
A *realization* relationship connects a class with an interface that supplies its behavioral specification. It is rendered by a dashed line with a hollow triangle towards the specifier.

Enumeration



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

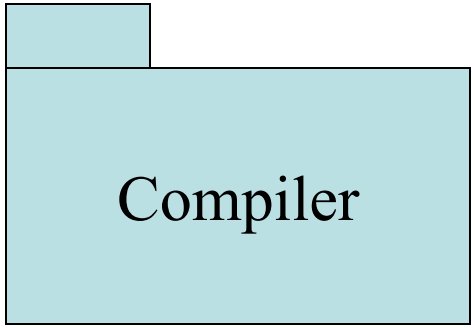
Exceptions



Exceptions can be modeled just like any other class.

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

Packages



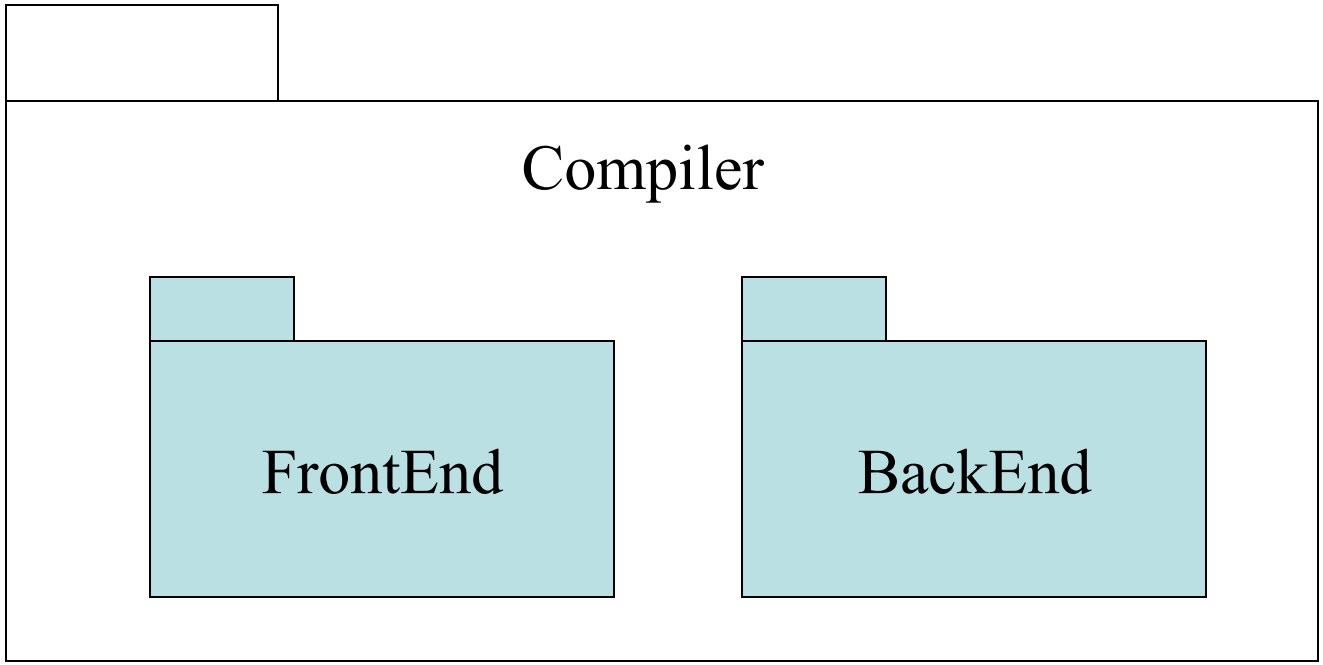
A *package* is a container-like element for organizing other elements into groups.

A package can contain classes and other packages and diagrams.

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

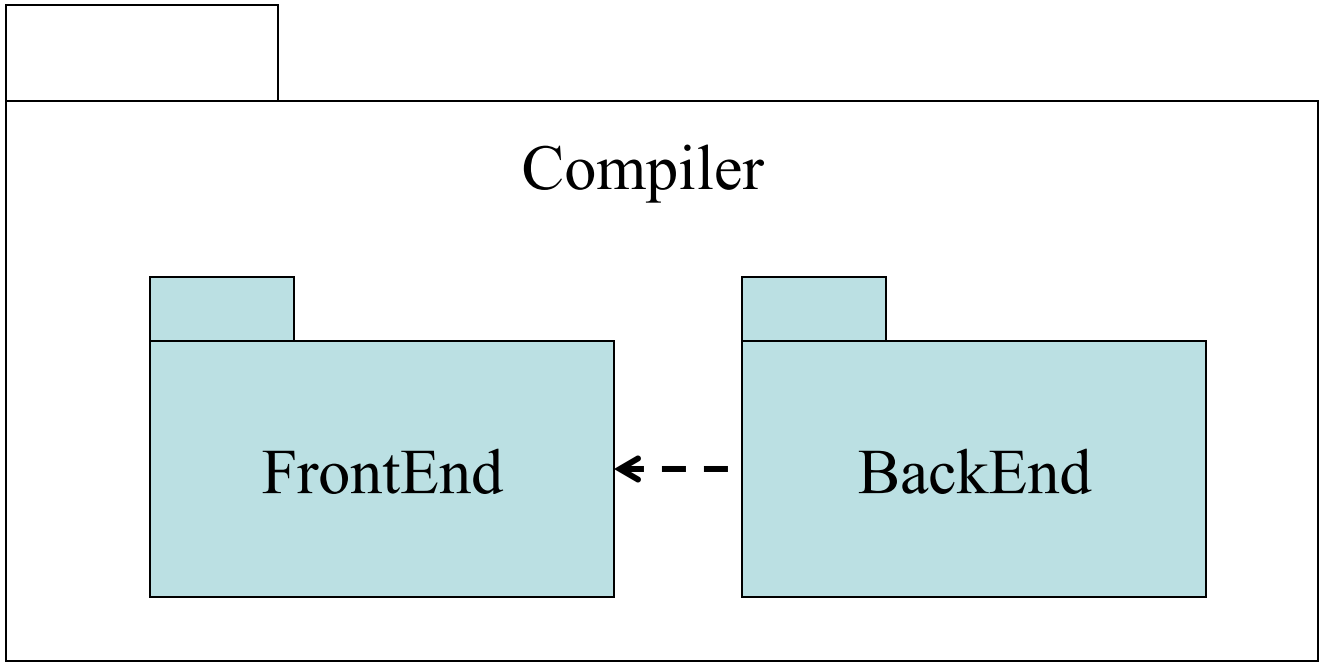
Packages (Cont'd)

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

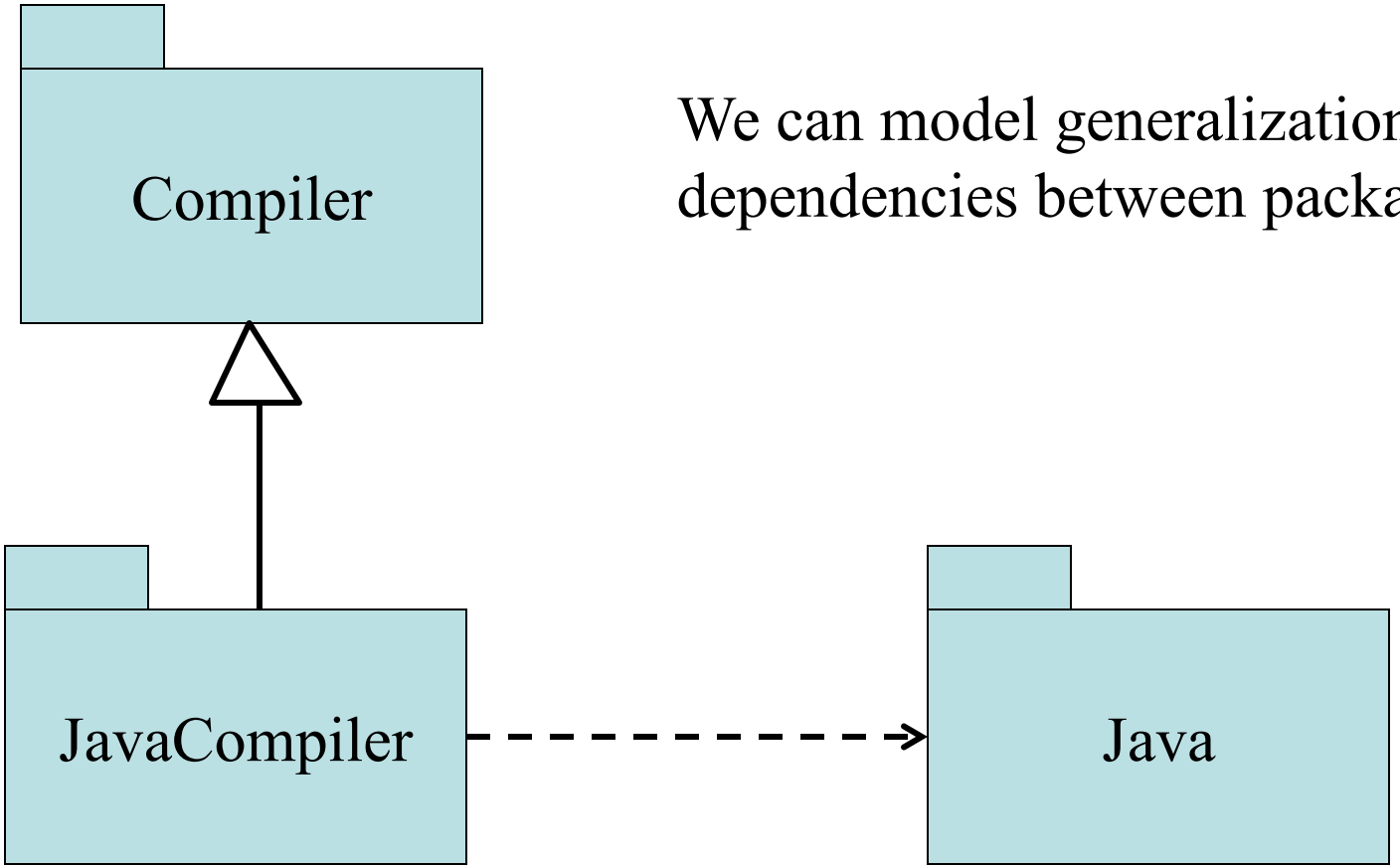


Packages (Cont'd)

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.

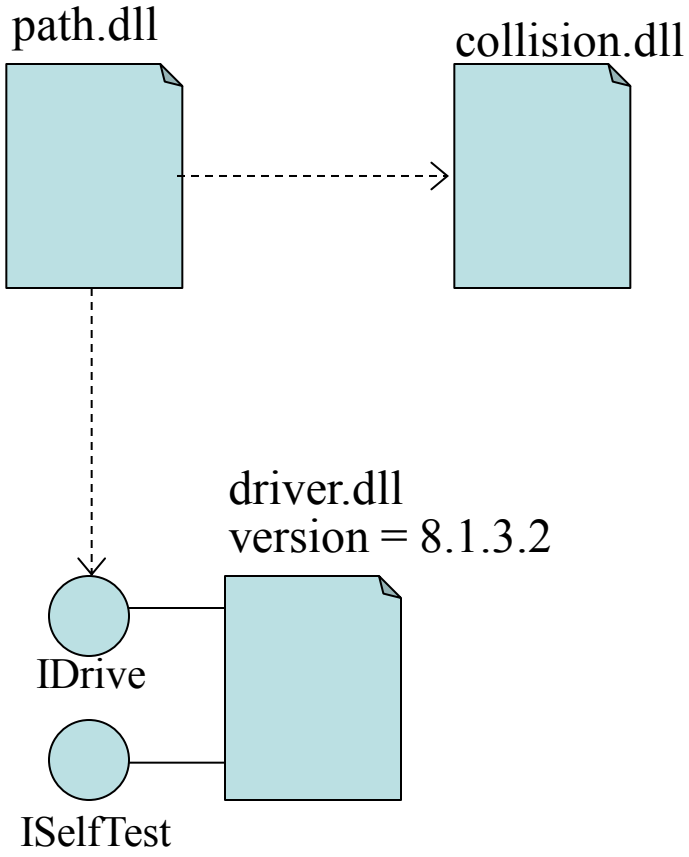
Component Diagram

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]

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.

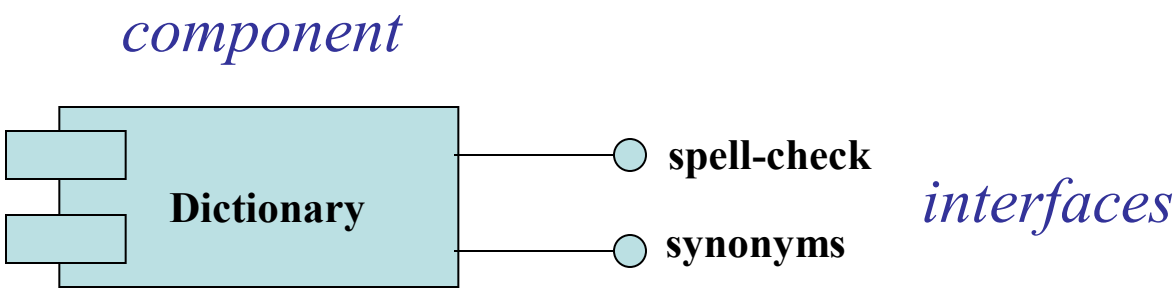
Use deployment diagrams to model the ***static deployment view*** of a system. This involves modeling the topology of the hardware on which the system executes.

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

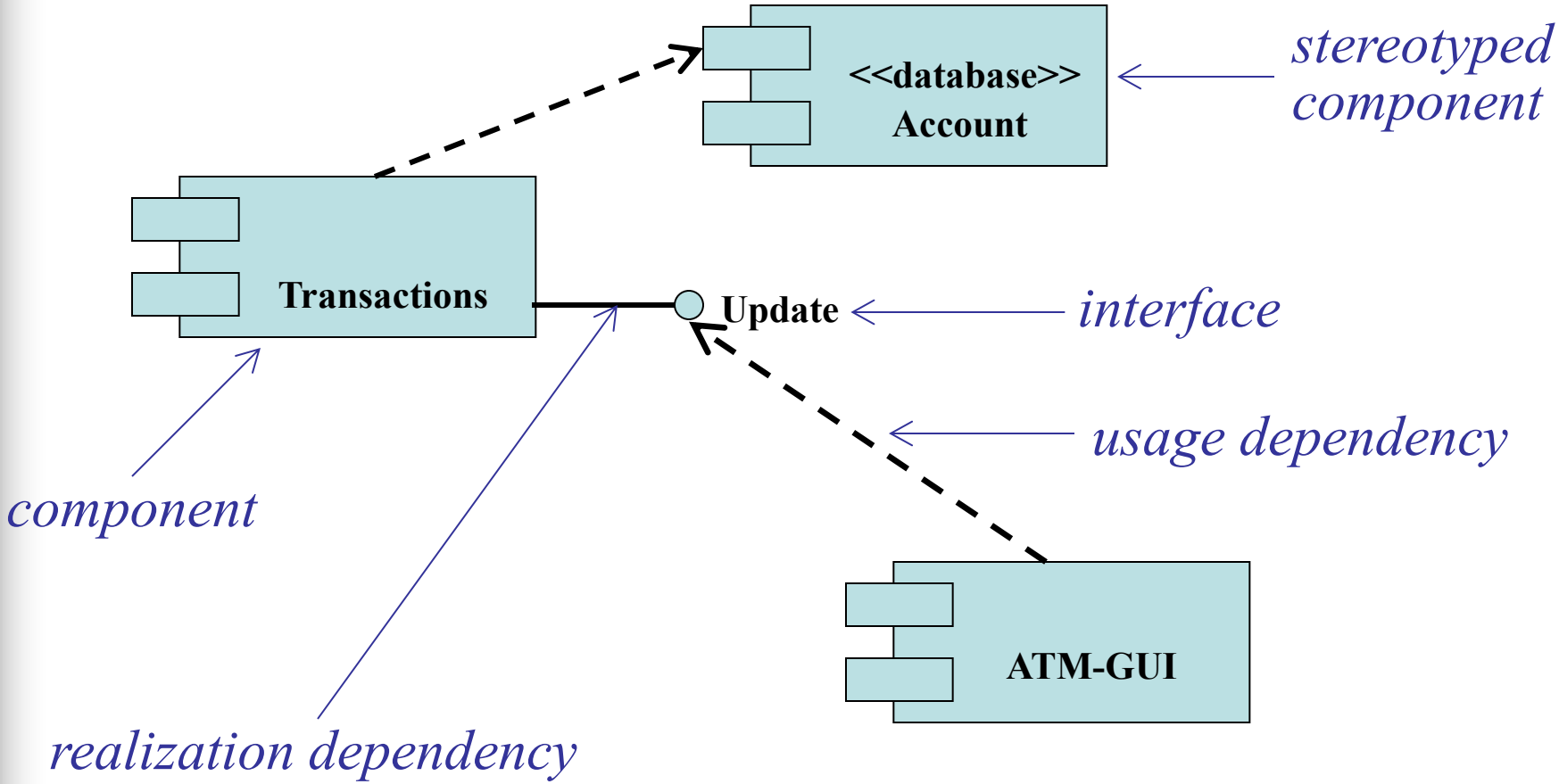
Deployment Diagram

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]*

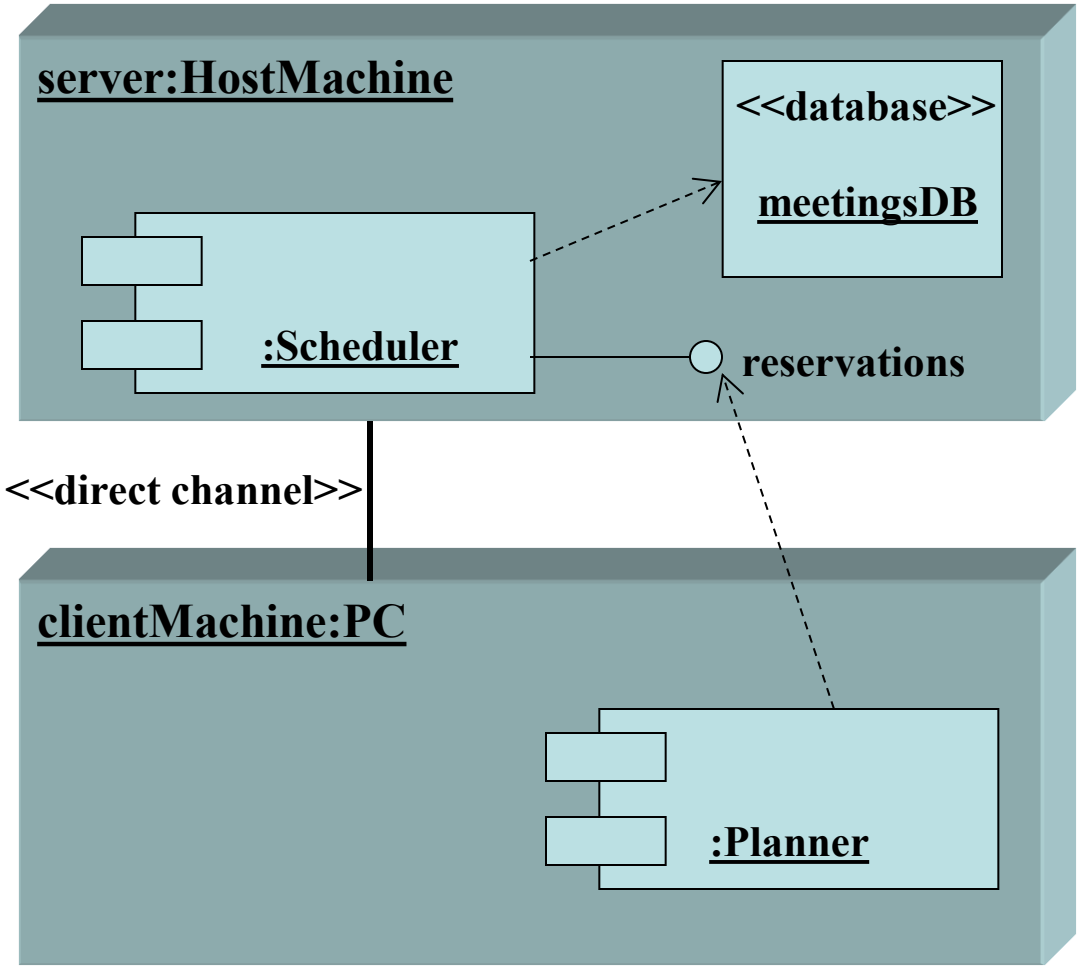


Deployment Diagram



[Rumbaugh,99]

Deployment Diagram



Deployment diagram of a client-server system.

[Rumbaugh,99]

Software Design

Dynamic Modeling using the Unified Modeling Language (UML)

Use Case

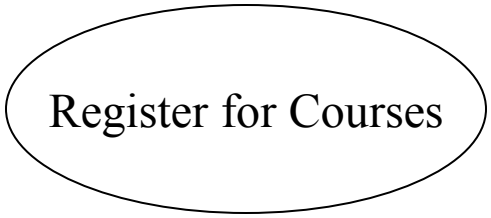
"A *use case* specifies the behavior of a system or a part of a system, and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor."

- *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."

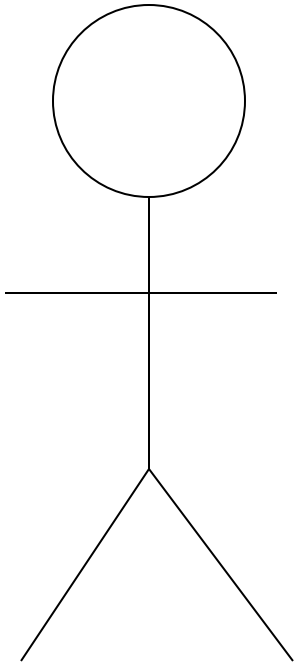
- *The UML Reference Manual, [Rumbaugh,99]*

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)

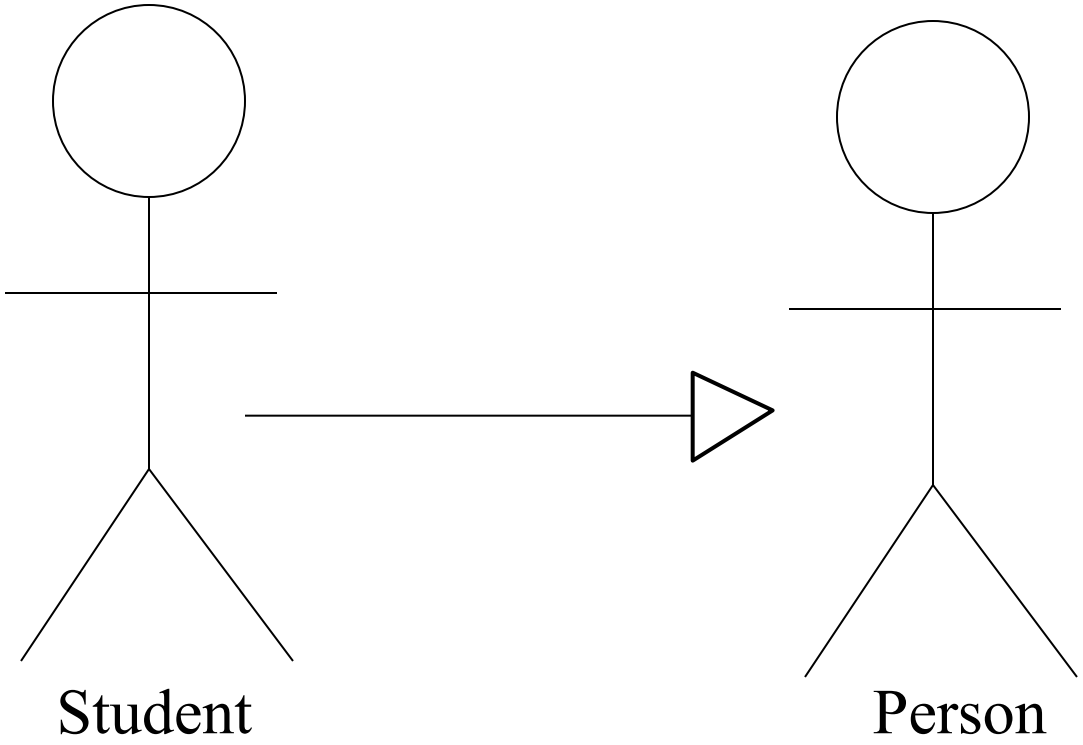


Student

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

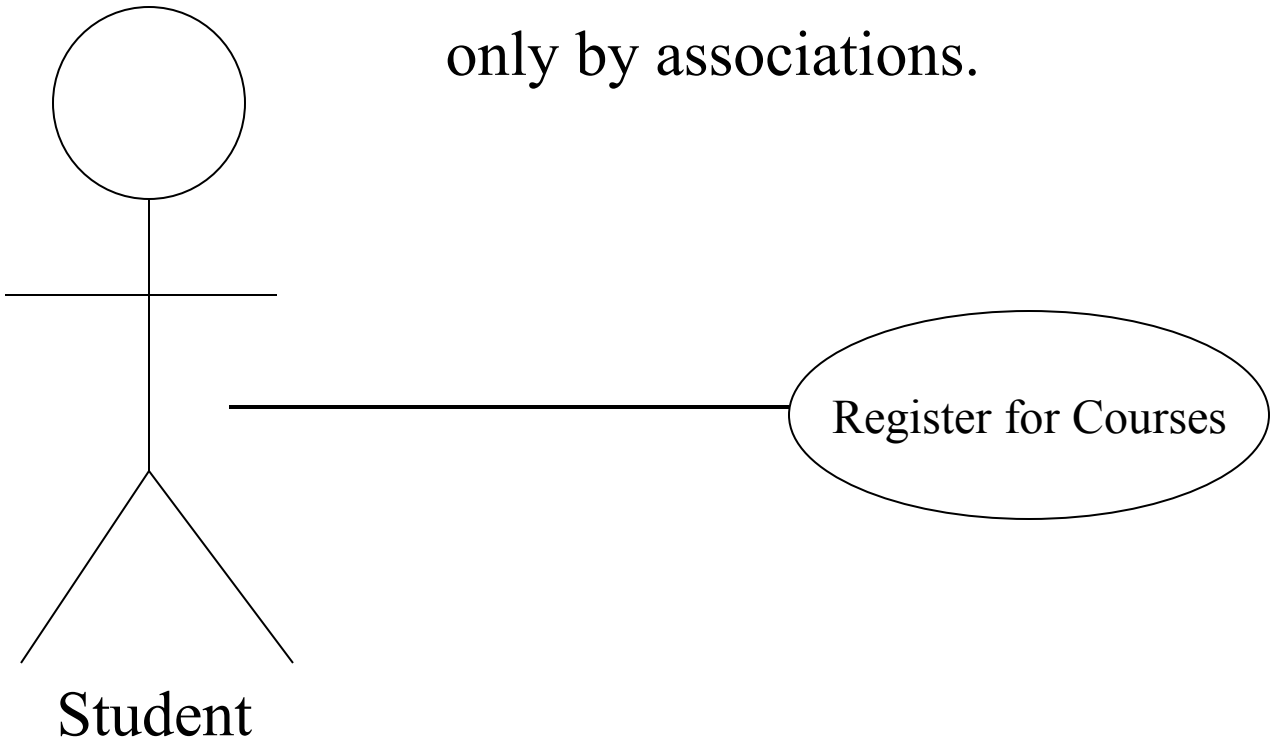
Use Case (Cont'd)

Actors can participate in a generalization relation with other actors.



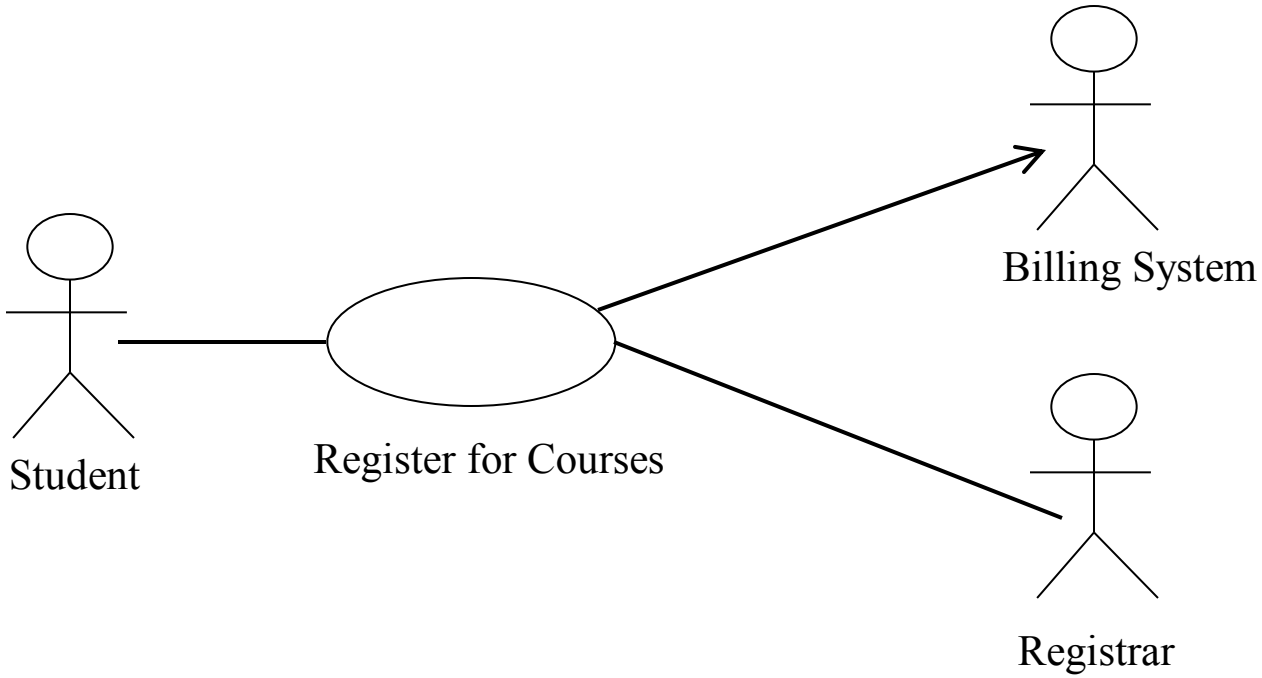
Use Case (Cont'd)

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.



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."

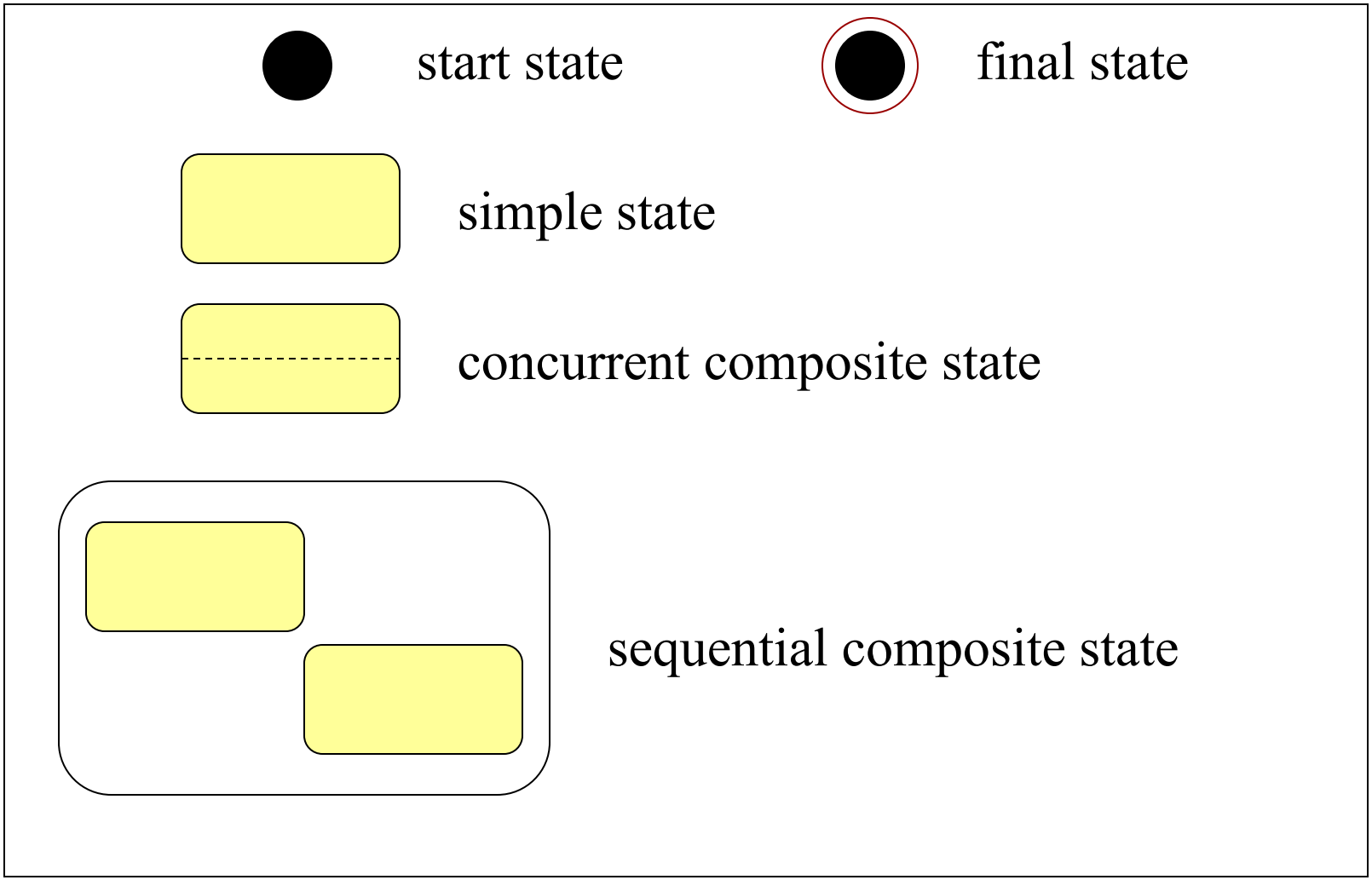
- *The UML Reference Manual, [Rumbaugh,99]*

State Machine

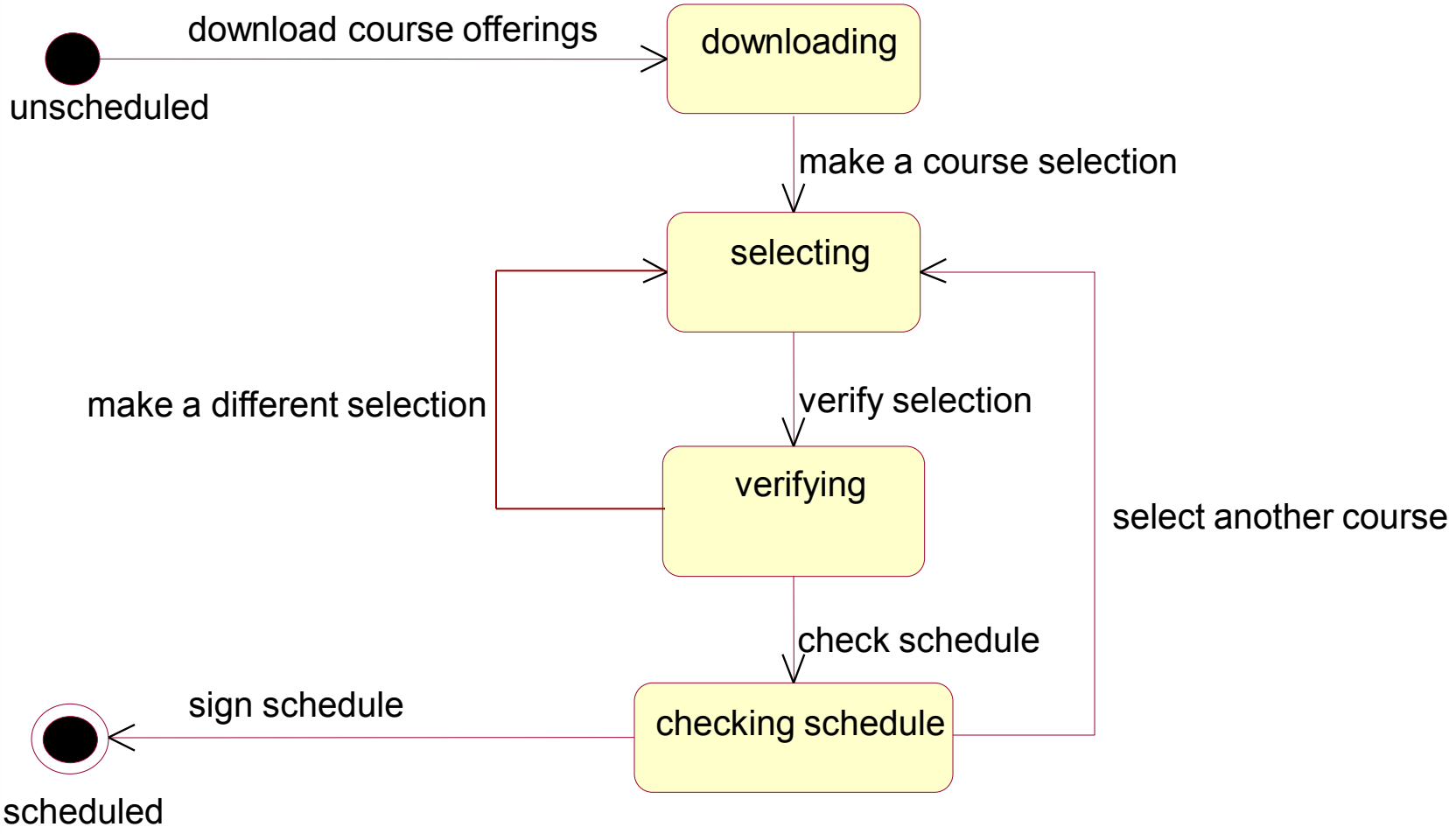
An object must be in some specific state at any given time during its lifecycle. An object transitions from one state to another as the result of some event that affects it. You may create a state diagram for any class, collaboration, operation, or use case in a UML model .

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

State Machine



State Machine



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.

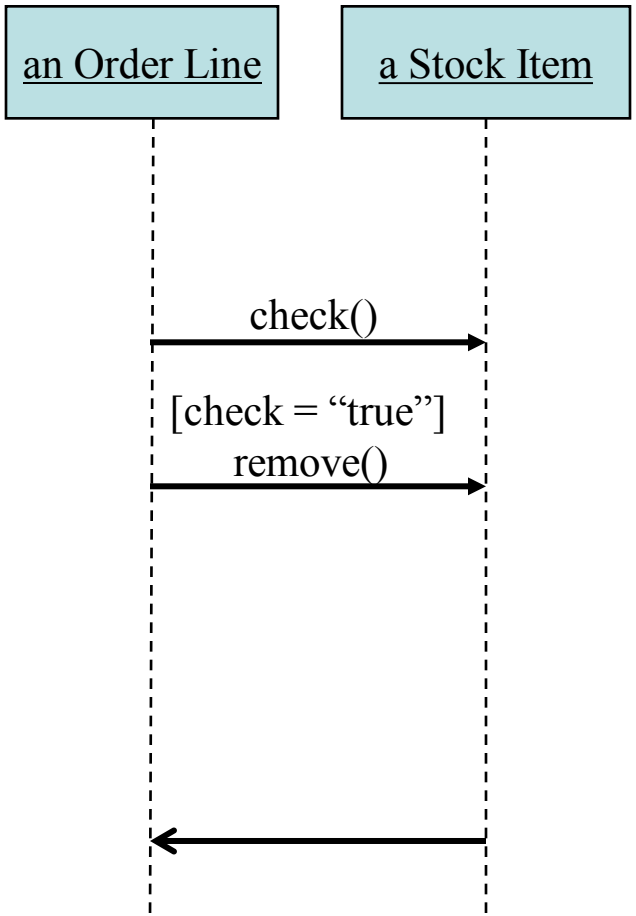
Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.

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

Sequence Diagram

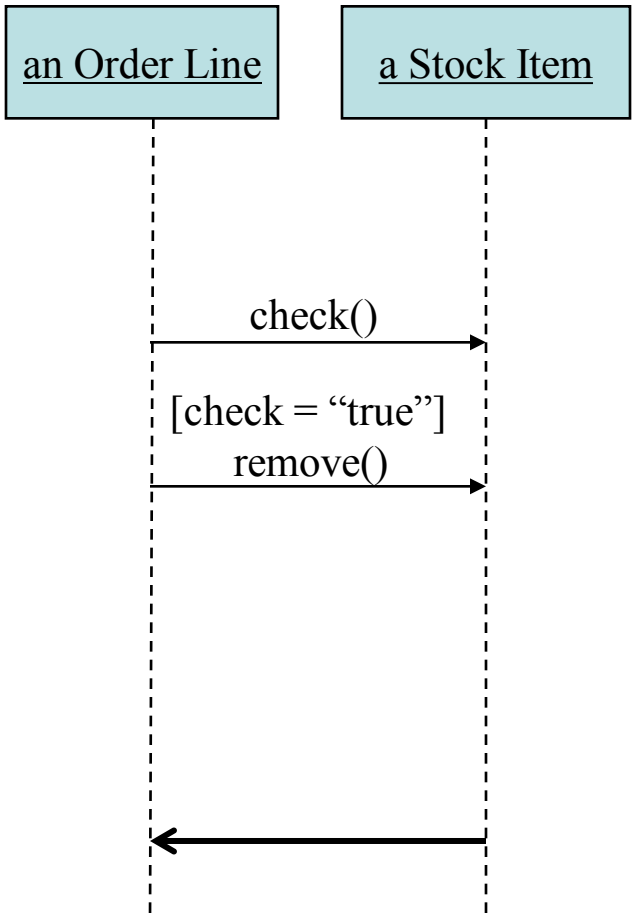
An object in a sequence diagram is rendered as a box with a dashed line descending from it. The line is called the *object lifeline*, and it represents the existence of an object over a period of time.

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.

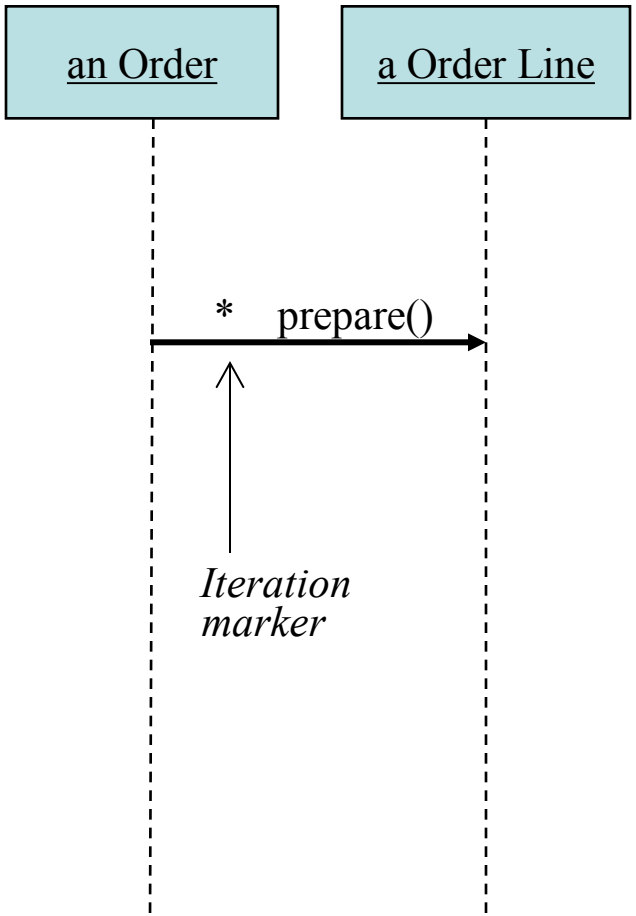
Sequence Diagram



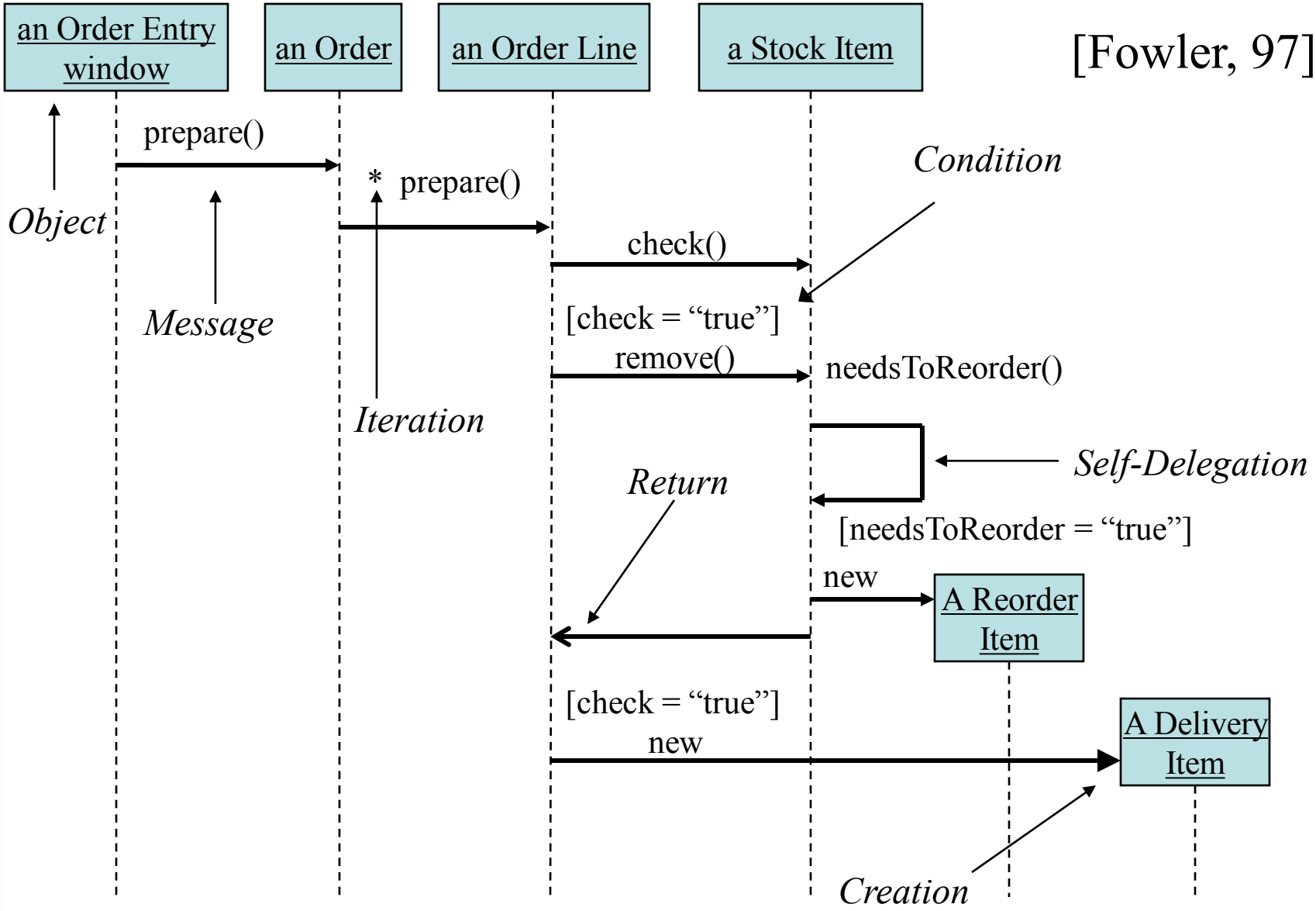
Notice that the bottom arrow is different. The arrow head is not solid, and there is no accompanying message.

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.

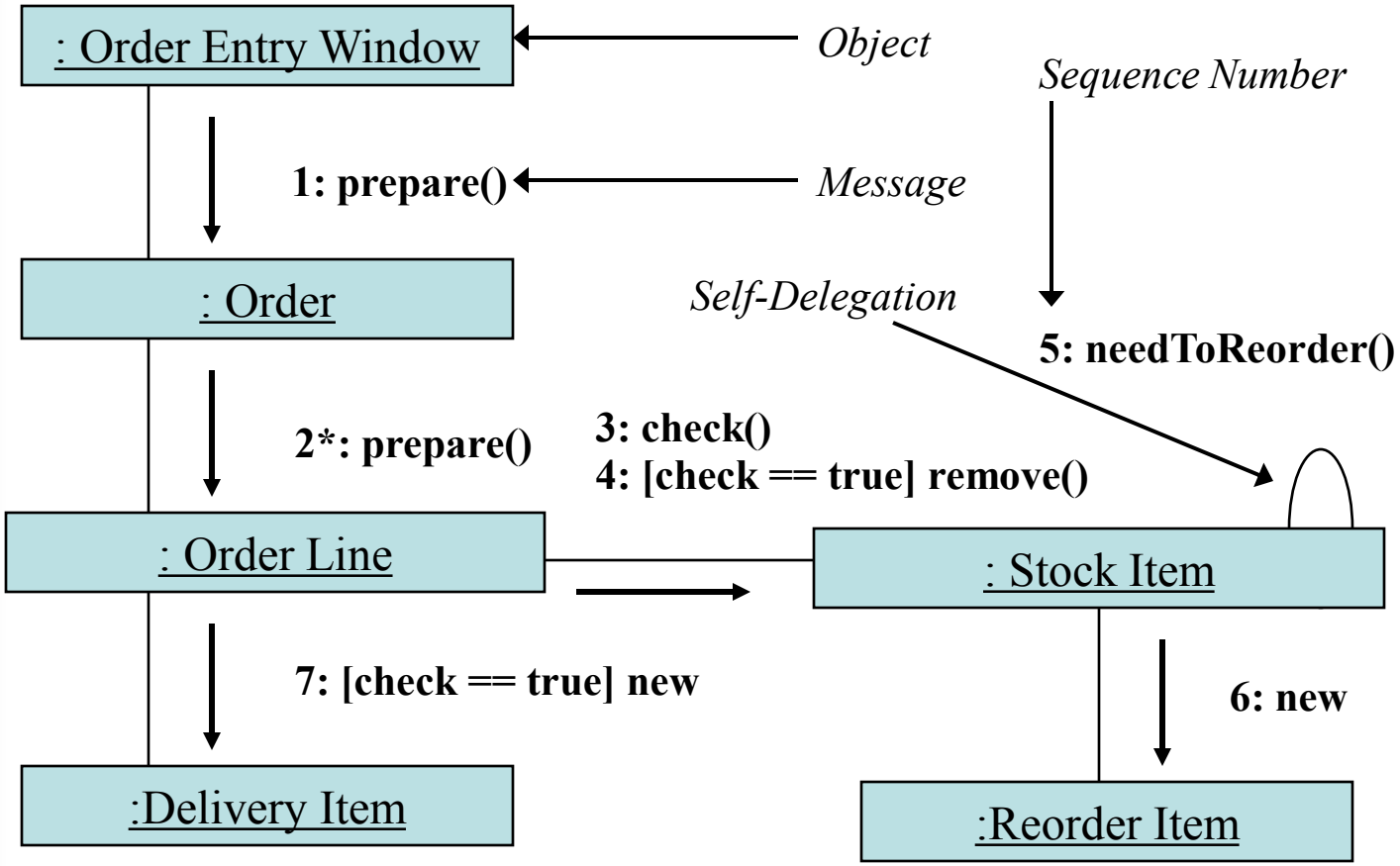

an Order Entry window
an Order
an Order Line
a Stock Item
[Fowler, 97]
prepare()
* prepare()
Condition
Object
check()
Message
[check = "true"]
remove()
needsToReorder()
Iteration
Self-Delegation
Return
[needsToReorder = "true"]
new
A Reorder Item
[check = "true"]
new
A Delivery Item
Creation

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]

Collaboration Diagram Sequence Diagram

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.

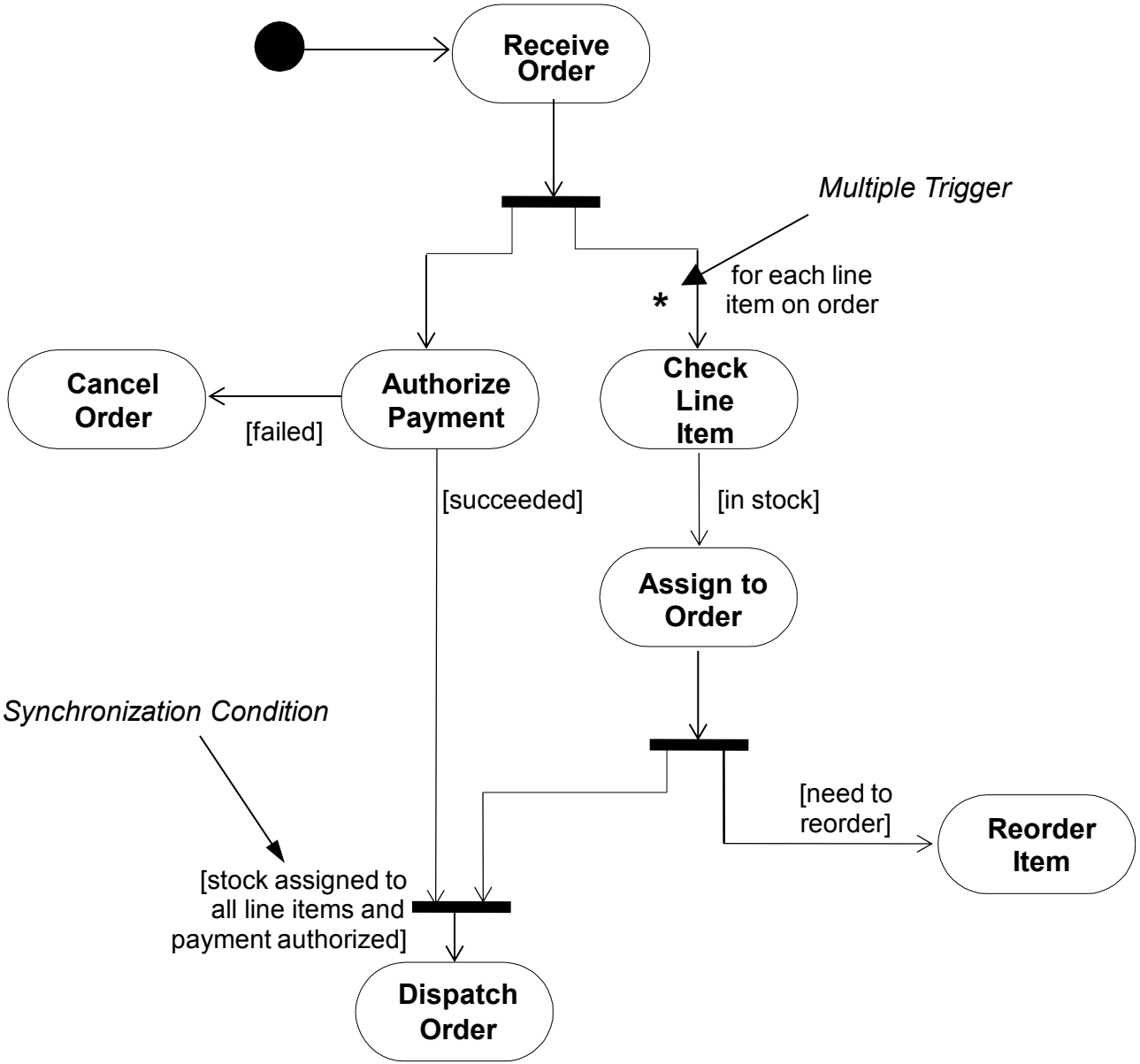
Activity Diagram

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
Authorize Payment
Check Line Item
[failed]
[succeeded]
[in stock]
Assign to Order
Synchronization Condition
[need to reorder]
Reorder Item
[stock assigned to all line items and payment authorized]
Dispatch Order