Software Design

Static Modeling using the Unified Modeling Language (UML)

Tools for creating UML diags

➢Eclipse Papyrus 2.0

➢Easy UML (Plugin for Netbeans IDE)

➢Visual Paradigm

➢https://www.predictiveanalyticstoday.com/open-source-free-unified-modeling-language-uml-tools/

➢Rational Rose

- http://www.rational.com/

UML Methodology

➢ as a blueprint: a complete design to be implemented

- sometimes done with CASE (Computer-Aided Software Engineering) tools

➢ as a programming language: with the right tools, code can be auto-generated and executed from UML

Classes

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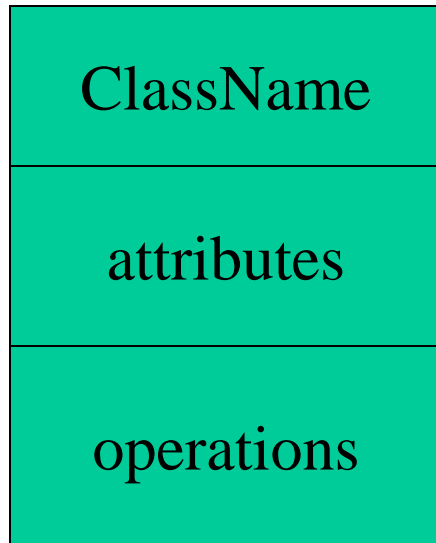
A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

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

Class Names

ClassName
attributes
operations

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

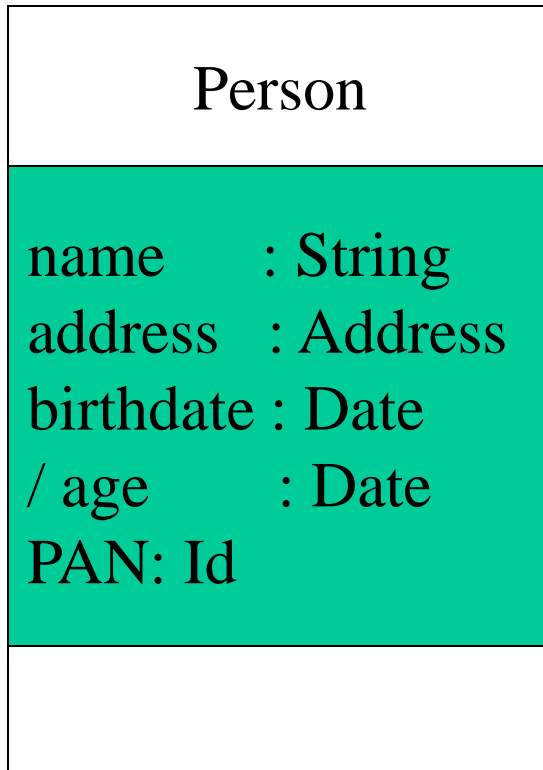| Person |
| --- |
| name   : String
address   : Address
birthdate : Date
PAN   : Id |
| |
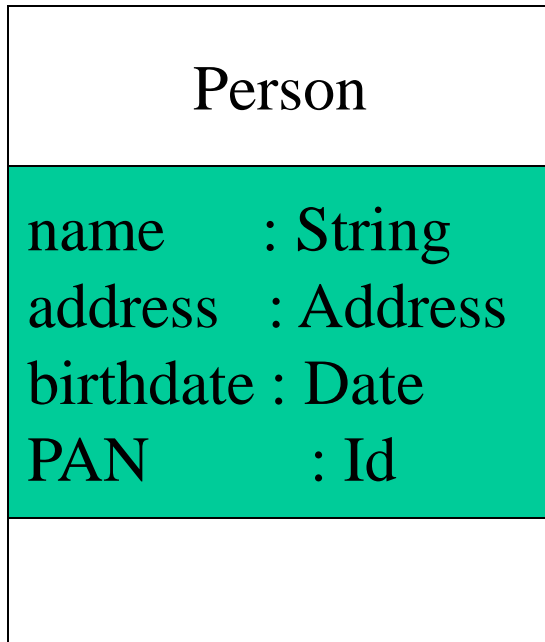
An *attribute* is a named property of a class that describes the object being modeled. In the class diagram, attributes appear in the second compartment just below the name-compartment.

Class Attributes (Cont'd)

Person
name   : String
address   : Address
birthdate : Date
/ age   : Date
PAN: Id

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:

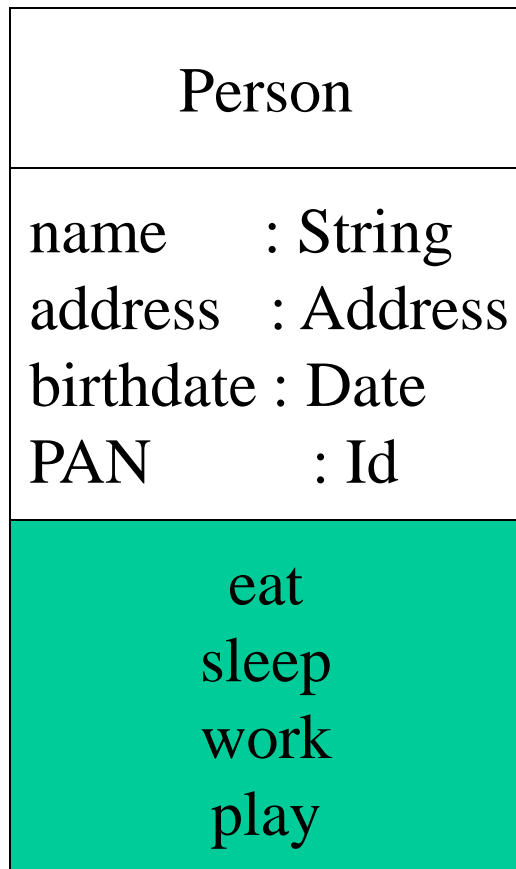/ age : Date

Class Attributes (Cont'd)

Person
+ name   : String # address   : Address # birthdate : Date / age   : Date - PAN   : Id
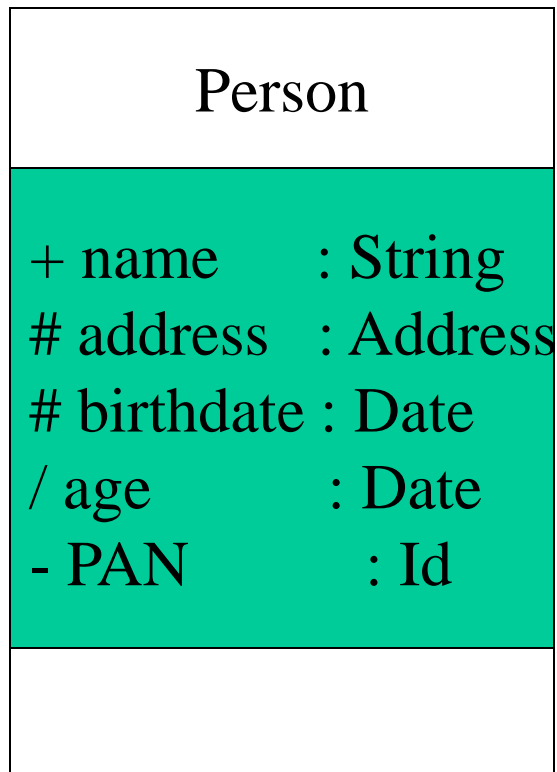
Attributes can be:
+ public
# protected
- private
/ derived

underline static attributes

Class Operations

Person
name : String address : Address birthdate : Date PAN : Id
eat sleep work play

Operations describe the class behavior and appear in the third compartment.
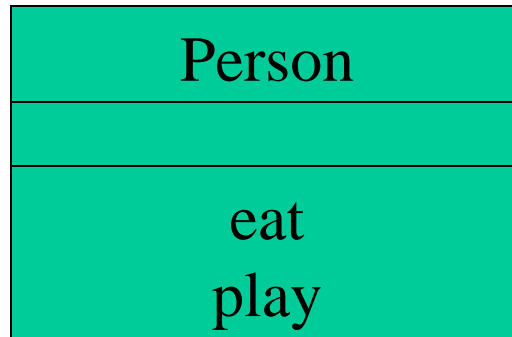
Class Operations (Cont'd)

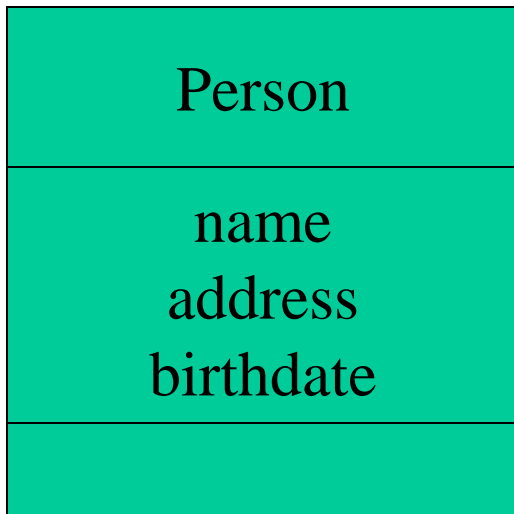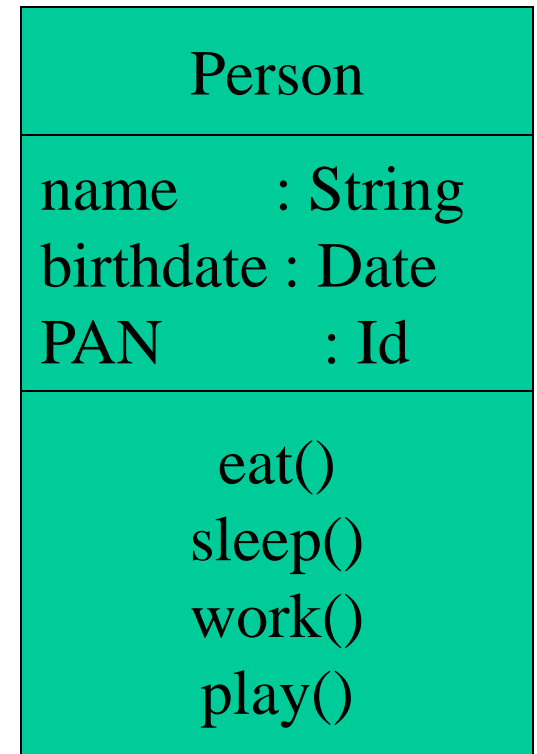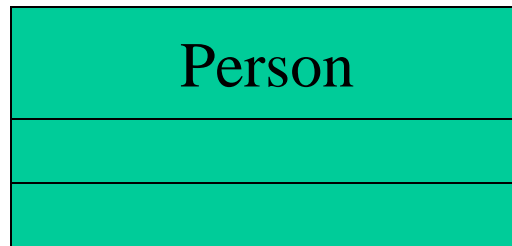| PhoneBook |
|---|
| |
| newEntry (n : Name, a : Address, p : PhoneNumber, d : Description)
getPhone (n : Name, a : Address) : PhoneNumber |

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.

Person

Person

Person
name   : String
birthdate : Date
PAN   : Id
eat()
sleep()
work()
play()

Person
name
address
birthdate

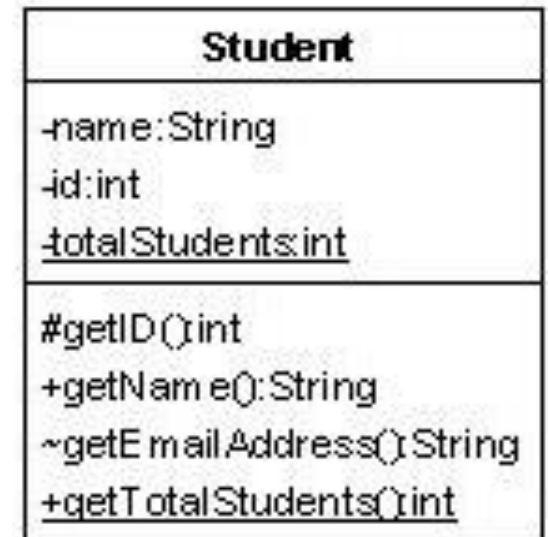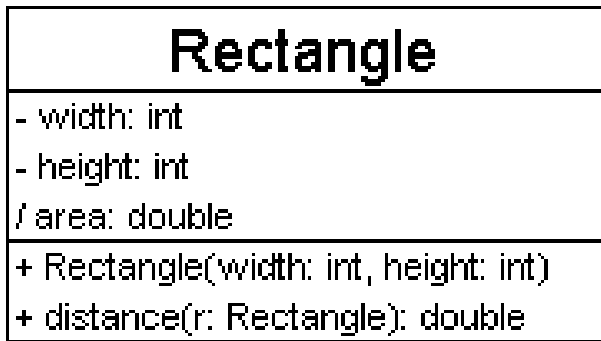| Person |
|--------|
| |
| eat |
| play |

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.

SmokeAlarm
Responsibilities -- sound alert and notify guard station   when smoke is detected.   -- indicate battery state

Software Design (UML)
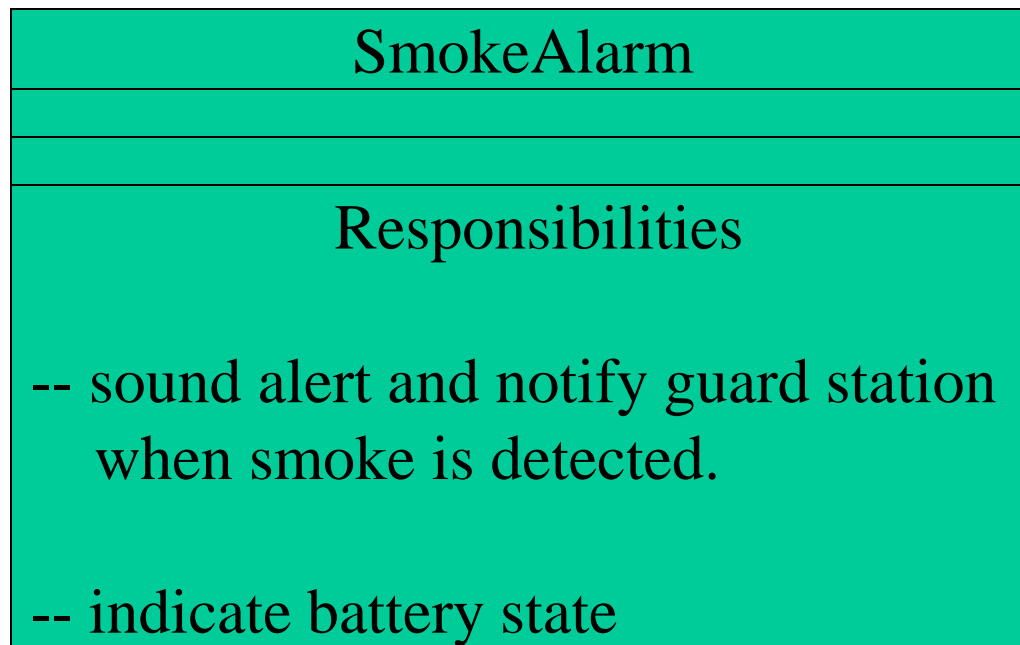
Example

➢ attributes (optional)
  should include all fields of the object

➢ operations / methods (optional)
  • may omit trivial (get/set) methods
    o but don't omit any methods from an interface!
  • should not include inherited methods

Rectangle
- width: int
- height: int
/ area: double
+ Rectangle(width: int, height: int)
+ distance(r: Rectangle): double

Student
-name:String
-id:int
totalStudents:int
#getID():int
+getName():String
~getEmailAddress():String
+getTotalStudents():int

Software Design (UML)
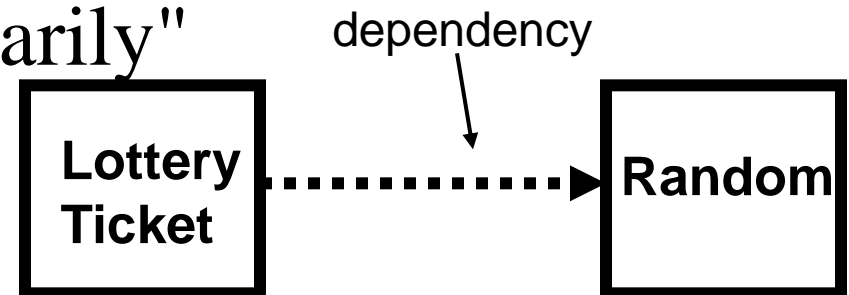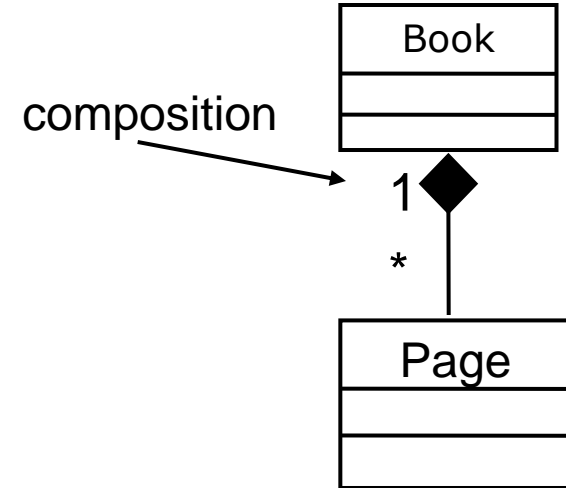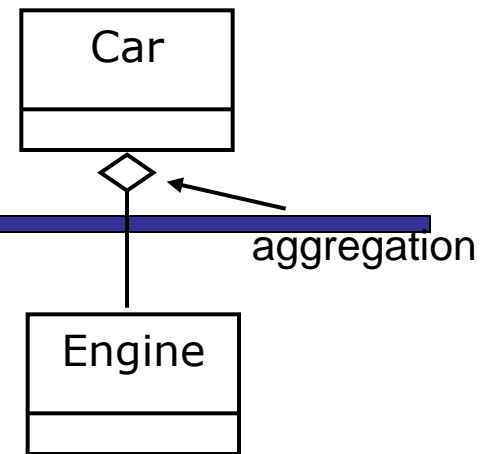
Relationships

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

➢ **generalization**: an inheritance relationship
  o inheritance between classes
  o interface implementation

➢ **association**: a usage relationship
  o dependency
  o aggregation
  o composition

Association types

Car

➤**aggregation**: "is part of"
  – *Weak relationship*

aggregation

Engine

  – symbolized by a clear white diamond

Book

➤**composition**: "is entirely made of"

composition

  – *stronger version* of aggregation

1

  – the parts live and die with the whole

*

  – symbolized by a black diamond

Page

➤**dependency**: "uses temporarily"

dependency

  – symbolized by dotted line

Lottery Ticket ┄┄┄➤ **Random**

  – often is an implementation detail, not an intrinsic part

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*.

```
┌─────────────────────────┐
│ CourseSchedule          │
├─────────────────────────┤              ┌──────────────────┐
│                         │ - - - - - -> │ Course           │
├─────────────────────────┤              └──────────────────┘
│ add(c : Course)         │
│ remove(c : Course)      │
└─────────────────────────┘
```

Generalization Relationships

Person

Student
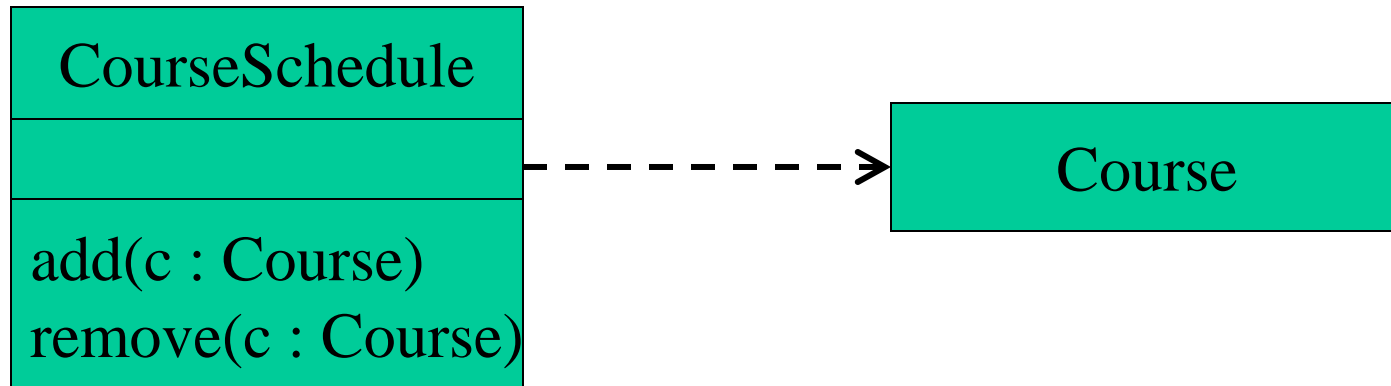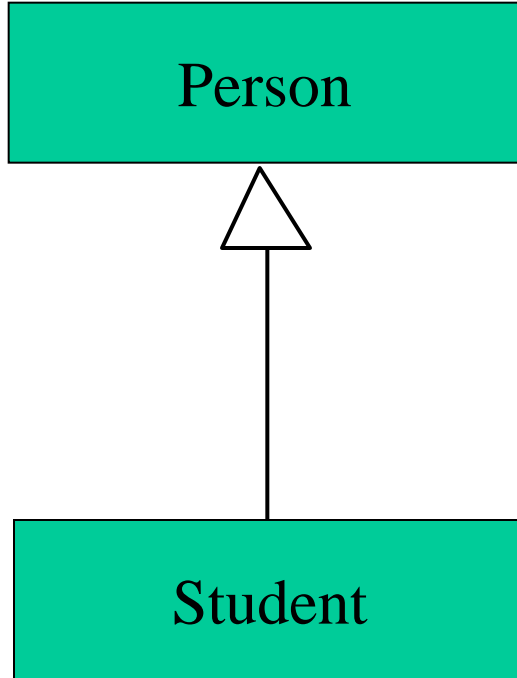
A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.
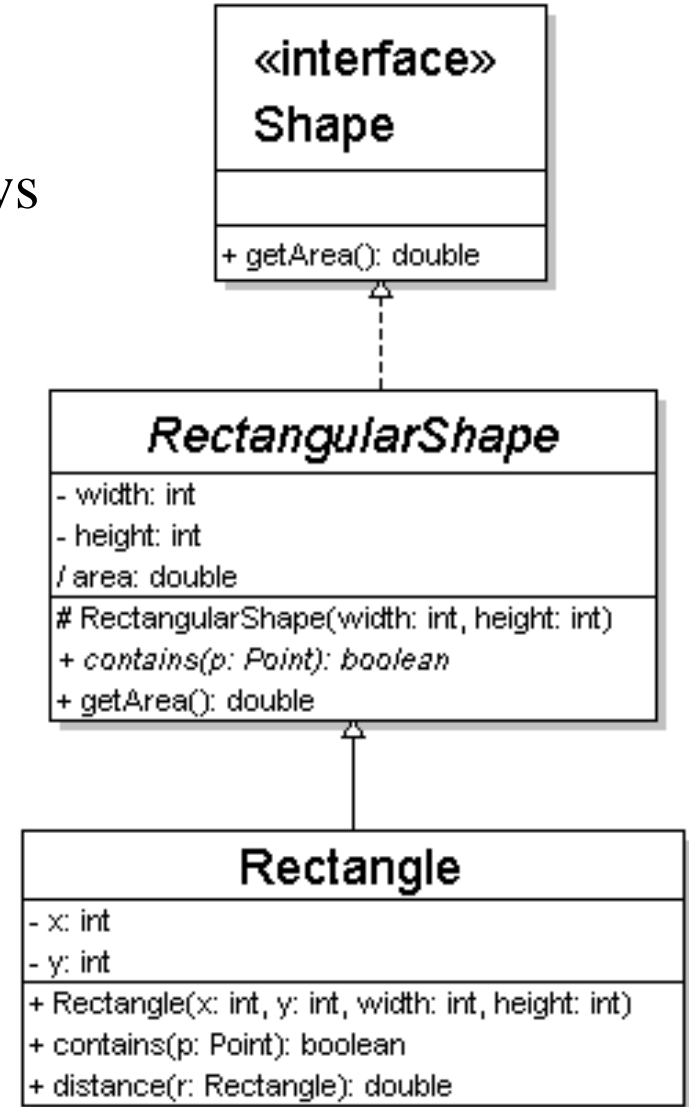
Generalization Relationships (Cont'd)

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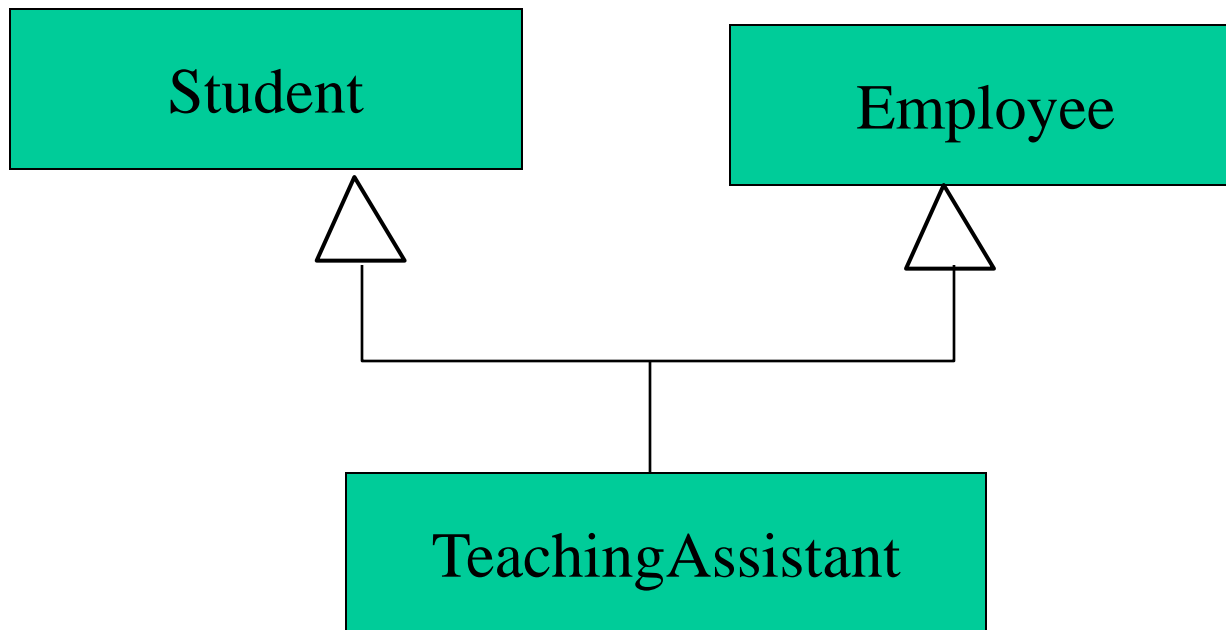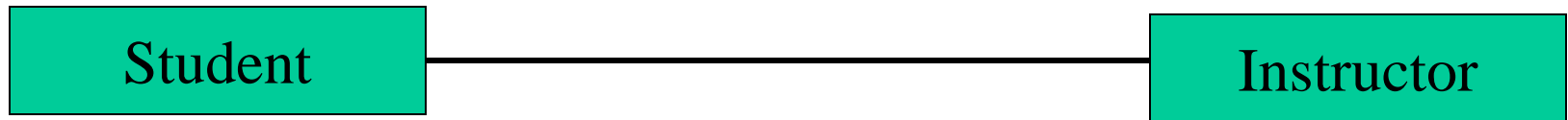
Generalization Relationships (Cont'd)

➢ generalization (inheritance) relationships

- hierarchies drawn top-down with arrows pointing upward to parent
- line/arrow styles differ, based on whether parent is a(n):
  - <u>class</u>:
    solid line, black arrow
  - <u>abstract class</u>:
    solid line, white arrow
  - <u>interface</u>:
    dashed line, white arrow

```
«interface»
Shape
─────────────────
─────────────────
+ getArea(): double
```

```
RectangularShape
─────────────────────────────────
- width: int
- height: int
/ area: double
─────────────────────────────────
# RectangularShape(width: int, height: int)
+ contains(p: Point): boolean
+ getArea(): double
```

```
Rectangle
─────────────────────────────────────────
- x: int
- y: int
─────────────────────────────────────────
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double
```

Association Relationships

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

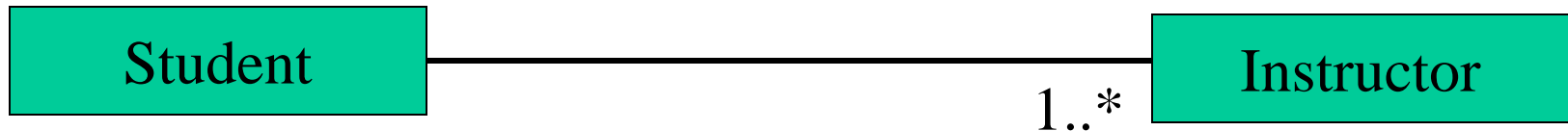An *association* denotes that link.

Student		Instructor

Association Relationships (Cont'd)

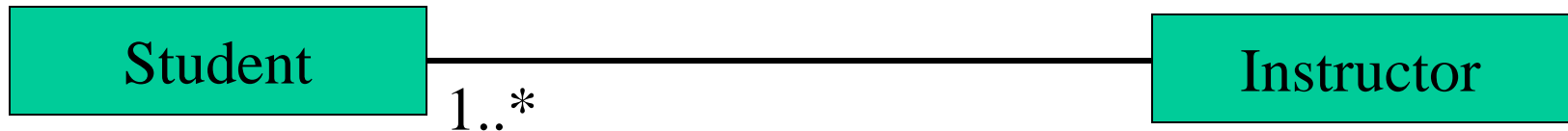We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

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

| Student | —————————— | Instructor |

1..*

Association Relationships (Cont'd)

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

| Student | 1..* ———————— | Instructor |

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.*

Student	teaches learns from	Instructor
	1..* 1..*	
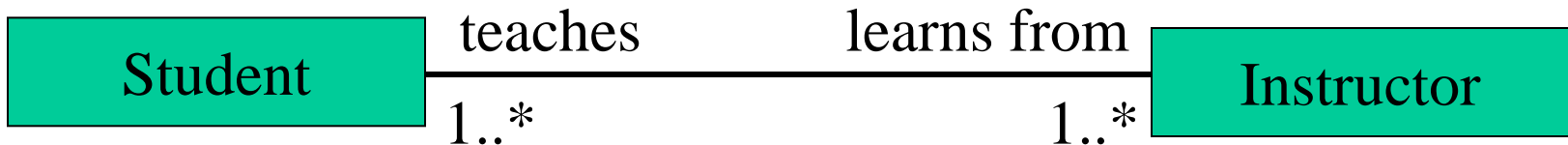
Association Relationships (Cont'd)

We can also name the association.

| Student | —— membership —— | Team |

1..*   1..*
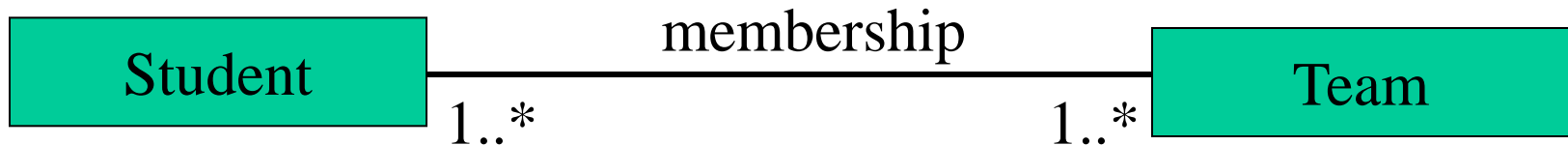
Association Relationships (Cont'd)

We can specify dual associations.

Student	member of	Team
	1..* 1..*	
	1 president of 1..*	
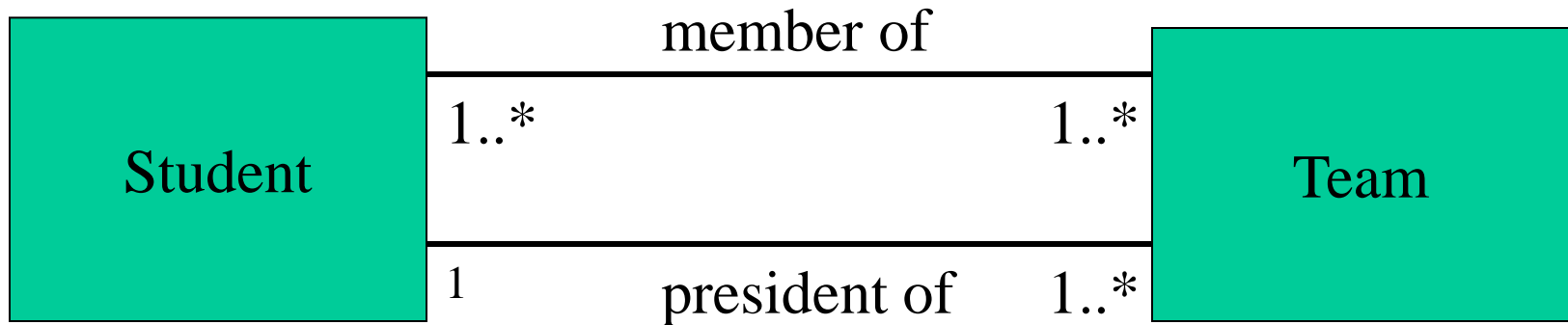
Association Relationships (Cont'd)

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*.

| Router | → | DomainNameServer |
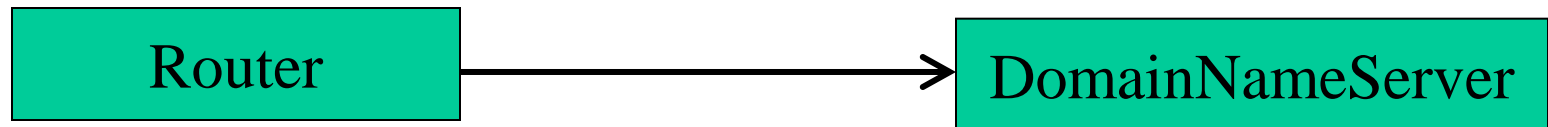
Association Relationships (Cont'd)

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

```
            ┌──────────────────┐
            │   Registration   │
            ├──────────────────┤
            │   modelNumber    │
            │   serialNumber   │
            │   warrentyCode   │
            ├──────────────────┤
            │                  │
            └──────────────────┘
                     ┊
                     ┊
   ┌───────────┐     ┊      ┌───────────┐
   │  Product  │─────┼──────│  Warranty │
   └───────────┘            └───────────┘
```
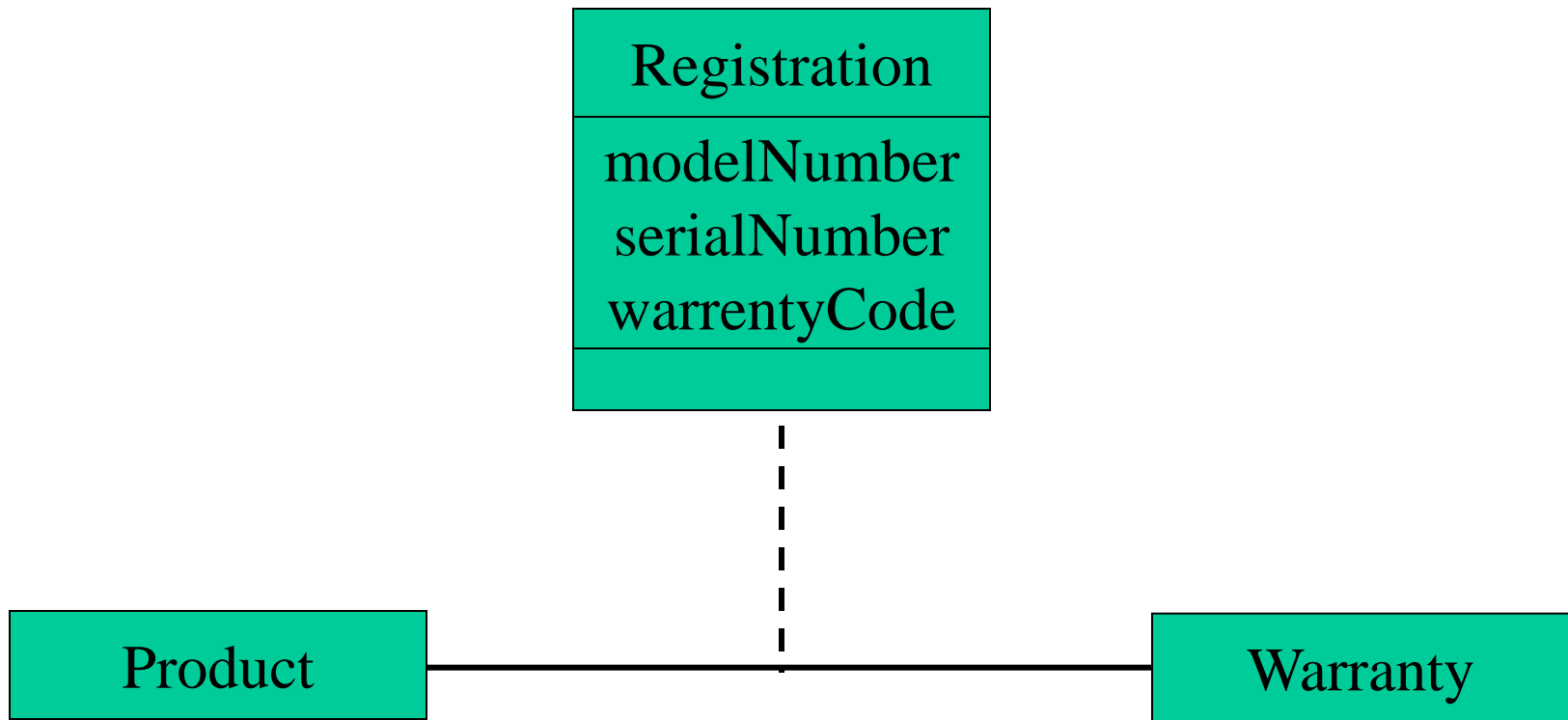
Software Design (UML)

Association Relationships (Cont'd)

A class can have a *self association*.

next

LinkedListNode

previous
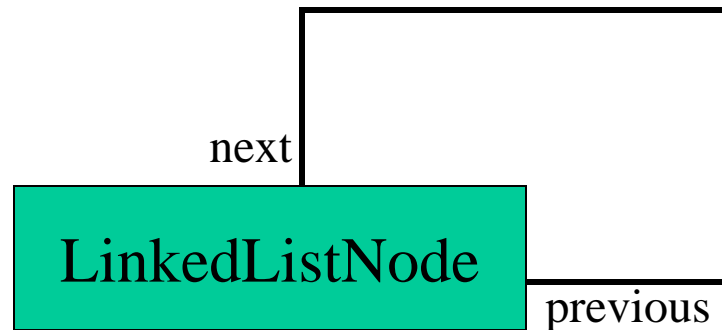
Association Relationships (Cont'd)

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.
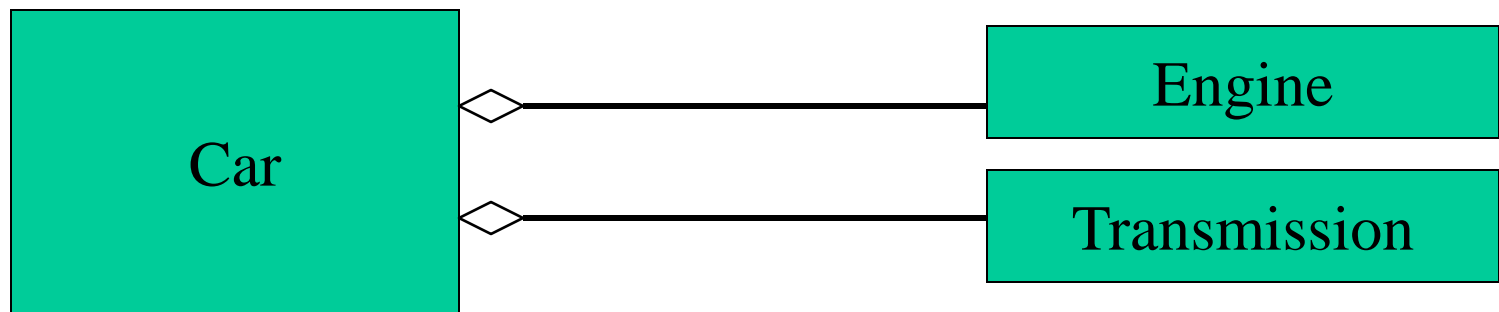
Association Relationships (Cont'd)

A *composition* 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.

Window		
◆ 1 — 1	Scrollbar
◆ 1 — 1	Titlebar
◆ 1 — 1 .. *	Menu

Software Design (UML)
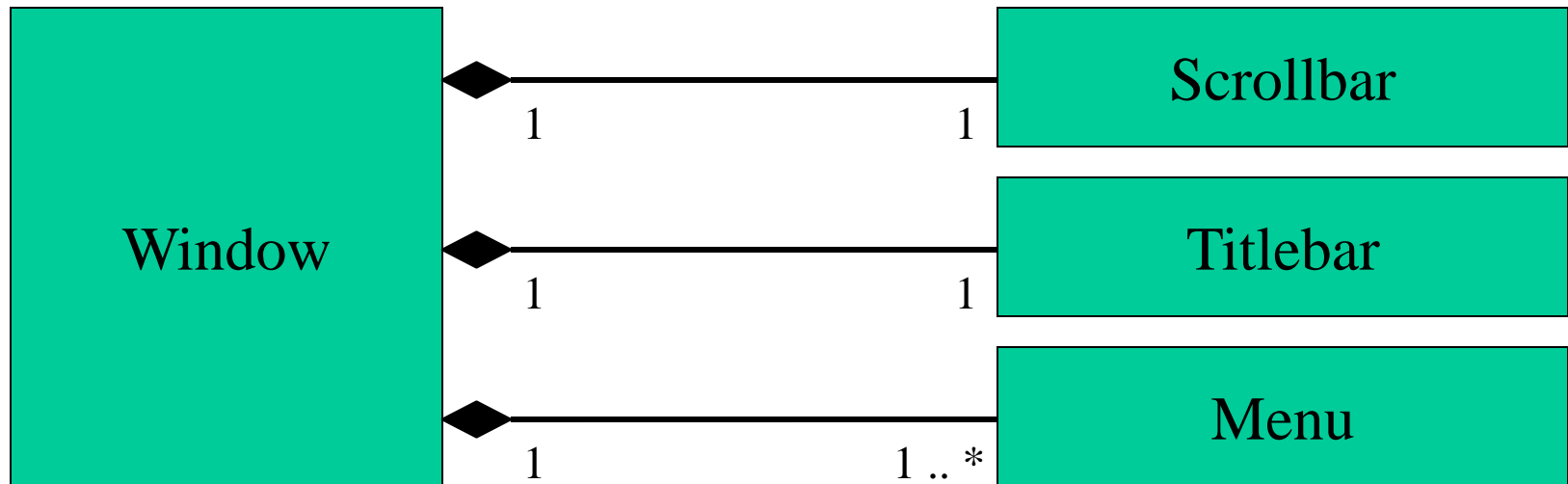
Association Relationships (Cont'd)
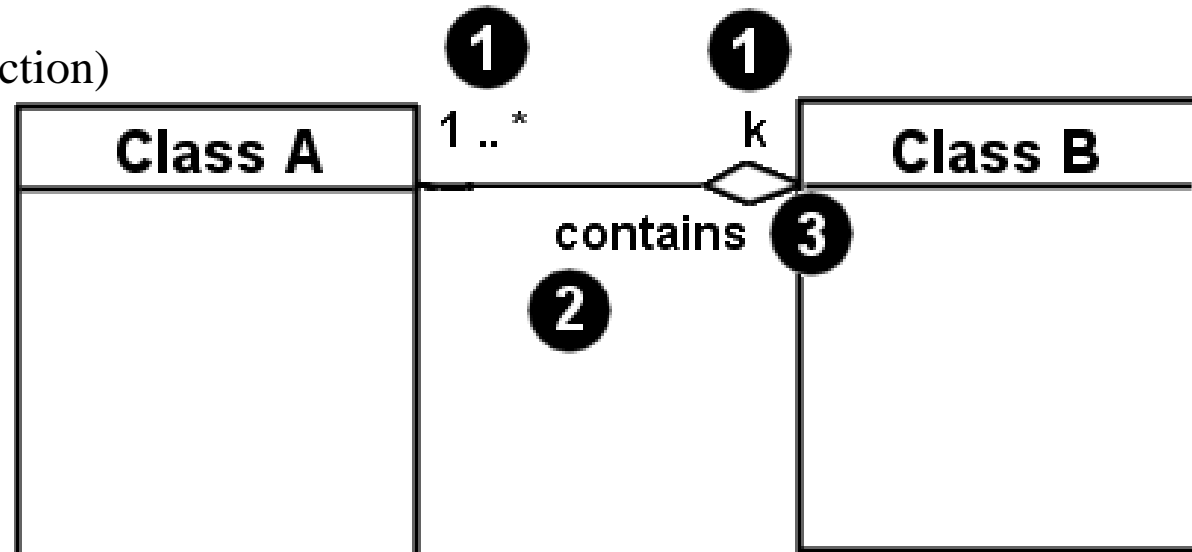
➢ associational (usage) relationships

1. multiplicity   (how many are used)

- *   ⇒ 0, 1, or more
- 1   ⇒ 1 exactly
- 2..4   ⇒ between 2 and 4, inclusive
- 3..*   ⇒ 3 or more

2. name   (what relationship the objects have)

3. navigability   (direction)

Class A		Class B

1 **1**

1 .. *   k

contains **3**

2

Examples

➢ one-to-one
  ▪ each student must carry exactly one ID card

Student
- idCard: IDCard

1 —— carries —— 1

IDCard
- name: String
- id: int
- password: String

➢ one-to-many
  ▪ one rectangle list can contain many rectangles

Rectangle
- x: int
- y: int
+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double

* —— contains —— 1

RectangleList
- list: ArrayList
+ add(r: Rectangle)
+ clear()
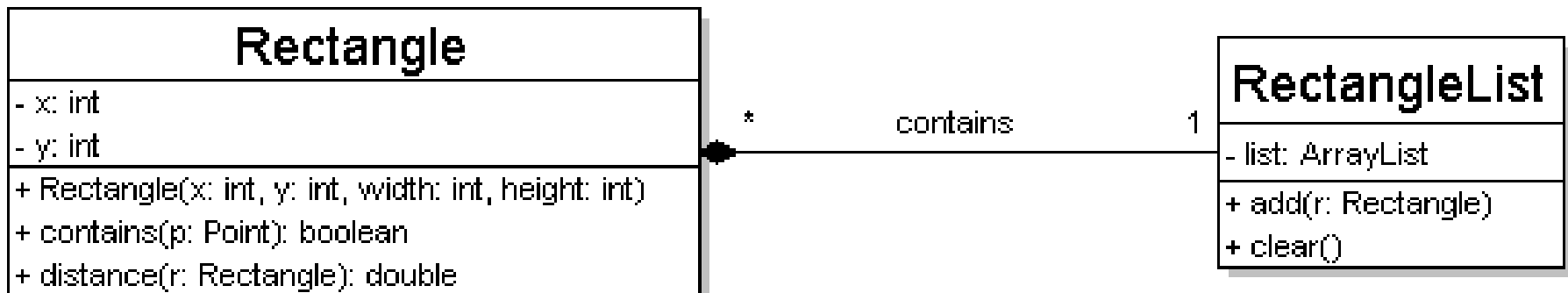
Interfaces

<<interface>>
ControlPanel

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

<<interface>> ControlPanel
getChoices : Choice[]
makeChoice (c : Choice)
getSelection : Selection

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

Interface Realization Relationship

<<interface>>
ControlPanel

specifier

implementation

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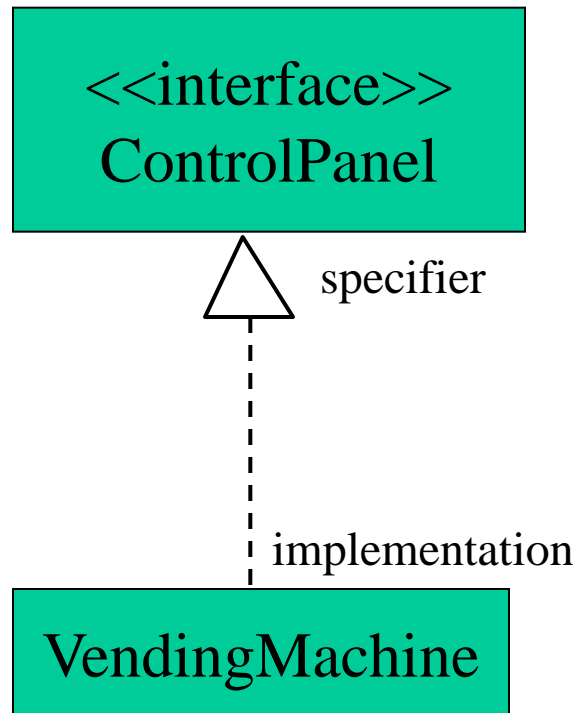
A *realization* relationship connects a class with an interface that supplies its behavioral specification. It is rendered by a dashed line with a hollow triangle towards the specifier.

Interfaces

inputStream

FileWriter

{file must not be locked}

File
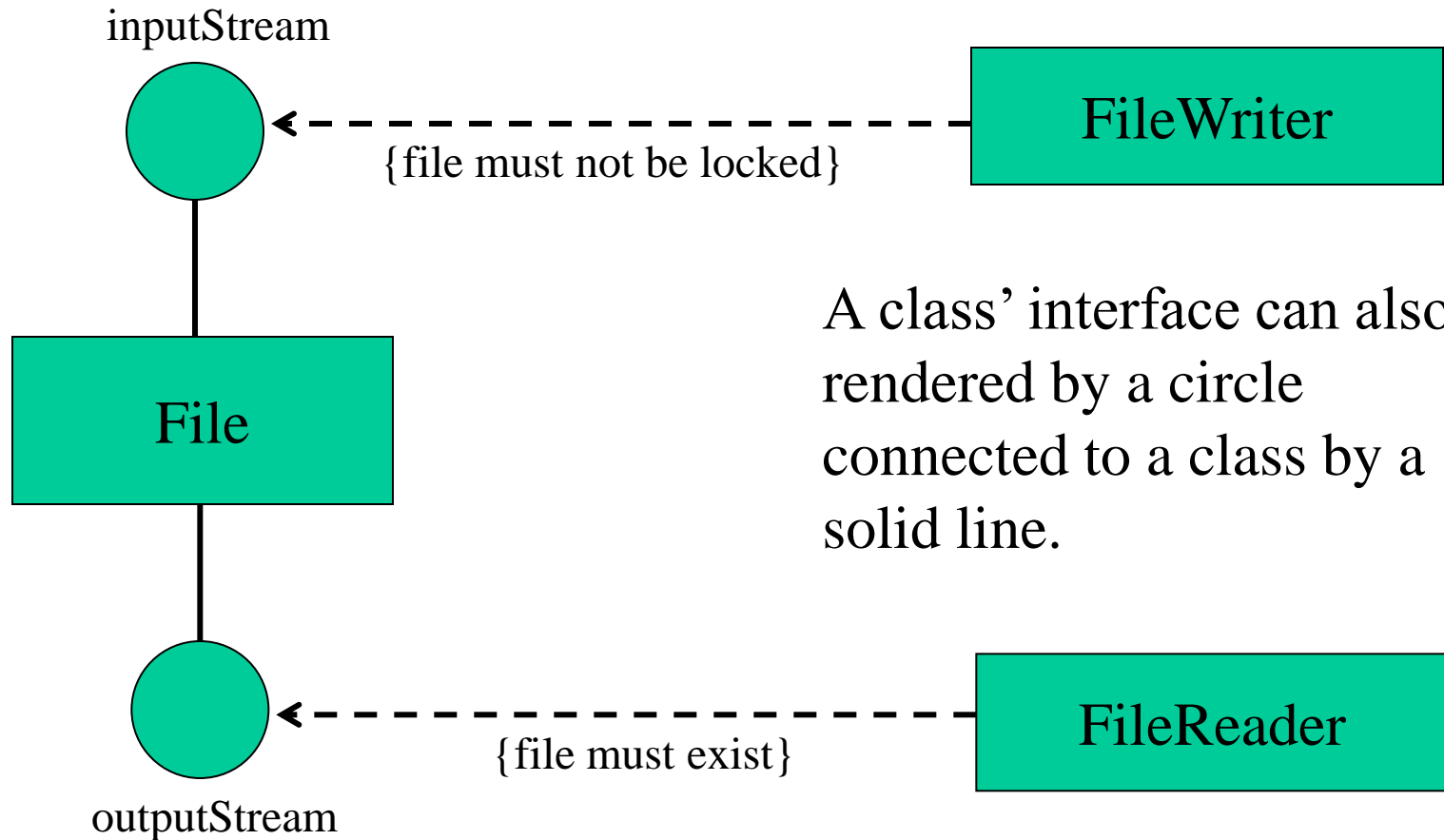
A class' interface can also be rendered by a circle connected to a class by a solid line.

FileReader

{file must exist}

outputStream

Parameterized Class

T

LinkedList

T

1 .. *
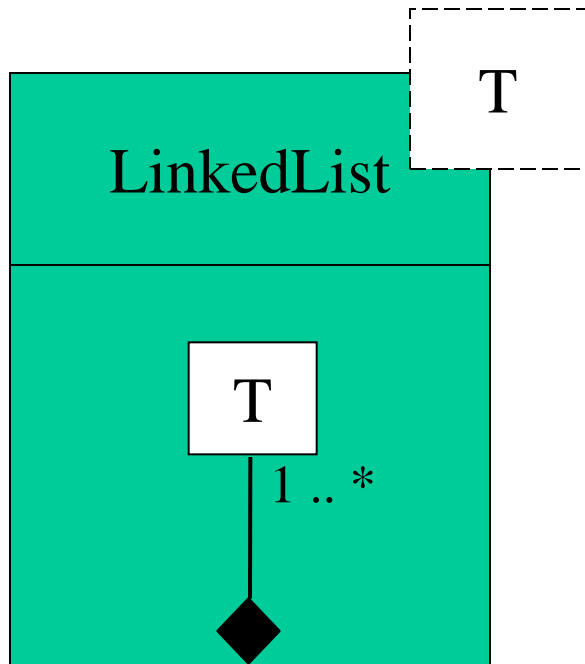
A *parameterized class* or *template* defines a family of potential elements.

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)

T

LinkedList

T

1..*

◆
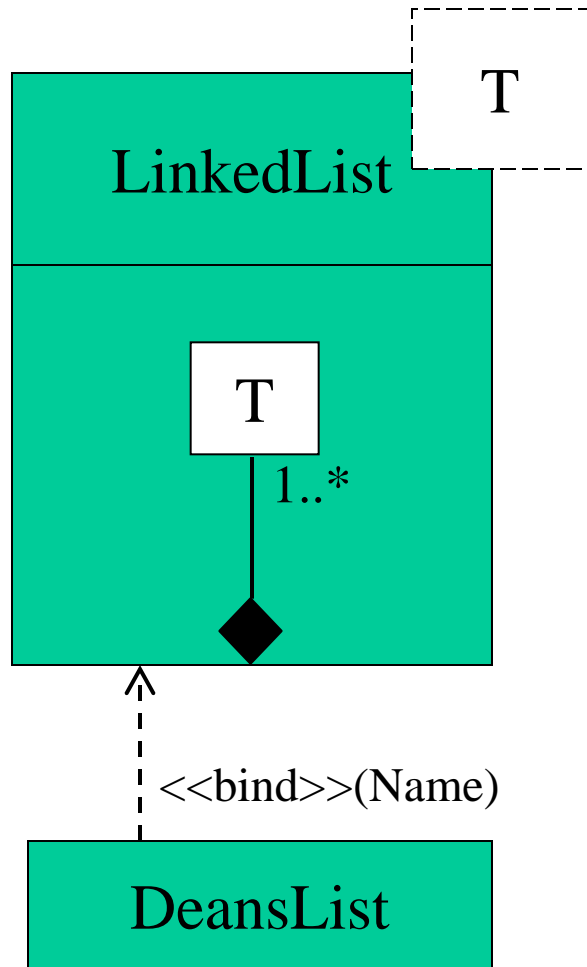
<<bind>>(Name)

DeansList

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

<<enumeration>>
Boolean
false
true

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

Exceptions


```
┌─────────────────────────┐
│     <<exception>>       │
│       Exception         │
├─────────────────────────┤
├─────────────────────────┤
│      getMessage()       │
│    printStackTrace()    │
└─────────────────────────┘
```

```
┌─────────────────────────┐
│     <<exception>>       │
│      KeyException       │
└─────────────────────────┘
```

```
┌─────────────────────────┐
│     <<exception>>       │
│      SQLException       │
└─────────────────────────┘
```
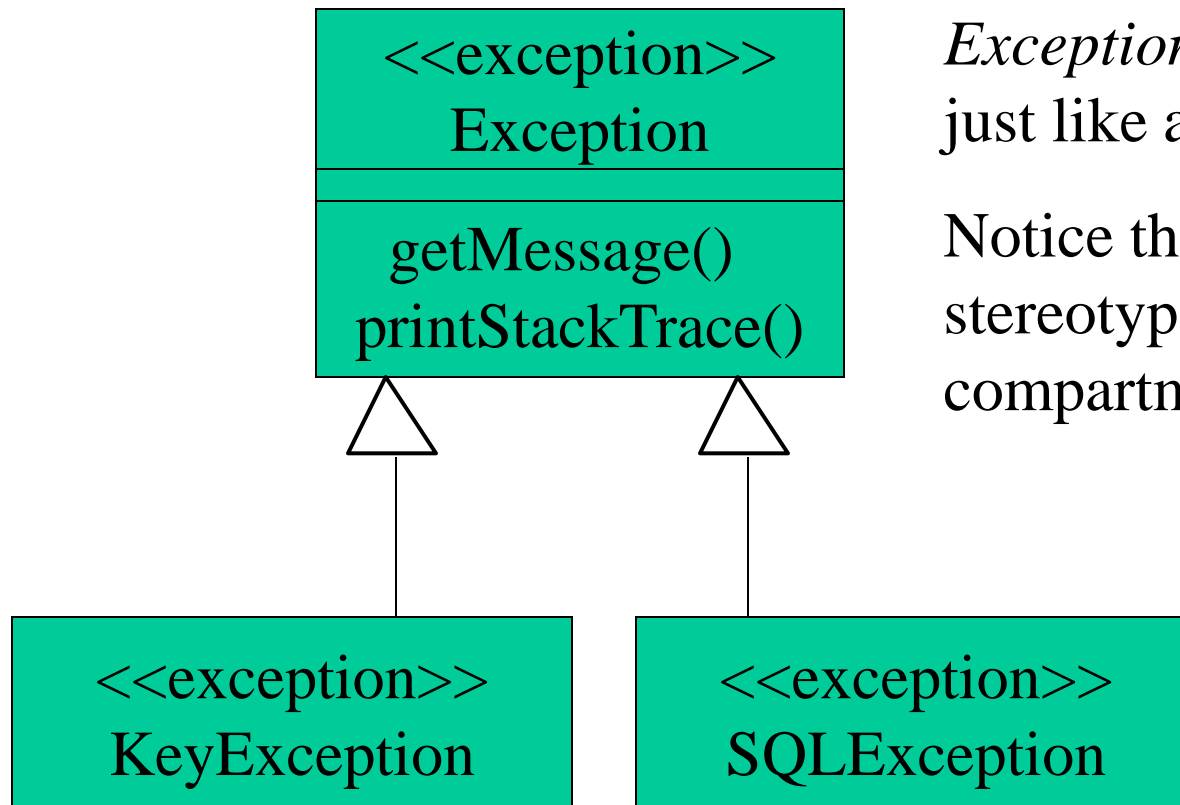
Exceptions can be modeled just like any other class.

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

Software Design (UML)

Packages

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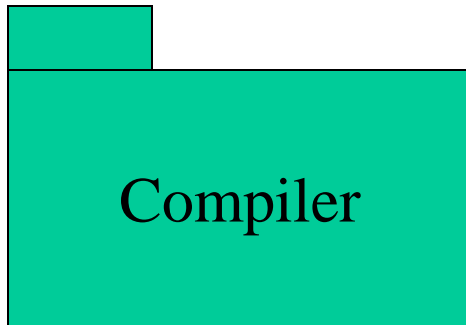
A *package* is a container-like element for organizing other elements into groups.

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.

Compiler
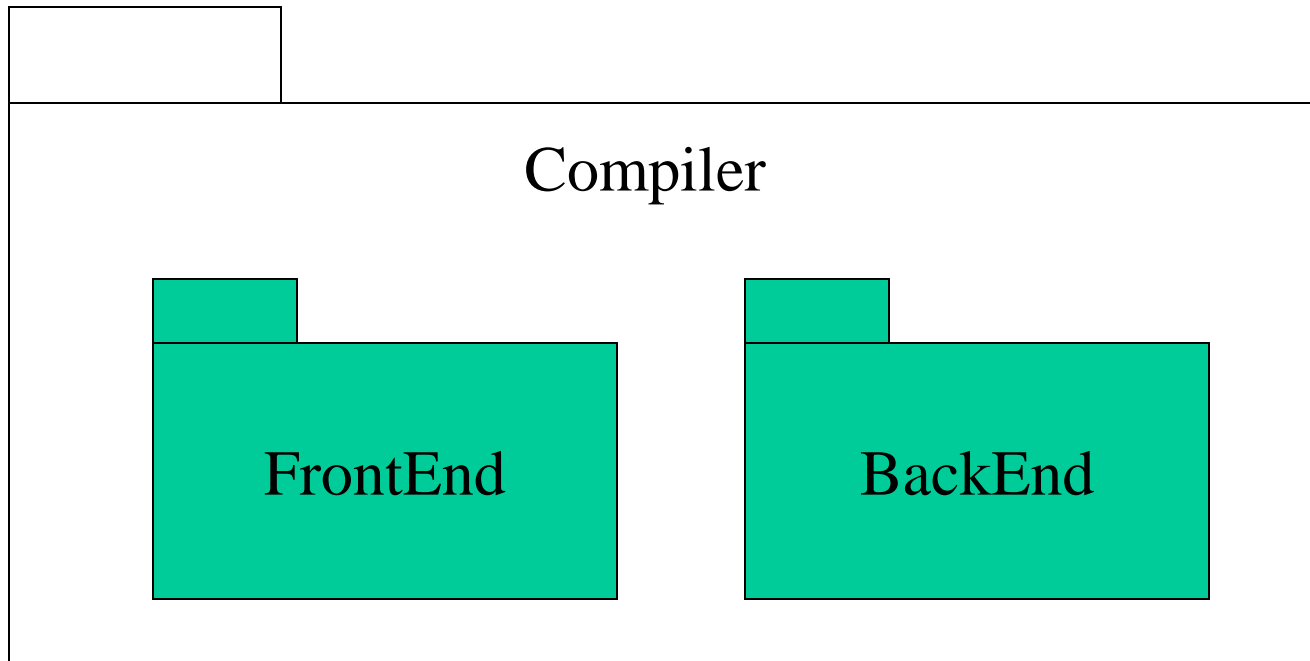
FrontEnd

BackEnd

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.

Compiler

JavaCompiler - - - - - - - → Java

Class diagram example 1

TheVotingProgram

VoterAuthentication
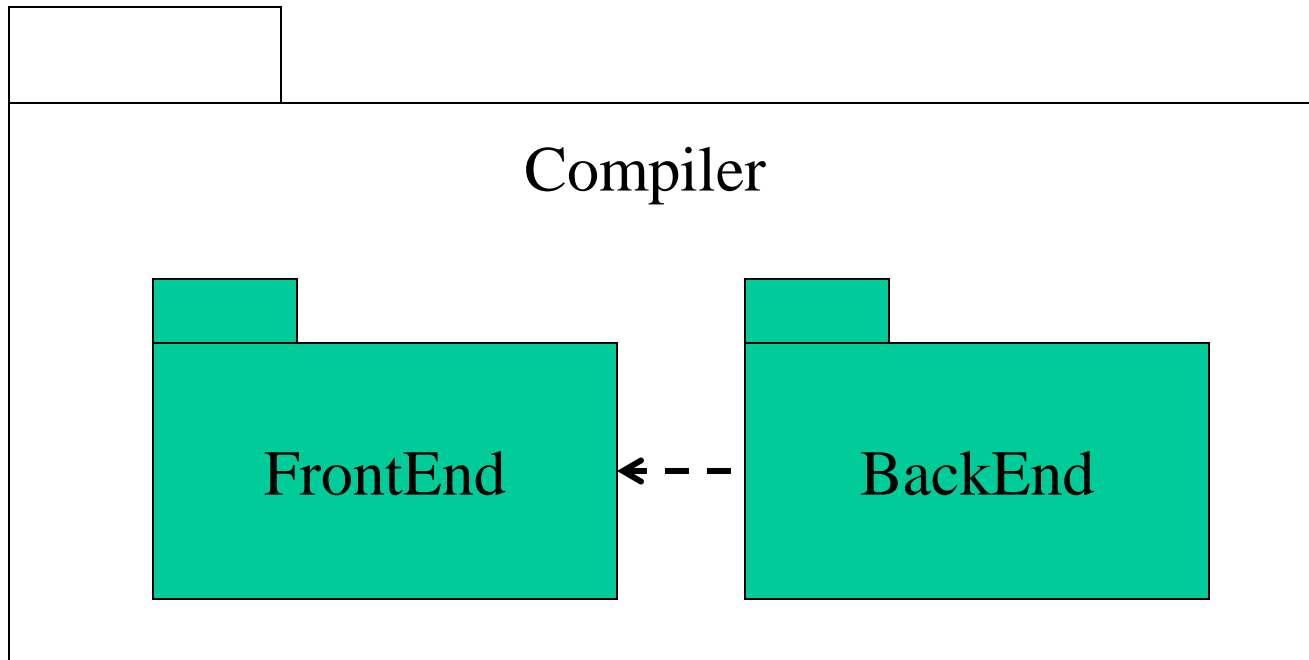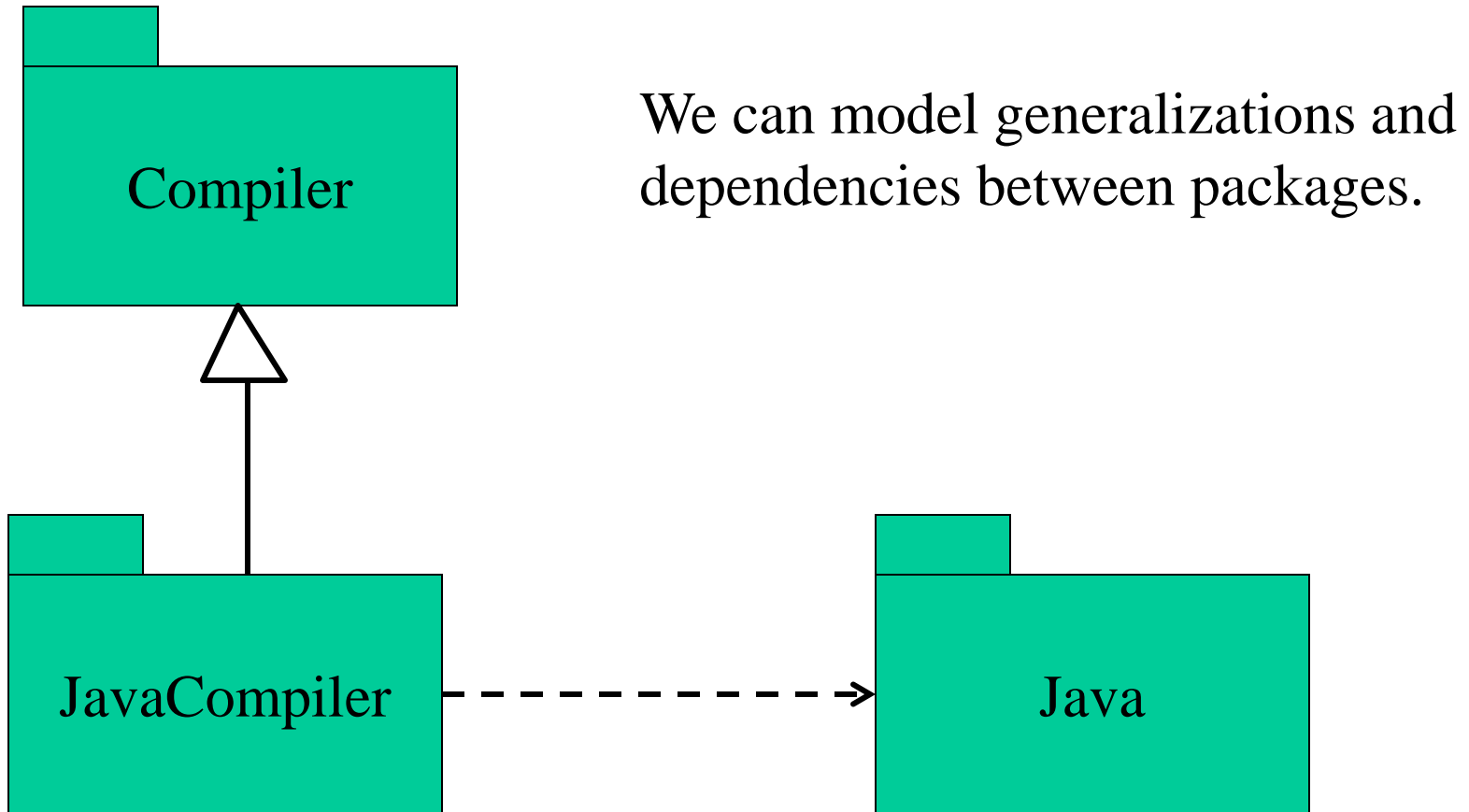- voterPersonalInfo: VoterPersonalInformation
- voterID: String
- voterPassword: securePW

BallotCreation
- ballotName: String
- candidates: String [];
- displayBallot():void
- createBallot():void

this is only a small
subset of the actual
package ...

VoterPersonalIdentification
- voterLastName: String
- voterFirstName: String
- voterMiddleName: String
- voterSSN: String
- voterAddress1: String
- voterAddress2: String
- voterCity: String
- voterState: String
- voterZIP: String
- +validateZipCode(voterZIP:String):String
- +validateState(parameter0VoterState:String):St...

securePW
- PWEntered: JPasswordField
- securePW(PW:securePW):securePW

Class diagram example 2

Multiplicity

Customer

1

Class

Simple
Aggregation

Abstract
Class

Rental Invoice

Rental Item

1..*

1

0..1

Generalization

Composition

Simple
Association

DVD Movie

VHS Movie

Video Game

Checkout Screen

Software Design (UML)

Class diagram example 3

```
┌─────────────────────────────┐             ┌──────────────────────────────────┐
│        StudentBody          │             │            Student               │
├─────────────────────────────┤   1   100   ├──────────────────────────────────┤
│                             │─────────────│ - firstName : String             │
├─────────────────────────────┤             │ - lastName : String              │
│ + main (args : String[])    │             │ - homeAddress : Address          │
└─────────────────────────────┘             │ - schoolAddress : Address        │
                                            ├──────────────────────────────────┤
                                            │ + toString() : String            │
     ┌─────────────────────────┐           └──────────────────────────────────┘
     │        Address          │◇─────────
     ├─────────────────────────┤
     │ - streetAddress : String│
     │ - city : String         │
     │ - state : String        │
     │ - zipCode : long        │
     ├─────────────────────────┤
     │ + toString() : String   │
     └─────────────────────────┘
```

Software Design (UML)

Other Diagrams

- ➢ Use Case Diagrams

- ➢ Sequence diagrams

- ➢ Component Diagram

- ➢ Collaboration diagrams

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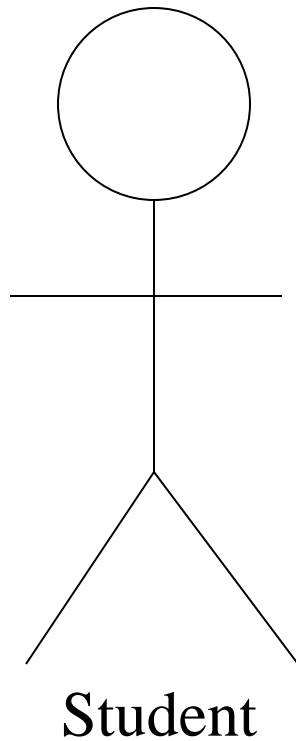 *- The UML Reference Manual, [Rumbaugh,99]*

Software Design (UML)

Use Case (Cont'd)

Register for Courses
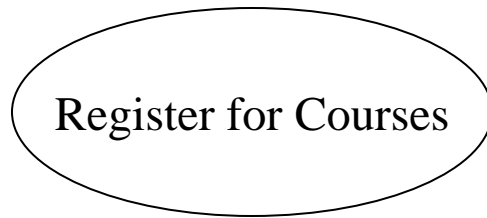
A use case is rendered as an ellipse in a use case diagram. A use case is always labeled with its name.
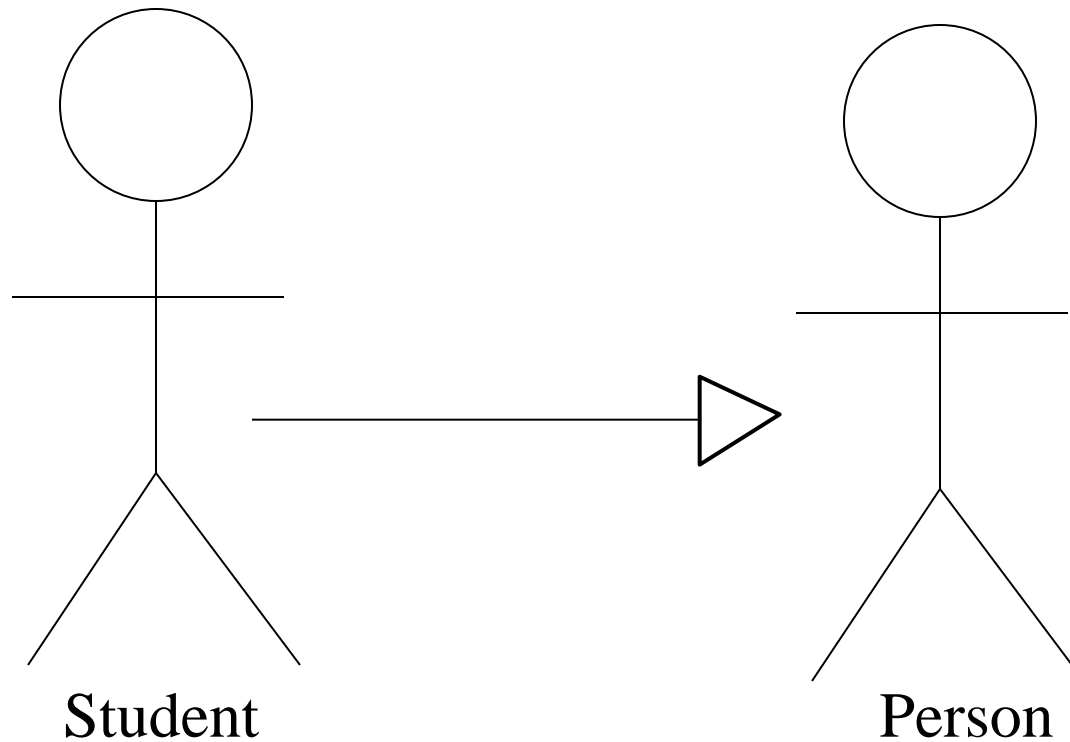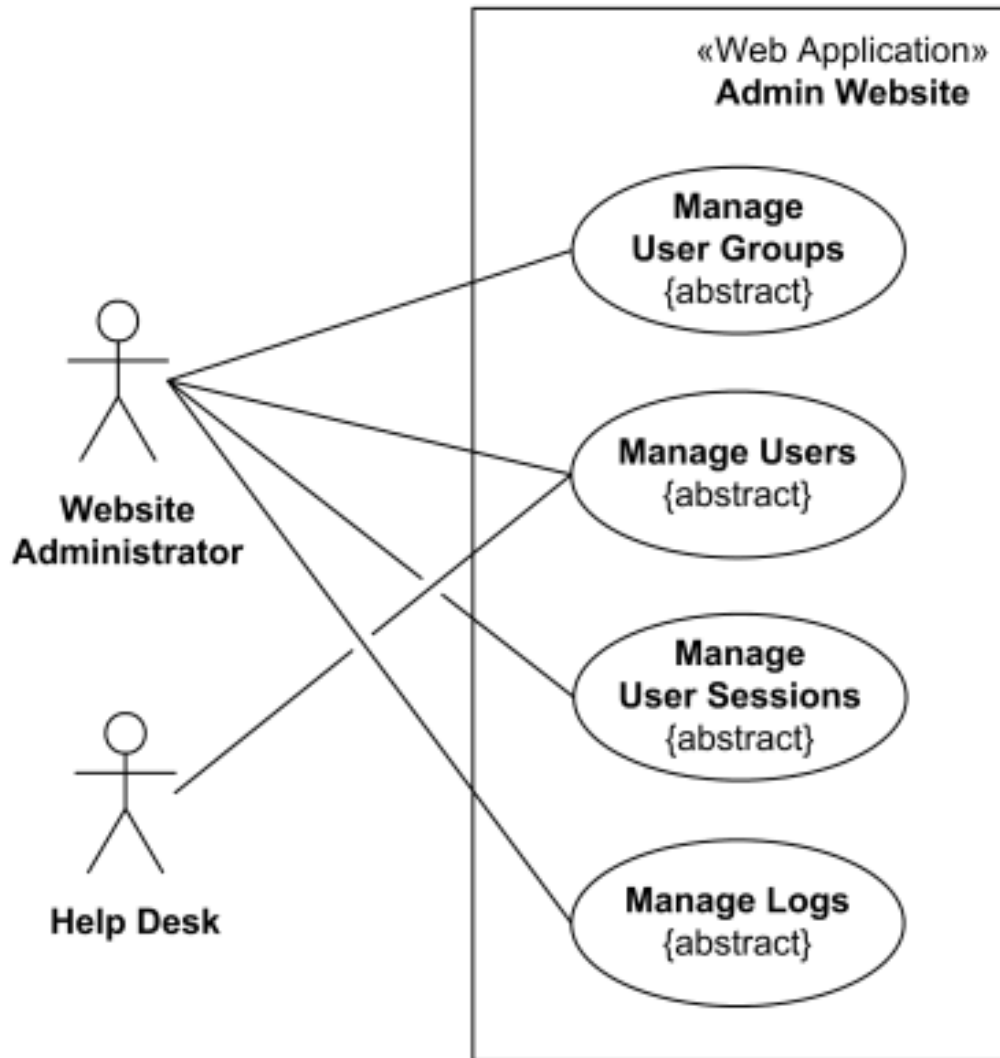
Use Case (Cont'd)

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)

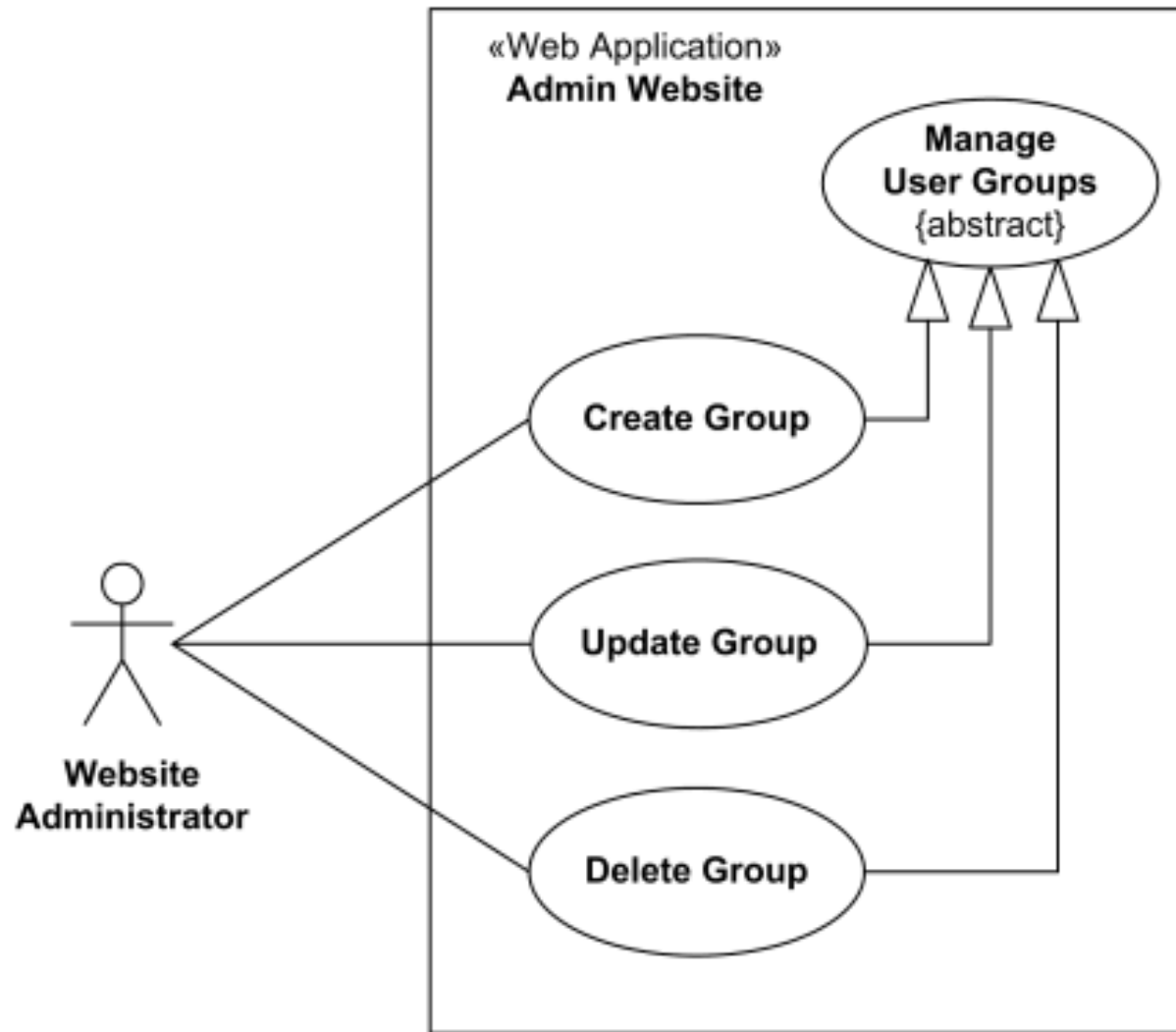Actors can participate in a generalization relation with other actors.

Student   Person

Use Case (Cont'd)

«Web Application»
Admin Website

Website administration

- Manage User Groups {abstract}
- Manage Users {abstract}
- Manage User Sessions {abstract}
- Manage Logs {abstract}

Website Administrator

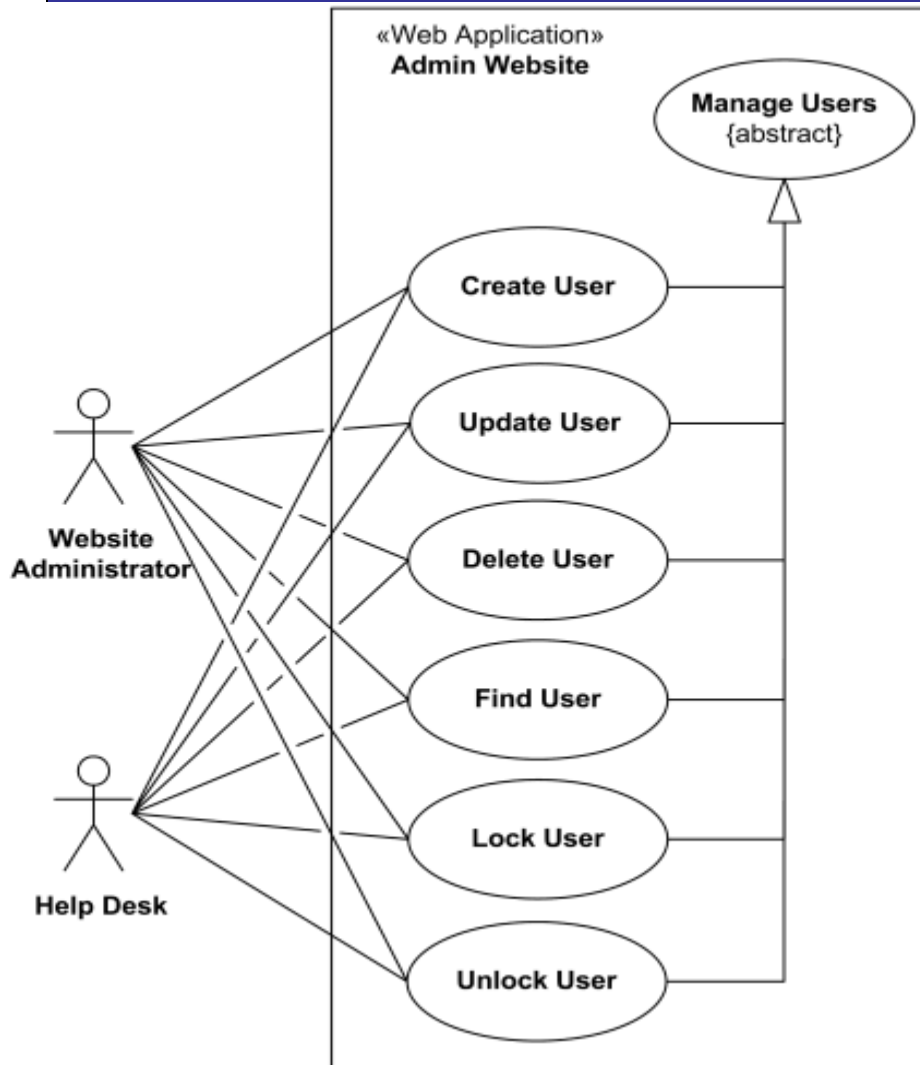Help Desk

Software Design (UML)

Use Case (Cont'd)

Website administration

Software Design (UML)

Use Case (Cont'd)

Website administration

Software Design (UML)

Use Case (Cont'd)

«module»
Hospital Reception

Hospital Management

- Schedule Patient Appointment
- Schedule Patient Hospital Admission
- «extend»
- «extend»
- Patient Registration
- «include»
- Patient Hospital Admission
- Outpatient Hospital Admission
- Inpatient Hospital Admission
- «include»
- Bed Allotment
- File Insurance Forms / Claims
- File Medical Reports

Receptionist

uml-diagrams.org

Software Design (UML)
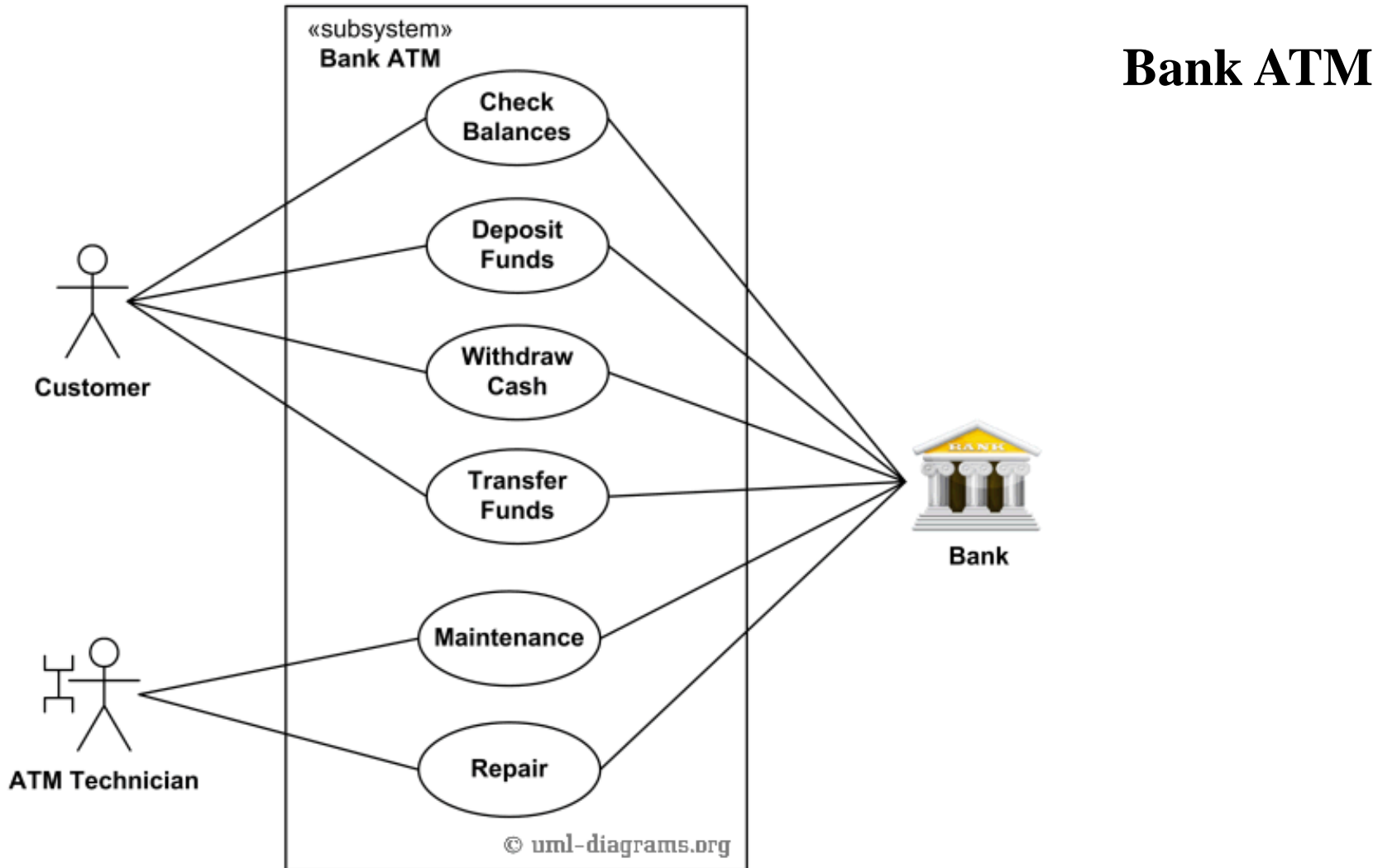
Use Case (Cont'd)

Software Design (UML)

Use Case (Cont'd)

Bank ATM

Software Design (UML)
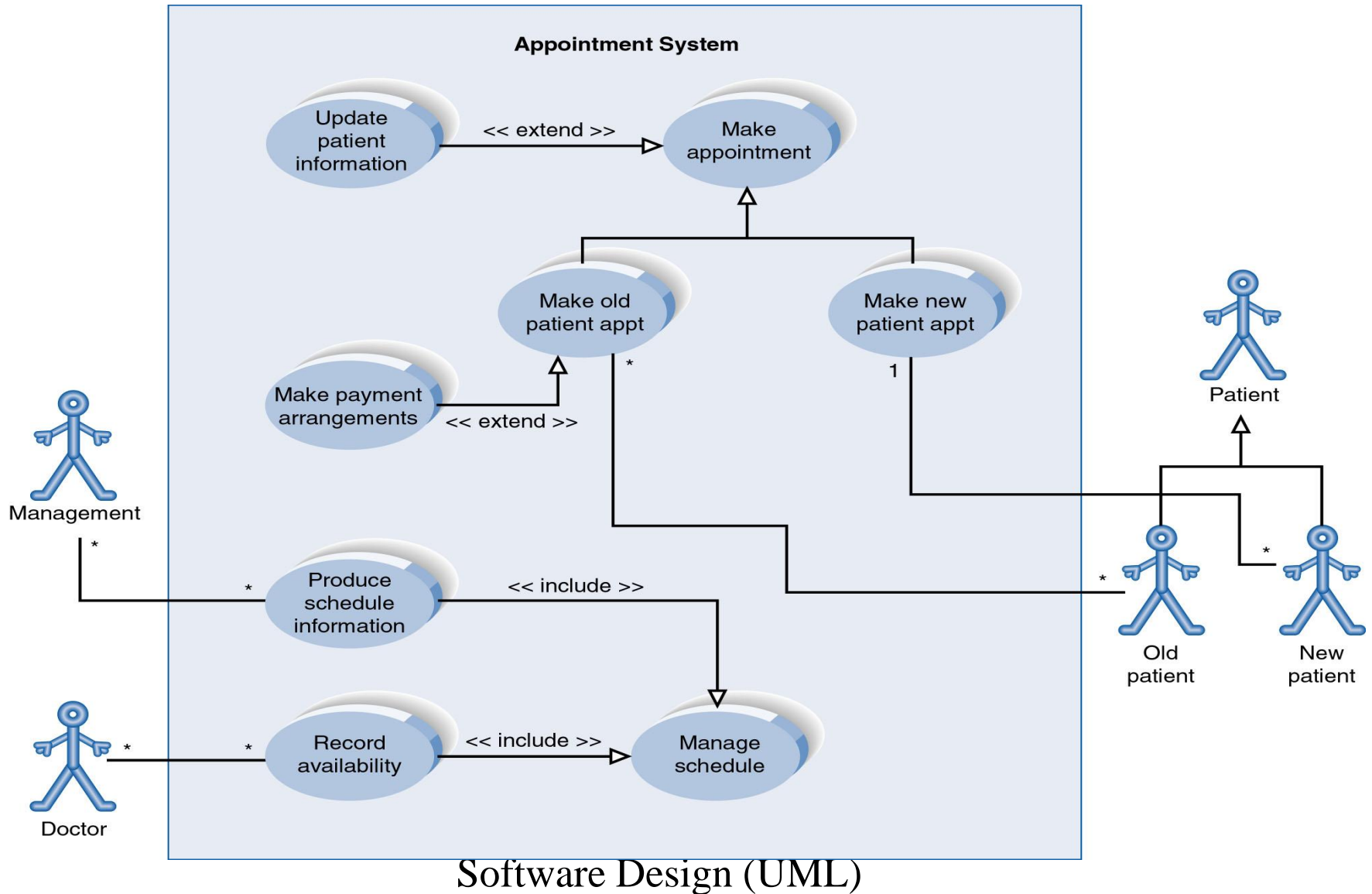
State Machine

"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]

Software Design (UML)

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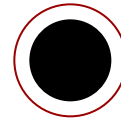
There can be only one start state in a state diagram, but there may be many intermediate and final states.
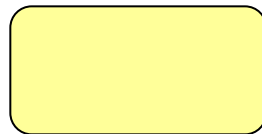
State Machine

➢ Protocol State Machines

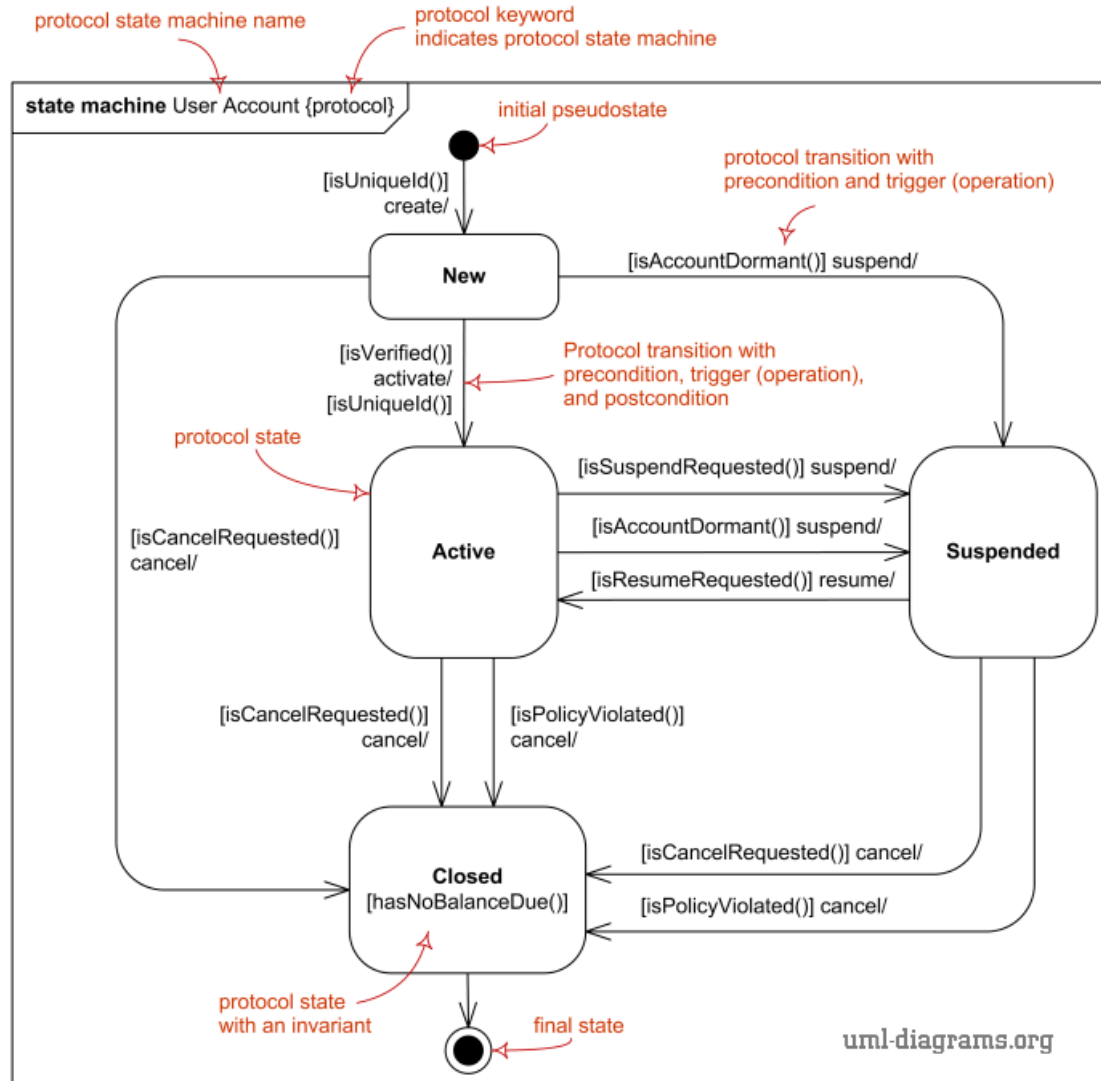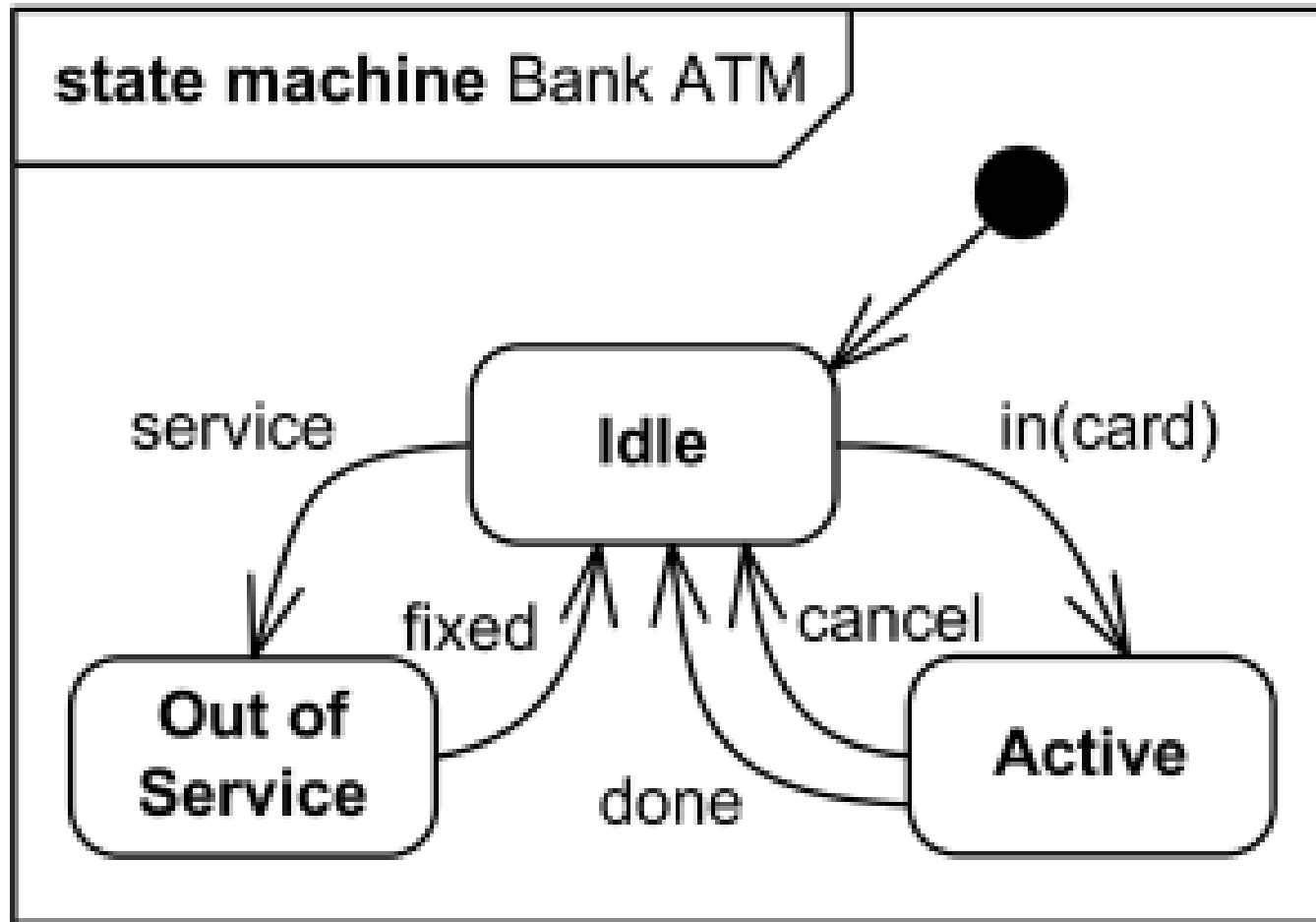➢ Behavioural State machines

start state

final state

simple state

Protocol State Machine

protocol state machine name

protocol keyword
indicates protocol state machine

state machine User Account {protocol}

initial pseudostate

[isUniqueId()]
create/

protocol transition with
precondition and trigger (operation)

New

[isAccountDormant()] suspend/

[isVerified()]
activate/
[isUniqueId()]

Protocol transition with
precondition, trigger (operation),
and postcondition

protocol state

[isSuspendRequested()] suspend/

Active

[isAccountDormant()] suspend/

Suspended

[isCancelRequested()]
cancel/

[isResumeRequested()] resume/

[isCancelRequested()]
cancel/

[isPolicyViolated()]
cancel/

Closed
[hasNoBalanceDue()]

[isCancelRequested()] cancel/

[isPolicyViolated()] cancel/

protocol state
with an invariant

final state
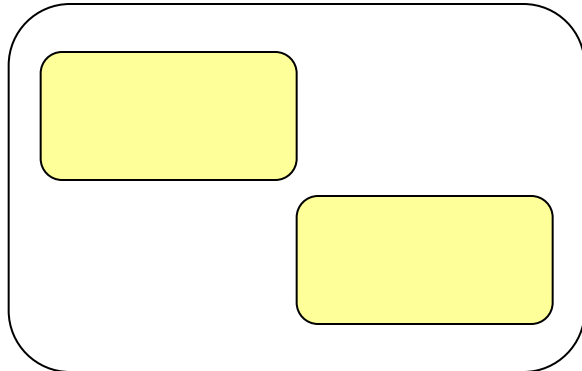
uml-diagrams.org

Software Design (UML)

Behavioral State Machine

Software Design (UML)
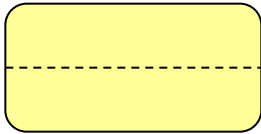
Composite States
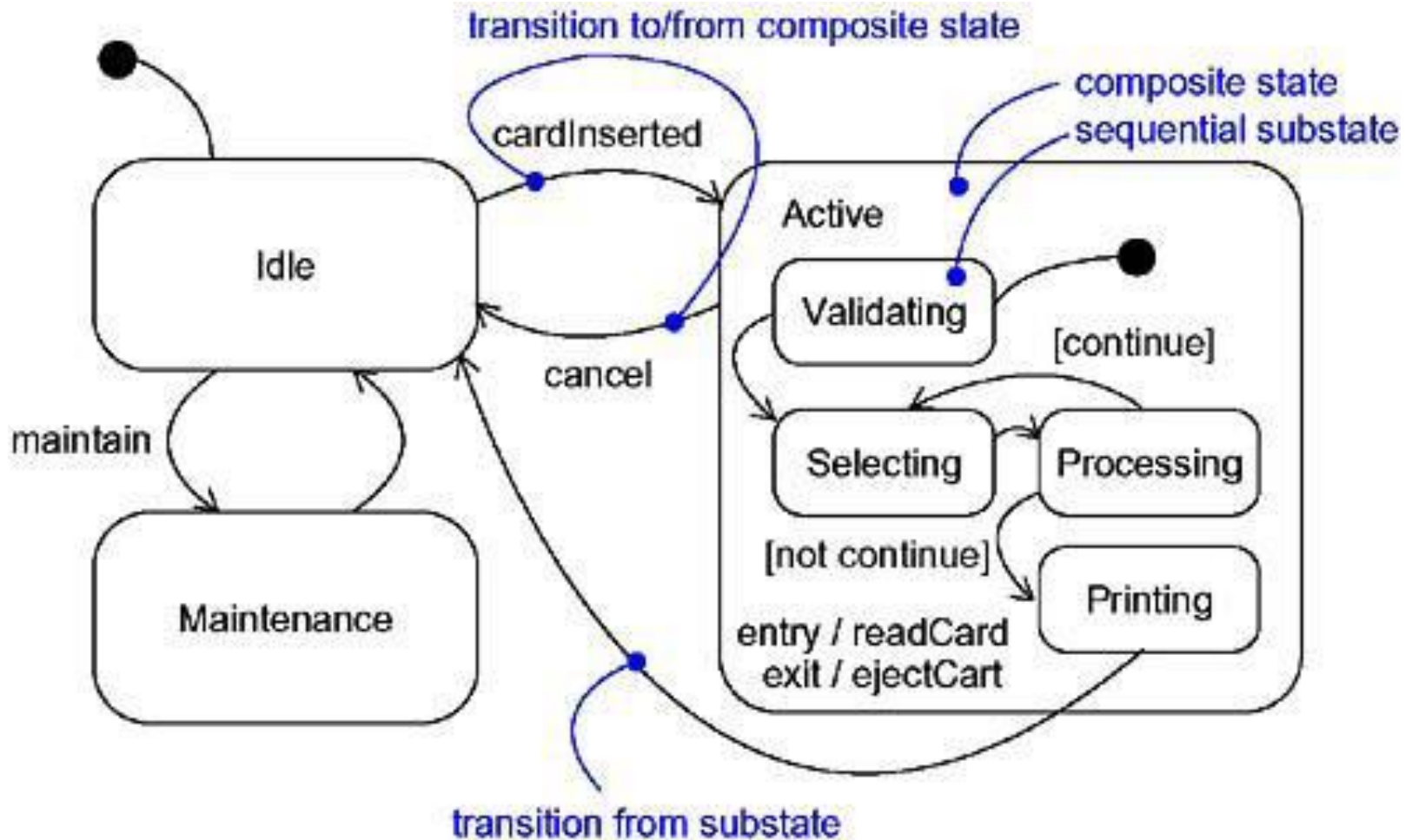
concurrent composite state

sequential composite state
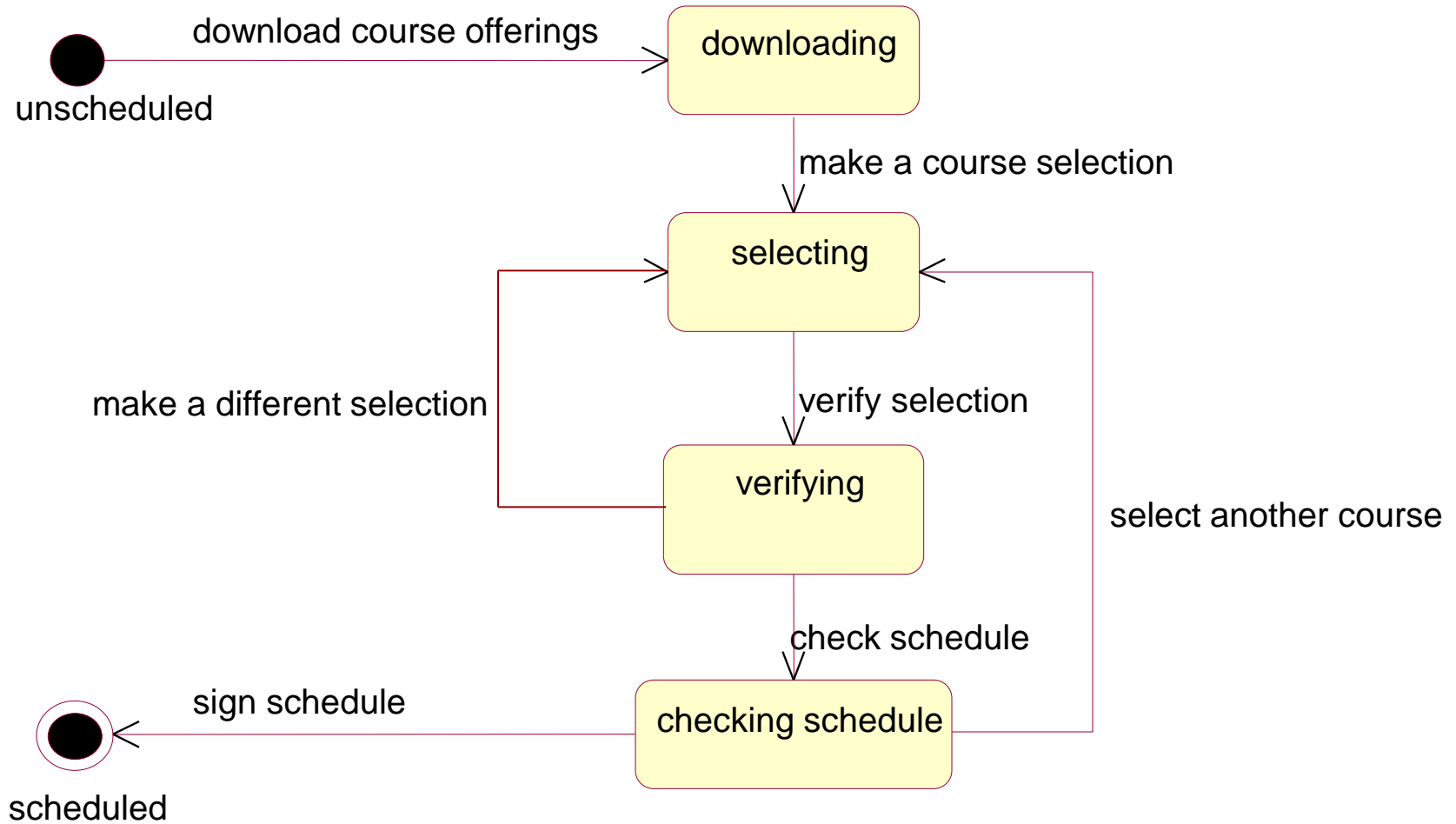
Sequential Composite State

Concurrent Composite State
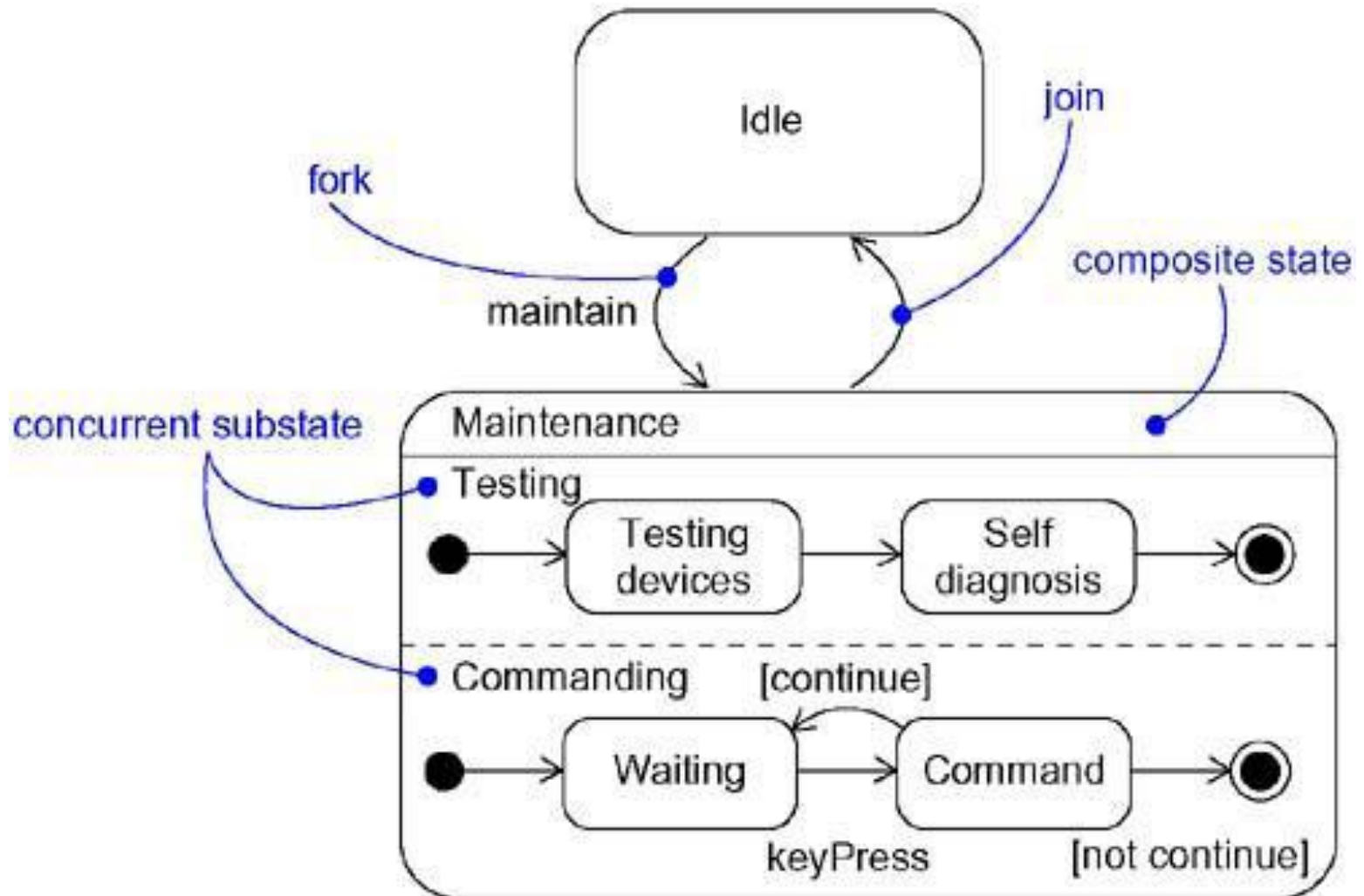
Advanced States and Transitions

Software Design (UML)
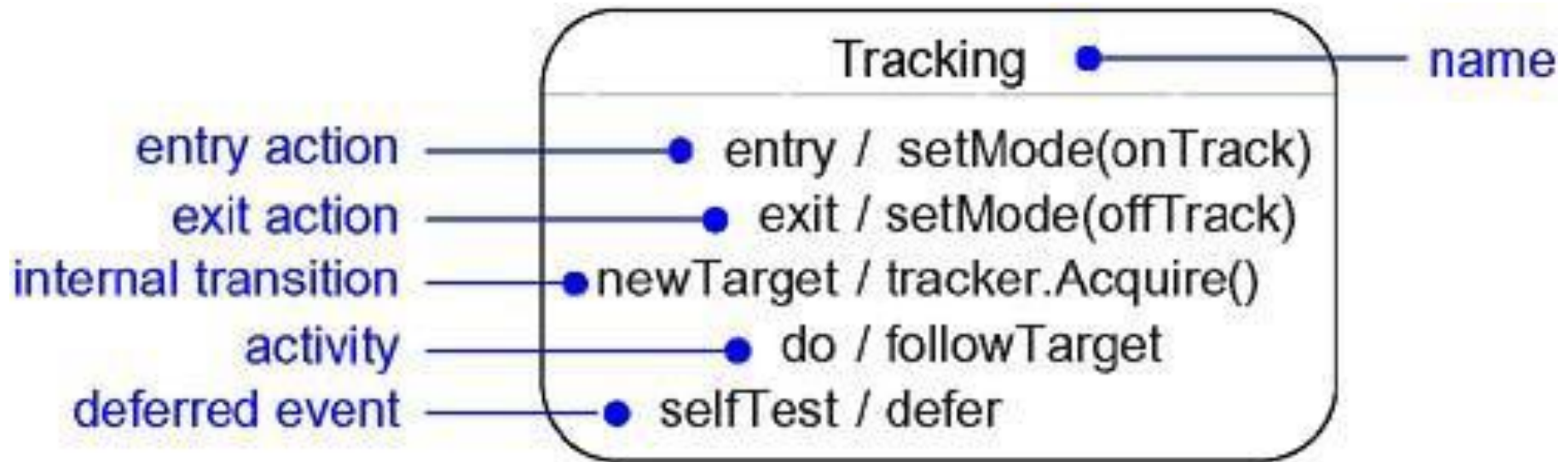
State Machine

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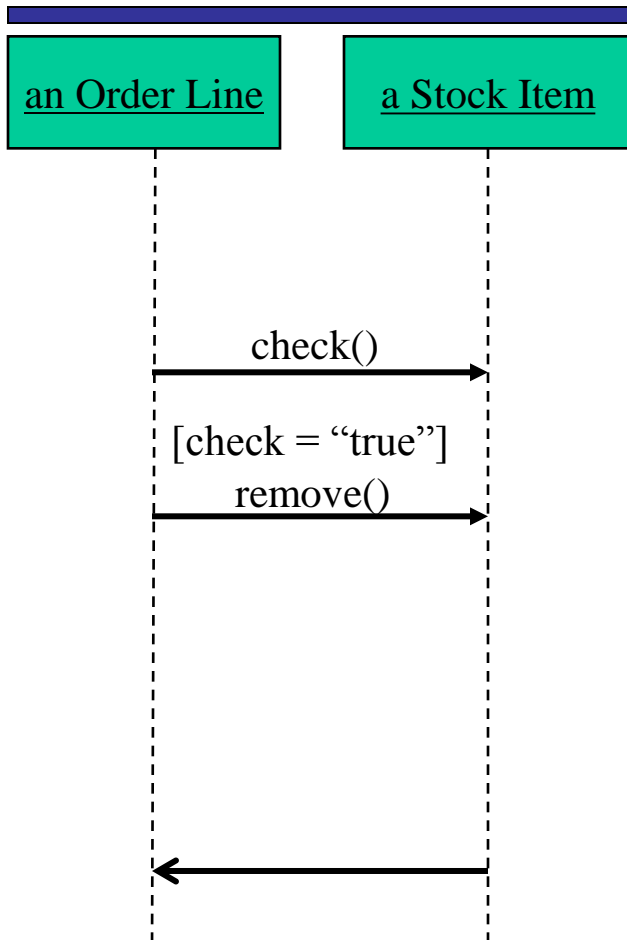
- *The UML User Guide, [Booch,99]*

Sequence Diagram

an Order Line

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

an Order Line   **a Stock Item**

check()

[check = "true"]
remove()

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

an Order Line	a Stock Item

check()

[check = "true"]
remove()
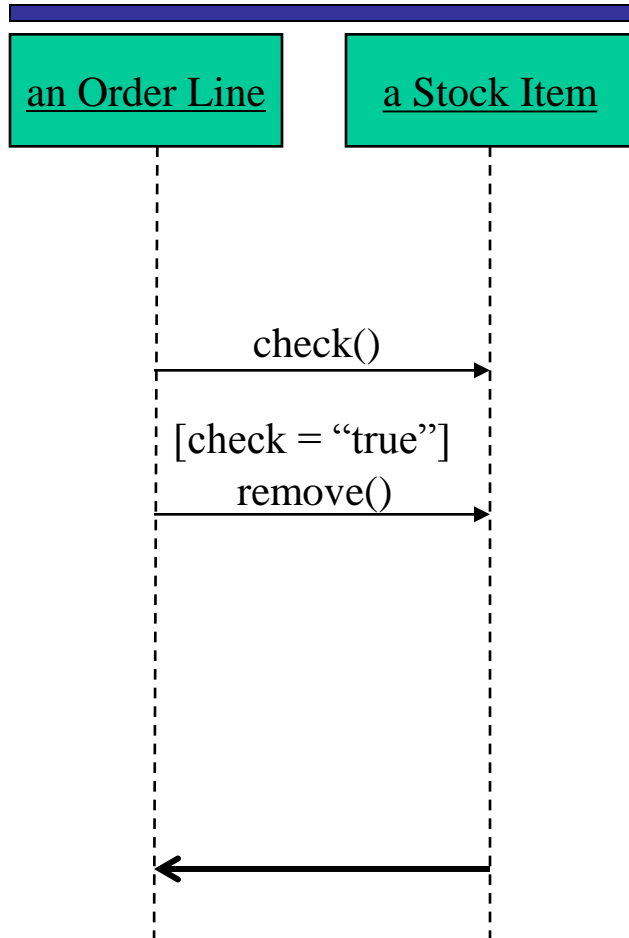
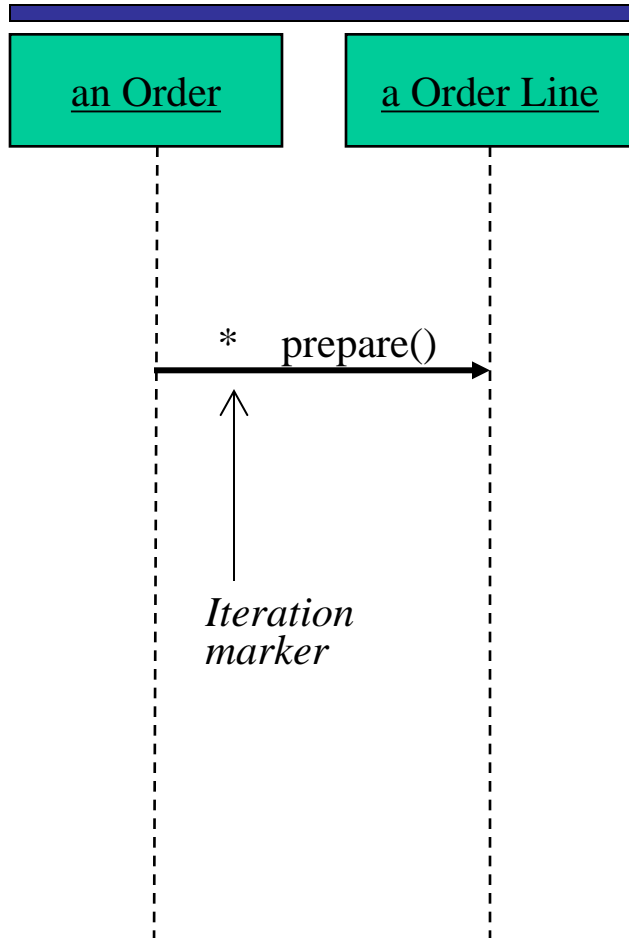Notice that the bottom arrow is different. The arrow head is not solid, and there is no accompanying message.

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

Sequence Diagram

an Order | a Order Line

* prepare()

Iteration marker

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

Object

Message

prepare()

* prepare()

Iteration

check()

Condition

[check = "true"]
remove()

needsToReorder()

Self-Delegation

[needsToReorder = "true"]

Return

new

A Reorder Item

[check = "true"]
new

A Delivery Item

Creation

Software Design (UML)

Example

Software Design (UML)
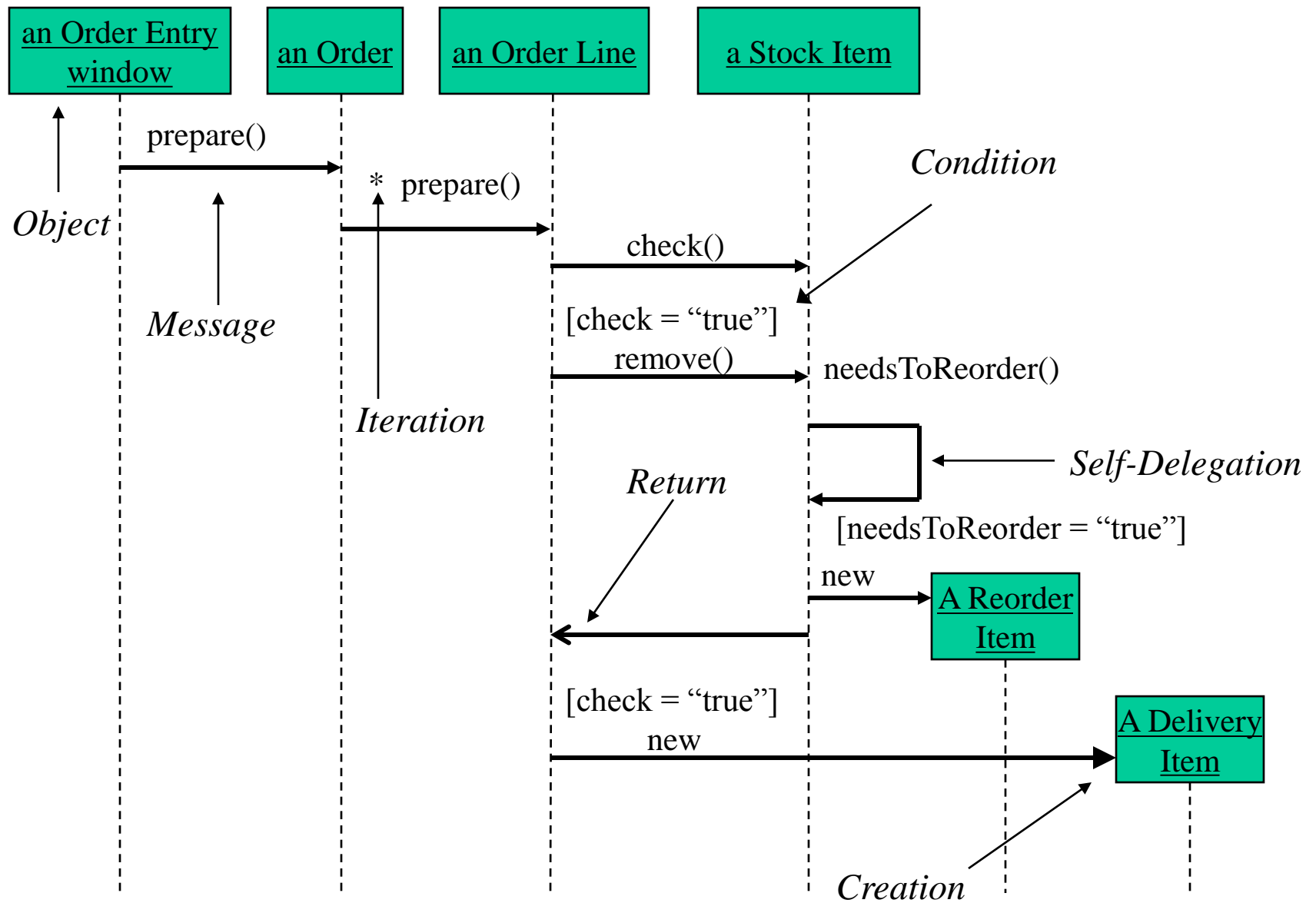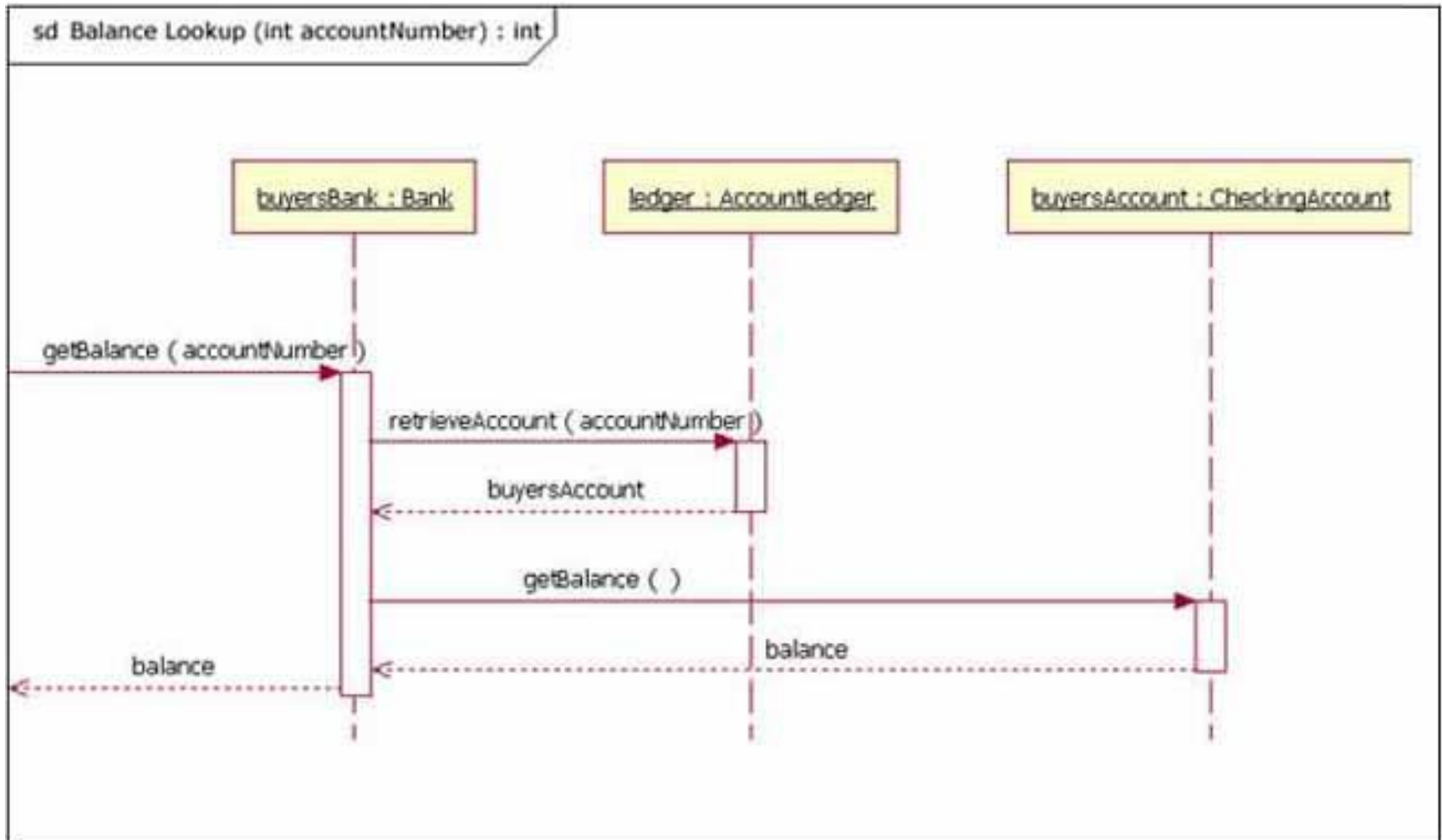
Collaboration Diagram

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

: Order Entry Window ← *Class*

Sequence Number

1: prepare() ← *Message*

: Order

Self-Delegation

5: needToReorder()

2*: prepare()   **3: check()**
4: [check == true] remove()

: Order Line   : Stock Item

7: [check == true] new   **6: new**

:Delivery Item   :Reorder Item

Software Design (UML)
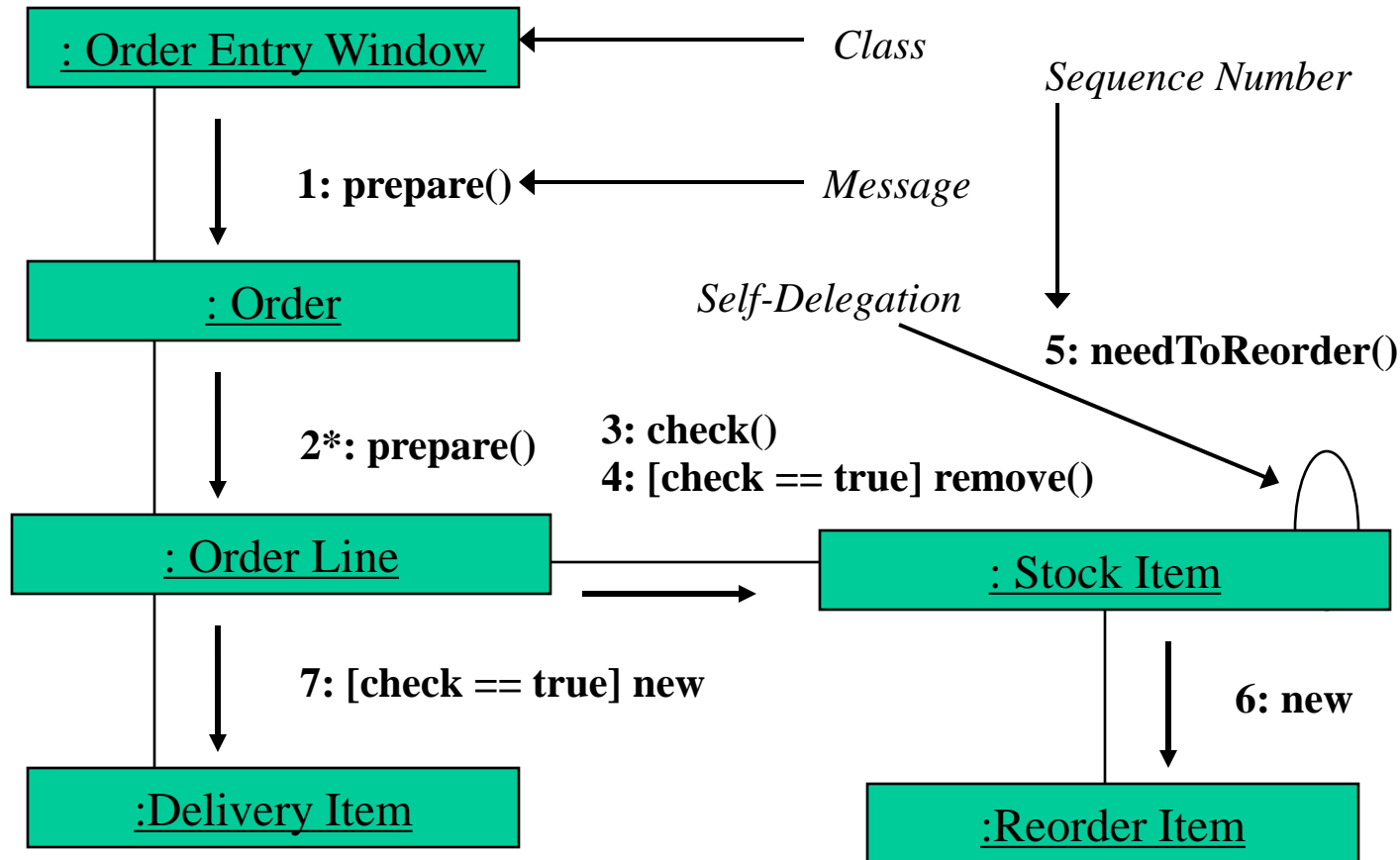
Collaboration Diagram
Sequence Diagram

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.
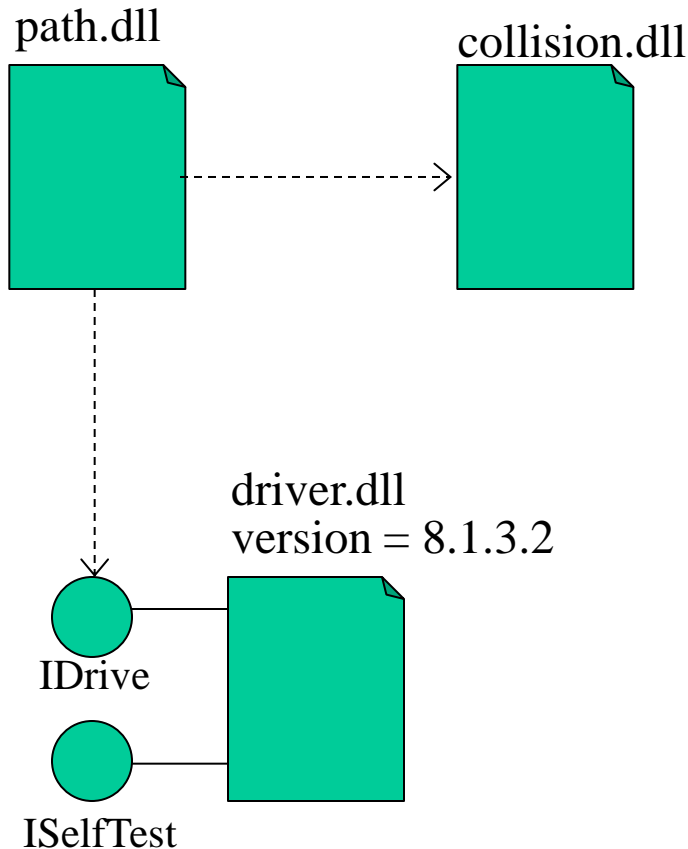
Component Diagram

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

path.dll

collision.dll

driver.dll
version = 8.1.3.2

IDrive

ISelfTest

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

[Booch,99]

Software Design (UML)

Component Diagram

signal.h
version = 3.5

signal.h
version = 4.0

signal.h
version = 4.1

"parent"

"parent"
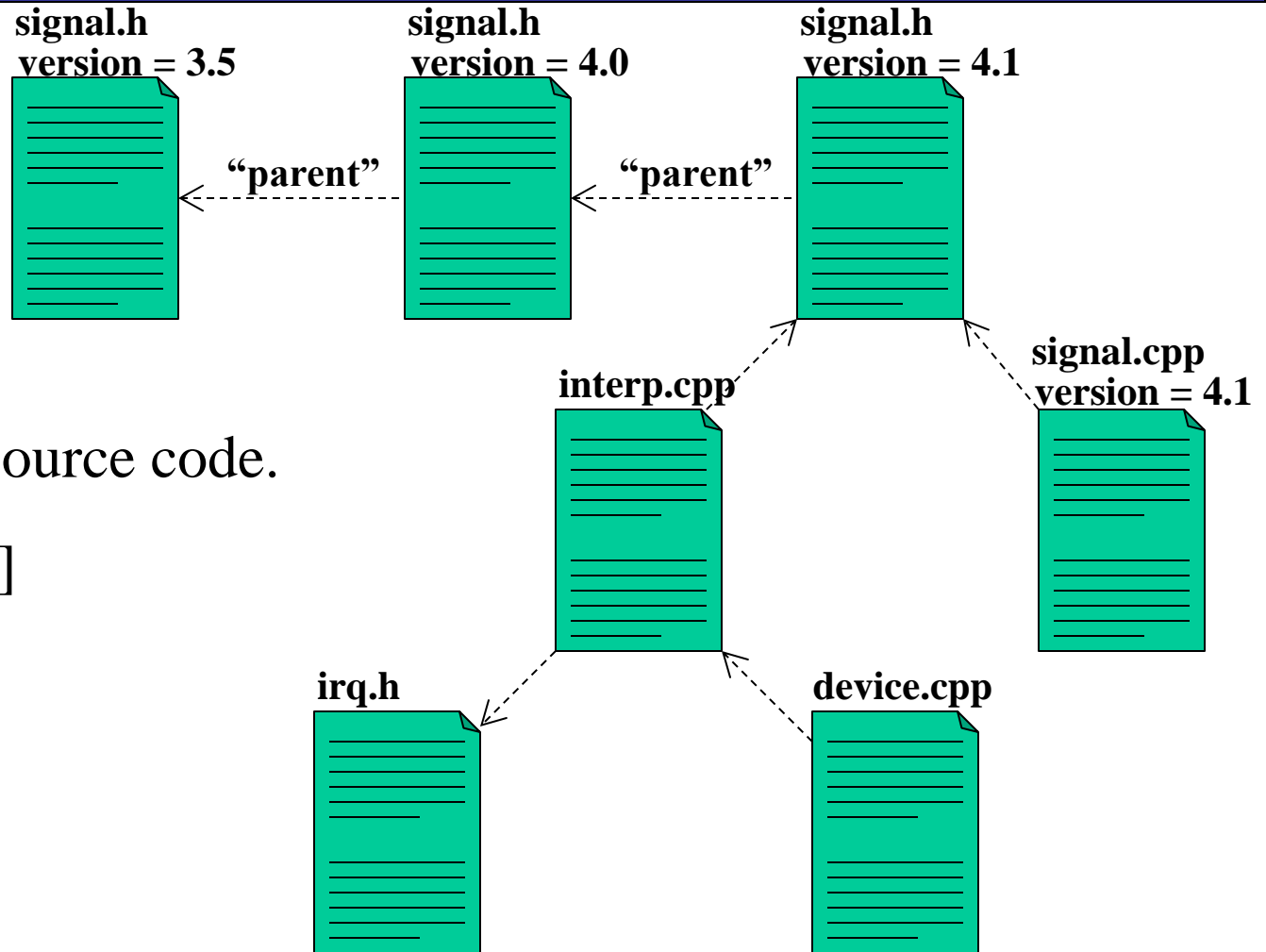
Modeling source code.

[Booch, 99]

interp.cpp

signal.cpp
version = 4.1

irq.h

device.cpp

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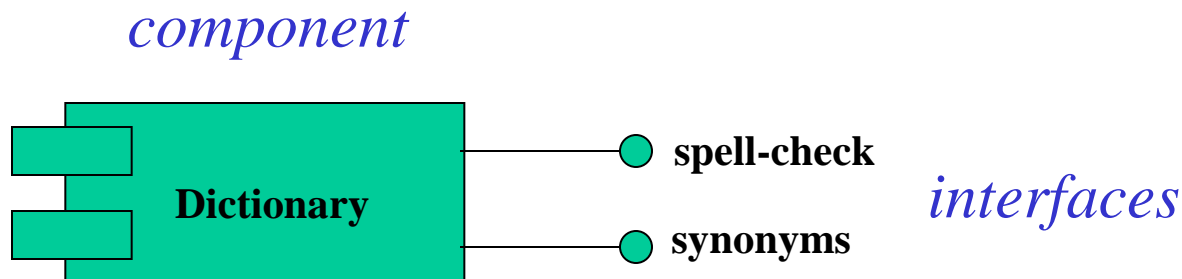
Use deployment diagrams to model the ***static deployment view*** of a system. This involves modeling the topology of the hardware on which the system executes.

- The UML User Guide, [Booch,99]

Deployment Diagram

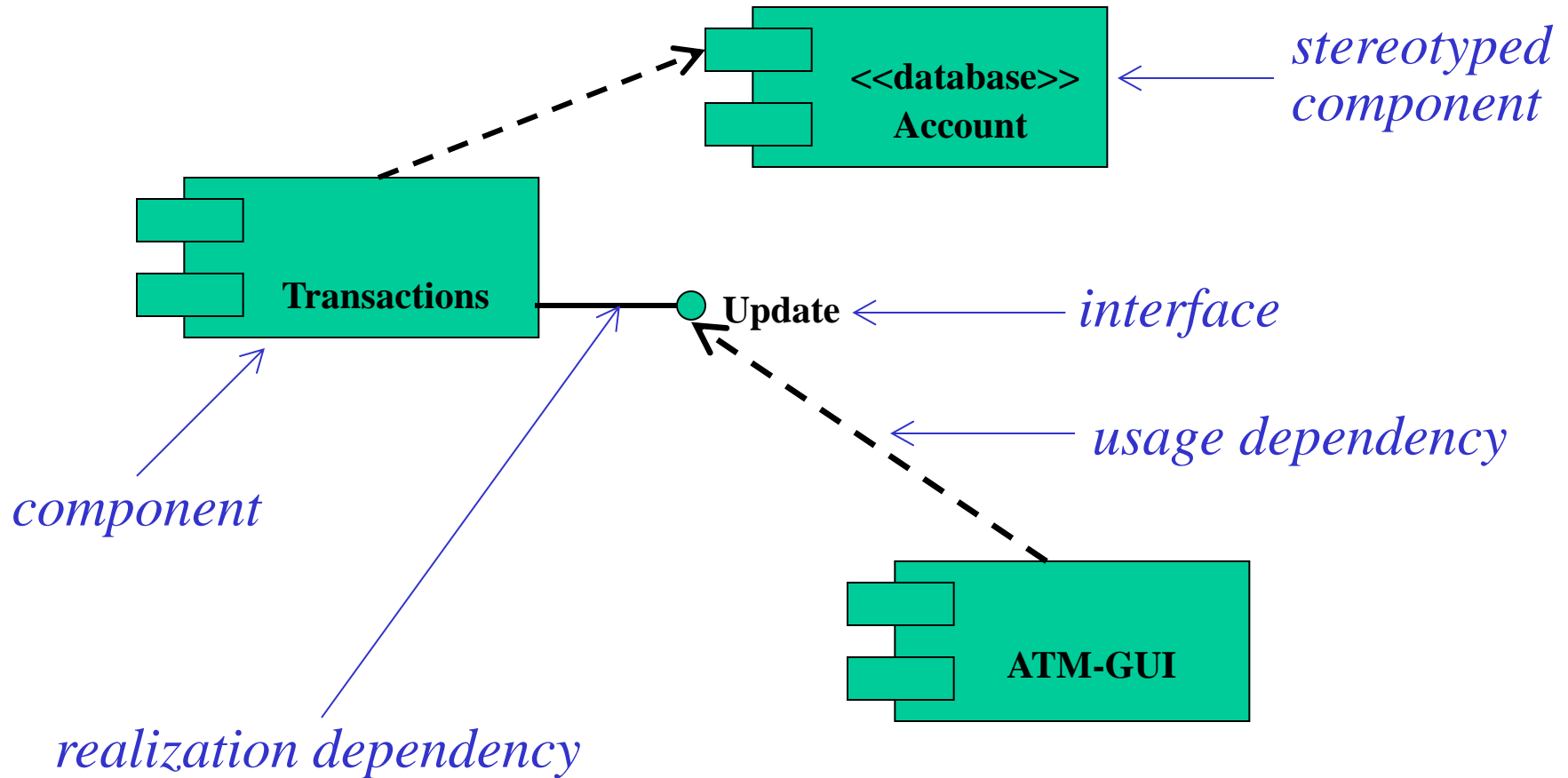A component is a physical unit of implementation with well-defined interfaces that is intended to be used as a replaceable part of a system. Well designed components do not depend directly on other components, but rather on interfaces that components support.

- The UML Reference Manual, [Rumbaugh,99]

component

spell-check

Dictionary

interfaces

synonyms

Software Design (UML)

Deployment Diagram

<<database>>
Account

stereotyped component

Transactions

Update

interface

component

usage dependency

ATM-GUI

realization dependency

[Rumbaugh,99]

Software Design (UML)

Deployment Diagram

server:HostMachine

<<database>>

meetingsDB

:Scheduler

reservations

<<direct channel>>

clientMachine:PC

:Planner
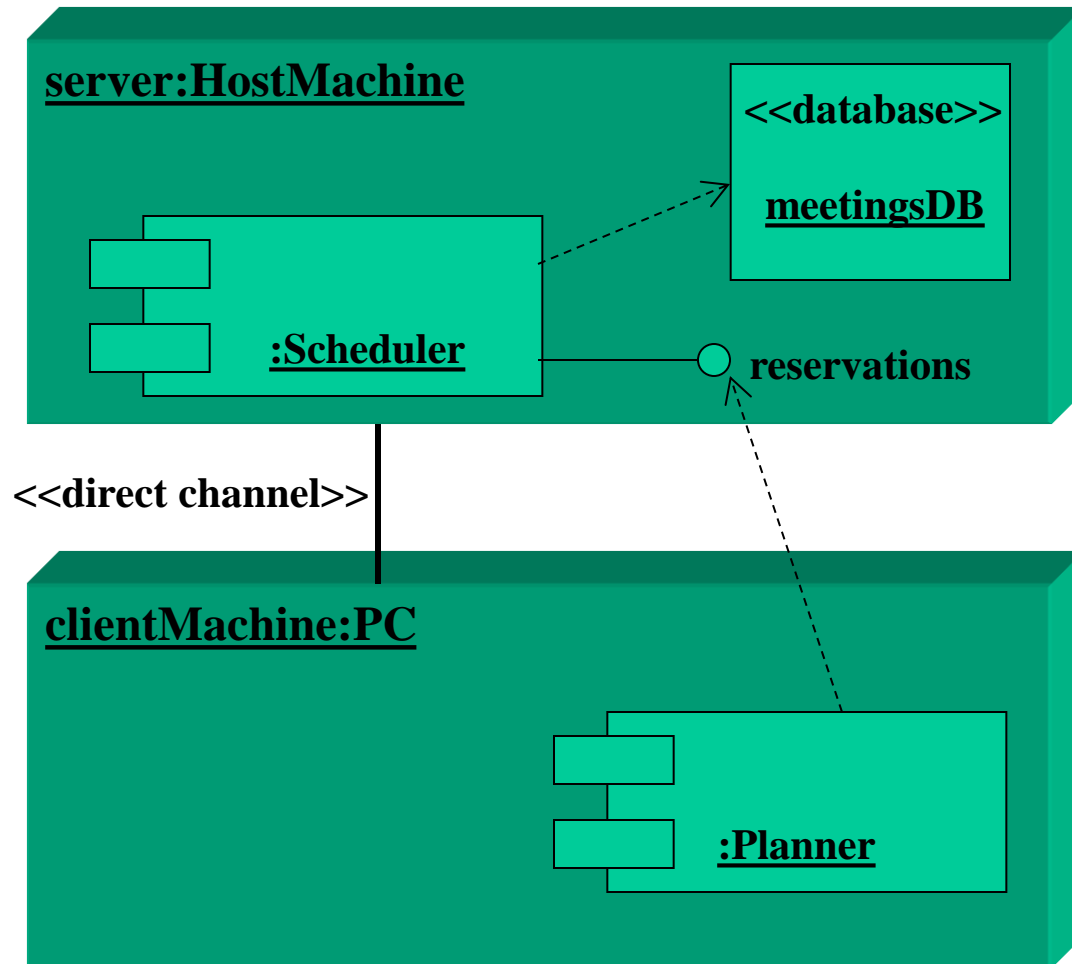
Deployment diagram of a client-server system.

[Rumbaugh,99]

Software Design (UML)

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]

Multiple Trigger

for each line
item on order

Receive
Order

Cancel
Order

[failed]

Authorize
Payment

[succeeded]

Check
Line
Item

*

[in stock]

Assign to
Order

Synchronization Condition

[stock assigned to
all line items and
payment authorized]

[need to
reorder]

Reorder
Item

Dispatch
Order

Software Design (UML)