OBJECT-ORIENTED AND CLASSICAL SOFTWARE ENGINEERING

EIGHTH EDITION, WCB/MCGRAW-HILL, 2011

STEPHEN R. SCHACH

Edited by Hyungbae Park

University of Central Missouri

CHAPTER 17

MORE ON UML

Chapter Overview

- UML is *not* a methodology
- Use-case diagrams
- Class diagrams
- Notes
- Interaction diagrams
- Statecharts
- Activity diagrams
- Packages
- Component diagrams

Chapter Overview (contd)

- Deployment diagrams
- Review of UML diagrams
- UML and iteration

The Current Version of UML

- Like all modern computer languages, UML is constantly changing
  - When this book was written, the latest version of UML was Version 2.0
  - By now, some aspects of UML may have changed
- UML is now under the control of the Object Management Group (OMG)
  - Check for updates at the OMG Web site, www.omg.org

17.1 UML Is *Not* a Methodology

- UML is an acronym for Unified Modeling Language
  - UML is therefore a *language*
- A language is simply a tool for expressing ideas
  - English - novels, poems, news reports, and even textbooks

UML Is *Not* a Methodology

- UML is a notation, not a methodology
  - It can be used in conjunction with any methodology
- UML is not merely a notation, it is *the* notation
- UML has become a world standard
  - Every information technology professional today needs to know UML

UML Is *Not* a Methodology (contd)

- The title of this chapter is “More on UML”
  - Surely it should be “All of UML”?
- The manual for Version 2.0 of UML is about 1200 pages long
  - Complete coverage is not possible
- But surely every information technology professional must know every aspect of UML?

UML Is *Not* a Methodology (contd)

- UML is a language
- The English language has over 100,000 words
  - We can manage fine with just a subset
- The small subset of UML presented in Chapters 7, 11, 13, and 14 is adequate for the purposes of this book
- The larger subset of UML presented in this chapter is adequate for the development and maintenance of most software products

Types of UML Diagrams

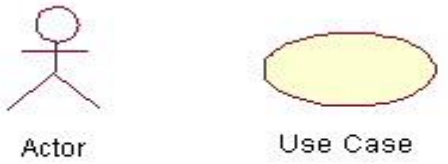
- Use Case Diagram
- Class Diagram
- Interaction Diagram
  - Sequence Diagram
  - Collaboration Diagram
- State Diagram

This is only a subset of diagrams … but are most widely used

17.4 Use-Case Diagrams

- A use case is a model of the interaction between
  - External users of a software product (actors) and
  - The software product itself
    - More precisely, an actor is a user playing a specific role
- A use-case diagram is a set of use cases
- Used for describing a set of user scenarios
- Mainly used for capturing *functional* requirements

Use-Case Diagram (contd)



- Actors:
  - A role that a user plays with respect to the system, including human users and other systems.
- Use case:
  - A set of scenarios that describing an interaction between a user and a system, including alternatives
- System boundary:
  - Rectangle diagram representing the boundary between the actors and the system

Use-Case Diagrams (contd)

- *Use cases* represent specific flows of events in the system

- *Use cases* are initiated by *actors* and describe the flow of events that these actors are involved in
  - Anything that interacts with a use case; it could be a human, external hardware (like a timer) or another system

Use-Case Diagram (contd)

- Which of these requirements should be represented directly in a use case?

1. Promotions may not run longer than 6 months.
2. Customers only become Preferred after 1 year
3. Response time is less than 2 seconds
4. Uptime requirement is 99.8% 7
5. Number of simultaneous users will be 200 max

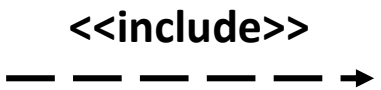
Use-Case Diagram (contd)

- Association:
  - Communication between an actor and a use case; Represented by a solid line

- Generalization: 
  - Relationship between one general use case and a special use case (used for defining special alternatives)
  - Represented by a line with a triangular arrow head toward the parent use case

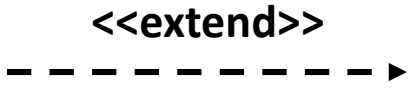
Use-Case Diagram (contd)

- Include:



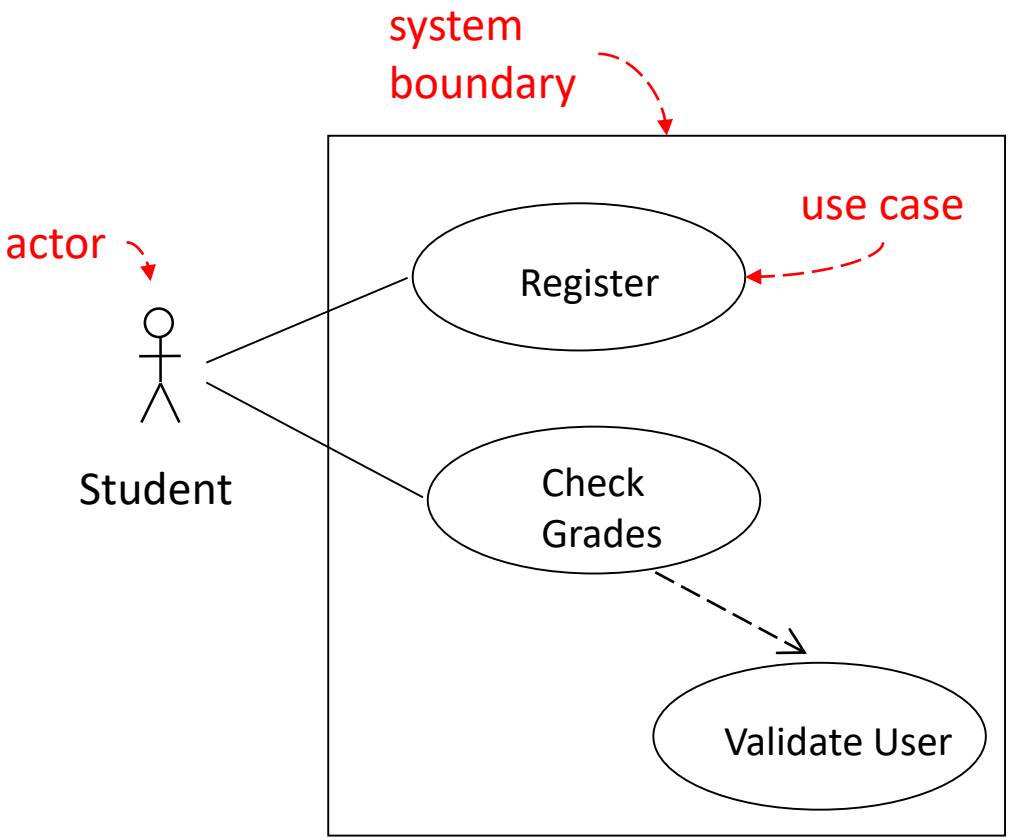
  - A dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case
  - The include relationship occurs when a chunk of behavior is similar across more than one use case

- 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

Use-Case Diagram (contd)



- A generalized description of how a system will be used
- Provides an overview of the intended functionality of the system

Use-Case Diagrams (contd)

- Generalization of actors is supported
  - The open triangle points toward the more general case

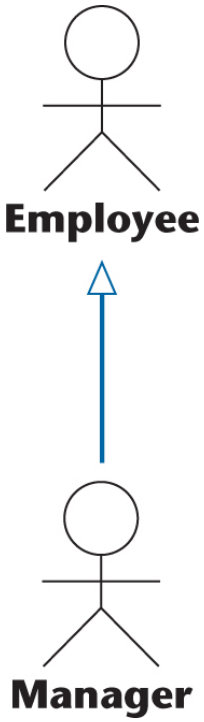
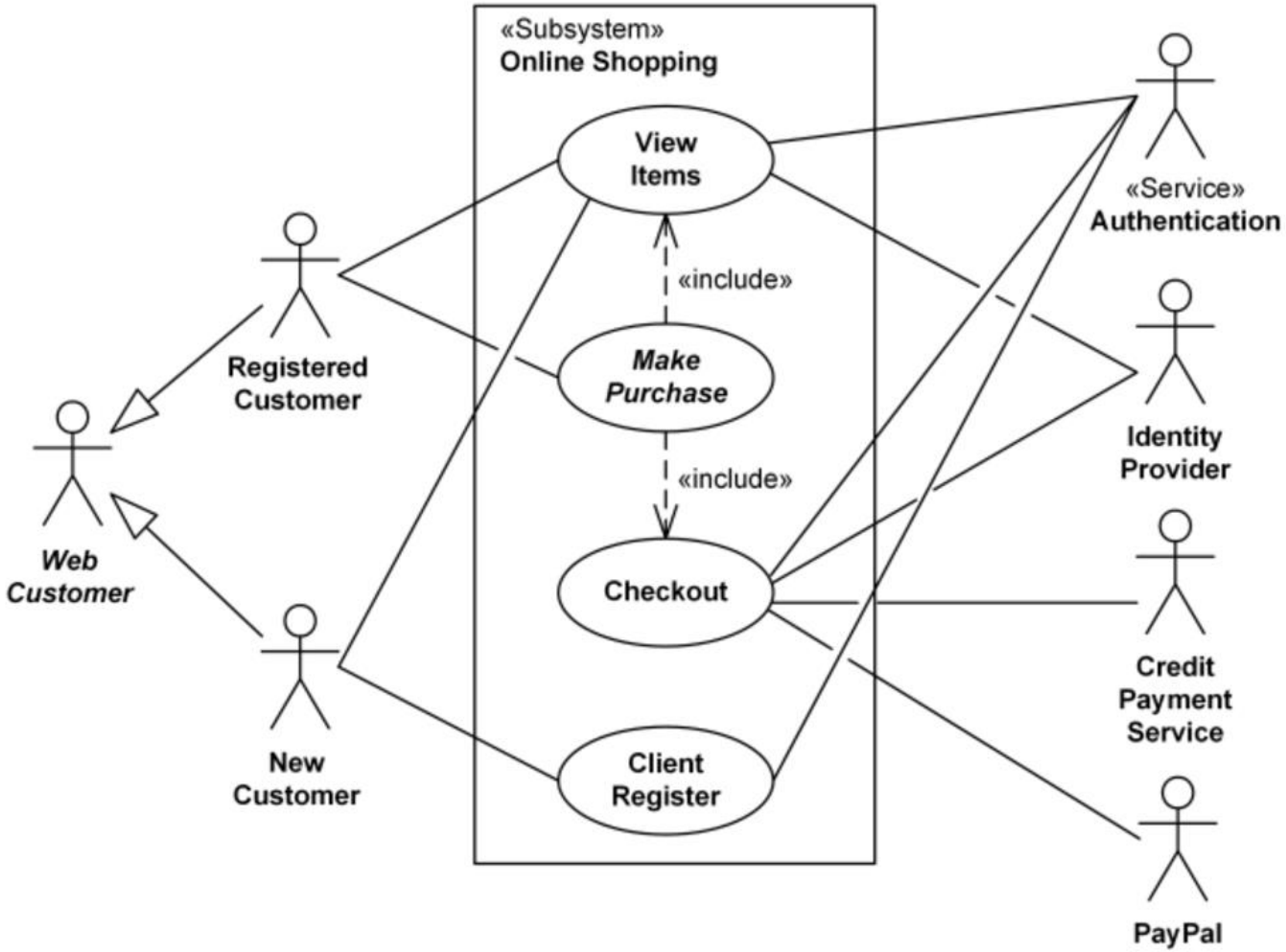


Figure 17.11

Use-Case Diagram (contd)

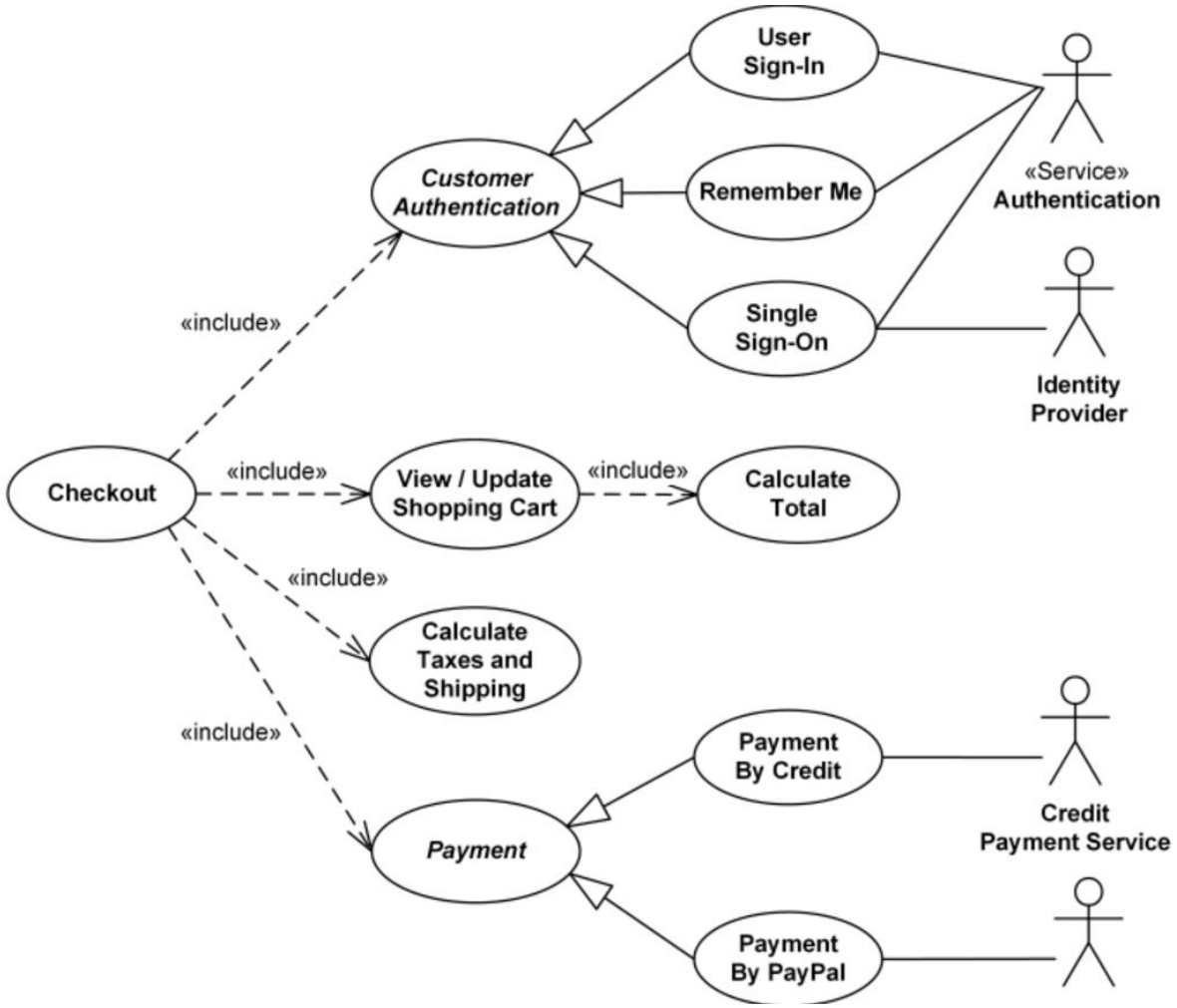


Uml-diagrams.org

Use-Case Diagram (contd)

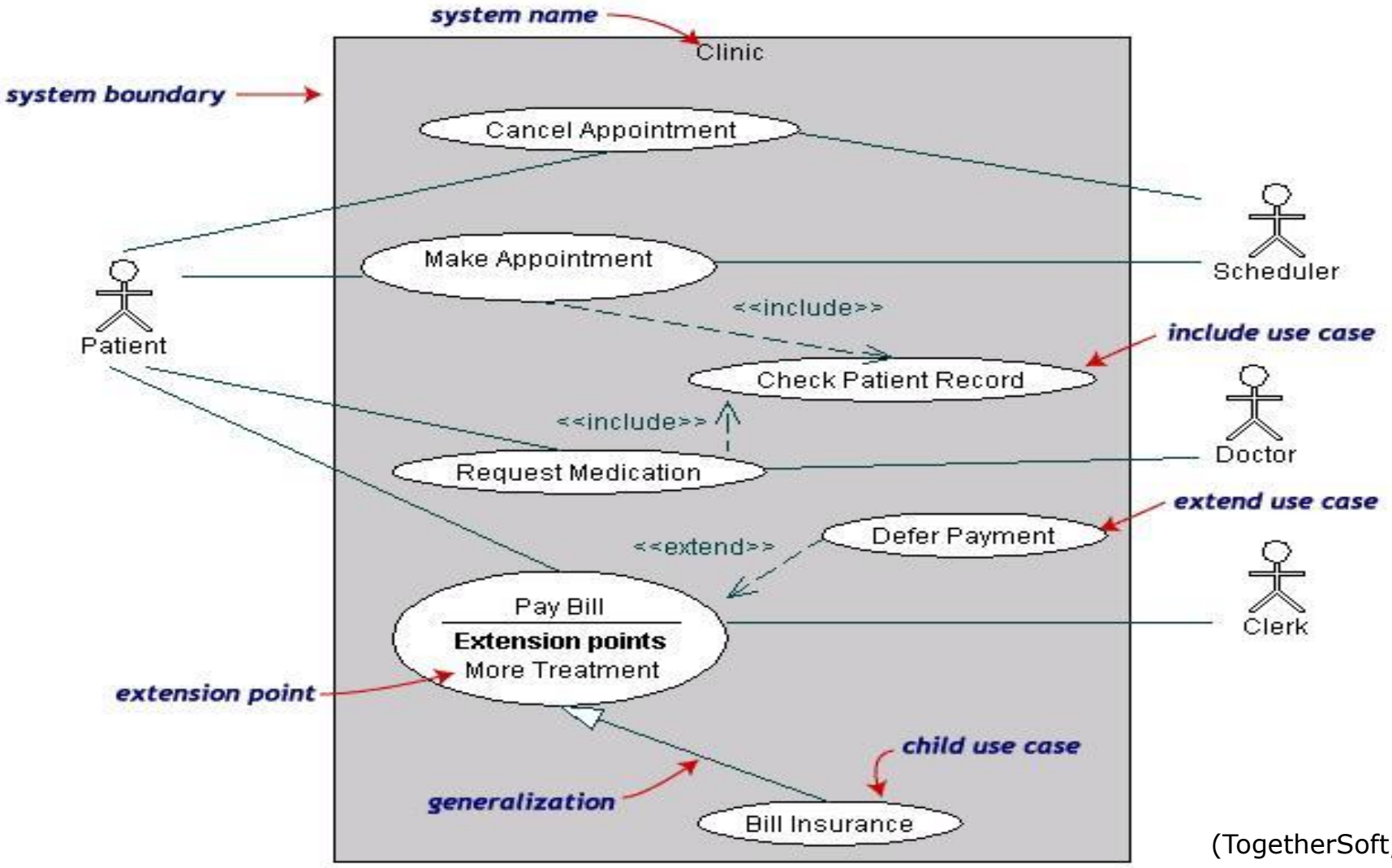
Uml-diagrams.org

Use-Case Diagram (contd)



Uml-diagrams.org

Use Case Diagram (contd)



(TogetherSoft, Inc)

Use Case Diagram (contd)

- Generalization
  - **Pay Bill** is a parent use case and **Bill Insurance** is the child use case
- Include
  - Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask
- Extend
  - The extension point is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point

17.2 Class Diagrams

- Used for describing ***structure and behavior*** in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

Class Diagrams (contd)

- A class diagram depicts classes and their interrelationships

- Here is the simplest possible class diagram

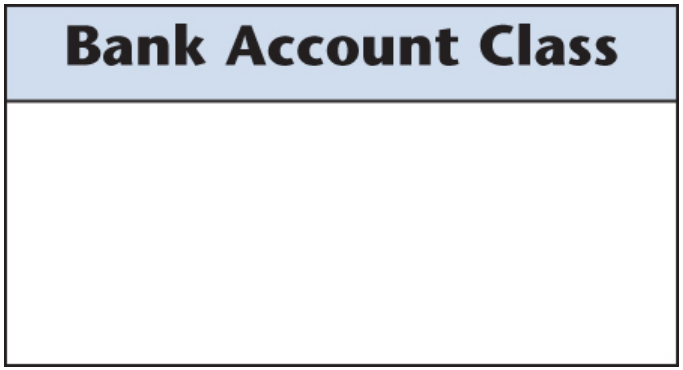


Figure 17.1

Class Diagram (contd)

- class name in top of box
  - write <> on top of interfaces' names – use italics for an abstract class name
- attributes (optional)
  - should include all fields of the object
- operations / methods (optional)
  - may omit trivial (get/set) methods

Class Diagrams (contd)

- Class diagram showing more details of **Bank Account Class**
  - **With an attribute and two operations**

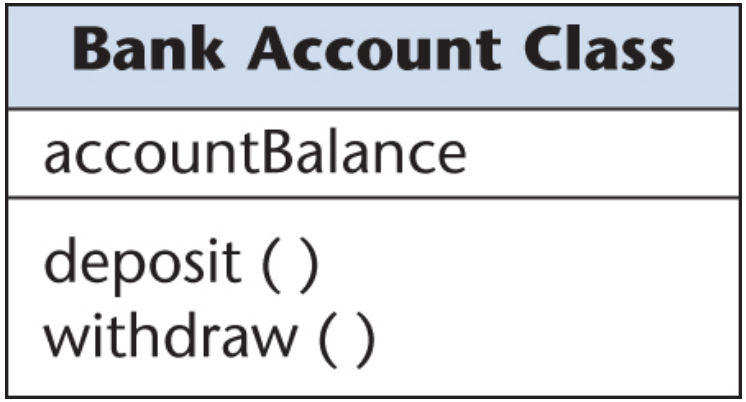
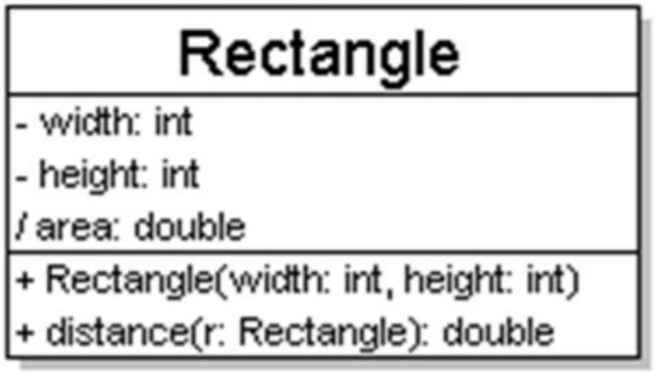


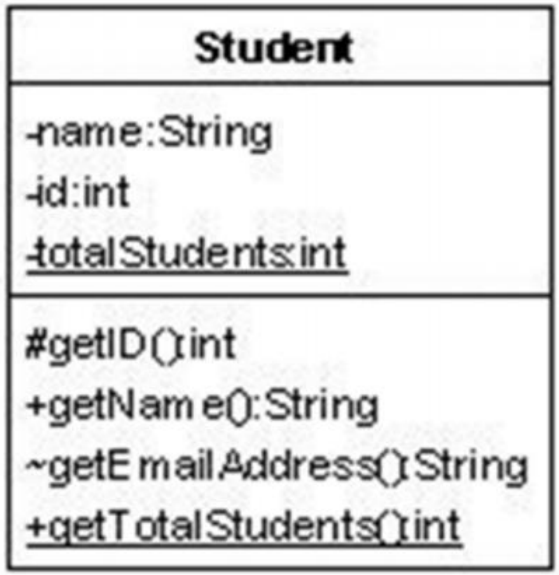
Figure 17.2

- Add as many (or as few) details as appropriate for the current iteration and incrementation

Class Diagram (contd)

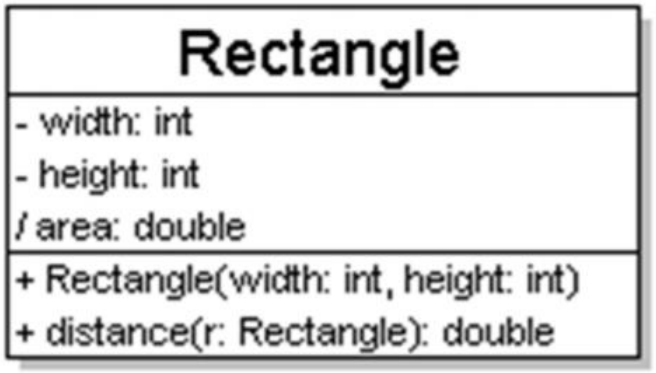
- attributes (fields, instance variables)
  - visibility name : type
  - visibility: + public
    # protected
    - private
    ~ package (default)
    / derived
  - attribute example:
    - balance : double = 0.00

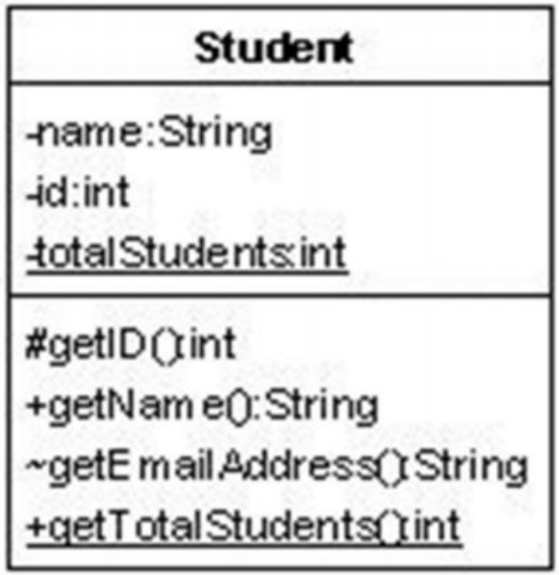




Class Diagram (contd)

- operations / methods
  - visibility name (parameters) : return_type
  - visibility: + public
    # protected
    - private
    ~ package (default)
  - parameter types listed as
    (name: type)
  - omit return_type on constructors and when return type is void
  - method example:
    + distance(p1: Point, p2: Point): double





Class Diagrams: Visibility Prefixes (contd)

- UML visibility prefixes (used for information hiding)
  - **Prefix +** indicates that an attribute or operation is ***public***
    - Visible everywhere
  - **Prefix –** denotes that the attribute or operation is ***private***
    - Visible only in the class in which it is defined
  - **Prefix #** denotes that the attribute or operation is ***protected***
    - Visible either within the class in which it is defined or within subclasses of that class

Class Diagrams: Visibility Prefixes (contd)

- Example:
  - Class diagram with visibility prefixes added

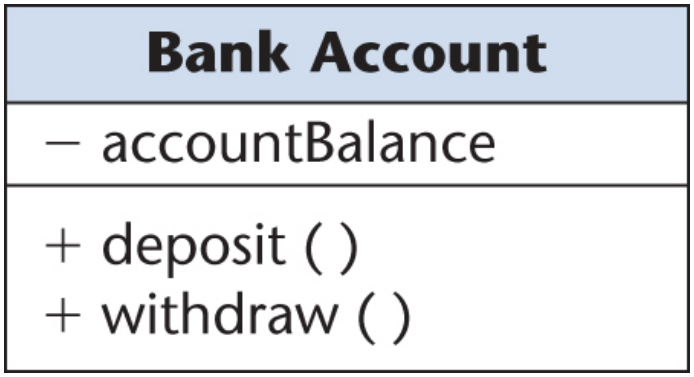


Figure 17.3

  - Attribute accountBalance is visible only within the **Bank Account Class**
  - Operations deposit and withdraw are accessible from anywhere within the software product

Relationships

- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (student enrolls in course)

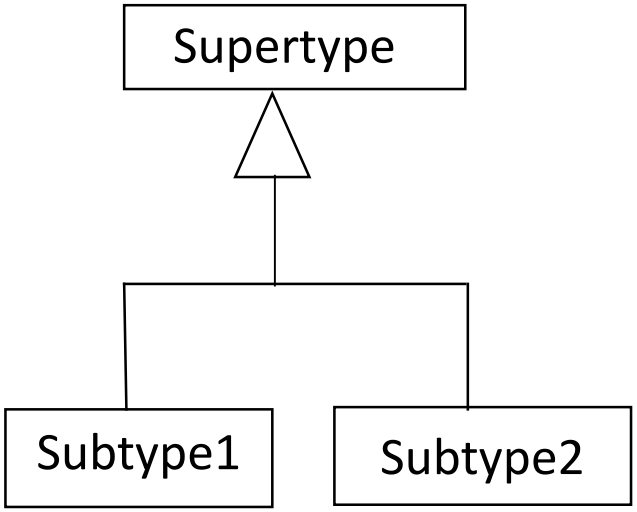
- Associations can be further classified as
  - Aggregation
  - Composition

17.2.4 Generalization

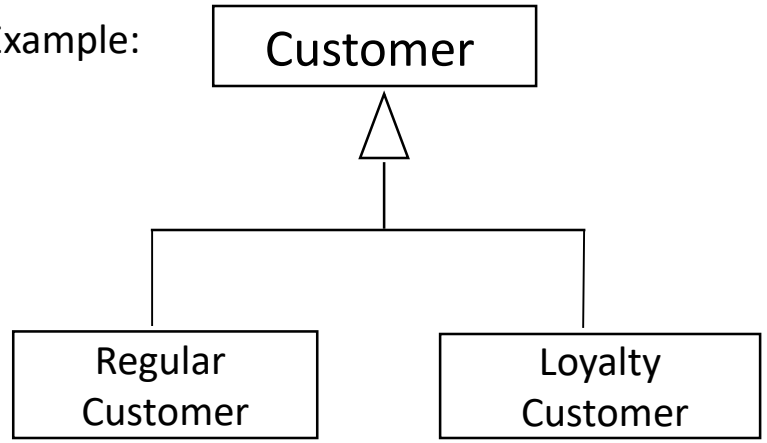
- Inheritance is a required feature of object orientation

- Inheritance is a special case of generalization
  - The UML notation for generalization is an open triangle
  - Sometimes the open triangle is labeled with a discriminator

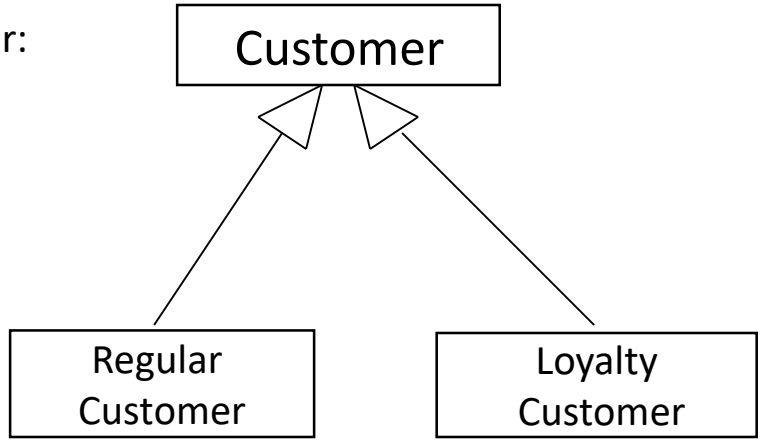
Generalization (contd)



Example:



or:



- Generalization expresses a parent/child relationship among related classes
- Used for abstracting details in several layers

Generalization (contd)

- Every instance of **Investment Class** or its subclasses has an attribute *investmentType* (the discriminator)
  - This attribute can be used to distinguish between instances of the subclasses

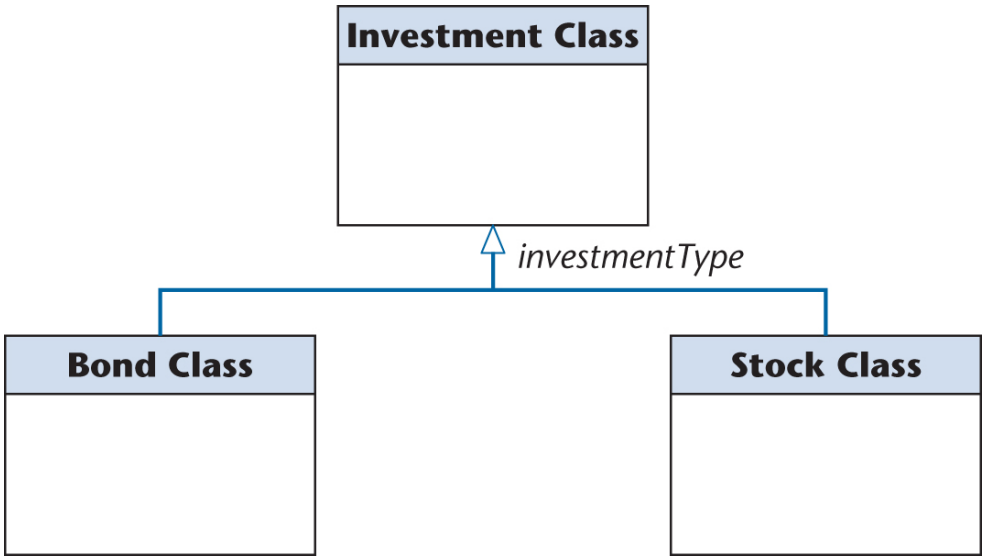
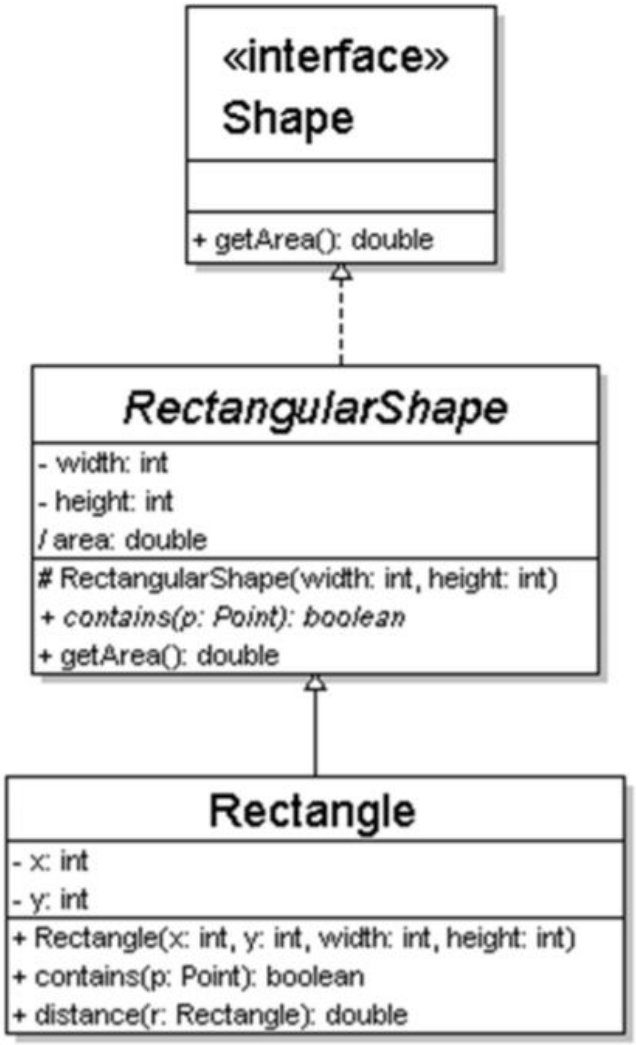


Figure 17.8

Generalization (contd)

- class: solid line, black arrow
- abstract class: solid line, white arrow
- interface: dashed line, white arrow



17.2.5 Association

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.

Association (contd)

- Example of association:

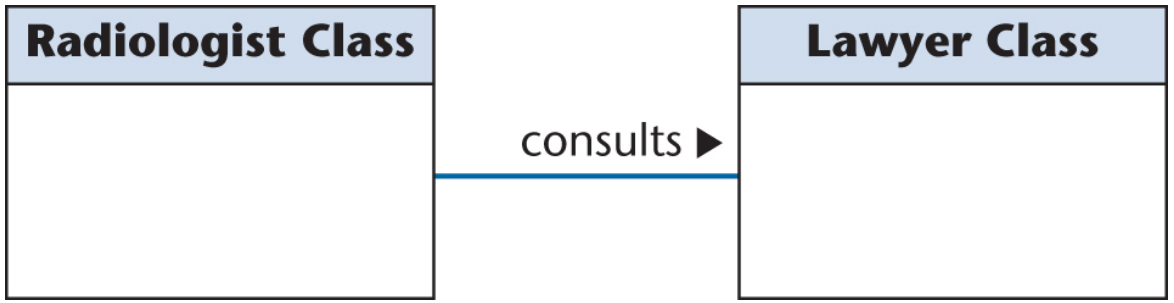


Figure 17.9

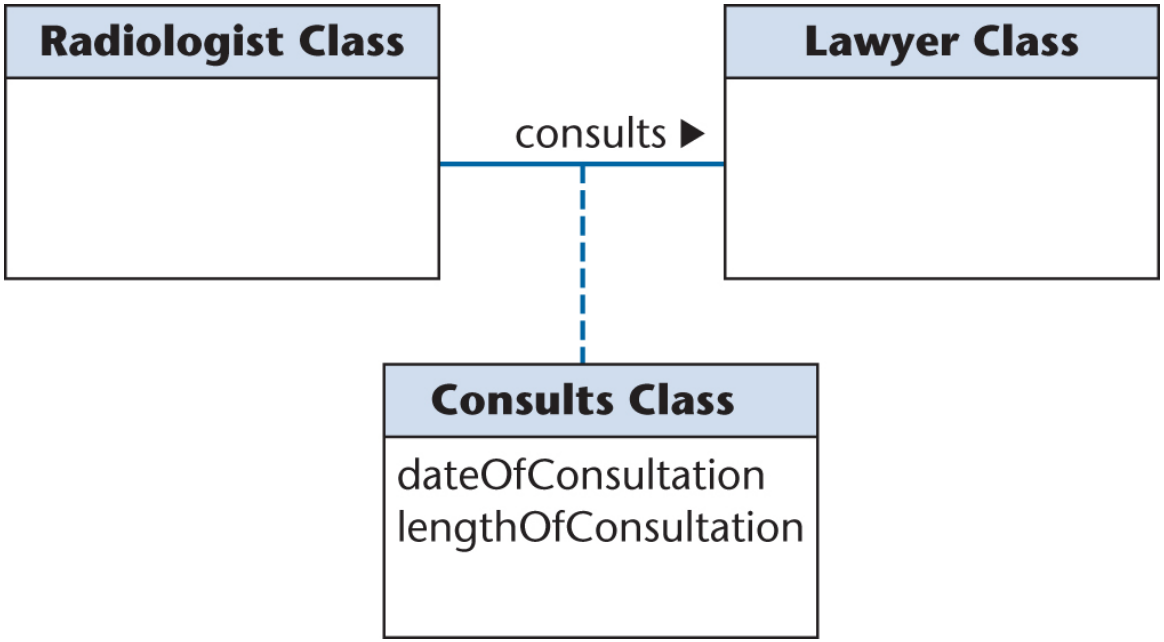
- A radiologist consults a lawyer
  - The optional navigation triangle shows the direction of the association

Association (contd)

- The association between the two classes may be modeled as a class
  - Example: Suppose the radiologist consults the lawyer on a number of occasions, each one for a different length of time
    - A class diagram is needed such as that depicted in the next slide

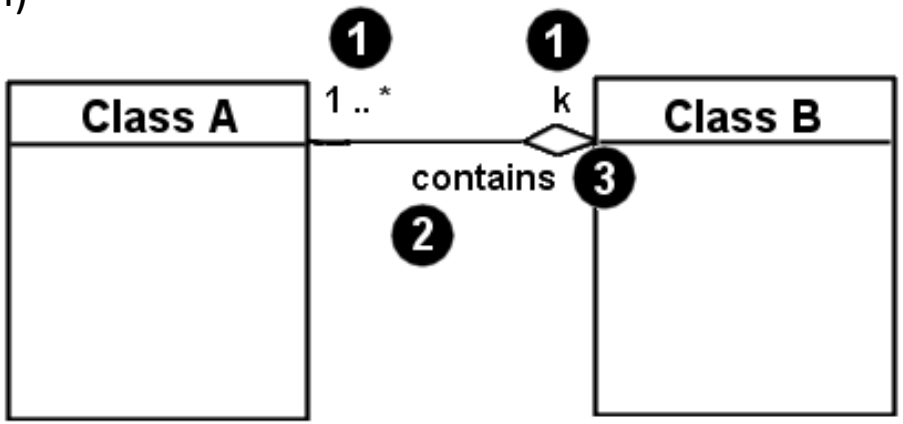
Association (contd)

- Class, which is called an association class
  - Because it is both an association and a class

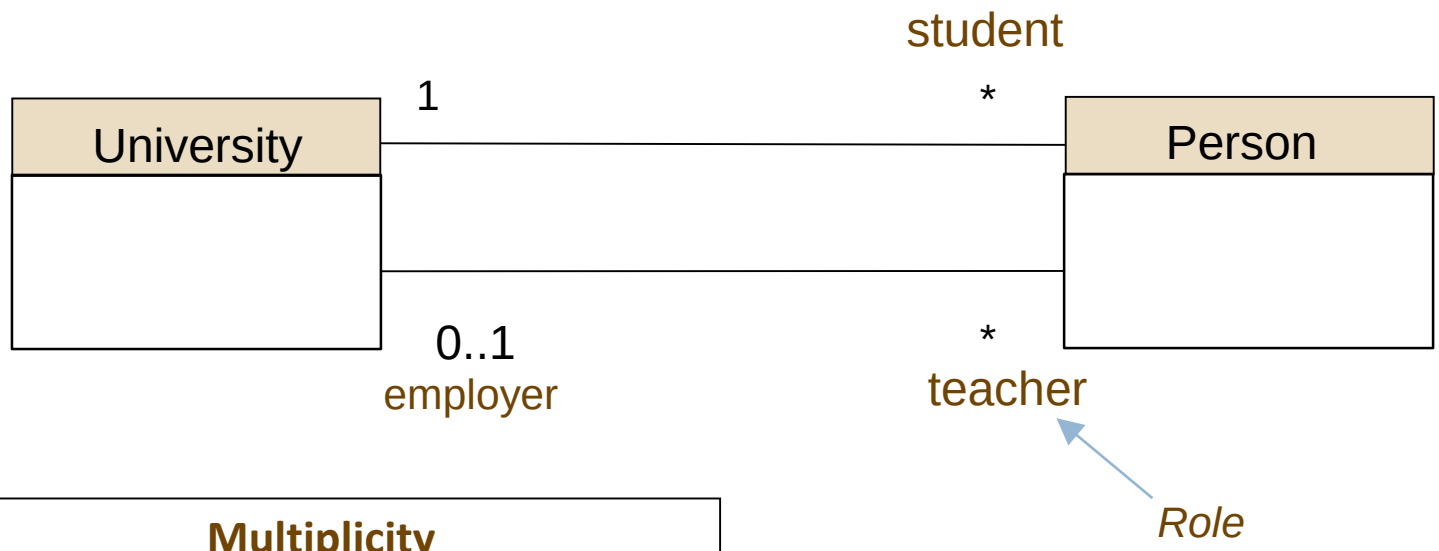


Association (contd)

- 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. Role name (what relationship the objects have)
  3. Navigability (direction)



17.2.2 Multiplicity



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

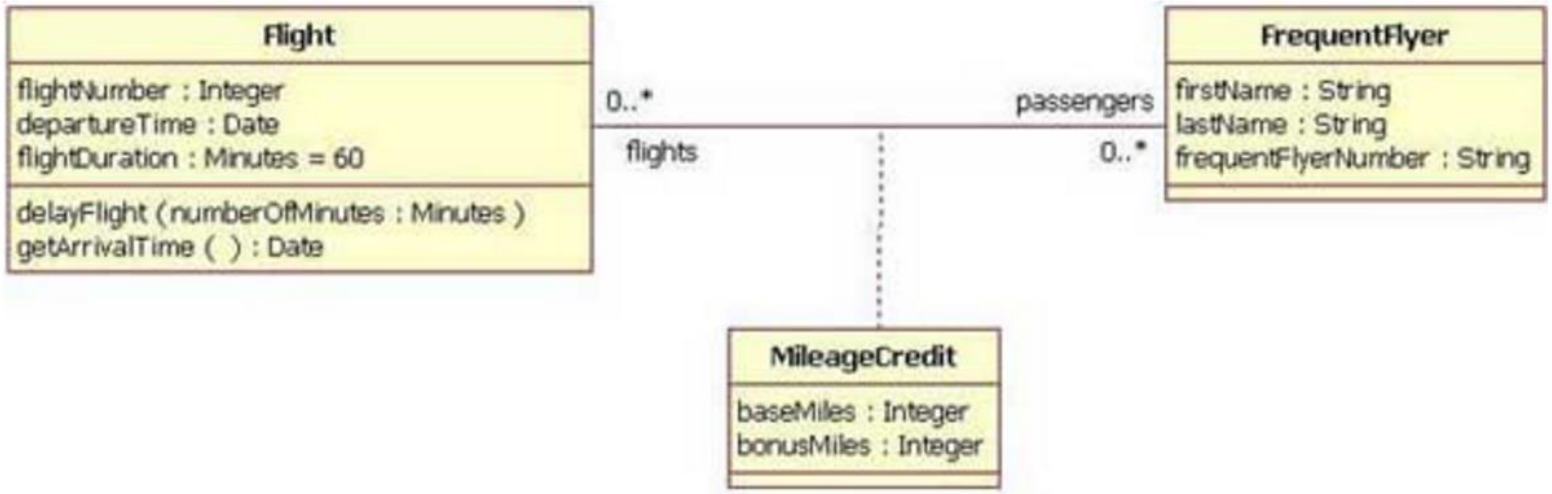
Role

"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."

Multiplicity (contd)

- The numbers next to the ends of the lines denote multiplicity
  - The number of times that the one class is associated with the other class

Multiplicity – more example



Multiplicity (contd)

- Example: "A car consists of one chassis, one engine, 4 or 5 wheels, an optional sun roof, zero or more fuzzy dice hanging from the rear-view mirror, and 2 or more seats"

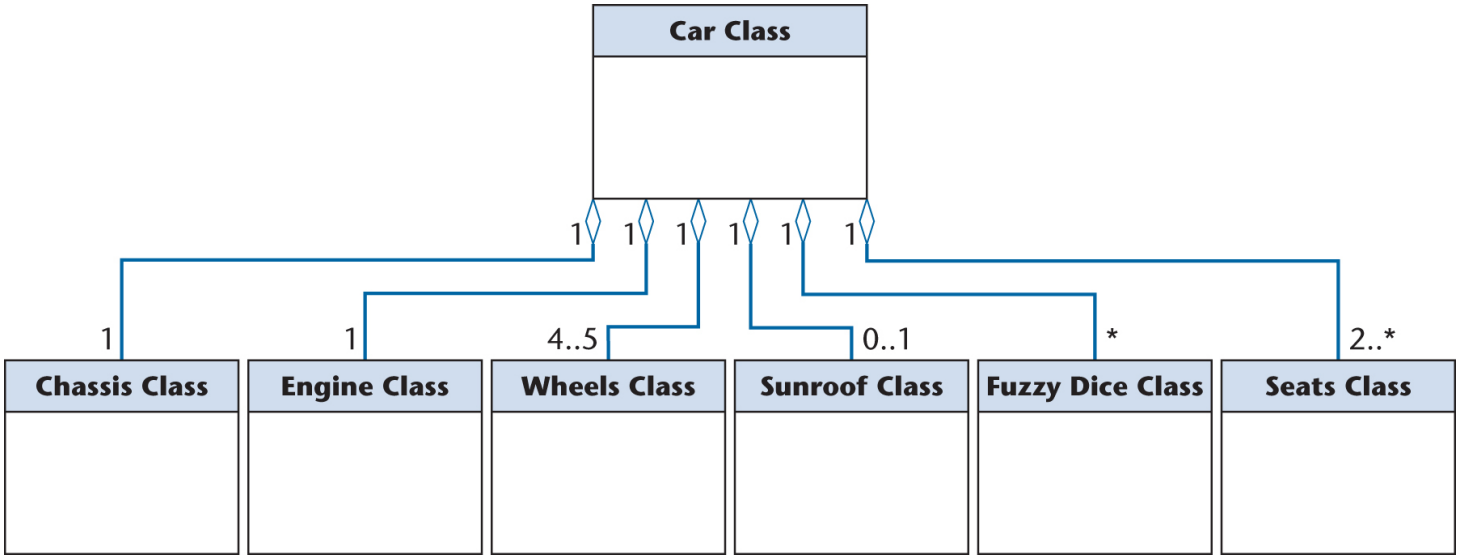


Figure 17.5

17.2.1 Aggregation

- Example: “A car consists of a chassis, an engine, wheels, and seats”

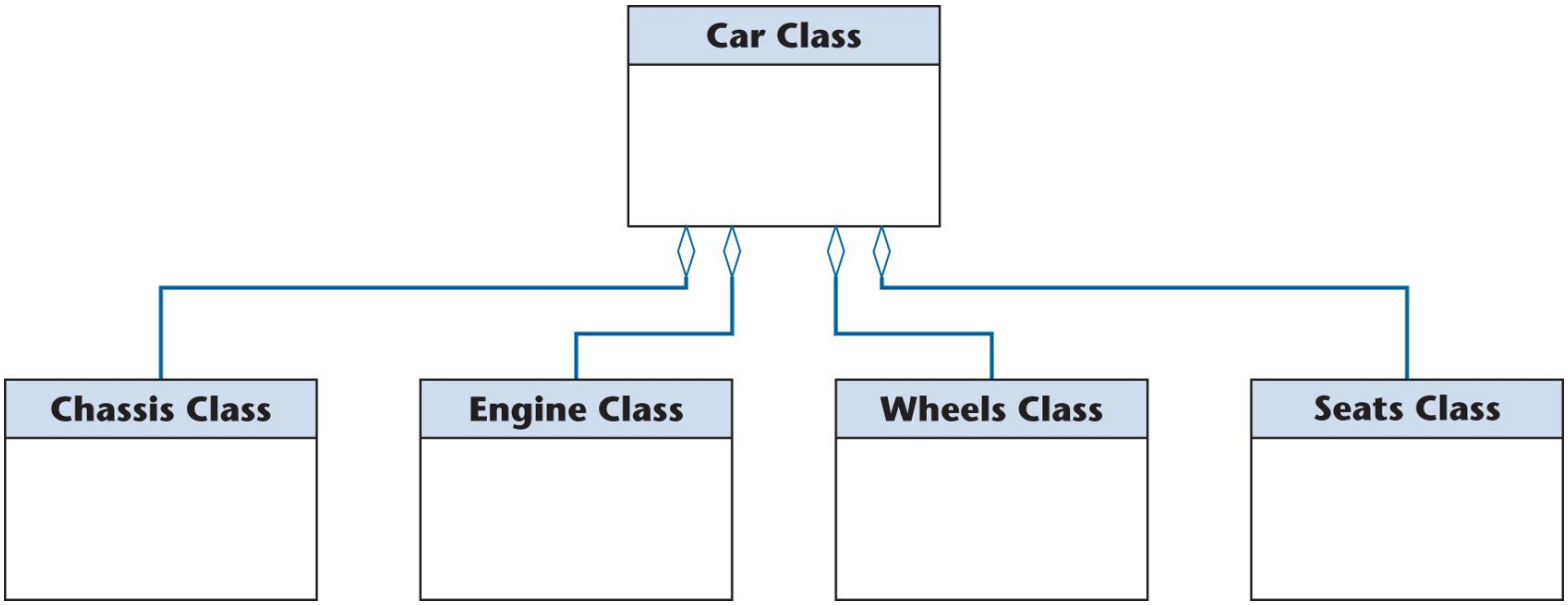


Figure 17.4

Aggregation (contd)

- The open diamonds denote aggregation
  - Aggregation is the UML term for the **part–whole** relationship

- The diamond is placed at the “whole” (car) end, not the “part” (chassis, engine, wheels, or seats) end of the line connecting a part to the whole

Multiplicity (contd)

- The line connecting **Chassis Class** to **Car Class**
  - The 1 at the “part” end of the line denotes that there is one chassis involved
  - The 1 at the “whole” end denotes that there is one car involved
- Each car has one chassis, as required
- Similar observations hold for the line connecting **Engine Class** to **Car Class**

Multiplicity (contd)

- The line connecting **Wheels Class** to **Car Class**
  - The 4..5 at the “part” end together with the 1 at the “whole” end denotes that each car has from 4 to 5 wheels (the fifth wheel is the spare)

- A car has 4 or 5 wheels, as required
  - Instances of classes come in whole numbers only

Multiplicity (contd)

- The line connecting **Sun Roof Class** to **Car Class**
  - Two dots .. denote a range, so the 0..1 means zero or one, the UML way of denoting “optional”
- A car has an optional sun roof, as required

Multiplicity (contd)

- The line connecting **Fuzzy Dice Class** to **Car Class**
  - The * by itself means zero or more

- Each car has zero or more fuzzy dice hanging from the rear-view mirror, as required

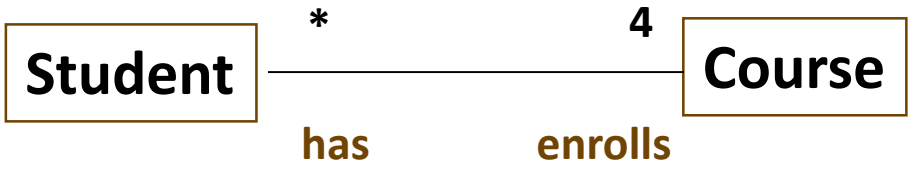
Multiplicity (contd)

- The line connecting **Seats Class** to **Car Class**
  - An asterisk in a range denotes “or more,” so the 2..* means 2 or more
- A car has two or more seats, as required

Multiplicity (contd)

- If the exact multiplicity is known, use it
  - Example: The 1 that appears in 8 places
- If the range is known, use the range notation
  - Examples: 0..1 or 4..5
- If the number is unspecified, use the asterisk
  - Example: *
- If the range has upper limit unspecified, combine the range notation with the asterisk notation
  - Example: 2..*

Multiplicity (contd): Implementation Example




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

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

17.2.3 Composition

- Aggregation example: Every chess board consists of 64 squares

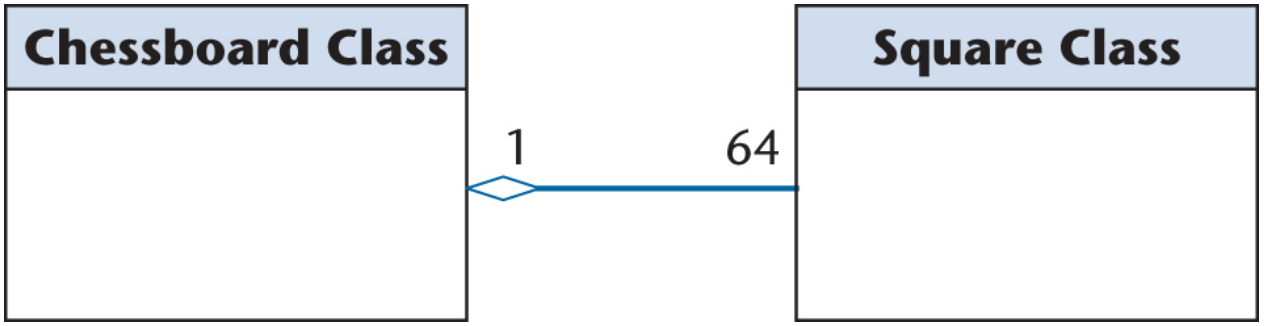


Figure 17.6

- This relationship goes further
  - It is an instance of *composition*, a stronger form of aggregation

Composition (contd)

- Aggregation
  - 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

Composition (contd)

- Composition is depicted by a solid diamond

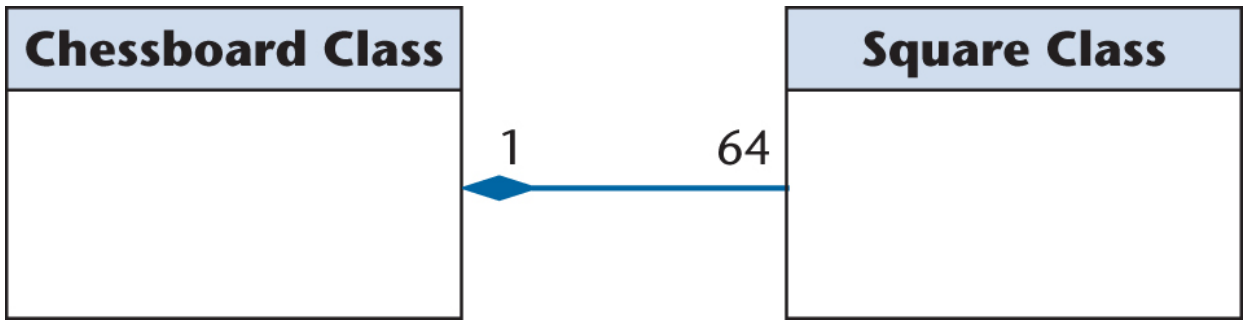
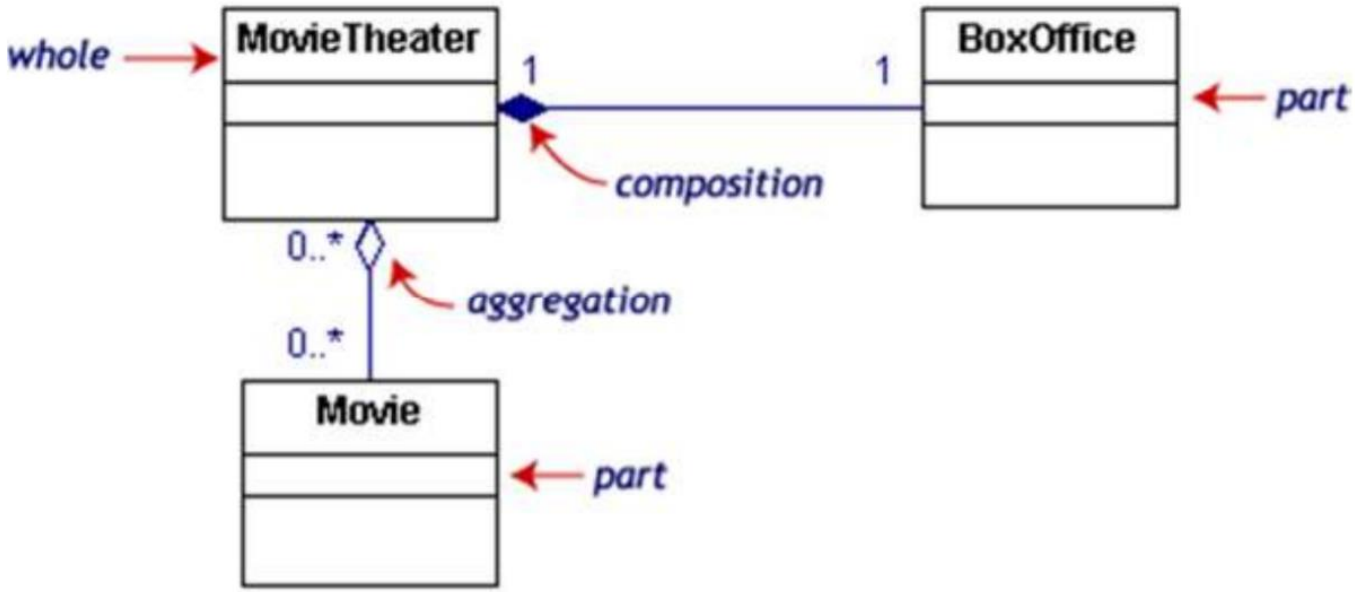


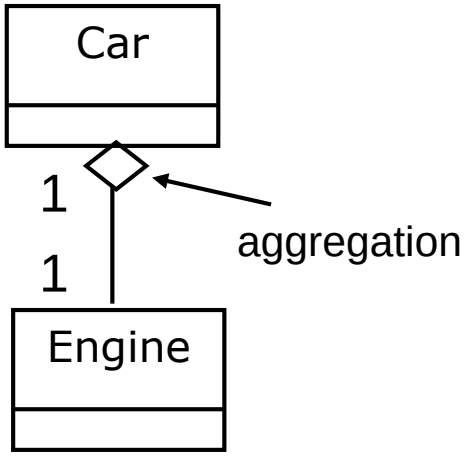
Figure 17.7

Composition/Aggregation Example

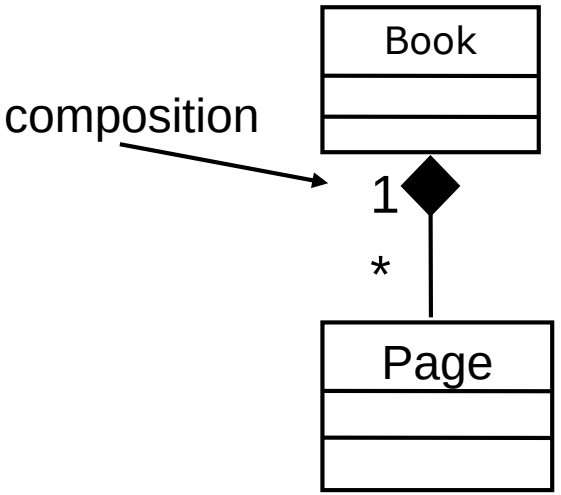


Summary of Association

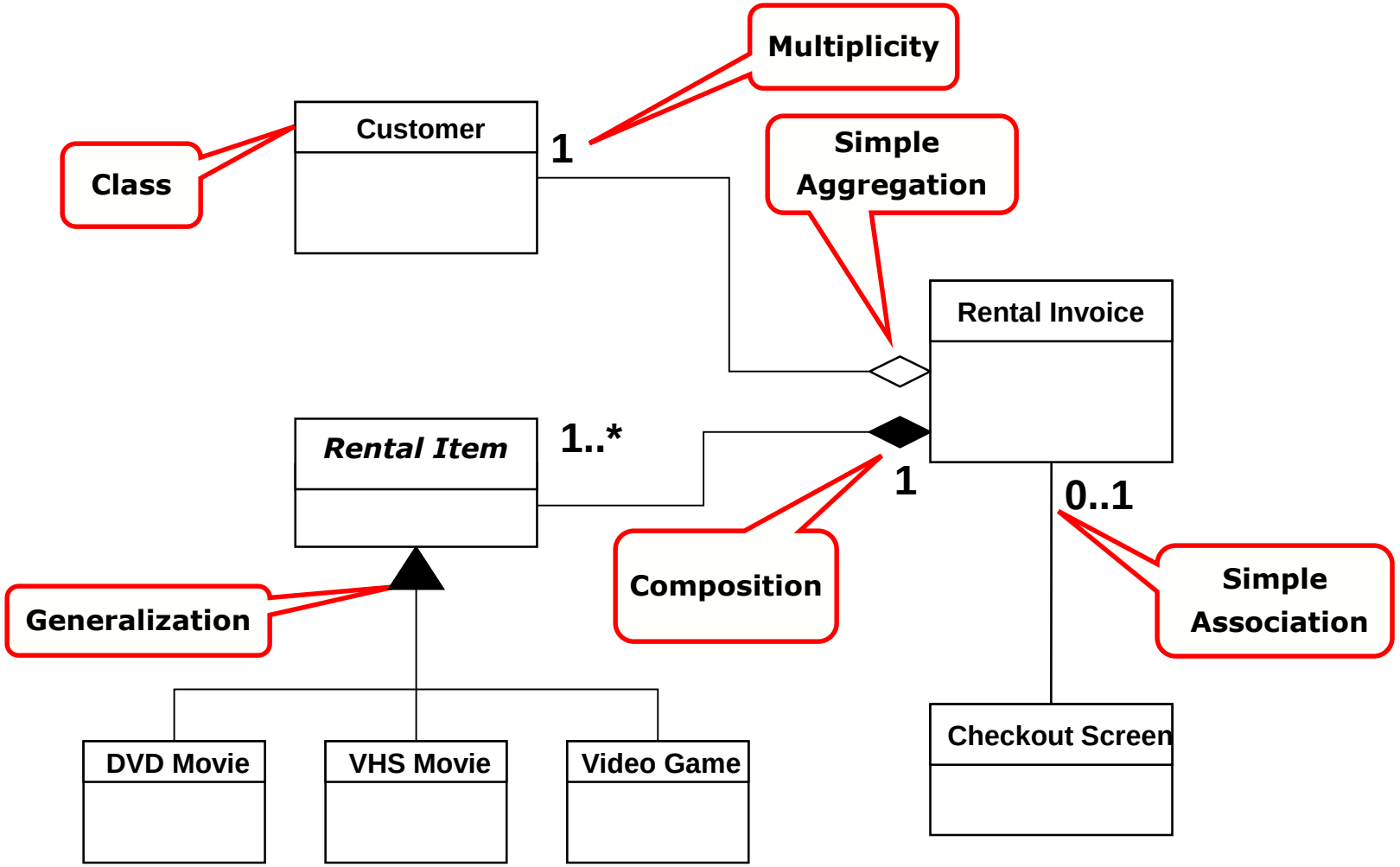
- **Aggregation**: "is part of"
  - Symbolized by a clear white diamond



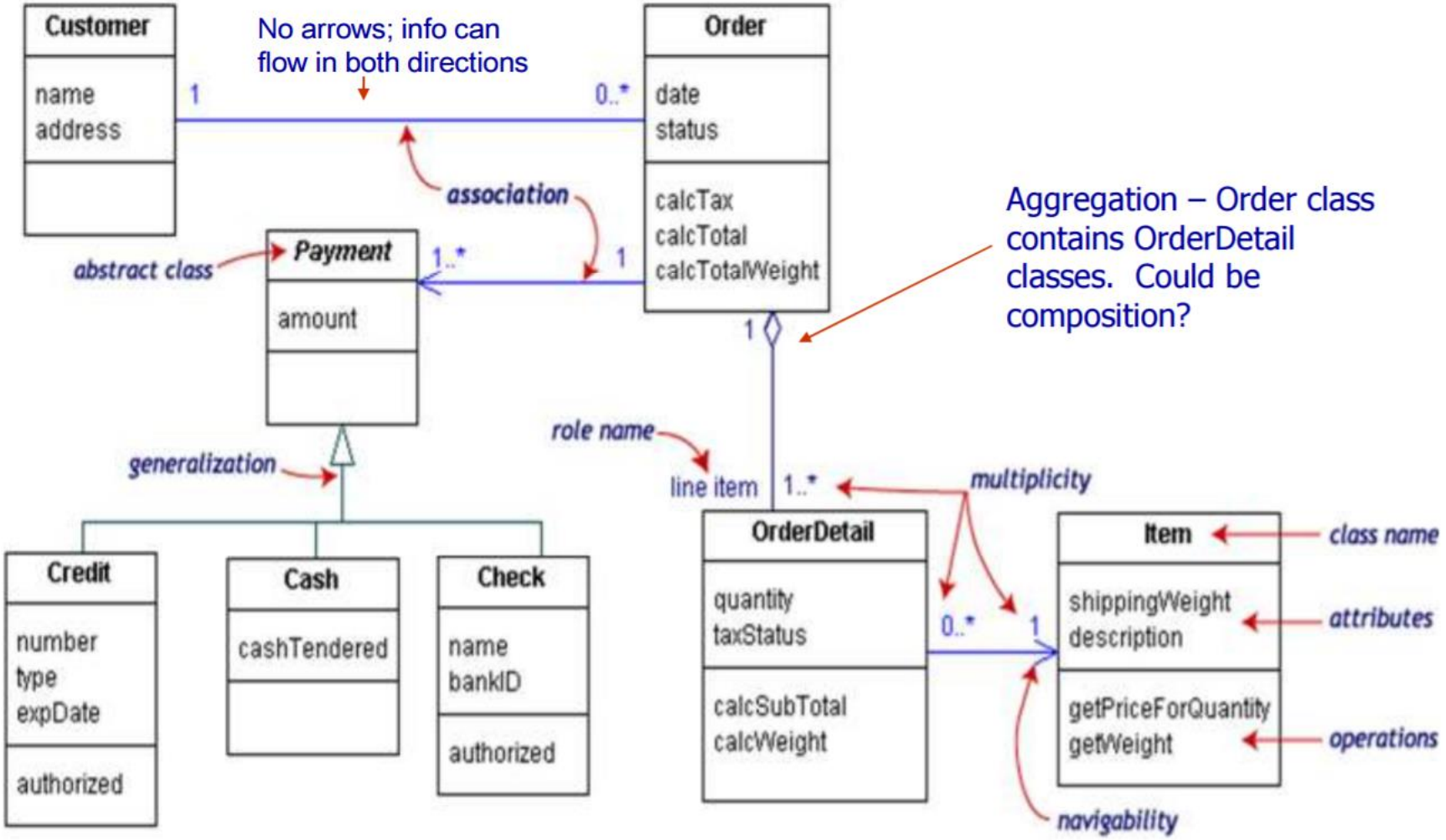
- **Composition**: "is entirely made of"
  - Stronger version of aggregation
  - The parts live and die with the whole
  - Symbolized by a black diamond



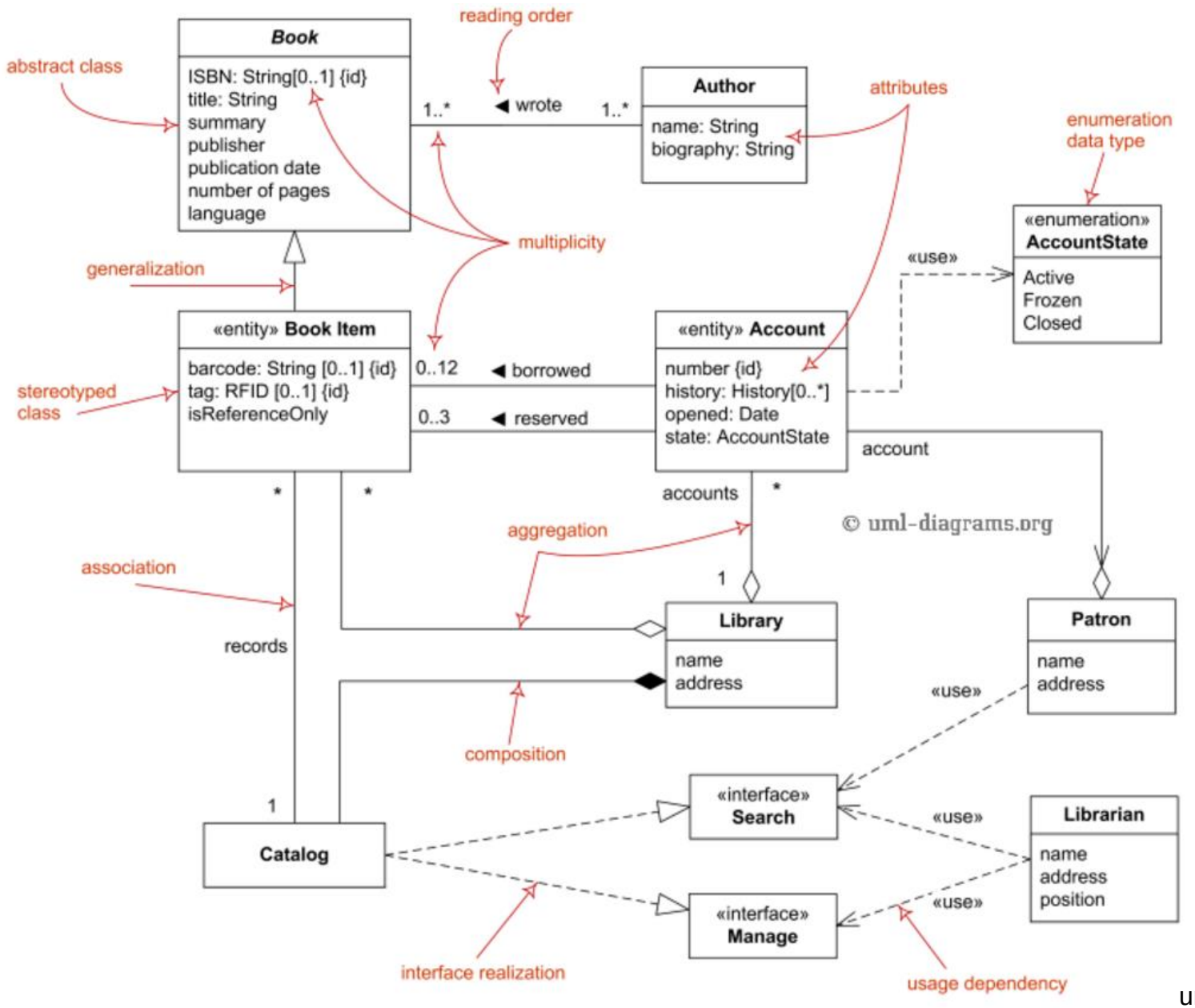
Class Diagram Example



Class Diagram Example

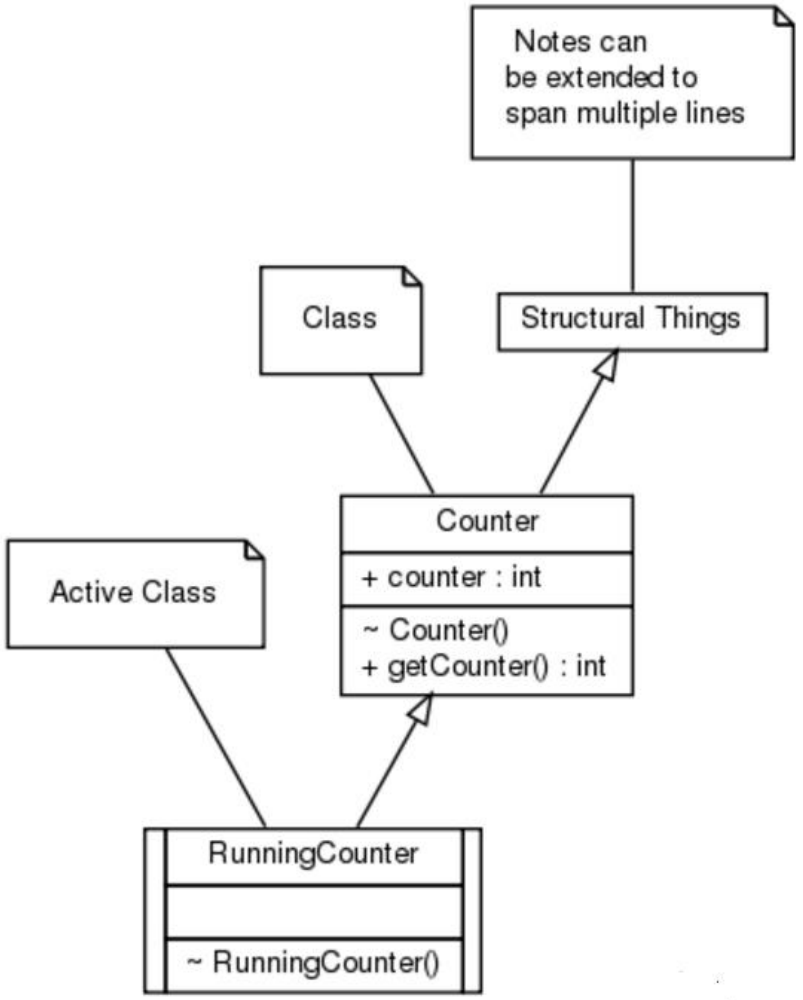


Class Diagram Example

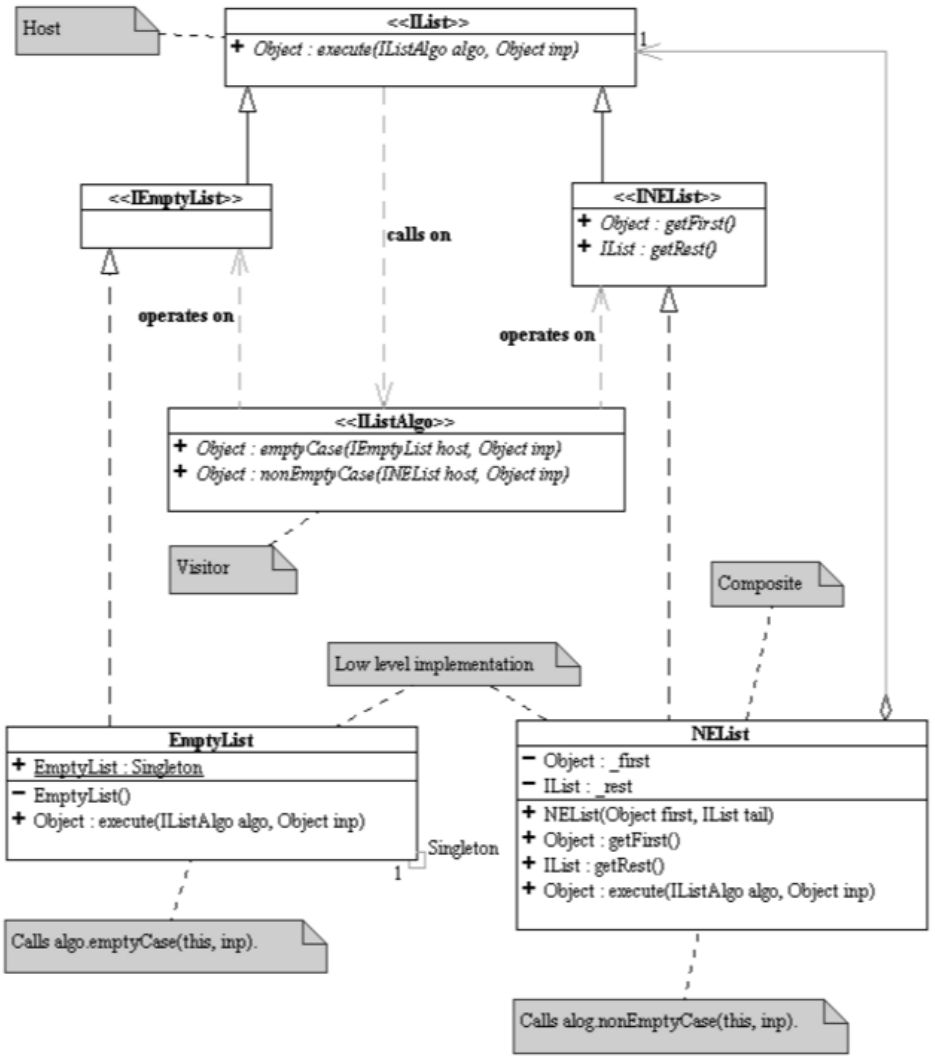


17.3 Notes

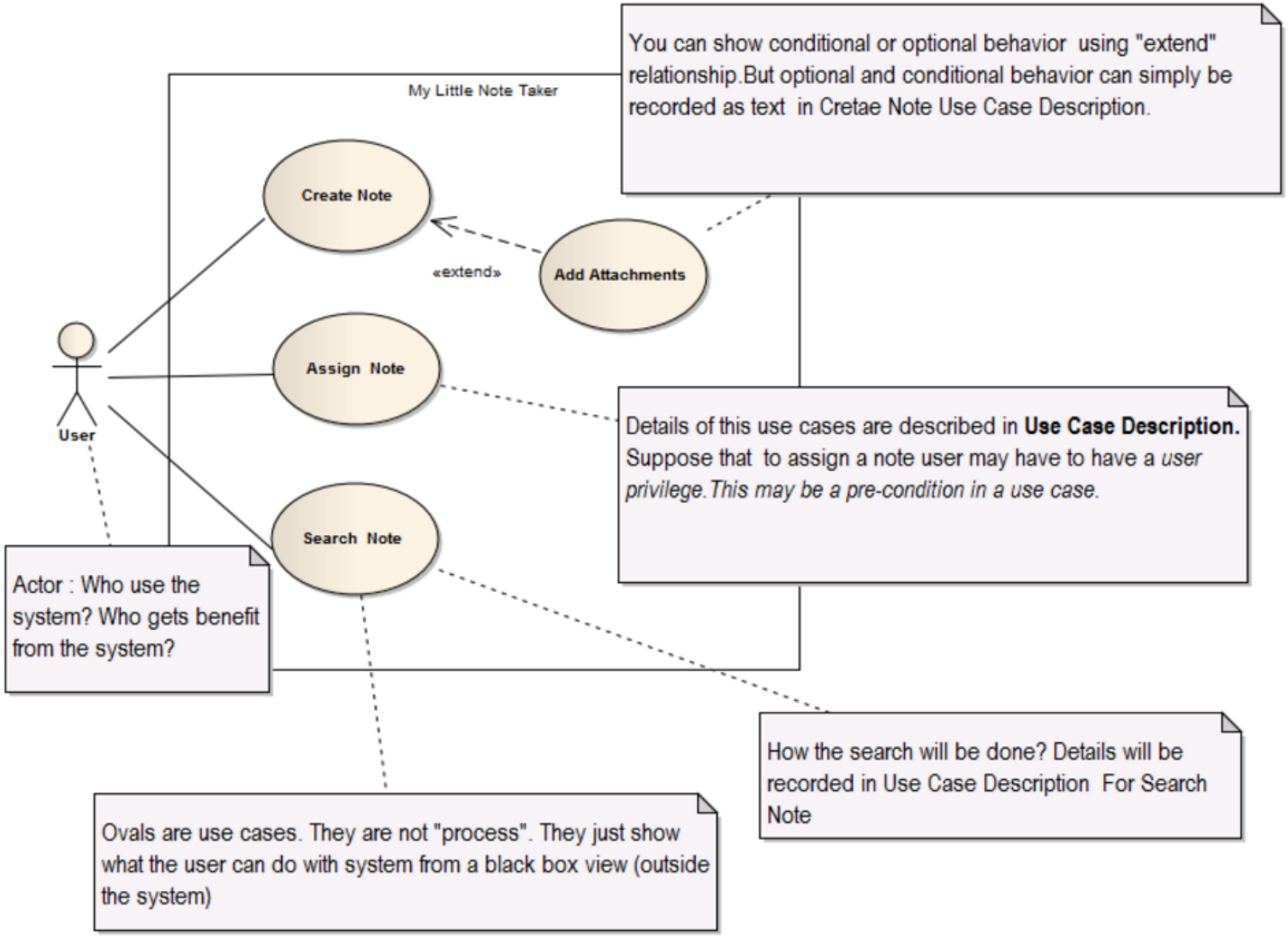
- A comment in a UML diagram is called a *note*
  - Depicted as a rectangle with the top right-hand corner bent over
  - A dashed line is drawn from the note to the item to which the note refers



Notes Example



Notes Example



17.6 Interaction Diagrams

- Interaction diagrams show how objects interact with one another

- UML supports two types of interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams

Sequence Diagrams (contd)

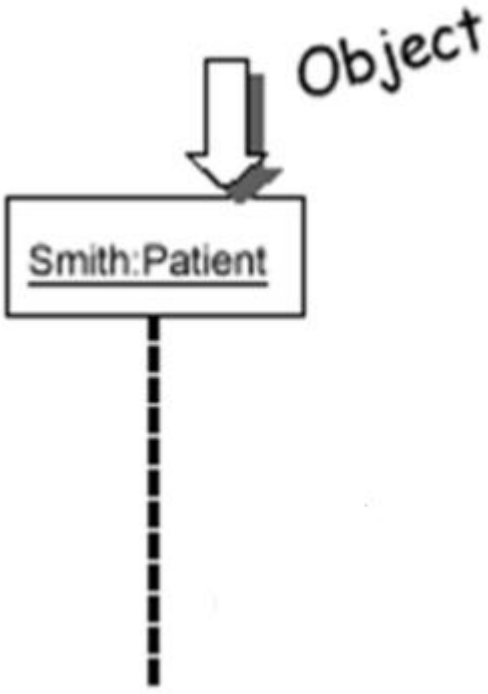
- Sequence diagrams demonstrate **the behavior of objects in a use case**
  - by describing the objects and the messages they pass
- The horizontal dimension shows the objects participating in the interaction
- The vertical arrangement of messages indicates their order
- The labels may contain the seq. # to indicate concurrency

Sequence Diagram - Objects

- An object: a square with object type, optionally preceded by object name and colon
  - write object's name if it clarifies the diagram
  - object's "life line" represented by dashed vert. line

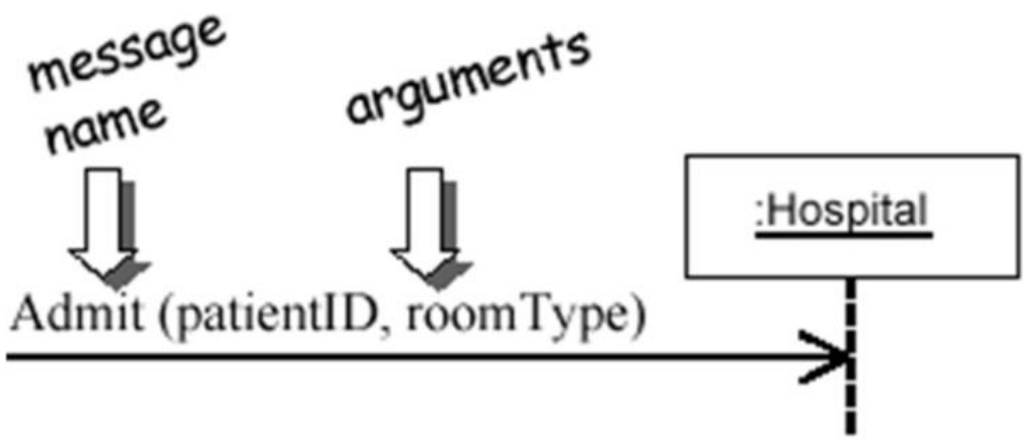
Syntax

<objectname>:<classname>



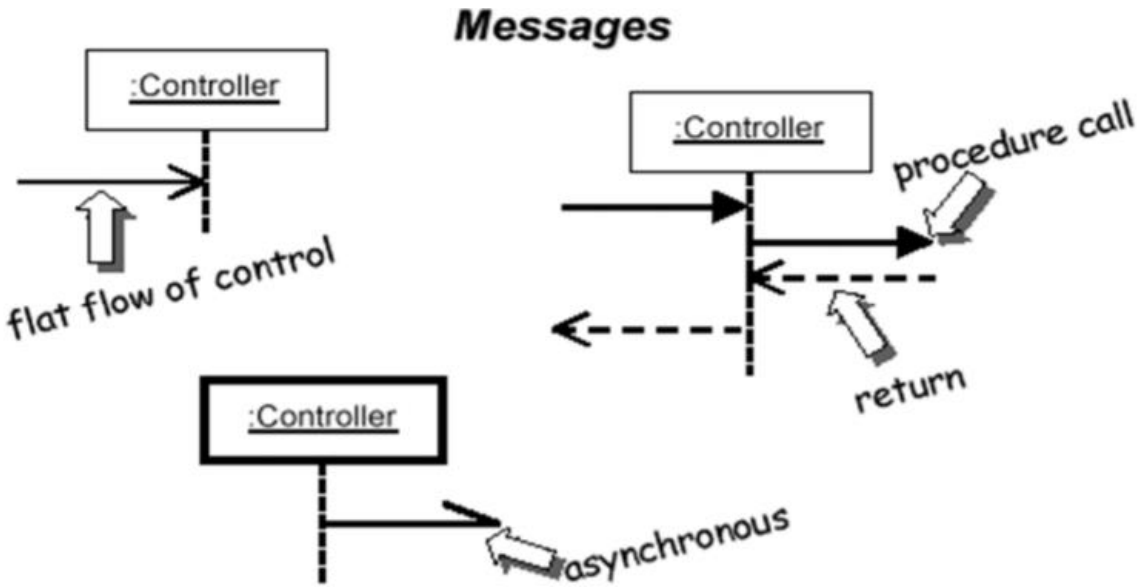
Sequence Diagram – Messages between Objects

- Message (method call): horizontal arrow to other object
  - write message name and arguments above arrow



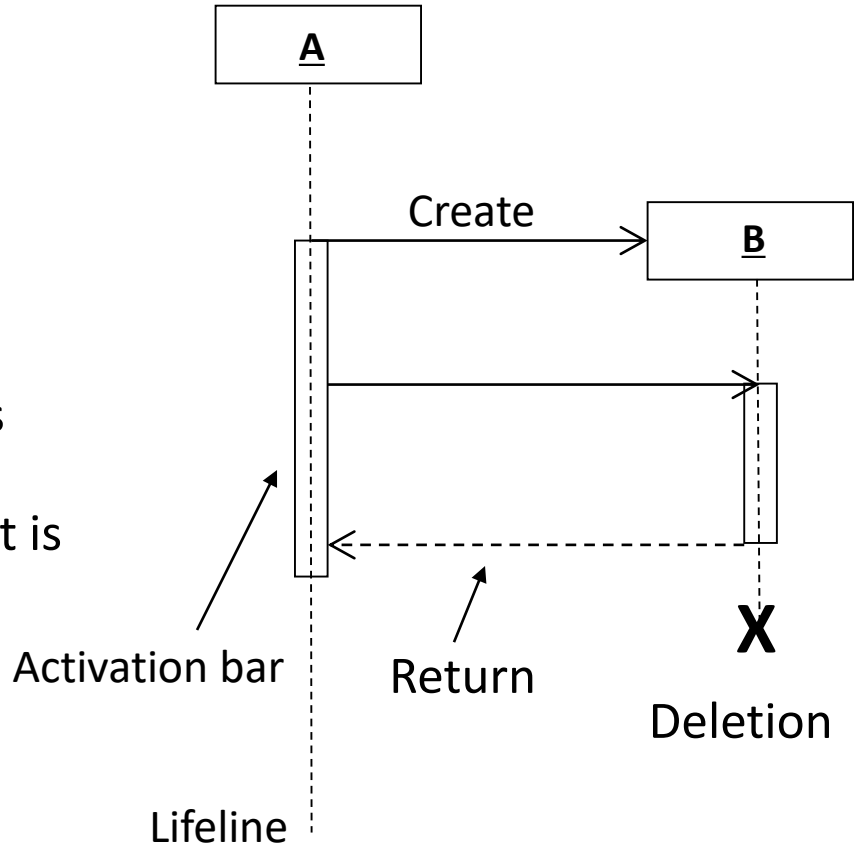
Sequence Diagram – Different Types of Messages

- Type of arrow indicates types of messages
  - dashed arrow back indicates return
  - different arrowheads for normal / concurrent (asynchronous) methods

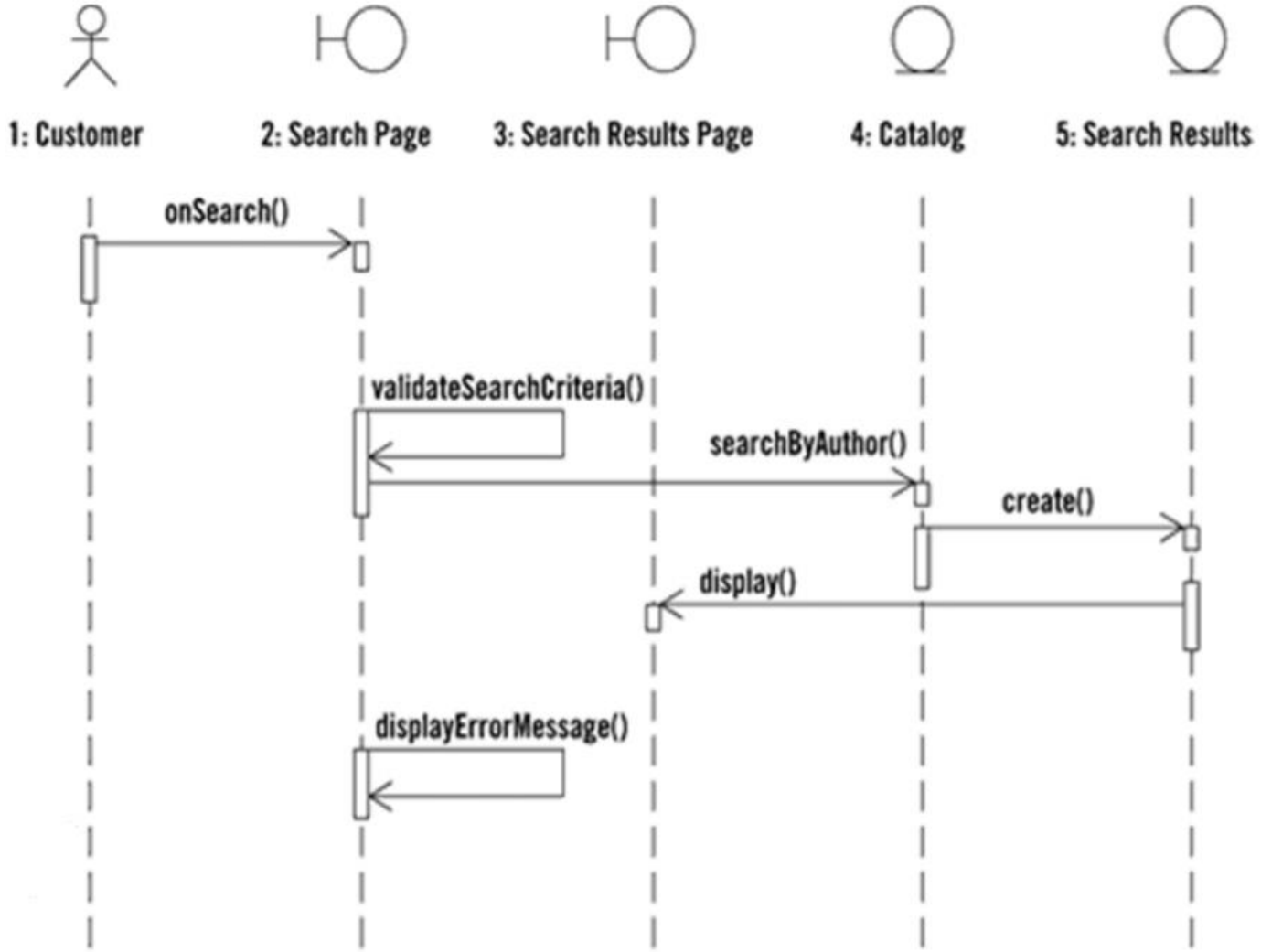


Sequence Diagrams (contd)

- 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 Example



Sequence Diagrams

- Example: Dynamic creation followed by destruction of an object

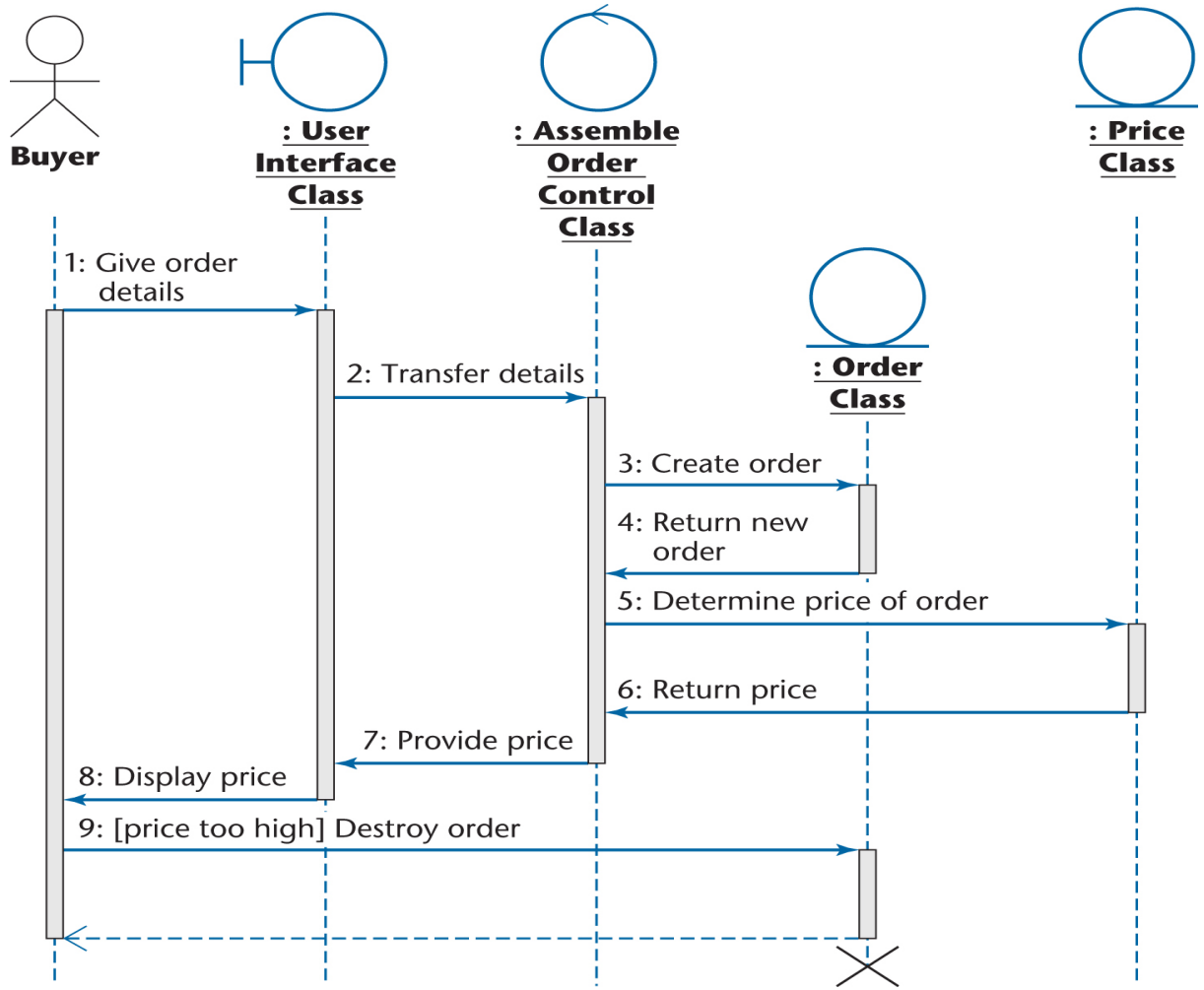


Figure 17.14

Sequence Diagrams (contd)

- The lifelines in the sequence diagram
  - An active object is denoted by a thin rectangle (activation box) in place of the dashed line

- Creation of the **: Order Class** object is denoted by the lifeline starting at the point of dynamic creation

- Destruction of that object after it receives message

  9: Destroy order

  is denoted by the heavy **X**

Sequence Diagrams (contd)

- A message is optionally followed by a message sent back to the object that sent the original message

- Even if there is a reply, it is not necessary that a specific new message be sent back
  - Instead, a dashed line ending in an open arrow indicates a *return* from the original message, as opposed to a new message

Sequence Diagrams (contd)

- There is a guard on the message
  - 9: [offer rejected] Destroy order
  - Message 9 is sent only if the buyer decides not to purchase the item because the price is too high
- A **guard** (condition) is something that is true or false
  - The message sent only if the guard is true
- The purpose of a guard
  - To ensure that the message is sent only if the relevant condition is true

Sequence Diagrams (contd)

- Sequence diagram for elevator

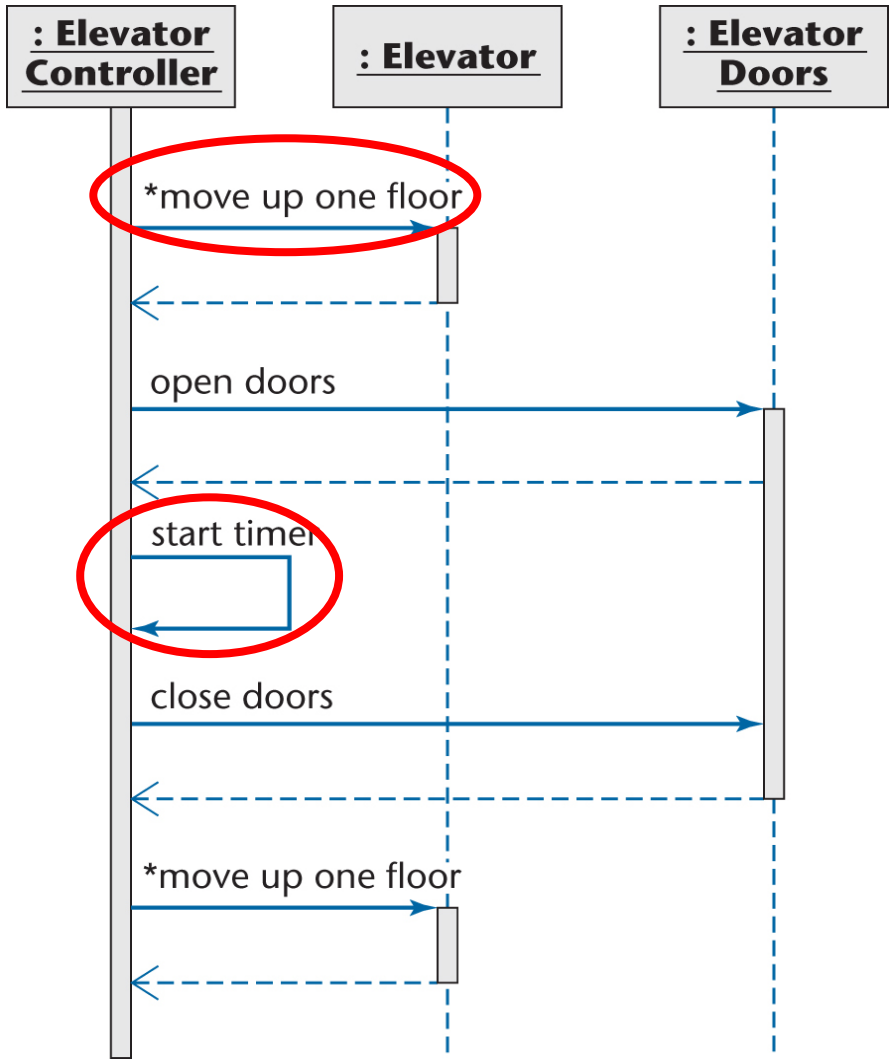


Figure 17.15

Sequence Diagrams (contd)

- **Iteration** an indeterminate number of times is modeled by an asterisk (Kleene star)

- Example: Elevator (see next slide)

  *move up one floor

  - The message means: “move up zero or more floors”

Sequence Diagrams (contd)

- An object can send a message to itself
  - A ***self-call***

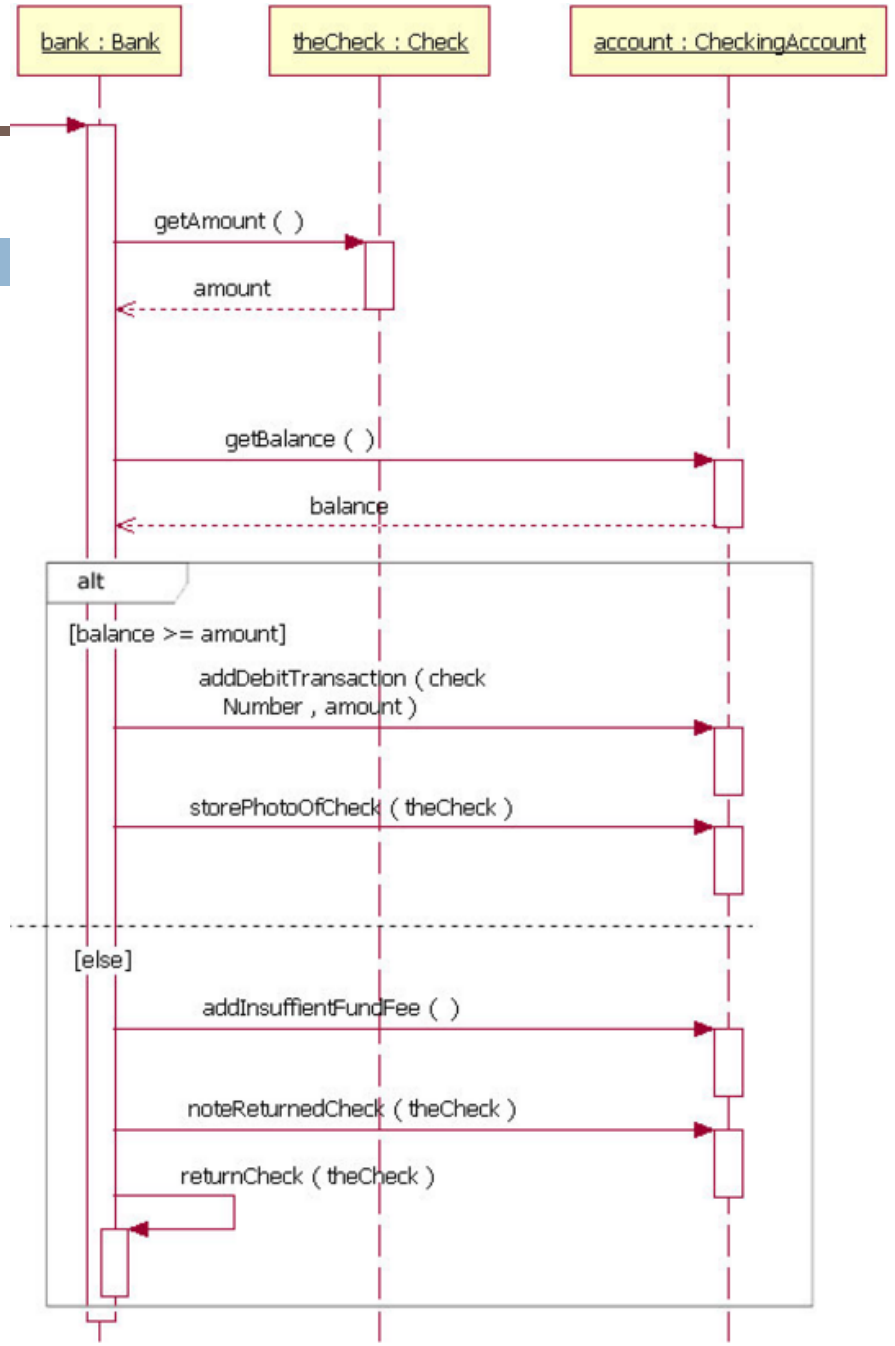
- Example:
  - The elevator has arrived at a floor
  - The elevator doors now open and a timer starts
  - At the end of the timer period the doors close again
  - The elevator controller sends a message to itself to start its timer — this self-call is shown in the previous UML diagram

Sequence Diagram – Selection/Loops

- Frame: box around part of a sequence diagram to indicate selection or loop
  - if → (opt) [condition]
  - if/else → (alt) [condition], separated by horizontal dashed line
  - loop → (loop) [condition or items to loop over]

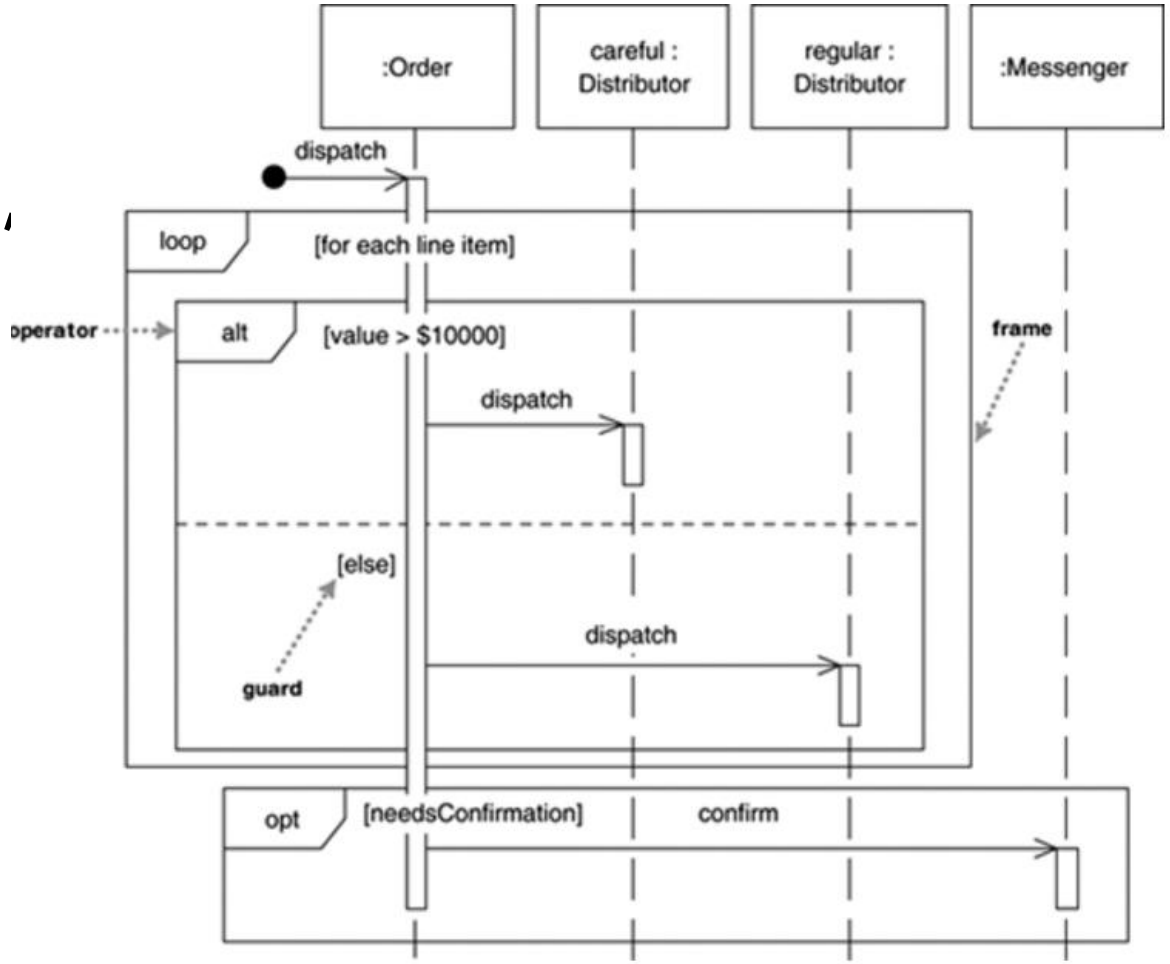
Sequence Diagram -

- if → (opt) [condition]
- if/else → (alt) [condition], separated by horizontal dashed line
- loop → (loop) [condition or items to loop over]



Sequence Diagram – Selection/Loops

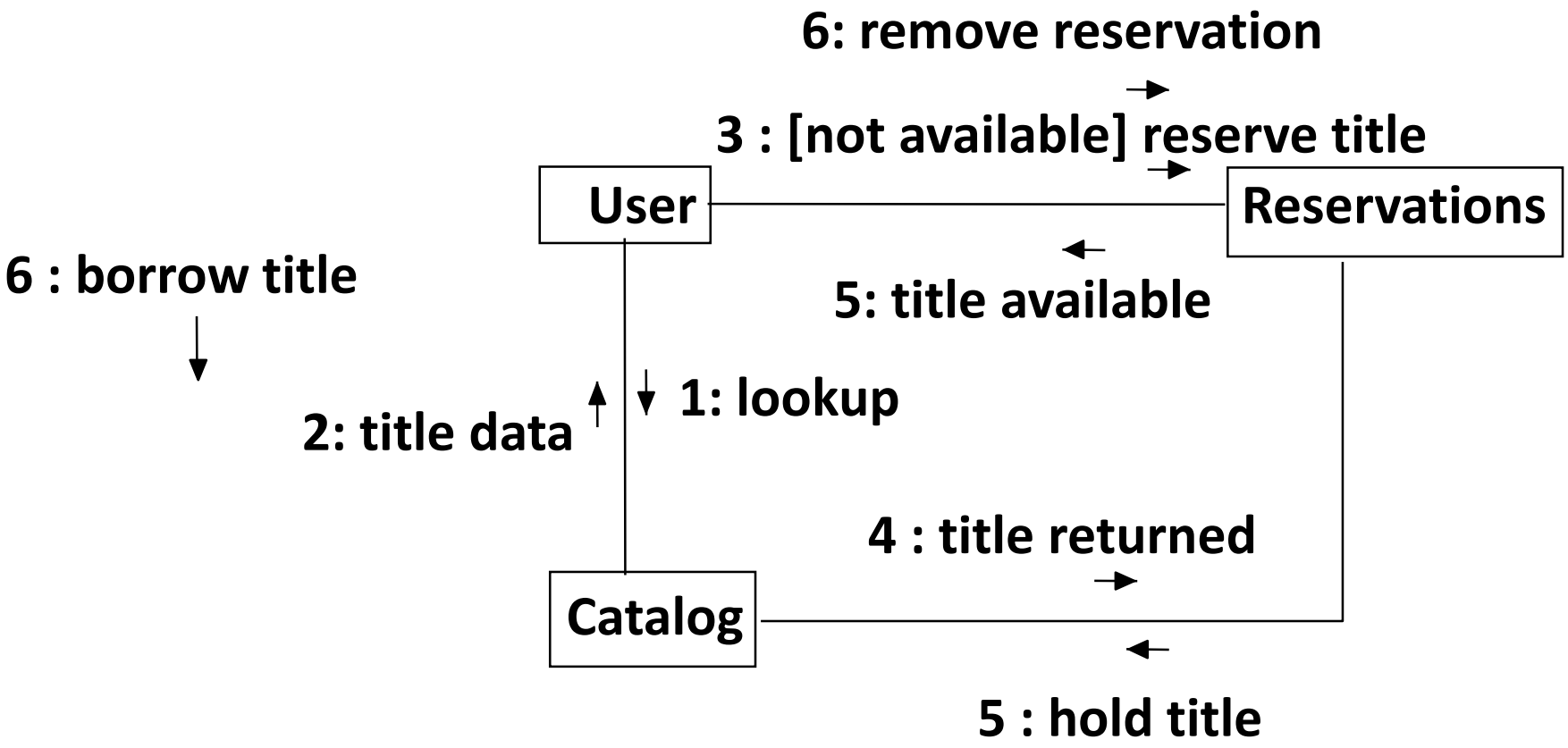
- if → (opt) [condition]
- if/else → (alt) [condition], separated by horizontal dashed line
- loop → (loop) [condition or items to loop over]



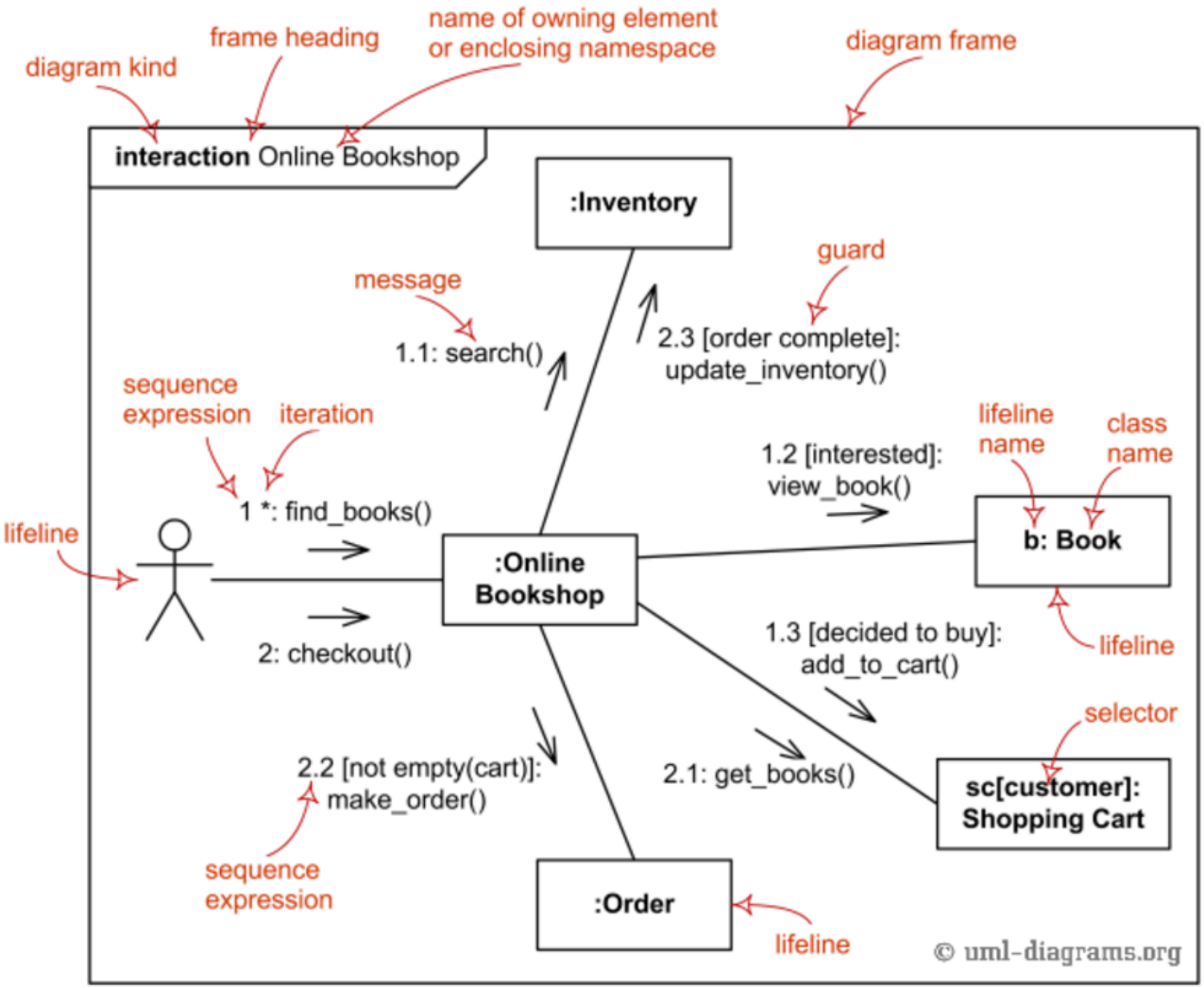
Collaboration Diagrams

- Collaboration diagrams are equivalent to sequence diagrams
  - All the features of sequence diagrams are equally applicable to collaboration diagrams
- Use a sequence diagram when the transfer of information is the focus of attention
- Use a collaboration diagram when concentrating on the classes

Collaboration Diagrams Example

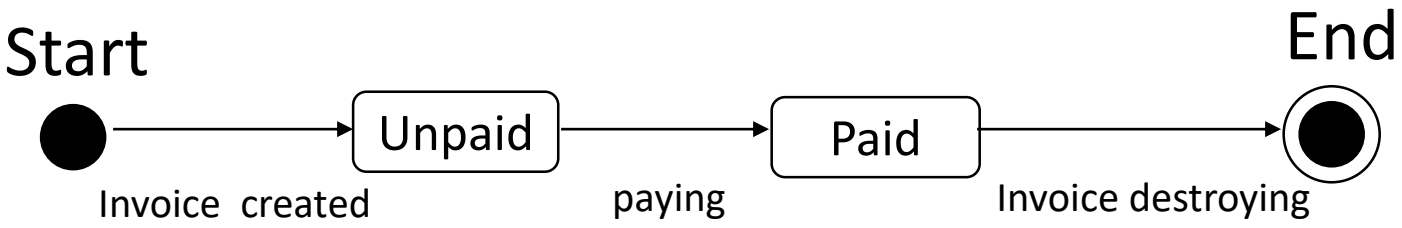


Collaboration Diagram Example



17.7 Statecharts

- 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



17.7 Statecharts

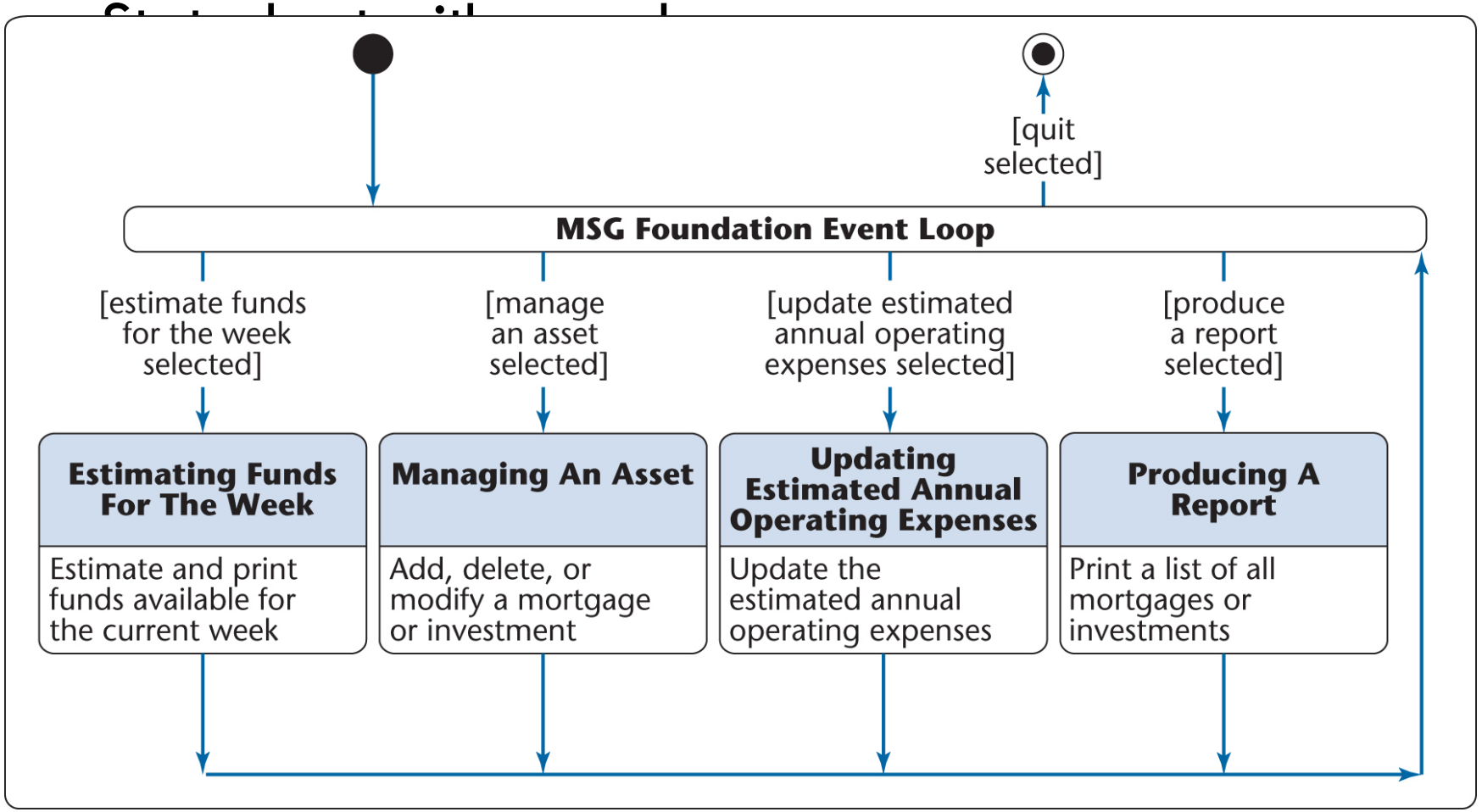


Figure 17.16

Statecharts (contd)

- An event also causes transitions between states

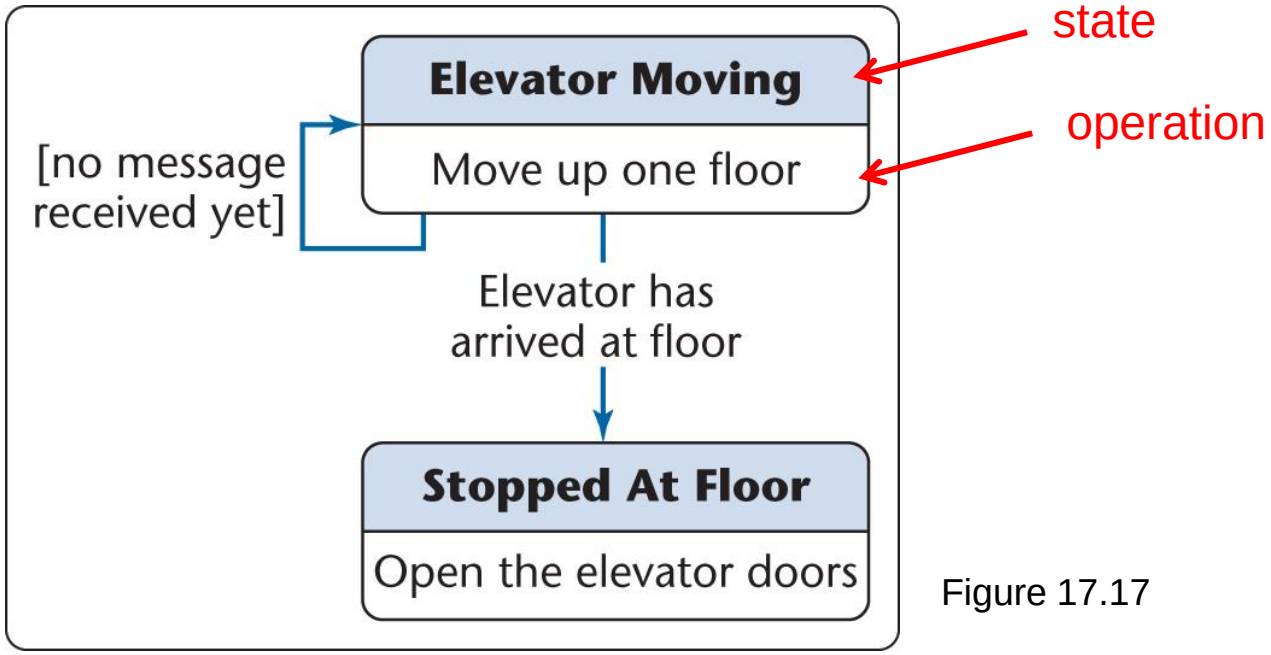
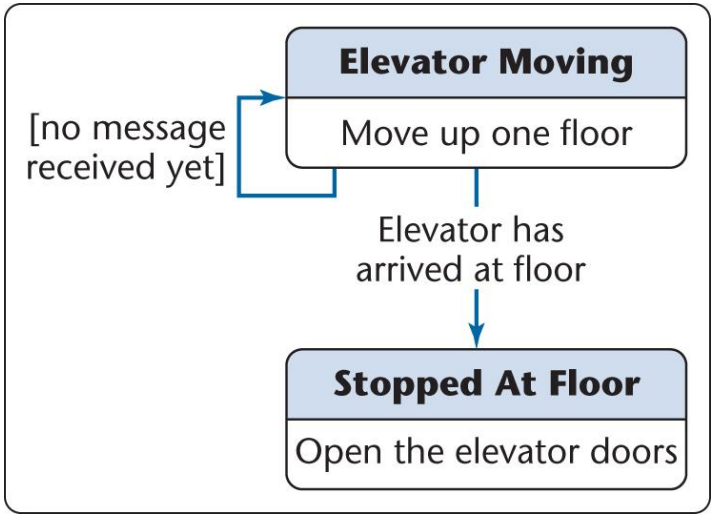


Figure 17.17

Statecharts (contd)

- The elevator is in state **Elevator Moving**
  - It performs operation
    - Move up one floor

    while guard [no message received yet] remains true, until it receives the message
    - Elevator has arrived at floor

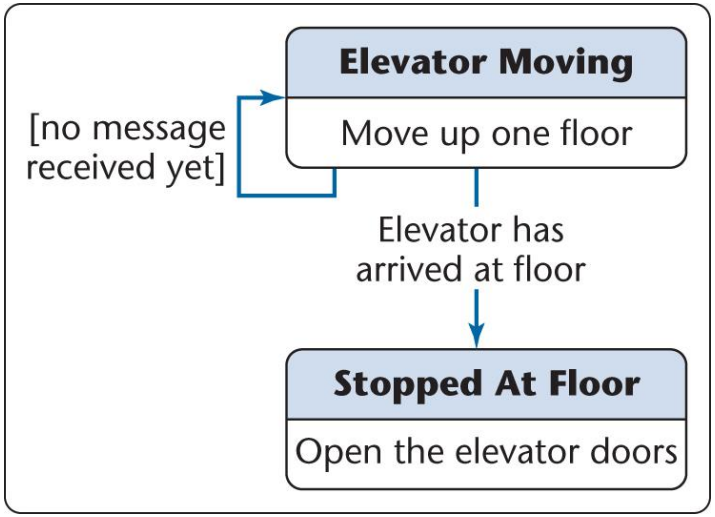


Statecharts (contd)

- Receipt of this message [event] causes the guard to be false

- It also enables a transition to state Stopped at Floor
  - In this state, activity
    - Open the elevator doors

    is performed



Statecharts (contd)

- The most general form of a transition label is
  - event [guard] / action
  - If
    - event

    has taken place and
    - [guard]

    is true, the transition occurs, and, while it is occurring,
    - action

    is performed

Statecharts (contd)

- Equivalent statement with the most general transition

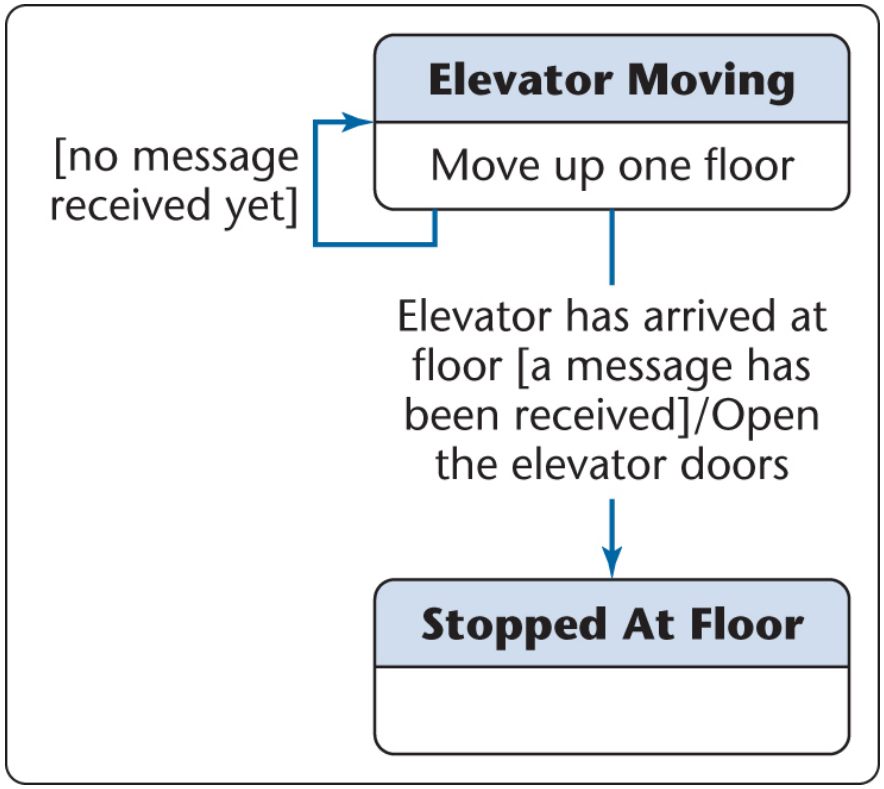


Figure 17.18

Statecharts (contd)

- There are two places where an action can be performed in a statechart
  - When a state is entered
    - Activity
  - As part of a transition
    - Action
- Technical difference:
  - An activity can take several seconds
  - An action takes places essentially instantaneously

Statecharts (contd)

- Superstates combine related states
  - States **A**, **B**, **C**, and **D** all have transitions to **Next State**
  - Combine them into superstate **ABCD Combined**
    - Now there is only one transition
    - The number of arrows is reduced from four to only one
  - States **A**, **B**, **C**, and **D** all still exist in their own right

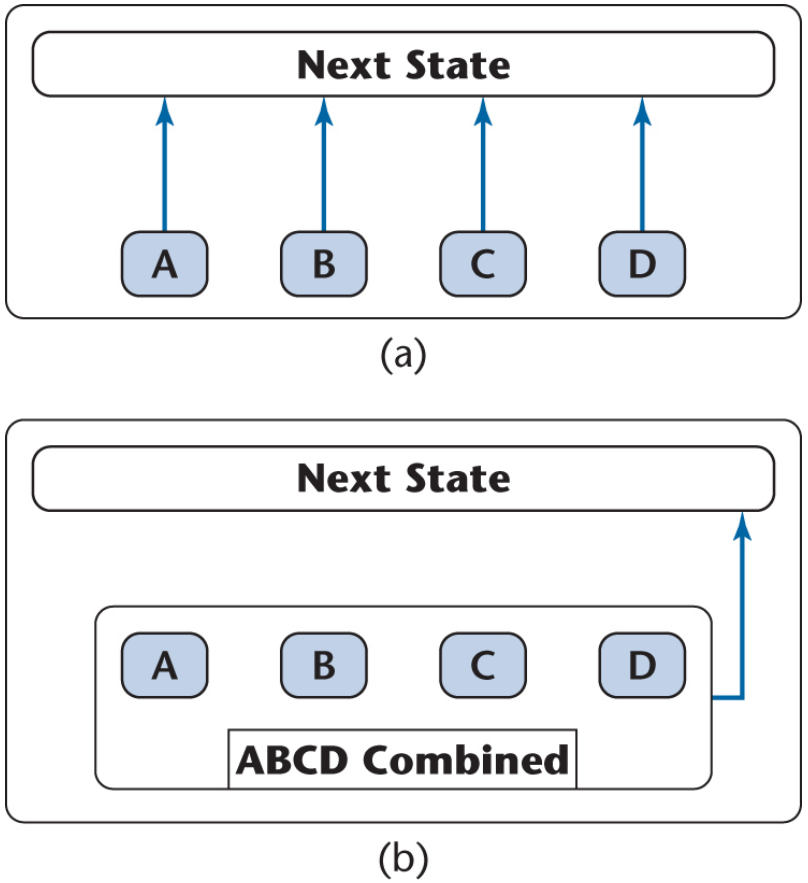


Figure 17.19

Statecharts (contd)

- Example: Four states are unified into **MSG Foundation Combined**

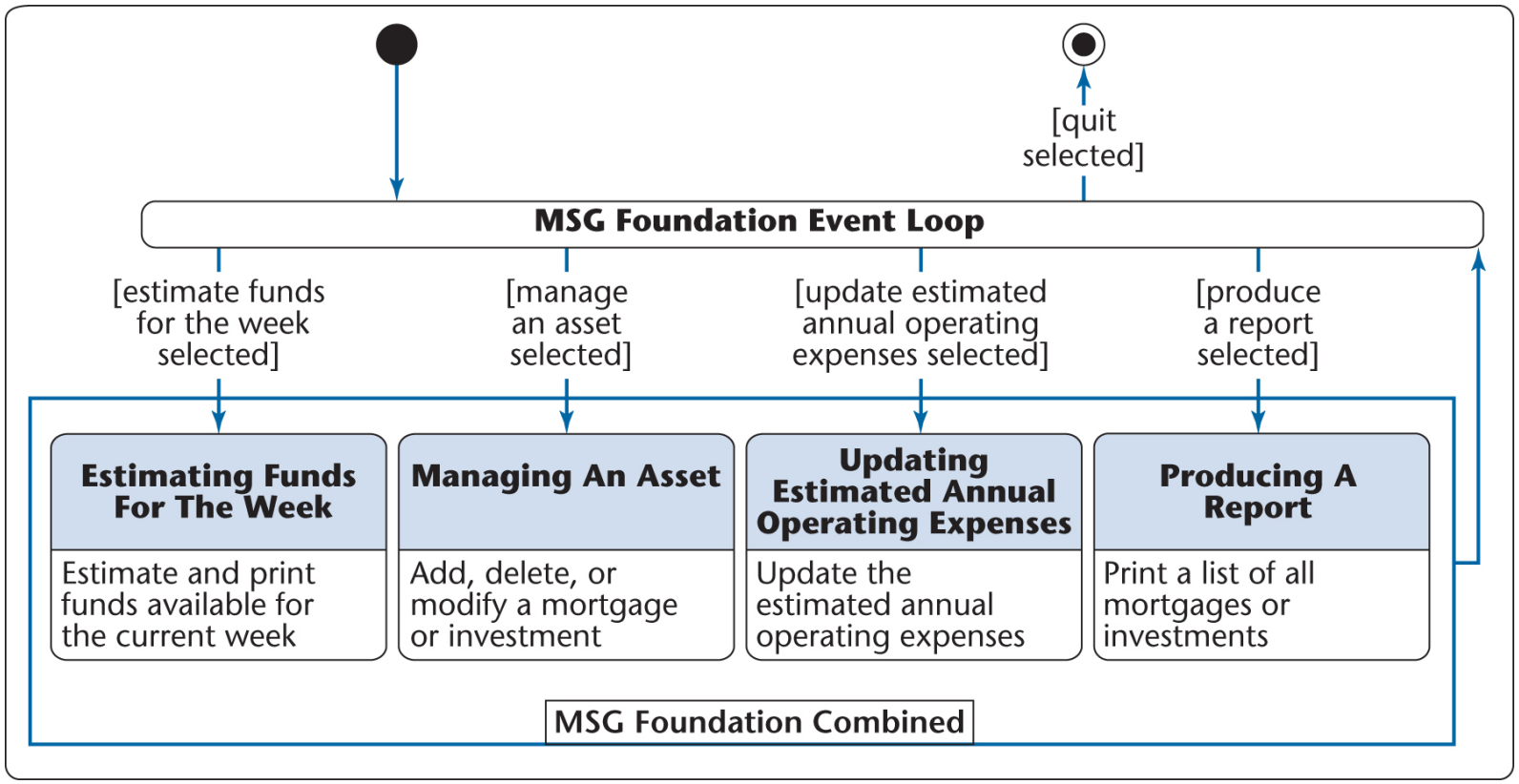


Figure 17.20

17.8 Activity Diagrams

- Activity diagrams show how various events are coordinated
  - Used when *activities are carried on in parallel*
- Example:
  - One diner orders chicken, the other fish
  - The waiter writes down their order, and hands it to the chef
  - The meal is served only when both dishes have been prepared

Activity Diagrams (contd)

- Example:

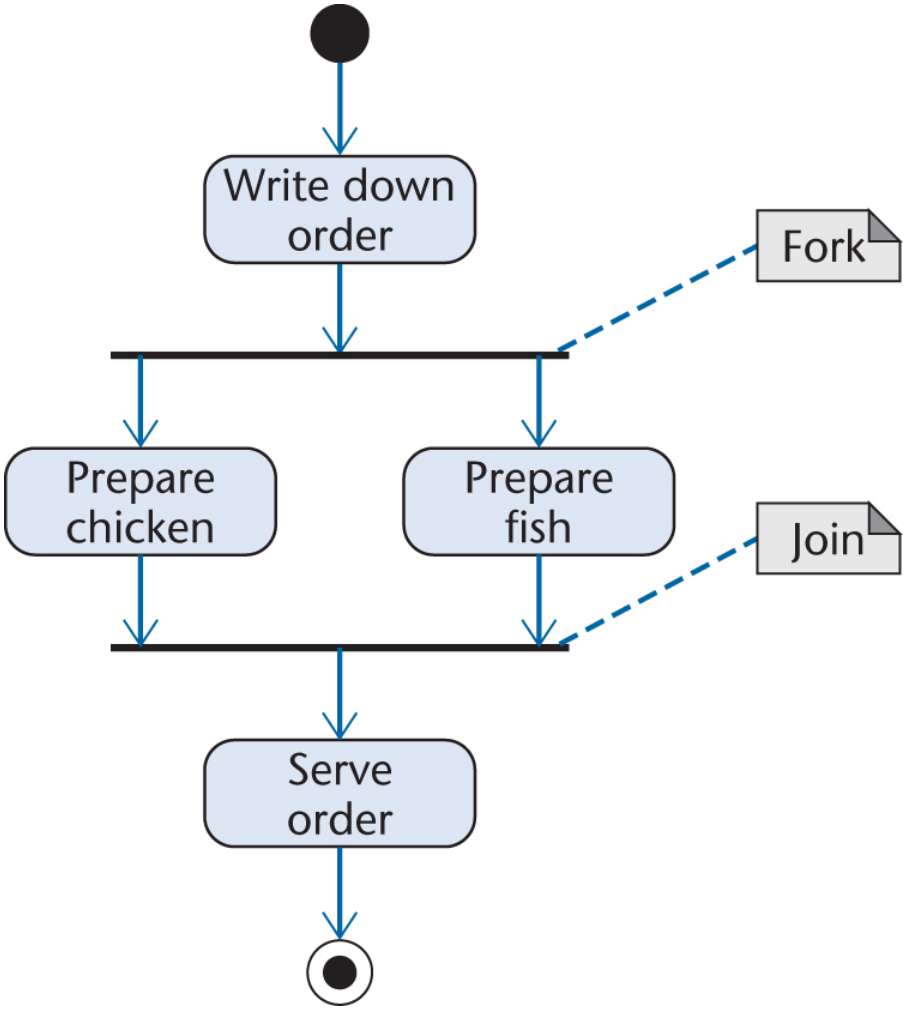


Figure 17.21

Activity Diagrams (contd)

- A *fork* has
  - One incoming transition, and
  - Many outgoing transitions, each of which starts an activity to be executed in parallel with the other activities

- A *join* has
  - Many incoming transitions, each of which lead from an activity executed in parallel with the other activities, and
  - One outgoing transition that is started when all the parallel activities have been completed

Activity Diagrams (contd)

- Example:
  - A company that assembles computers as specified by the customer

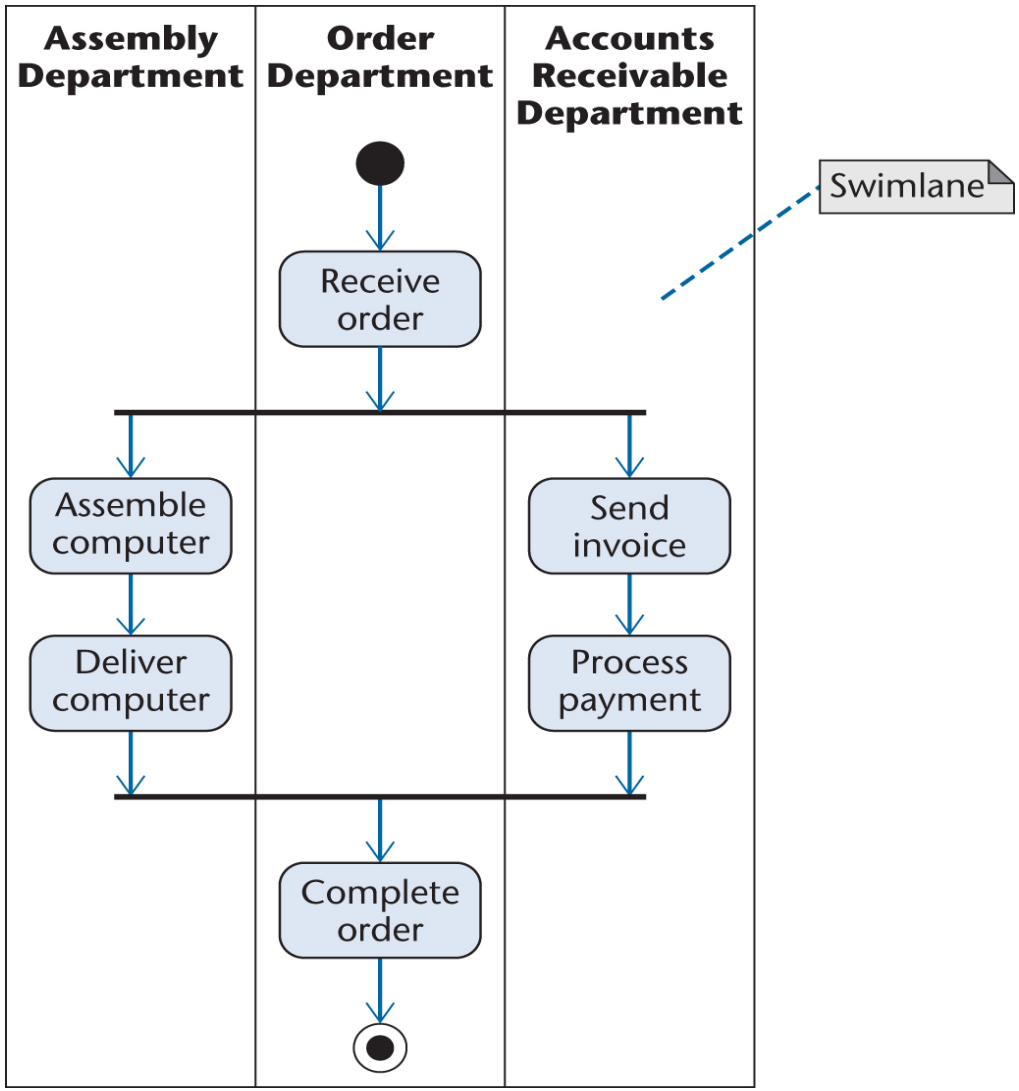
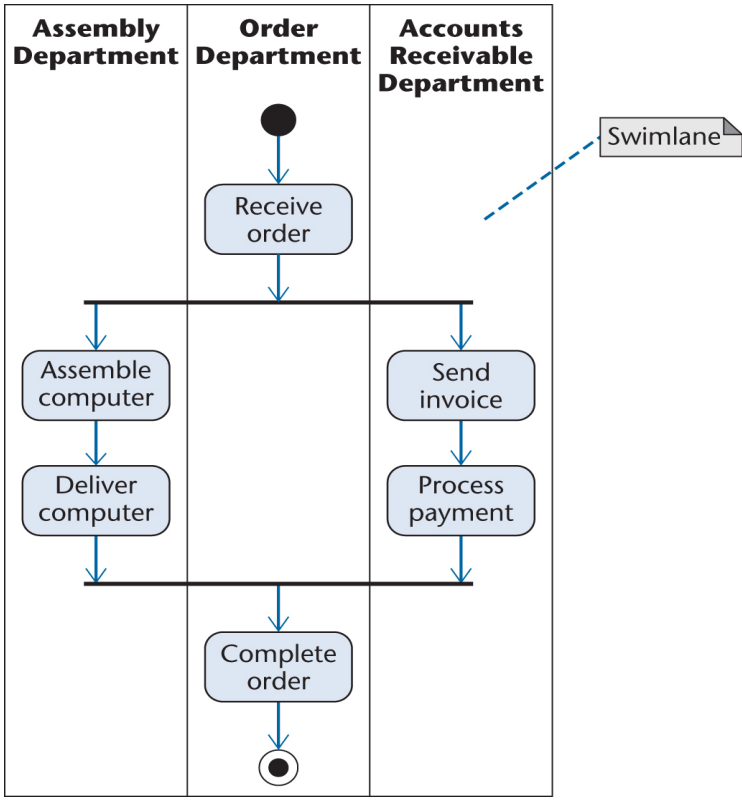


Figure 17.22

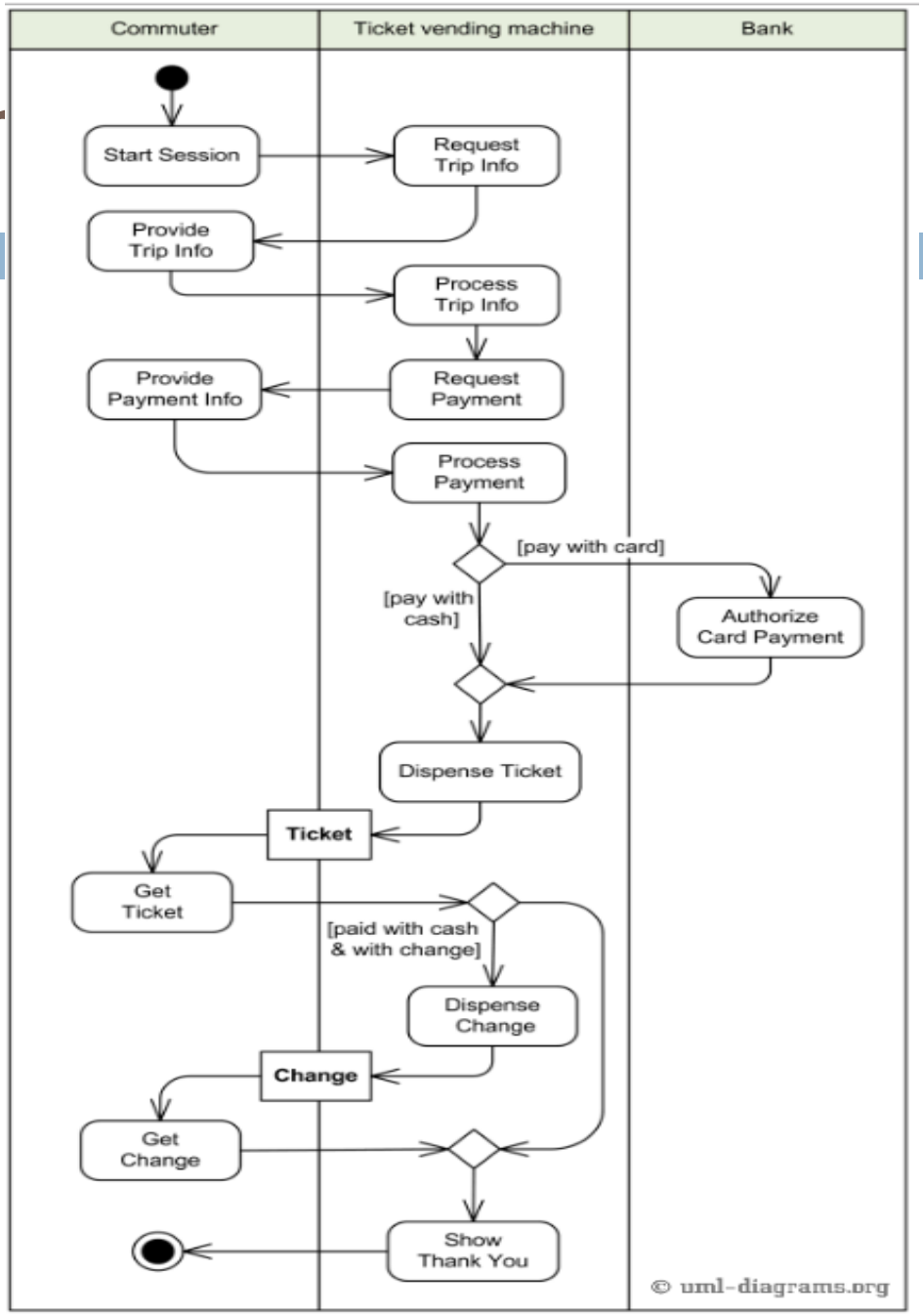
Activity Diagrams (contd)

- The three departments involved
  - Assembly Department
  - Order Department
  - Accounts Receivable Department

  are each in their own swimlane



Activity Diagram



17.9 Packages

- A large information system is decomposed into relatively independent packages
  - UML notation for a package

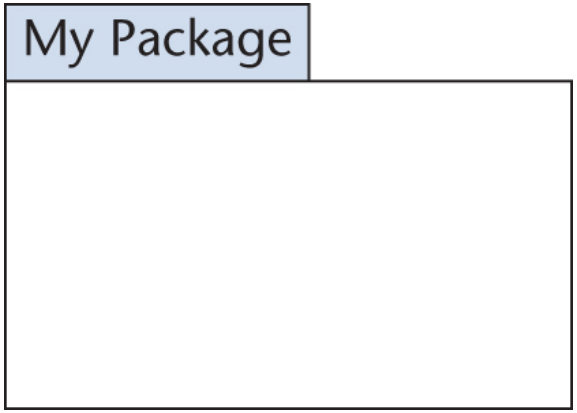


Figure 17.23

Packages (contd)

- A **package** is used to group elements, and provides a namespace for the grouped elements
  - A package is a namespace for its members, and may contain other packages

- **Owned members** of a package should all be package elements

Packages (contd)

- Example showing the contents of My Package

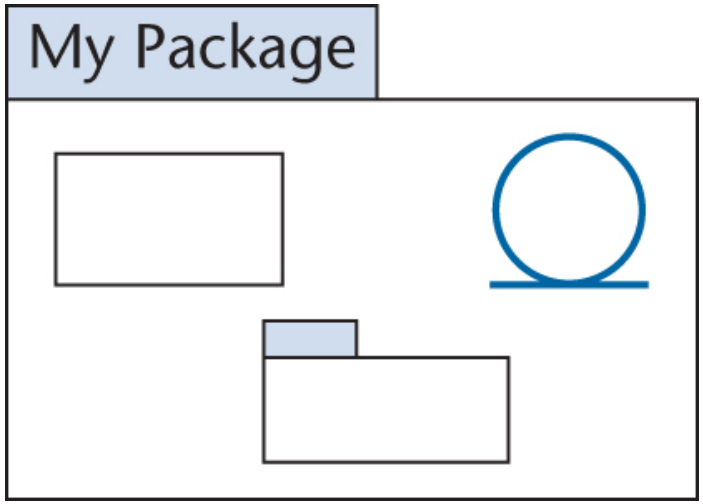
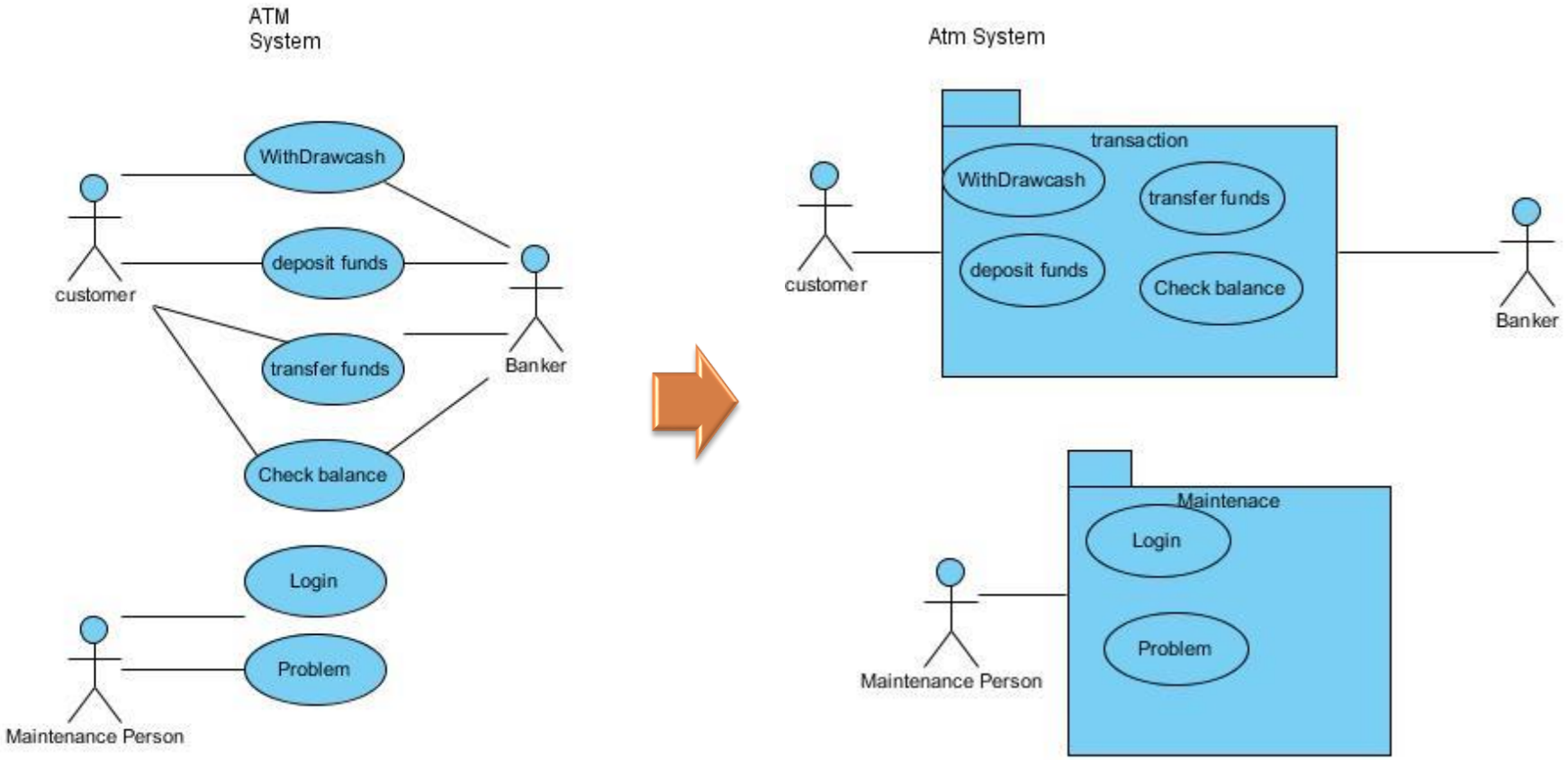


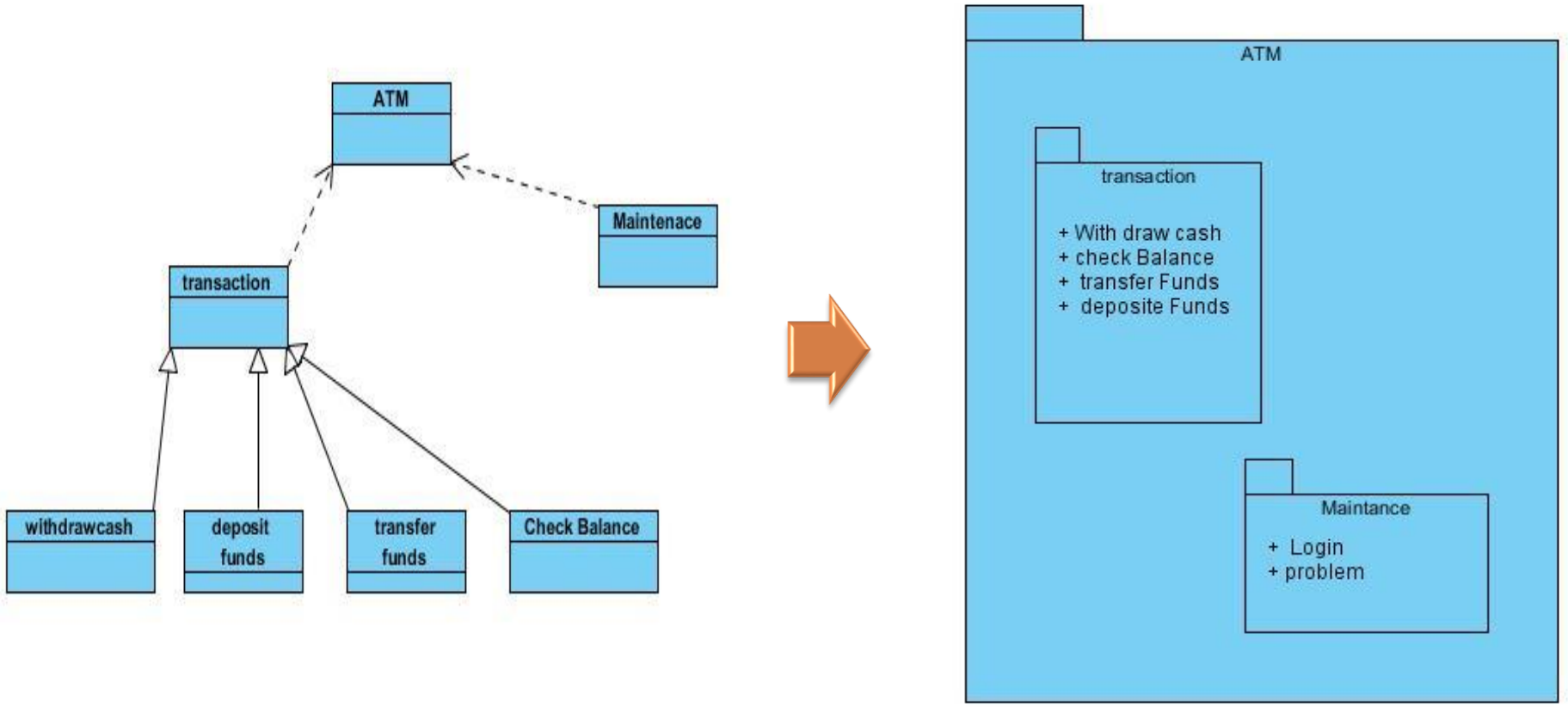
Figure 17.24

Packages Example



Use case Diagram → **Use case Package Diagram**

Packages Example (contd)



Class Diagram → **Class Package Diagram**

17.11 Deployment Diagrams

- A deployment diagram shows on which hardware component each software component is installed (or deployed)

- It also shows the communication links between the hardware components

Deployment Diagrams (contd)

- Example:

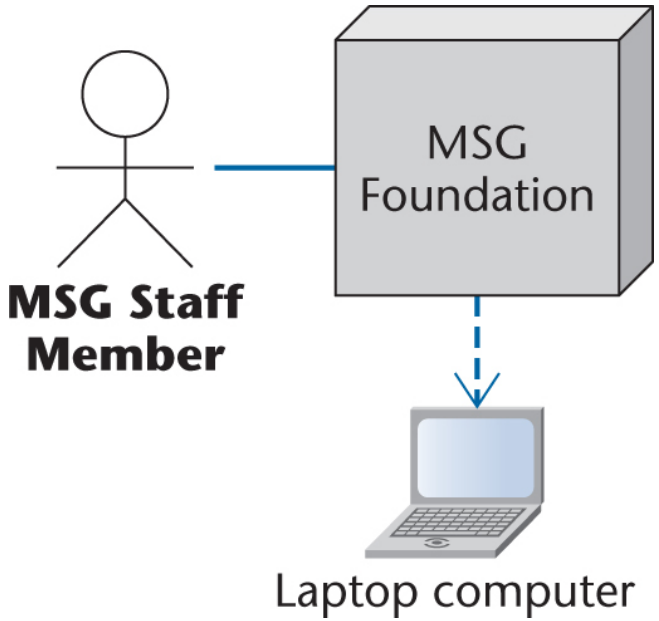
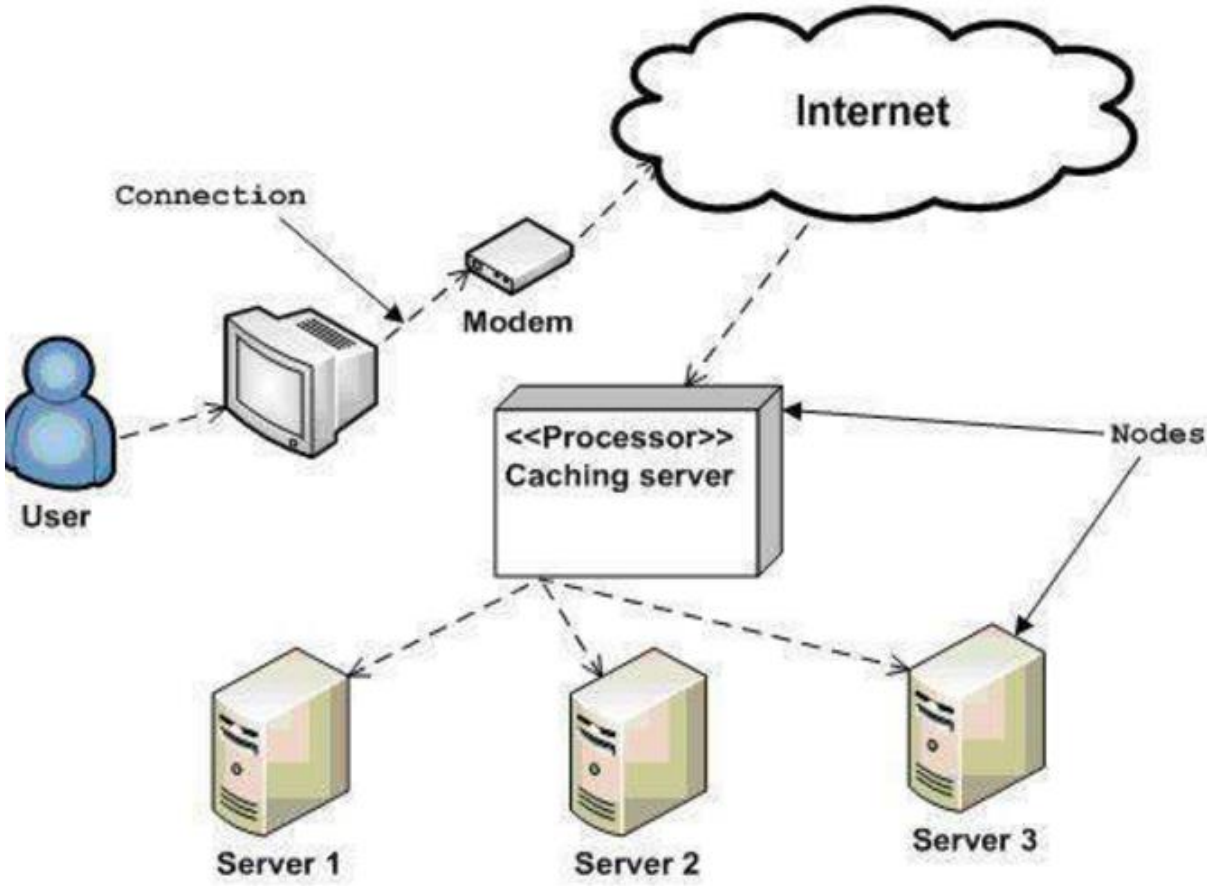


Figure 17.26

Deployment Diagram Example



17.12 Review of UML Diagrams

- A *use case* models the interaction between actors and the information system
- A *use-case diagram* is a single diagram that incorporates a number of use cases
- A *class diagram* is a model of the classes showing the static relationships between them
  - Including association and generalization

Review of UML Diagrams

- A *statechart* shows
  - States (specific values of attributes of objects),
  - Events that cause transitions between states (subject to guards), and
  - Actions taken by objects
- An *interaction diagram* (sequence diagram or collaboration diagram) shows how objects interact as messages are passed between them
- An *activity diagram* shows how events that occur at the same time are coordinated

17.13 UML and Iteration

- Every UML diagram consists of a small required part plus any number of options
  - Not every feature of UML is applicable to every information system
  - To perform iteration and incrementation, features have to be added stepwise to diagrams

- This is one of the many reasons why UML is so well suited to the Unified Process

UML Modeling Tools

- Rational Rose (www.rational.com) by IBM
- TogetherSoft Control Center, Borland (http://www.borland.com/together/index.html)
- ArgoUML (free software) (http://argouml.tigris.org/)
  OpenSource; written in java
- Others (http://www.objectsbydesign.com/tools/umltools_byCompany.html)