Recap - Lecture4

- Understanding USE CASE DIAGRAMS

Activity Diagram

- **Activity diagram** is basically a flowchart to represent the flow from one activity to another activity. The **activity** can be described as an **operation** of the system.

- **Activity diagrams** are used to describe business activities and software systems' functionality.

- **Activity diagrams** help people on the **business** and **development** sides of an organization come together.

Activity Diagram…

- The control flow is drawn from one operation to another.

- This flow can be sequential, branched (Fork), or concurrent (Join).

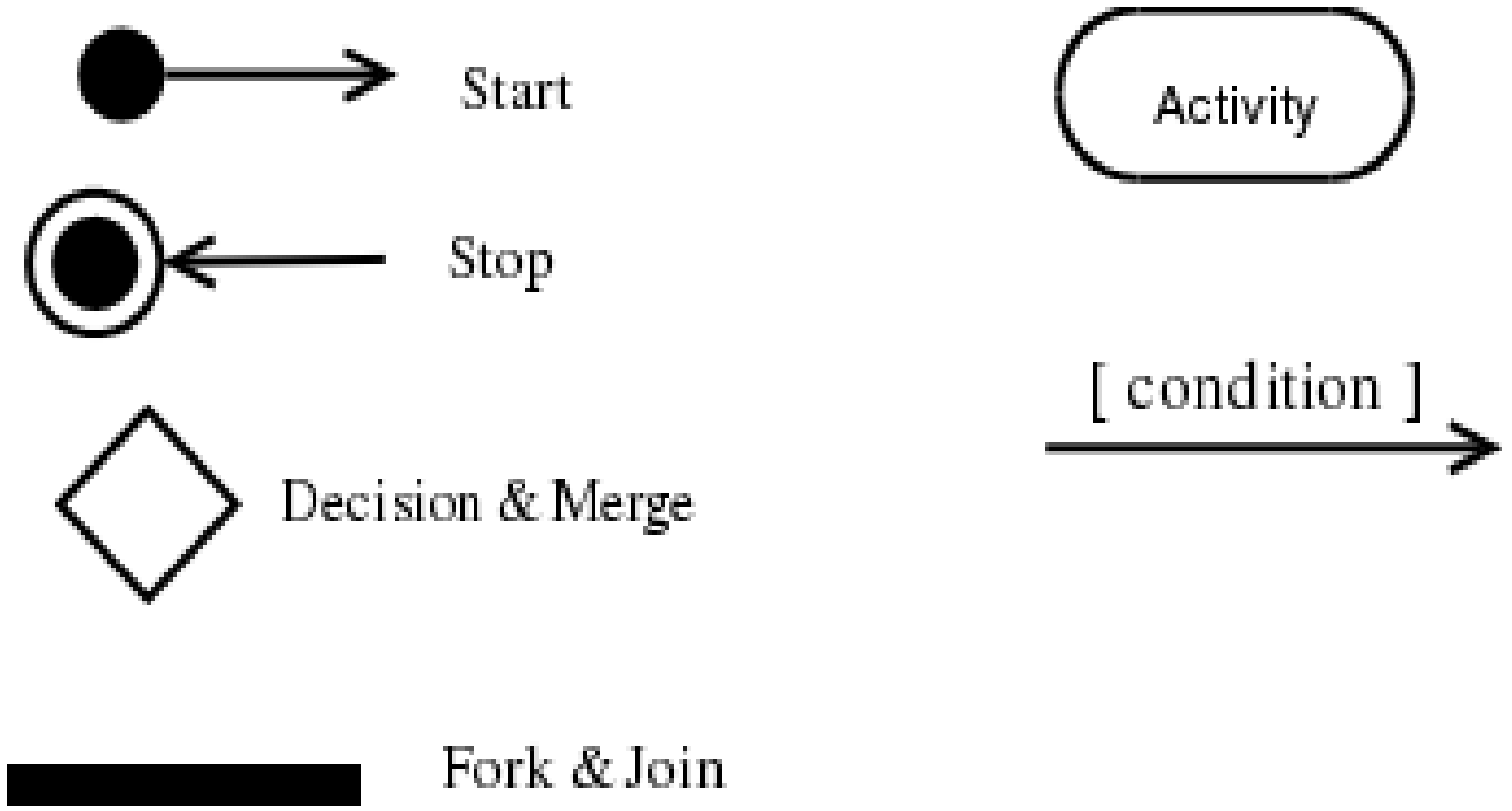
Why?

- Activity diagrams are needed for:-
  - Drawing the activity flow of a system.
  - Describe the sequence from one activity to another.
  - Describe the parallel, branched and concurrent flow of the system.

Drawing Activity Diagram

- To draw an activity diagram
  1. List all functions/activities performed by the system
  2. Show activities association with constraints and conditions

Symbols in UML Activity Diagrams



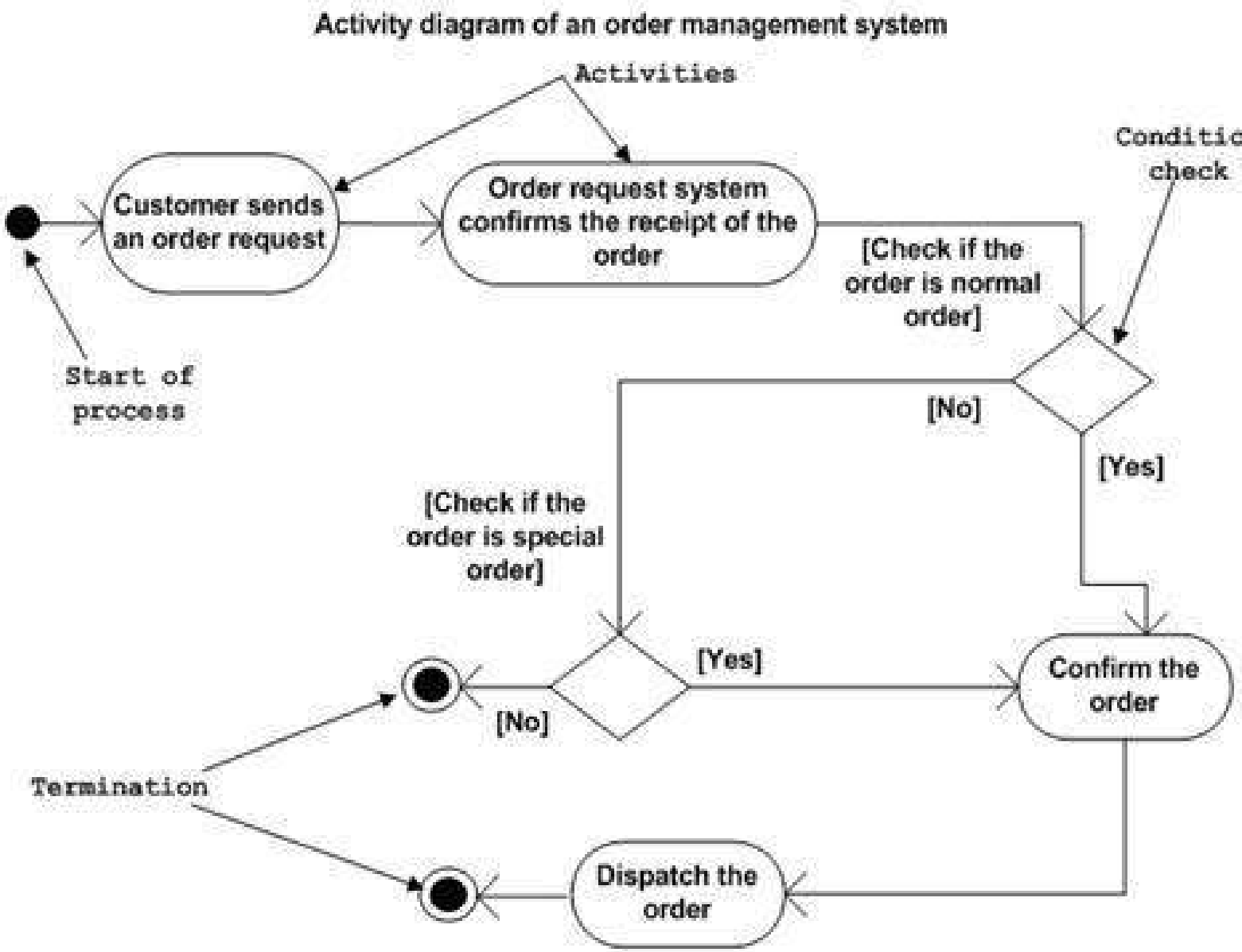
Example

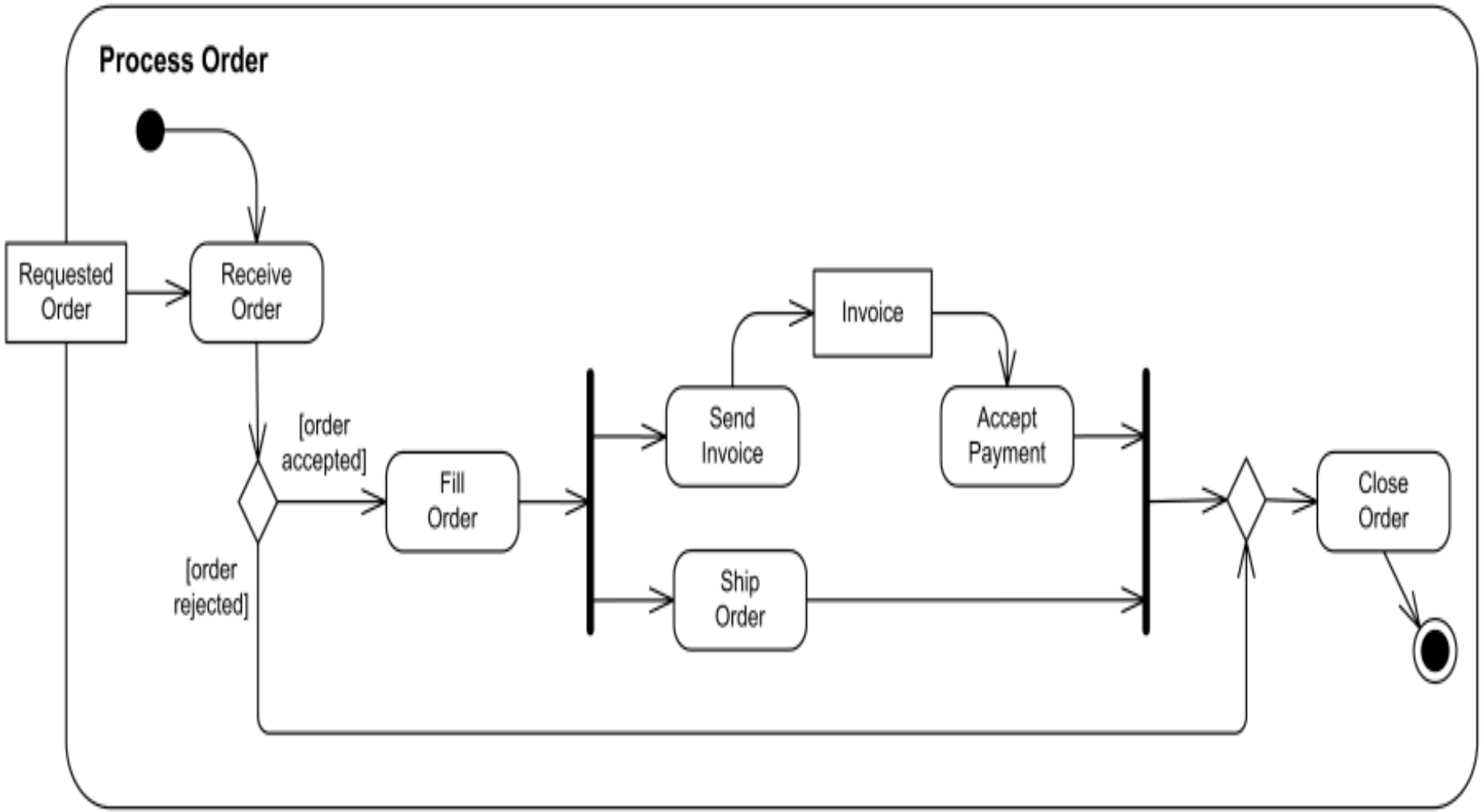
- Activity diagram for order management system

1. Four activities are identified which are associated with conditions
    i. Place order by the customer
    ii. Receipt of the order by the system
    iii. Confirm the order
    iv. Dispatch the order

Example...

2. After placing the order, condition checks are performed to check if it is **normal** or **special order**. After the type of order is identified, it **gets confirmed** then **dispatched** and the process terminates.


Activity diagram of an order management system
Activities
Condition check
Customer sends an order request
Order request system confirms the receipt of the order
[Check if the order is normal order]
Start of process
[No]
[Yes]
[Check if the order is special order]
[Yes]
[No]
Confirm the order
Termination
Dispatch the order


Process Order
Requested Order
Receive Order
[order accepted]
[order rejected]
Fill Order
Send Invoice
Invoice
Accept Payment
Ship Order
Close Order

SUMMARY

- Activity diagram is suitable for modeling activity flow of the system.
- An application can have multiple systems. Activity diagram also captures these systems and describes the flow from one system to another.
- Activity diagram gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person.
- The diagram has more impact on business understanding rather than on implementation details.

Structural Modeling

LECTURE 5

Key Ideas

- A structural or conceptual model describes the structure of the data that supports the business processes in an organization..

- The structure of data used in the system is represented through *Class-responsibility-collaboration (**CRC**) **cards** , class diagrams*, *and object diagrams*.

STRUCTURAL MODELS

Purpose of Structural Models

- Reduce the “semantic/logic gap” between the real world and the world of software
- Create a vocabulary for analysts and users
- Represent things, ideas, and concepts of importance in the application domain

Classes and Objects

- **Class** – **Template** to define specific instances or objects
- **Object** – **Instantiation** of a class
- **Attributes** – **Describes** the object
- **Behaviours** – specify what an object **can do**

Basic Characteristics of Object Oriented Systems

- Classes and Objects
- Methods and Messages
- Encapsulation and Information Hiding
- Inheritance
- Polymorphism

Classes and Objects

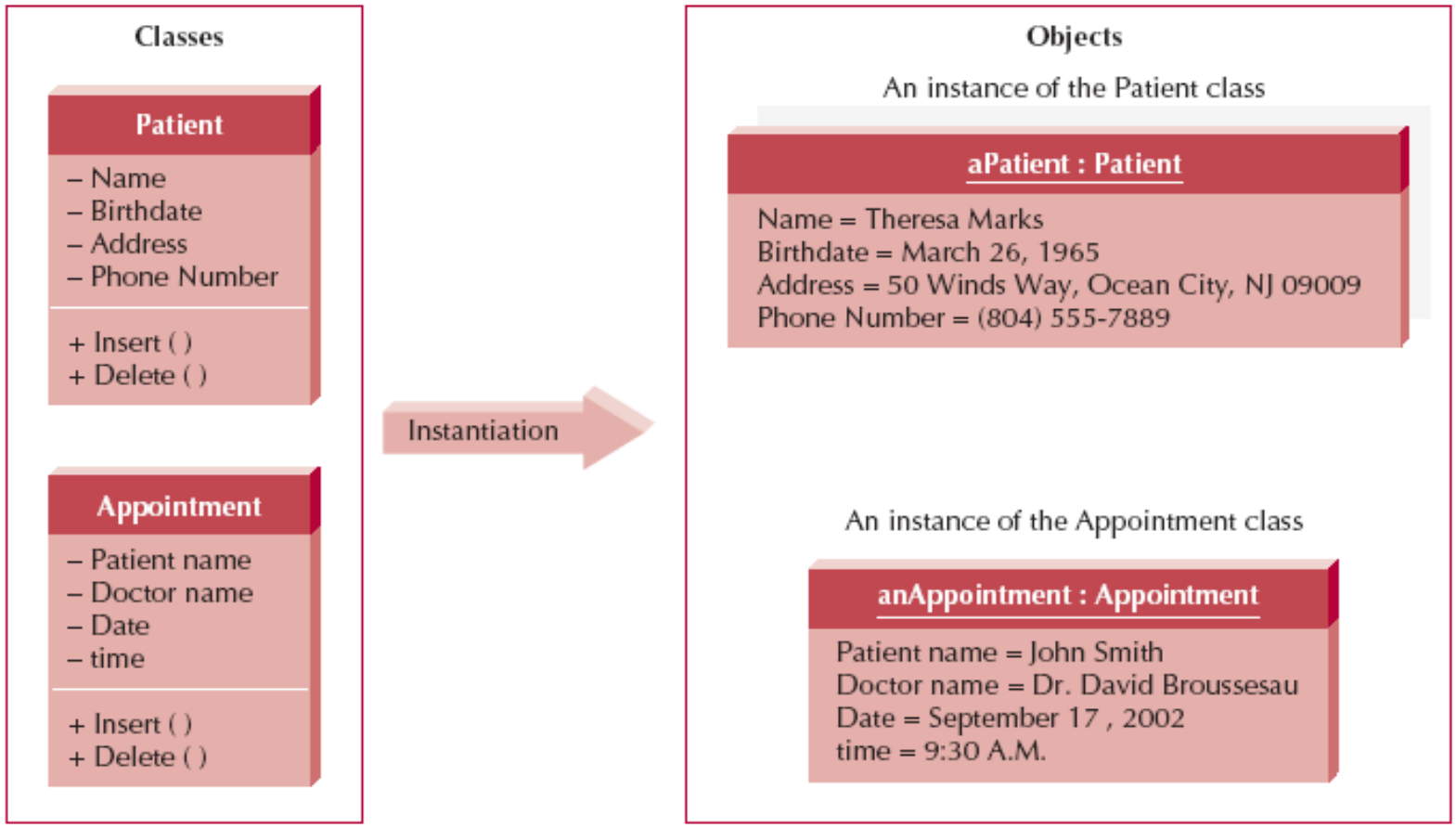


FIGURE 2-1 Classes and Objects

Class Diagram

- Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a **structural diagram.**

Class Diagram…

The purpose of the class diagram : –

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

How to Draw a Class Diagram?

The following points should be remembered while drawing a class diagram −

- The name of the class diagram should be meaningful to describe the aspect of the system.

- Each element and their relationships should be identified in advance.

- Responsibility (attributes and methods) of each class should be clearly identified

How to Draw a Class Diagram?...

- Use notes whenever required to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer/coder.

- Finally, before making the final version, the diagram should be drawn on plain paper and reworked as many times as possible to make it correct.

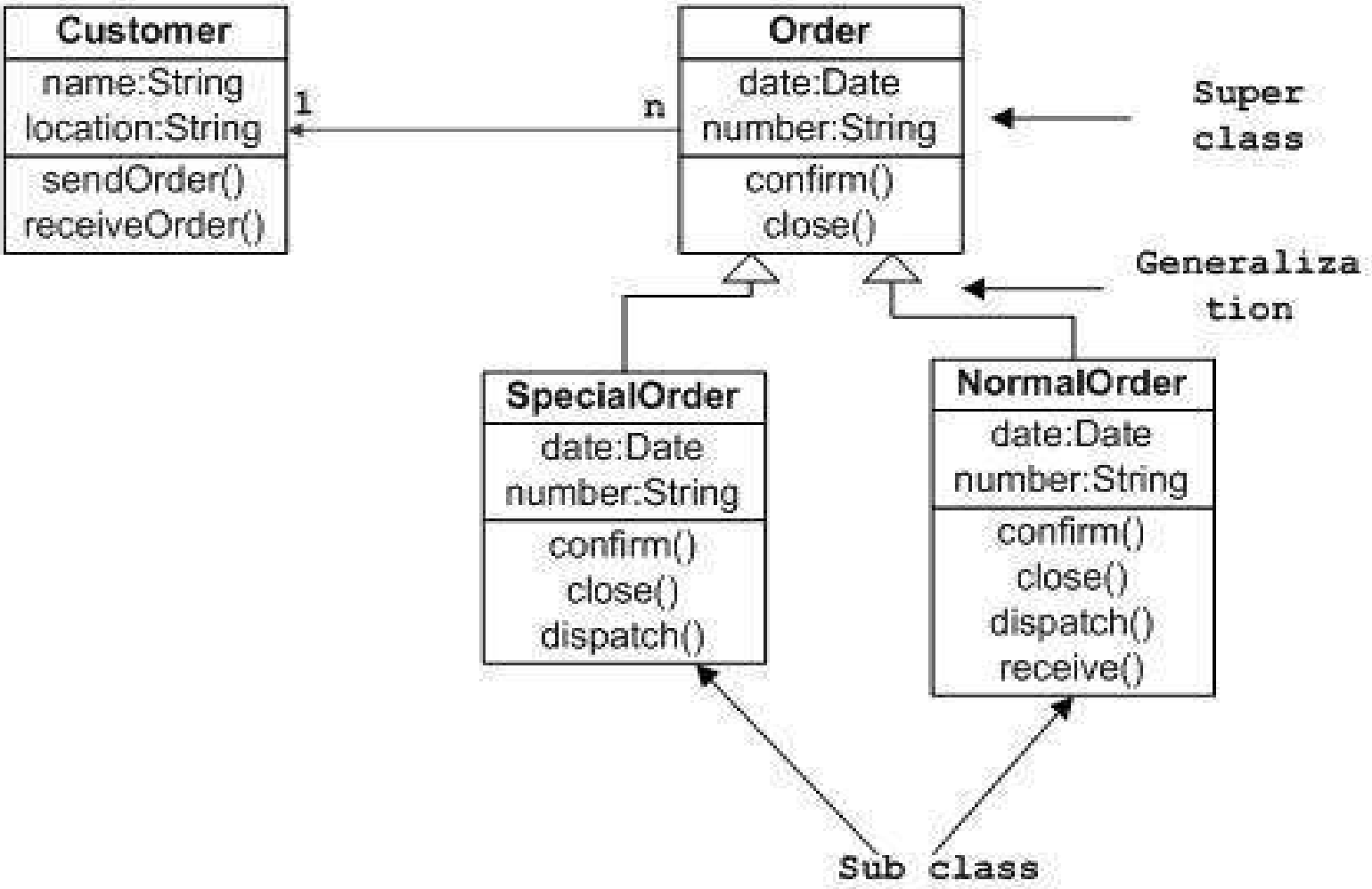
Class Diagram

- Example of an Order System of an application. It describes a particular aspect of the entire application.
- **First of all**, Order and Customer are identified as the two elements of the system.
- They have a **one-to-many relationship** because a customer can have multiple orders.
- **Order class** is an **abstract class** and it has two **concrete classes** (inheritance relationship) **SpecialOrder** and **NormalOrder.**

Class Diagram…

- The two inherited classes have all the properties as the Order class. In addition, they have additional functions like **dispatch ()** and **receive ().**
- See sample class diagram.

Sample Class Diagram

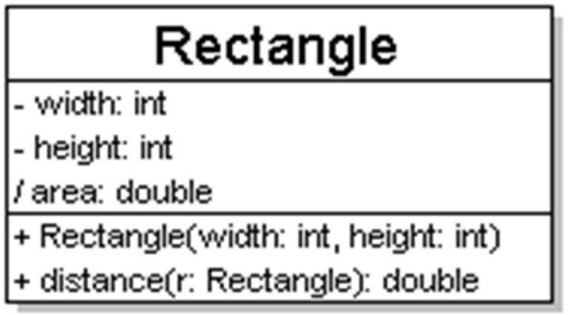


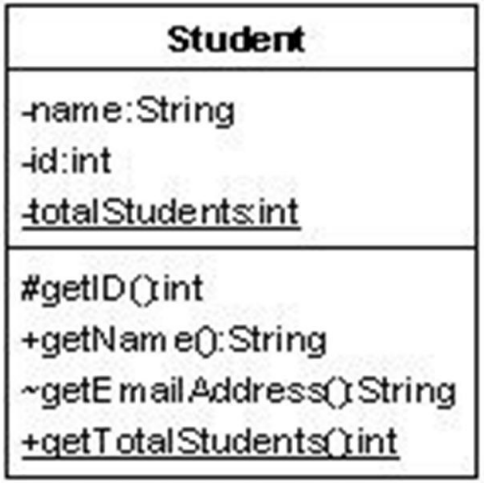
UML class diagrams

- **UML class diagram**: a picture of
  - the classes in an OO system
  - their fields and methods
  - connections between the classes
    - that interact or inherit from each other
- Not represented in a UML class diagram:
  - details of how the classes interact with each other
  - algorithmic details; how a particular behavior is implemented

Diagram of one class

- class name in top of box
  - write <<interface>> 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
    - but don't omit any methods from an interface! Because it gives operations for other classes to implement.
  - should not include inherited methods

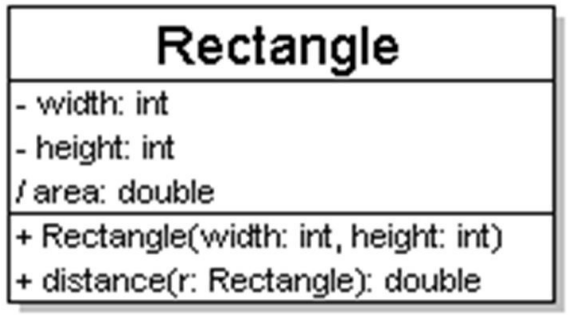


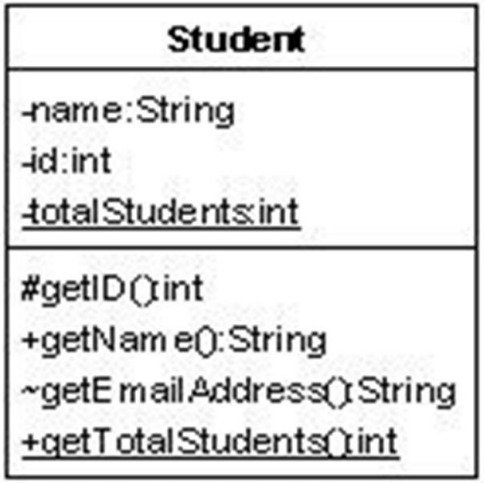


Class attributes (= fields)

- attributes (fields, instance variables)
  - *visibility name* : *type* [*count*] = *default_value*
  - visibility: + public
    # protected
    - private
    ~ package (default)
    / derived
  - underline static attributes (these are non changing attributes)
  - **derived attribute**: not stored, but can be computed from other attribute values
  - attribute example:

- balance : double = 0.00

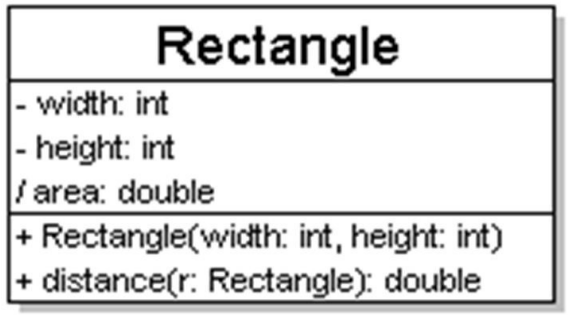


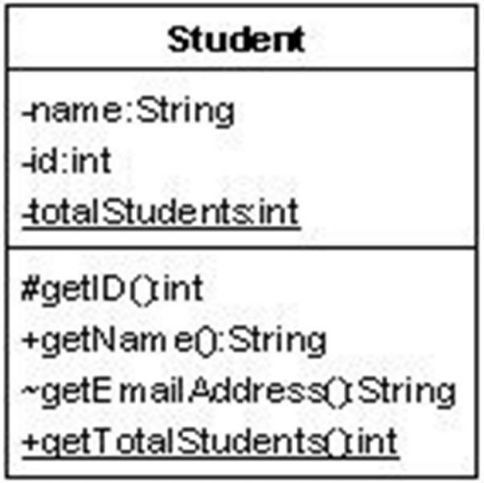


Class operations / methods

- operations / methods
  - *visibility name* (*parameters*) : *return_type*
  - visibility: + public
    # protected
    - private
    ~ package (default)
  - underline static methods
  - parameter types listed as (name: type)
  - method example:

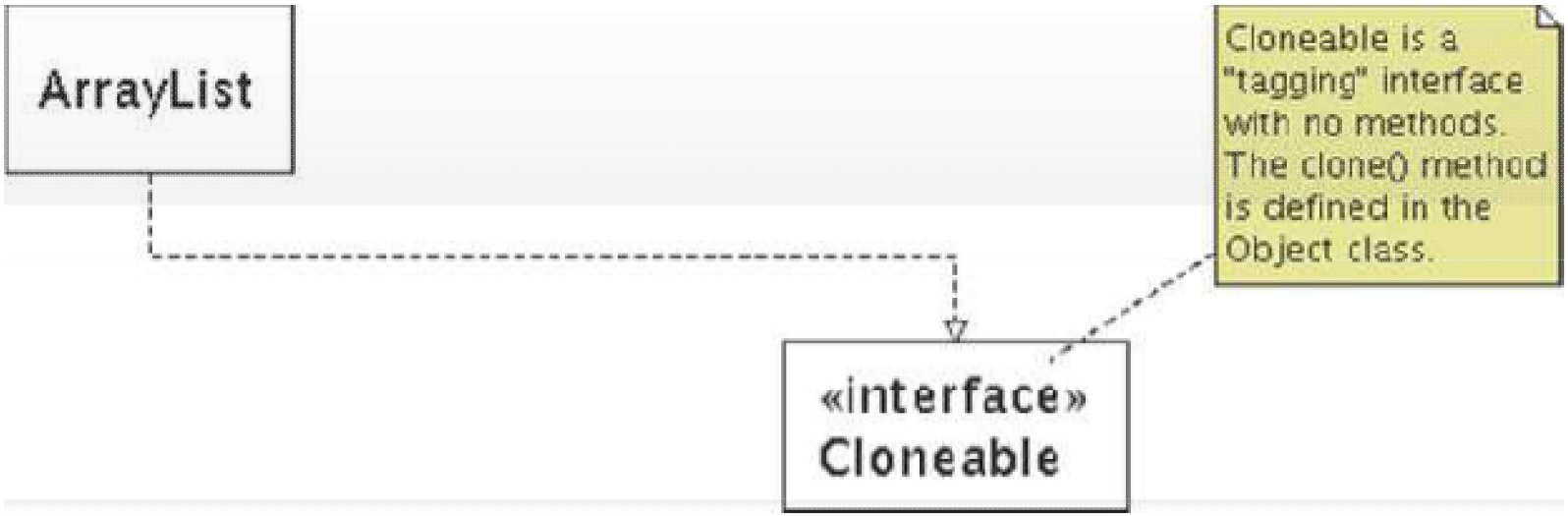
    + distance(p1: Point, p2: Point): double





Comments

- represented as a folded note, attached to the appropriate class/method/etc by a dashed line



Relationships between classes

- **generalization**: an inheritance relationship
  - inheritance between classes
  - interface implementation

- **association**: a usage relationship
  - dependency
  - aggregation
  - composition

Generalization (inheritance) relationships

- hierarchies drawn top-down
- arrows point upward to parent
- line/arrow styles indicate whether parent is a(n):
  - class:

solid line, black arrow

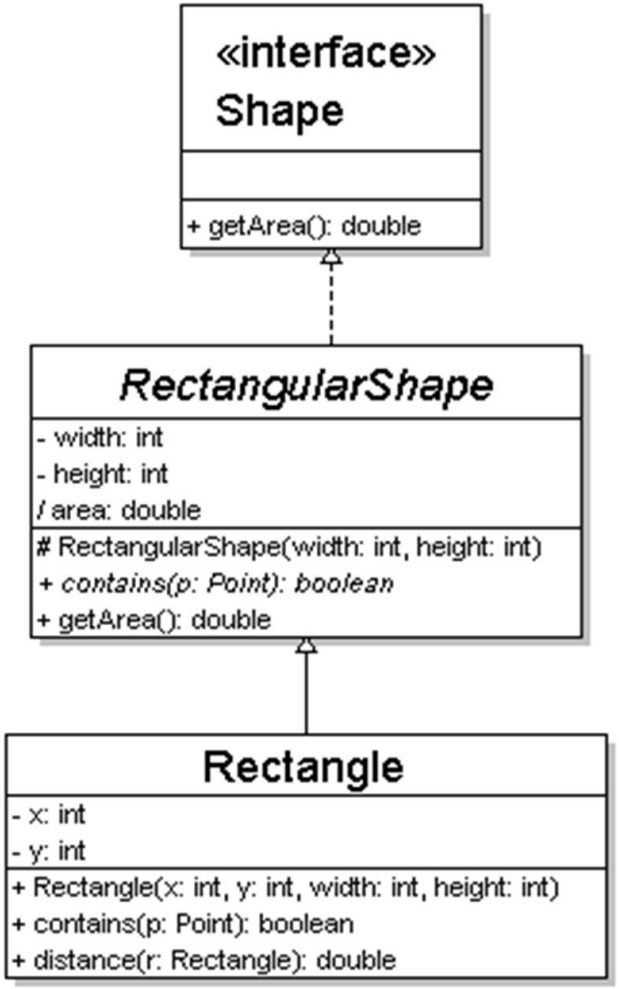
  - abstract class:

solid line, white arrow

  - interface:

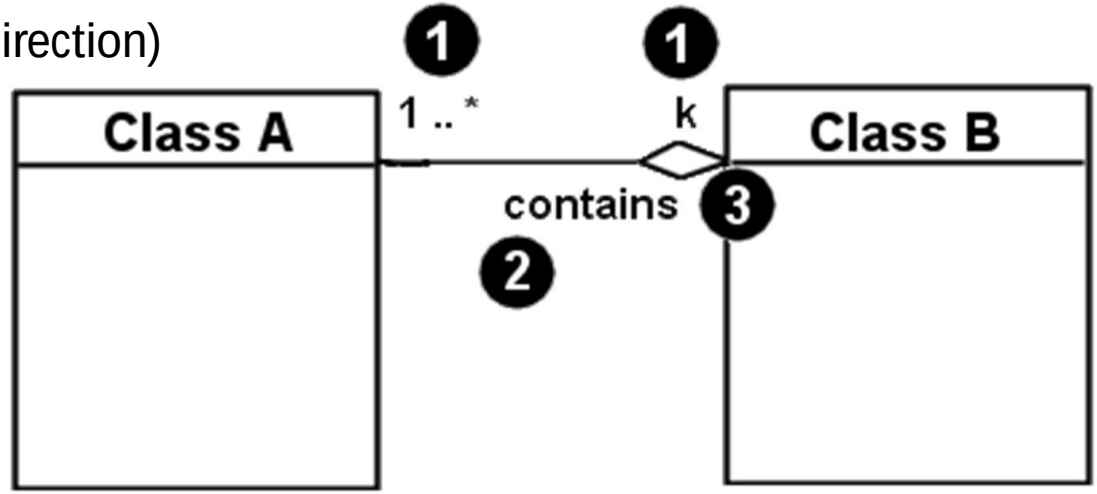
dashed line, white arrow

(**interfaces** are model elements that define sets of operations that other model elements, such as **classes**, or components must implement)



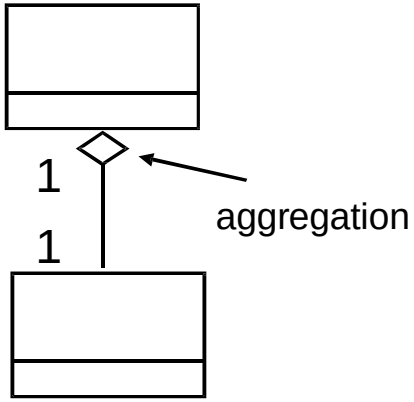
Associational relationships

- 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 (also written as "3..")
  2. name (what relationship the objects have)
  3. navigability (direction)

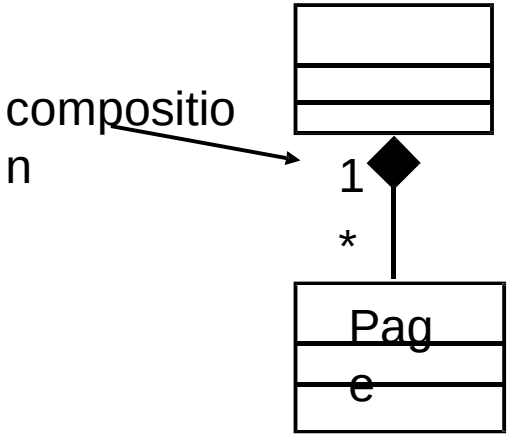


Association types

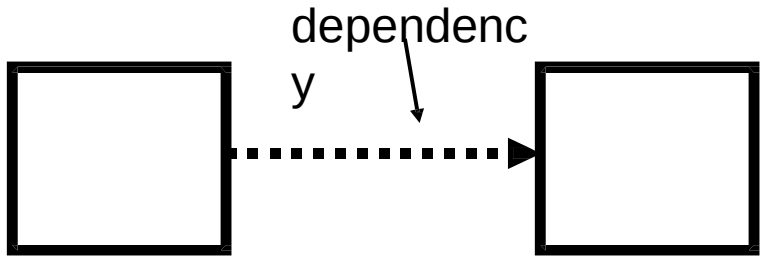
- **aggregation**: "is part of"
  - symbolized by a clear white diamond

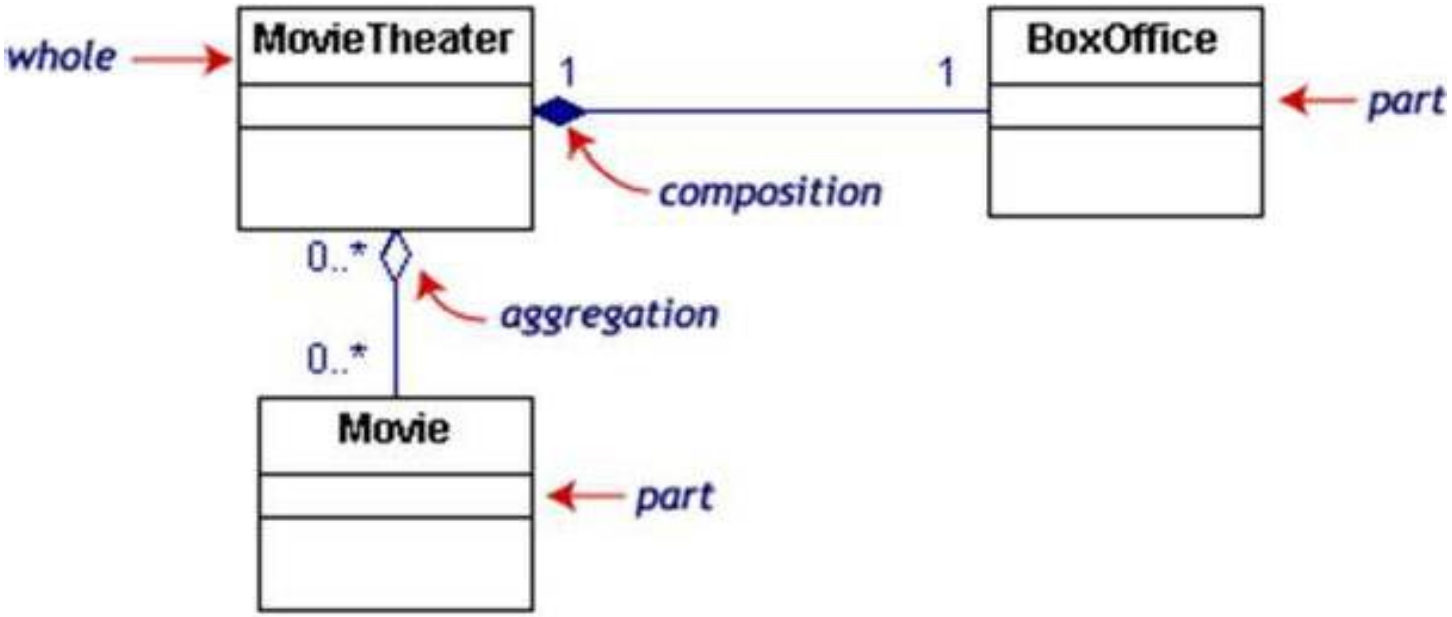


- **composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond

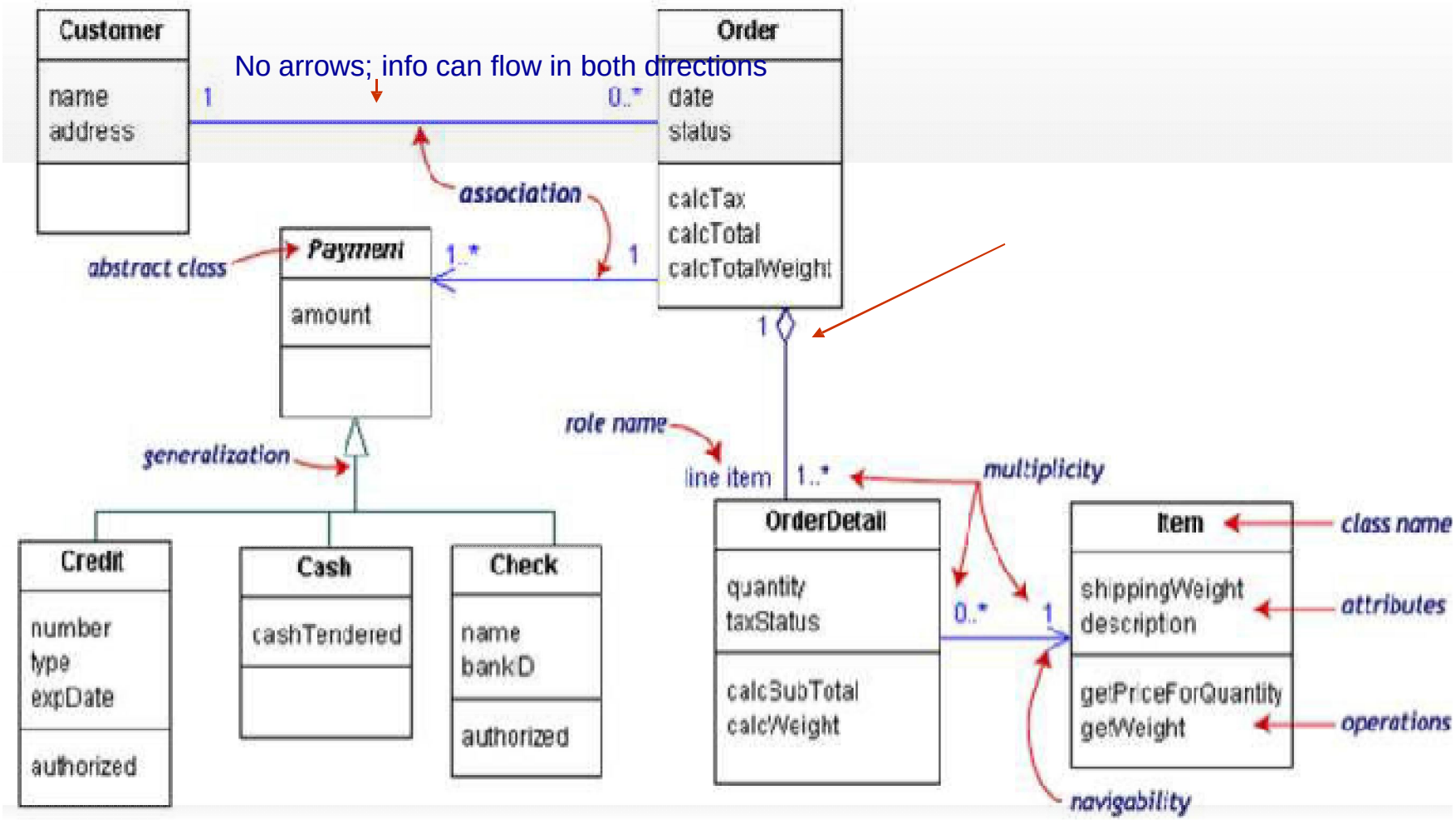


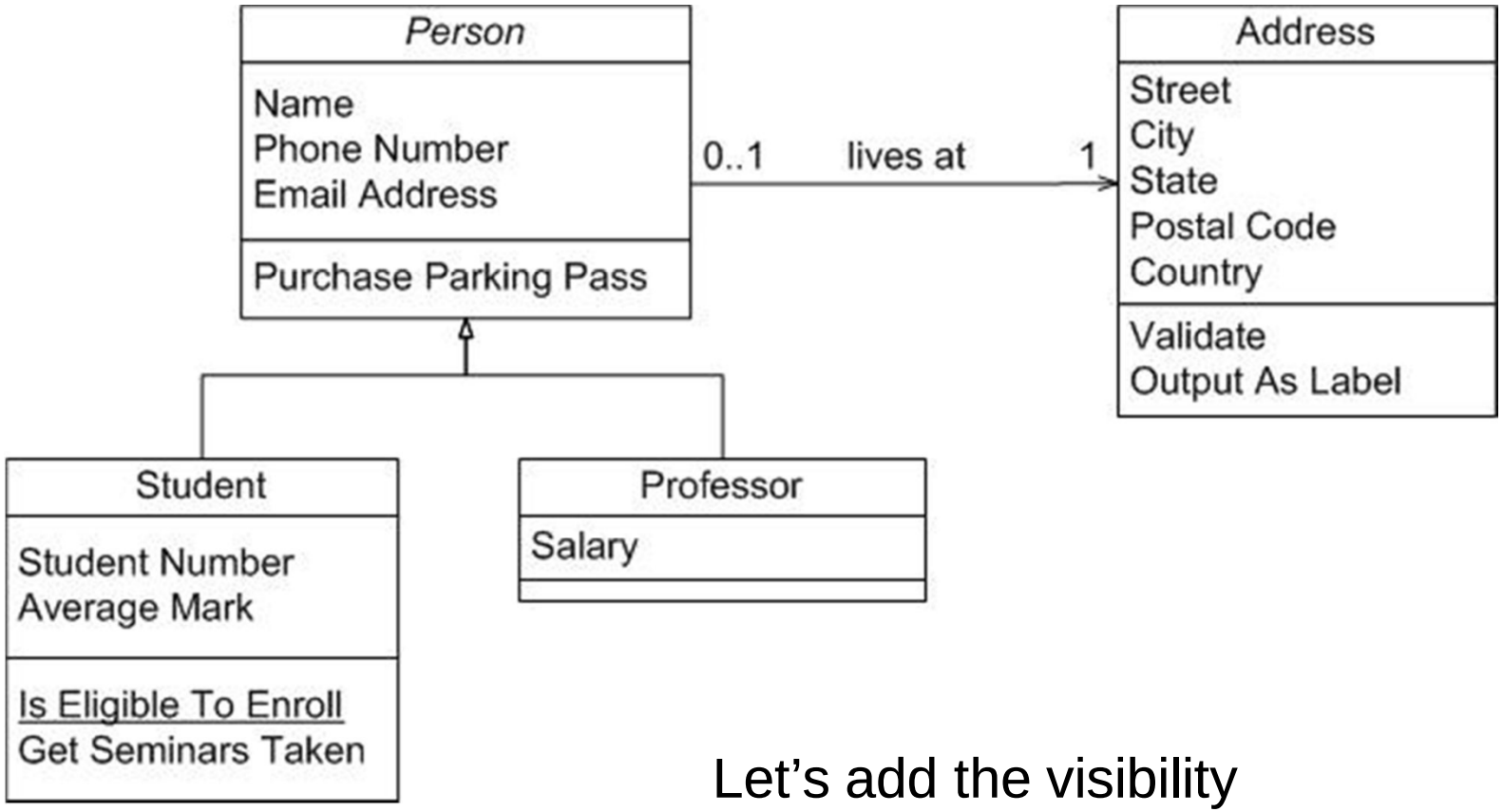
- **dependency**: "uses temporarily"
  - symbolized by dotted line
  - often is an implementation detail





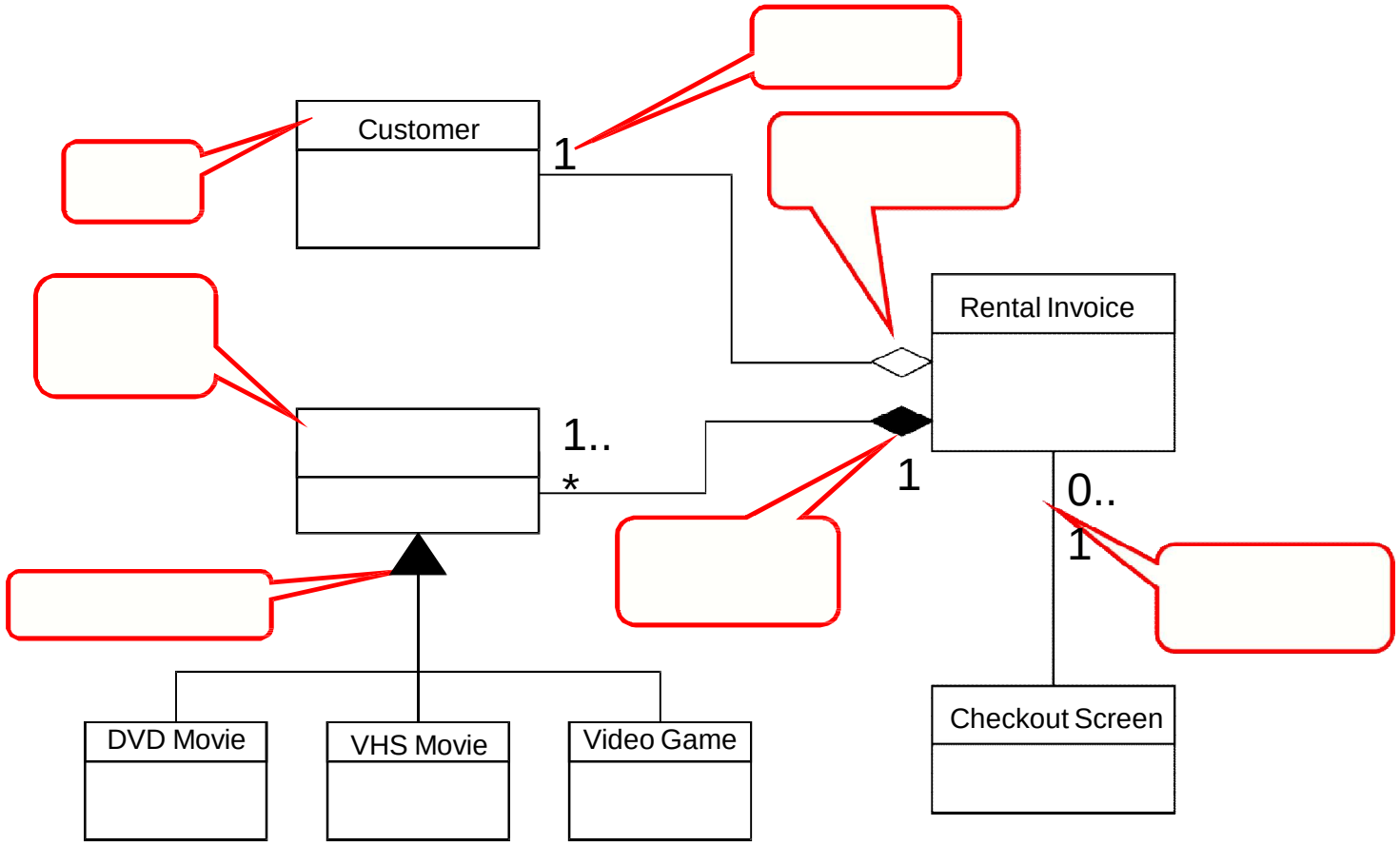
If the movie theater goes away
so does the box office => composition but
movies may still exist => aggregation


Customer
name
address
Order
date
status
calcTax
calcTotal
calcTotalWeight
No arrows; info can flow in both directions
1
0..*
association
1
1..*
abstract class
Payment
amount
generalization
role name
line item
1..*
multiplicity
1
Credit
number
type
expDate
authorized
Cash
cashTendered
Check
name
bankID
authorized
OrderDetail
quantity
taxStatus
calcSubTotal
calcWeight
0..*
1
Item
shippingWeight
description
getPriceForQuantity
getWeight
class name
attributes
operations
navigability



Let’s add the visibility attributes

Class diagram example: video store



CLASS-RESPONSIBILITY-COLLABORATION CARDS

Responsibilities and Collaborations

- Responsibilities
  - Knowing
  - Doing
- Collaboration
  - Objects working together to service a request

A CRC Card

Front:

Class Name: Patient	**ID:** 3	**Type:** Concrete, Domain
Description: An Individual that needs to receive or has received medical attention		**Associated Use Cases:** 2

Responsibilities	Collaborators
Make appointment	Appointment
Calculate last visit	
Change status	
Provide medical history	Medical history

Back of CRC Card

Attributes:

Amount (double)

Insurance carrier (text)

Relationships:

Generalization (a-kind-of): Person

Aggregation (has-parts): Medical History

Other Associations: Appointment

Object Diagrams

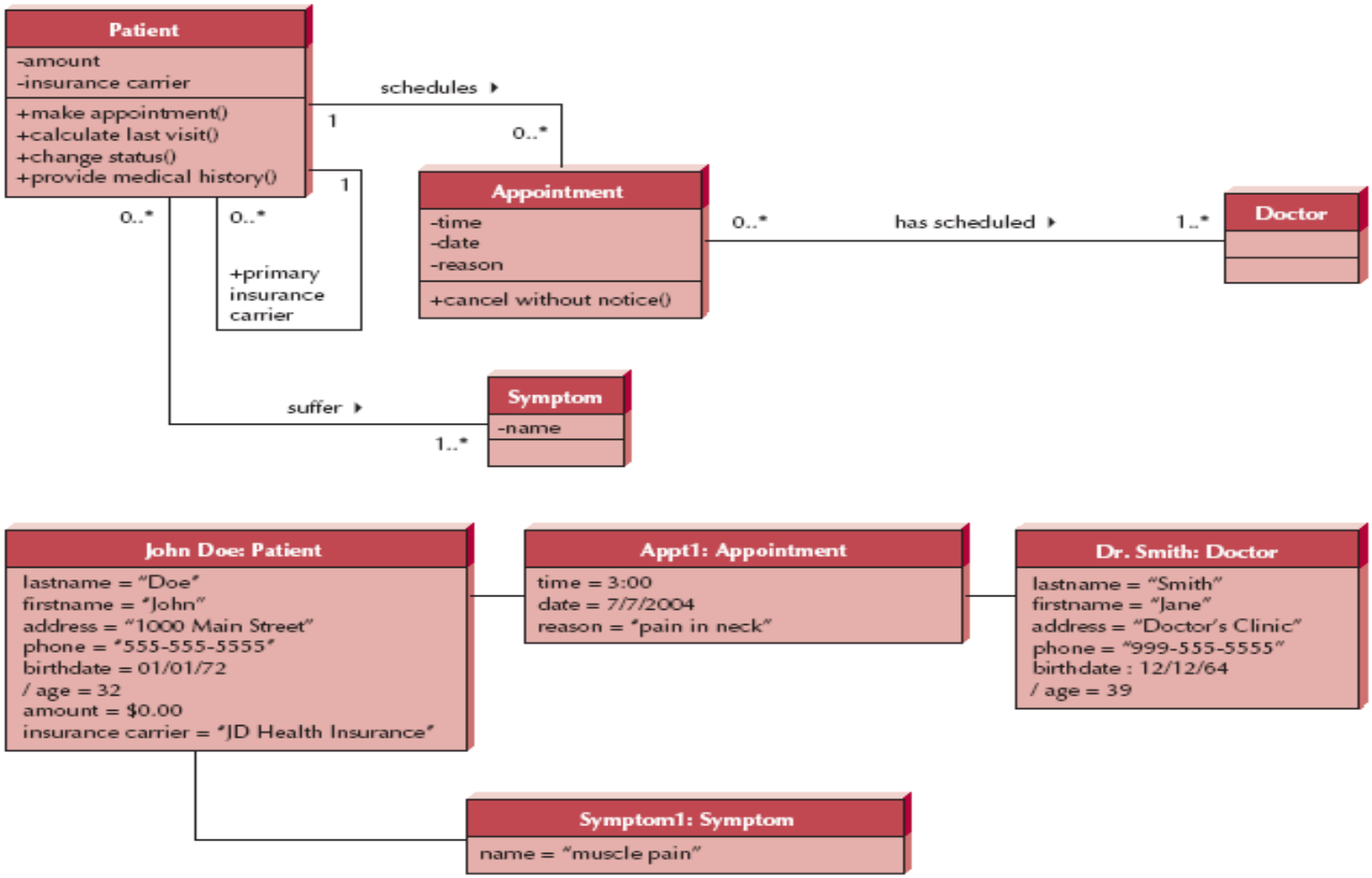


FIGURE 7-5 Example Object Diagram

Object Identification

- Textual analysis of use-case information
  - Nouns suggest classes
  - Verbs suggest operations
- Creates a rough first cut
- Common object list
- Incidents
- Roles

Patterns

- Useful groupings of classes that recur in various situations
- Contain groups of classes that collaborate or work together
- Enable reusability

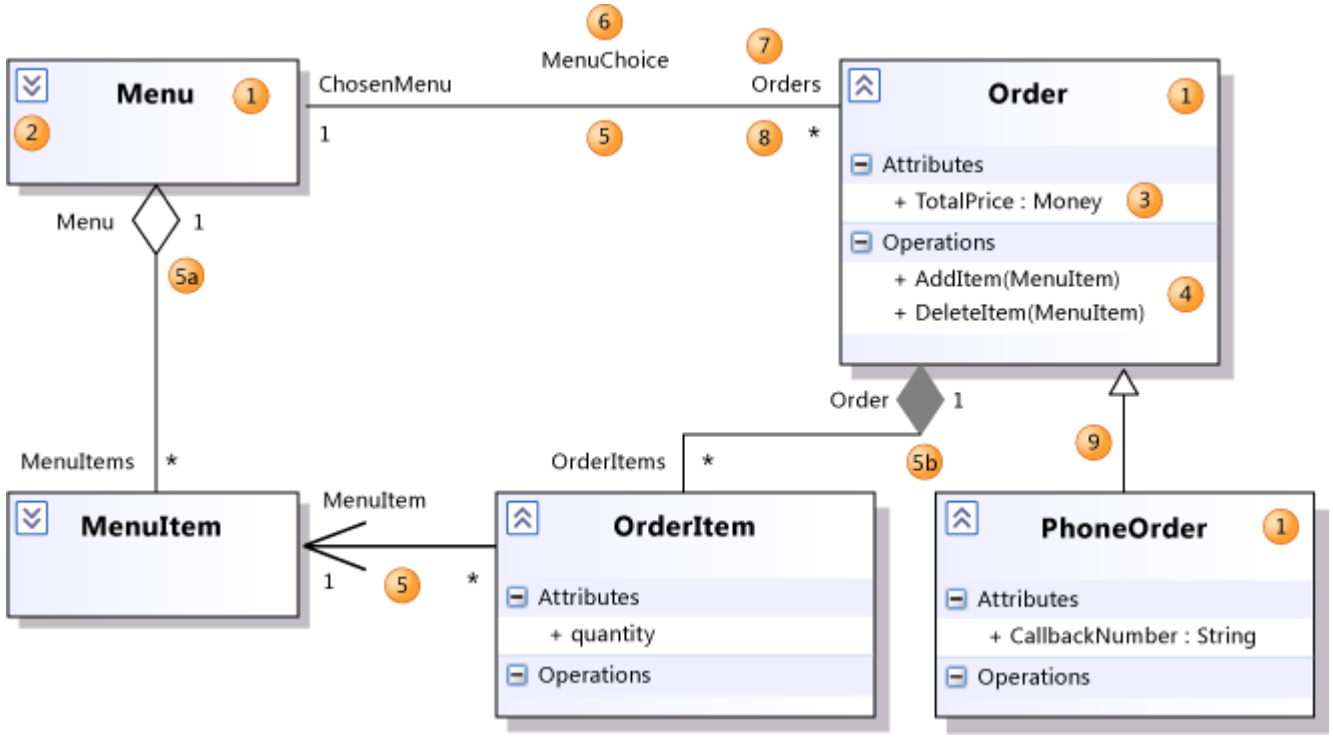
Steps for Object Identification and Structural Modelling

1. Create CRC cards by performing textual analysis on the use-cases.
2. Brainstorm additional candidate classes, attributes, operations, and relationships by using the common object list approach.
3. Role-play each use-case using the CRC cards.
4. Create the class diagram based on the CRC cards.
5. Review the structural model for missing and/or unnecessary classes, attributes, operations, and relationships.
6. Incorporate useful patterns.
7. Review the structural model.

Summary

- *CRC cards* capture essential elements of a class.
- *Class and object diagrams* show the underlying structure of an object-oriented system.
- Constructing the structural model is an iterative process involving: *textual analysis, brainstorming objects, role playing, creating the diagrams*, and *incorporating useful patterns.*

extras


Menu
1
2
ChosenMenu
1
6
MenuChoice
5
7
Orders
8
*
Order
1
Attributes
+ TotalPrice : Money
3
Operations
+ AddItem(MenuItem)
+ DeleteItem(MenuItem)
4
Menu
1
5a
Order
1
5b
9
MenuItems
*
OrderItems
*
MenuItem
1
5
*
MenuItem
OrderItem
Attributes
+ quantity
Operations
PhoneOrder
1
Attributes
+ CallbackNumber : String
Operations

- **5**: **Association**: A relationship between the members of two classifiers.
- **5a**: **Aggregation**: An association representing a shared ownership relationship. The**Aggregation** property of the owner role is set to **Shared**.
- **5b**: **Composition**: An association representing a whole-part relationship. The **Aggregation**property of the owner role is set to **Composite**.
- **9**: **Generalization**: The specific classifier inherits part of its definition from the general classifier. The general classifier is at the arrow end of the connector. Attributes, associations, and operations are inherited by the specific classifier. Use the **Inheritance** tool to create a generalization between two classifiers.
-