

Lecture 2, Modeling with UML



Compiled by: Dr. Osama Hosameldeen

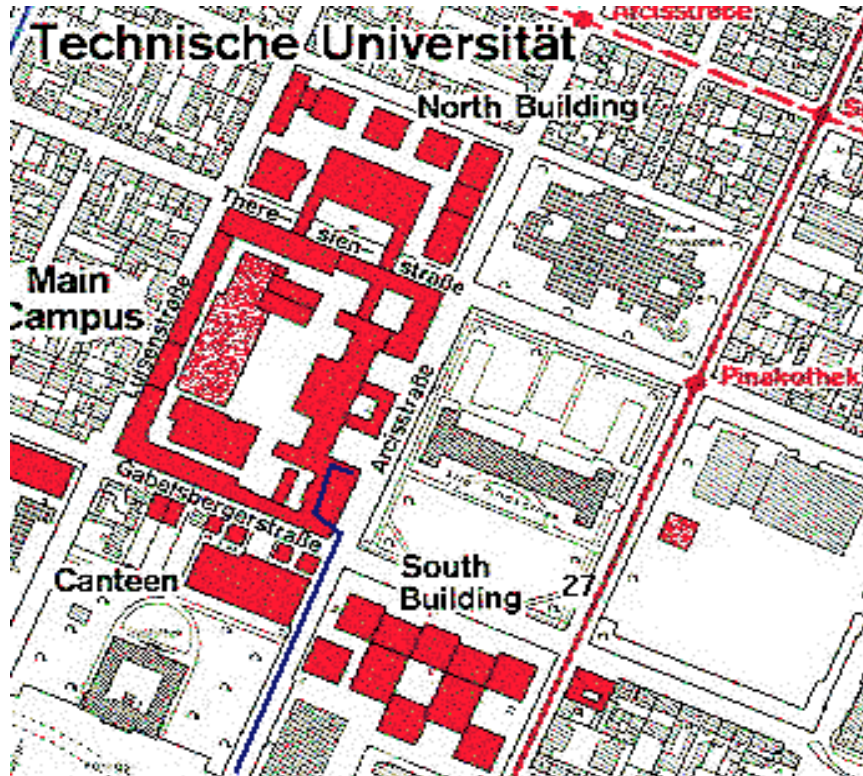
Overview: modeling with UML

- ♦ What is modeling?
- ♦ What is UML?
- ♦ Use case diagrams
- ♦ Class diagrams
- ♦ Sequence diagrams
- ♦ Activity diagrams

What is modeling?

- ♦ Modeling consists of building an abstraction of reality.
- ♦ Abstractions are simplifications because:
 - ♦ **They ignore irrelevant details and**
 - ♦ **They only represent the relevant details.**
- ♦ What is *relevant* or *irrelevant* depends on the purpose of the model.

Example: Street map, House blueprint



Why model software?

Why model software?

- ♦ Software is getting increasingly more complex
 - ♦ **Windows XP > 40 million lines of code**
 - ♦ **A single programmer cannot manage this amount of code in its entirety.**
- ♦ Code is not easily understandable by developers who did not write it
- ♦ We need simpler representations for complex systems
 - ♦ **Modeling is a means for dealing with complexity**

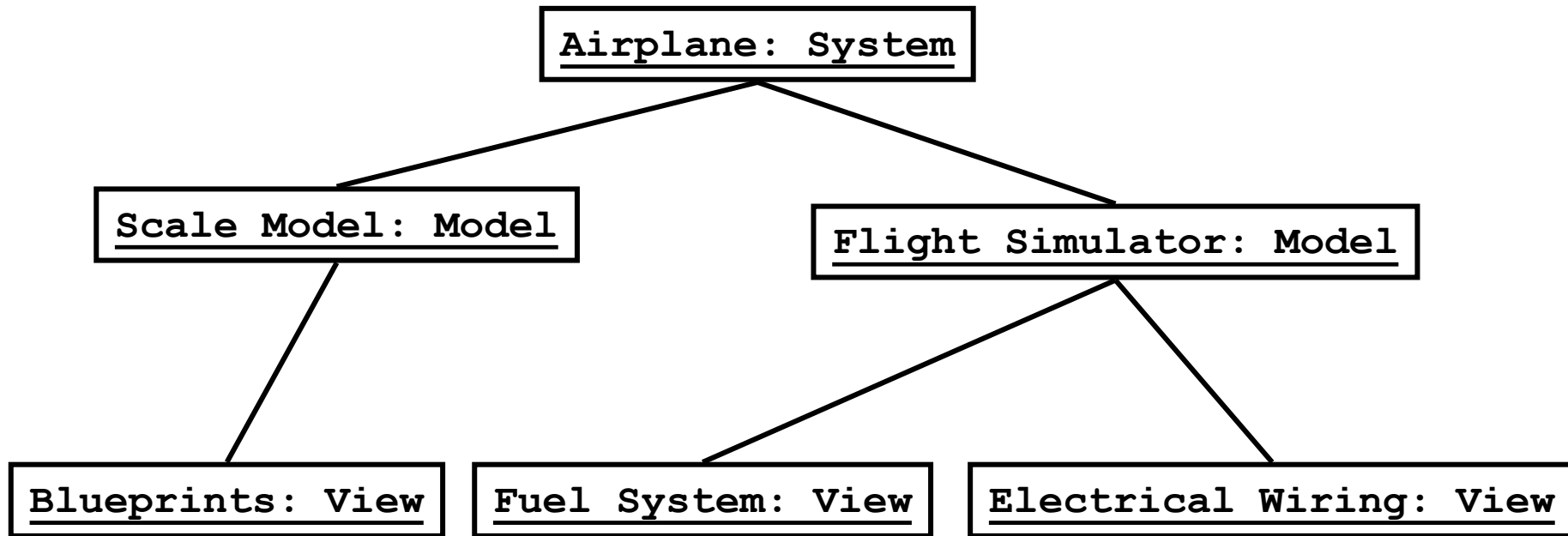
Systems, Models and Views

- ♦ A *model* is an abstraction describing a subset of a system
- ♦ A *view* depicts selected aspects of a model
- ♦ A *notation* is a set of graphical or textual rules for depicting views
- ♦ Views and models of a single system may overlap each other

Examples:

- ♦ System: Aircraft
- ♦ Models: Flight simulator, scale model
- ♦ Views: All blueprints, electrical wiring, fuel system

Models, Views and Systems (UML)




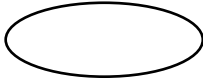
What is UML?

- ♦ UML (Unified Modeling Language)
 - ♦ **Nonproprietary standard for modeling software systems, Object Management Group (OMG)**
 - ♦ **Convergence of notations used in object-oriented methods**
 - ♦ **OMT (Rumbaugh et al.)**
 - ♦ **Booch (Grady Booch)**
 - ♦ **OOSE (Jacobson et al.)**
- ♦ Current Version: UML 2.4.1
- ♦ Open-Source tools: ArgoUML, list of tutorials
https://www.youtube.com/playlist?list=PL1FOYmrrT6nzt7J54_SnkbwwofM4RiaaA
- ♦ Commercial : StarUML

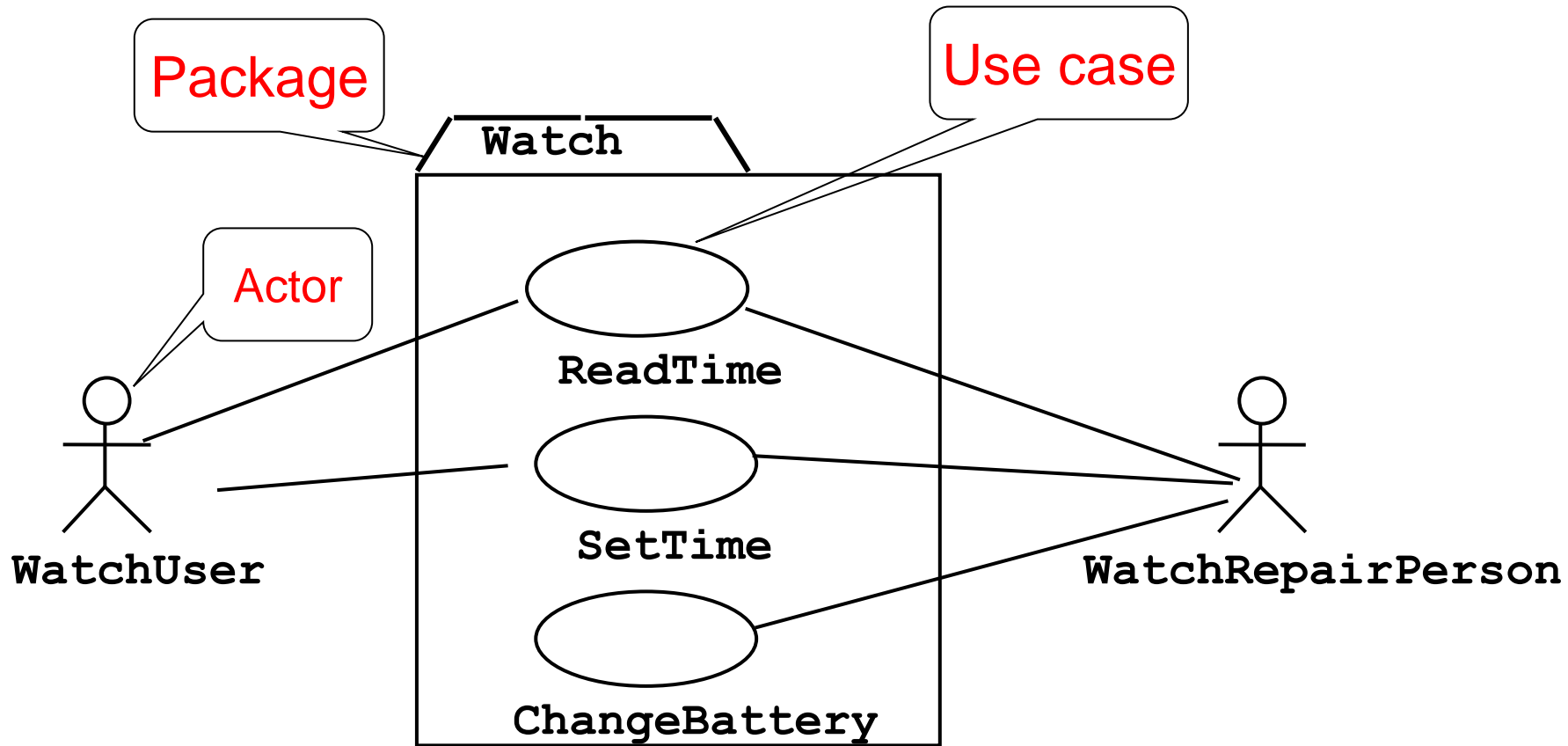
UML First Pass

- ◆ **Use case diagrams**
 - ◆ Describe the functional behavior of the system as seen by the user
 - ◆ Used during requirements elicitation
- ◆ **Class diagrams**
 - ◆ Describe the static structure of the system: Objects, attributes, associations
- ◆ **Sequence diagrams**
 - ◆ Describe the dynamic behavior between objects of the system
- ◆ **State diagrams**
 - ◆ Describe the dynamic behavior of an individual object

UML Core Conventions

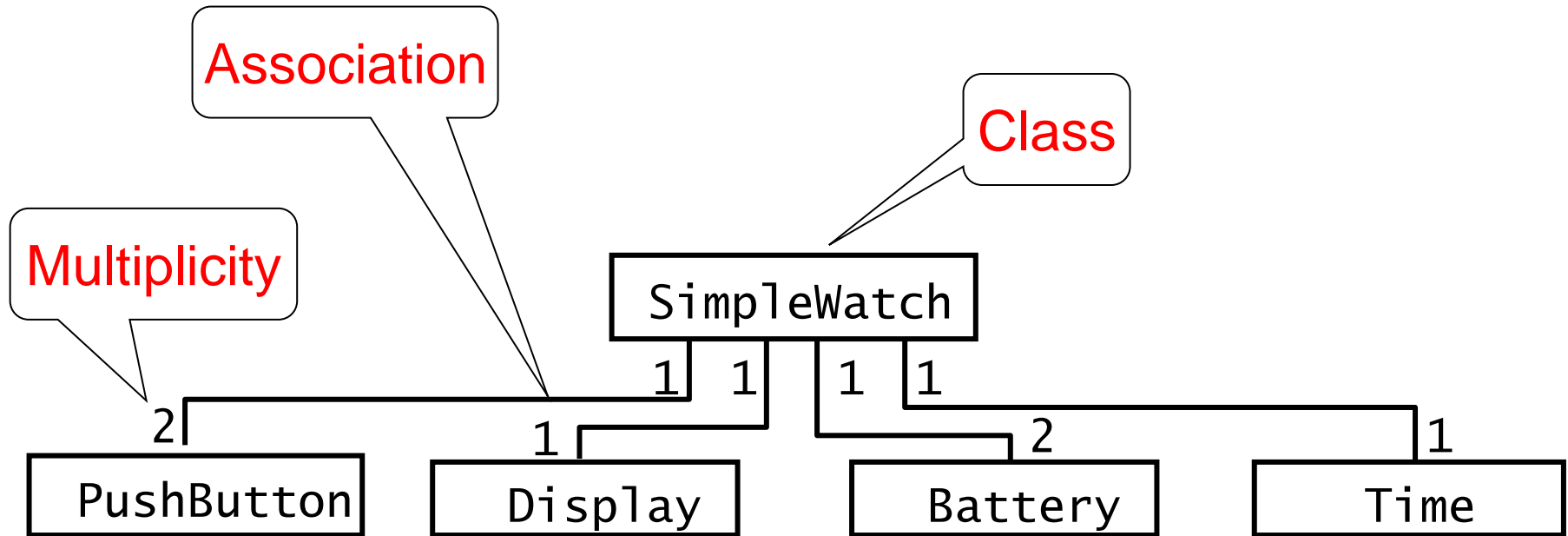
- ♦ All UML Diagrams are composed of graphs of nodes and edges
 - ♦ **Nodes are entities and drawn as rectangles or ovals**
 - ♦ **Rectangles** denote classes or instances 
 - ♦ **Ovals** denote functions 
- Names of Classes are not underlined
 - SimpleWatch
 - Firefighter
- Names of Instances are underlined
 - myWatch:SimpleWatch
 - Joe:Firefighter
- An edge between two nodes denotes a relationship between the corresponding entities

Historical Remark: UML 1 used packages



Use case diagrams represent the functionality of the system from user's point of view

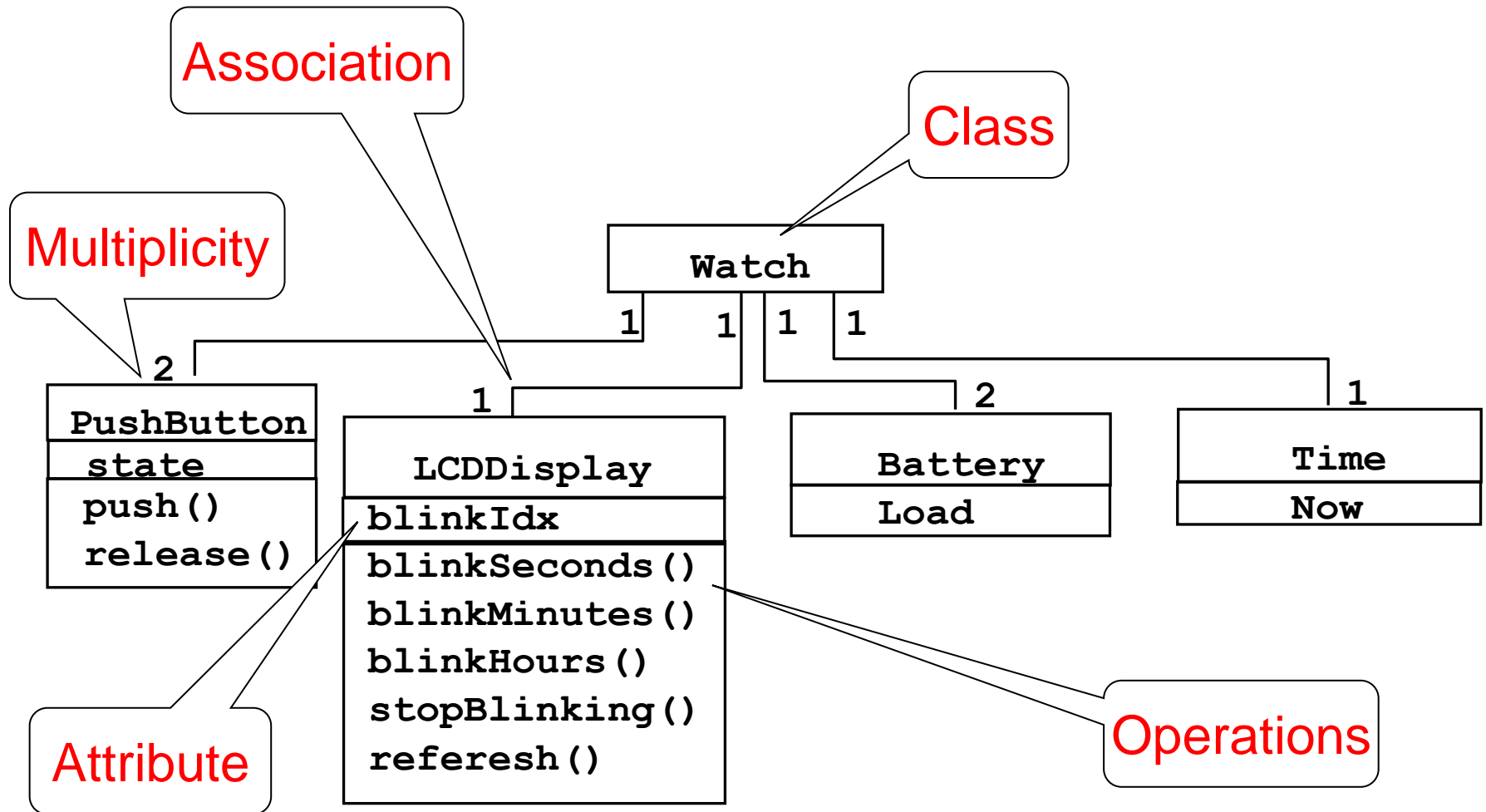
UML first pass: Class diagrams



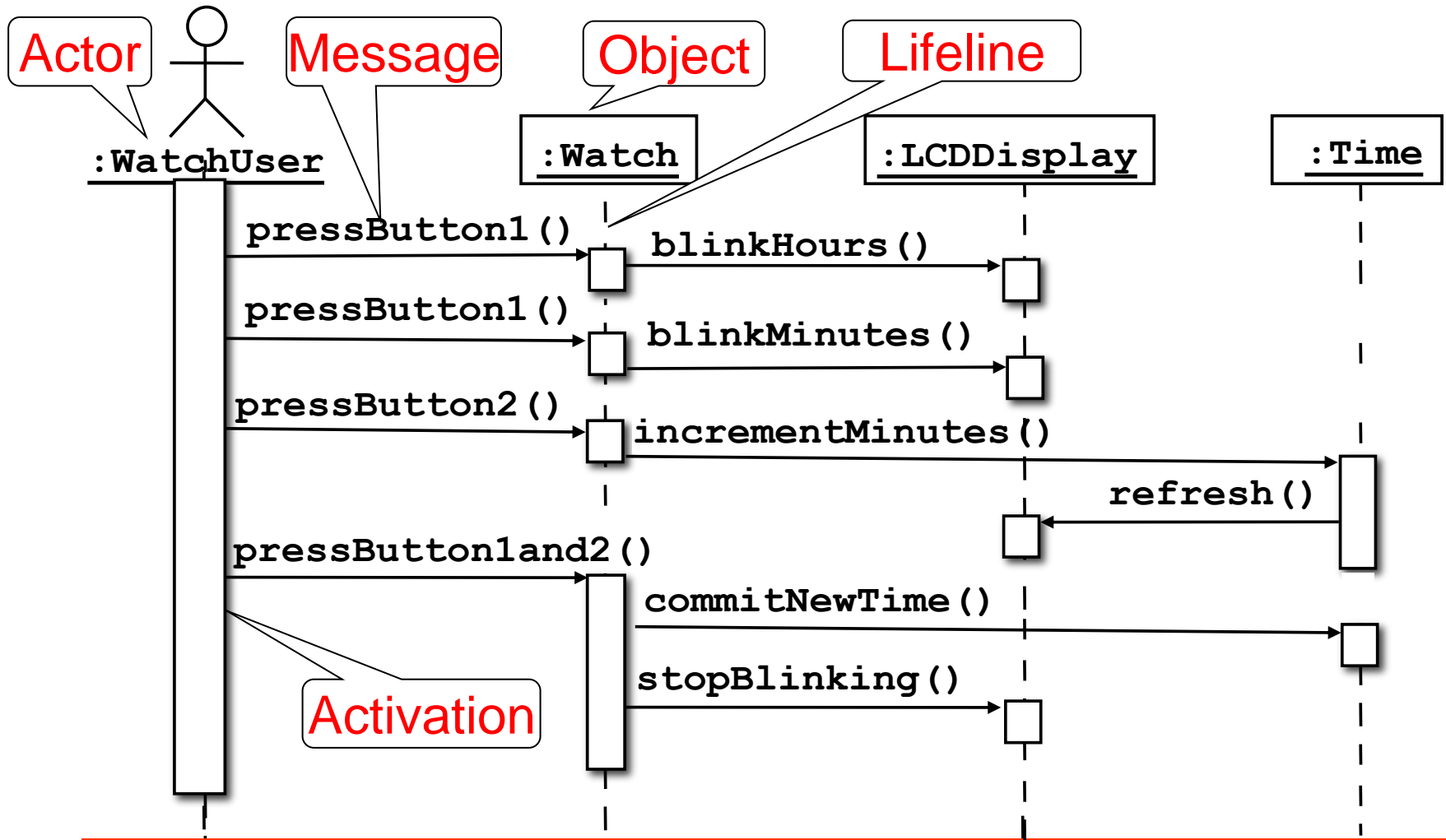
Class diagrams represent the structure of the system

UML first pass: Class diagrams

Class diagrams represent the structure of the system

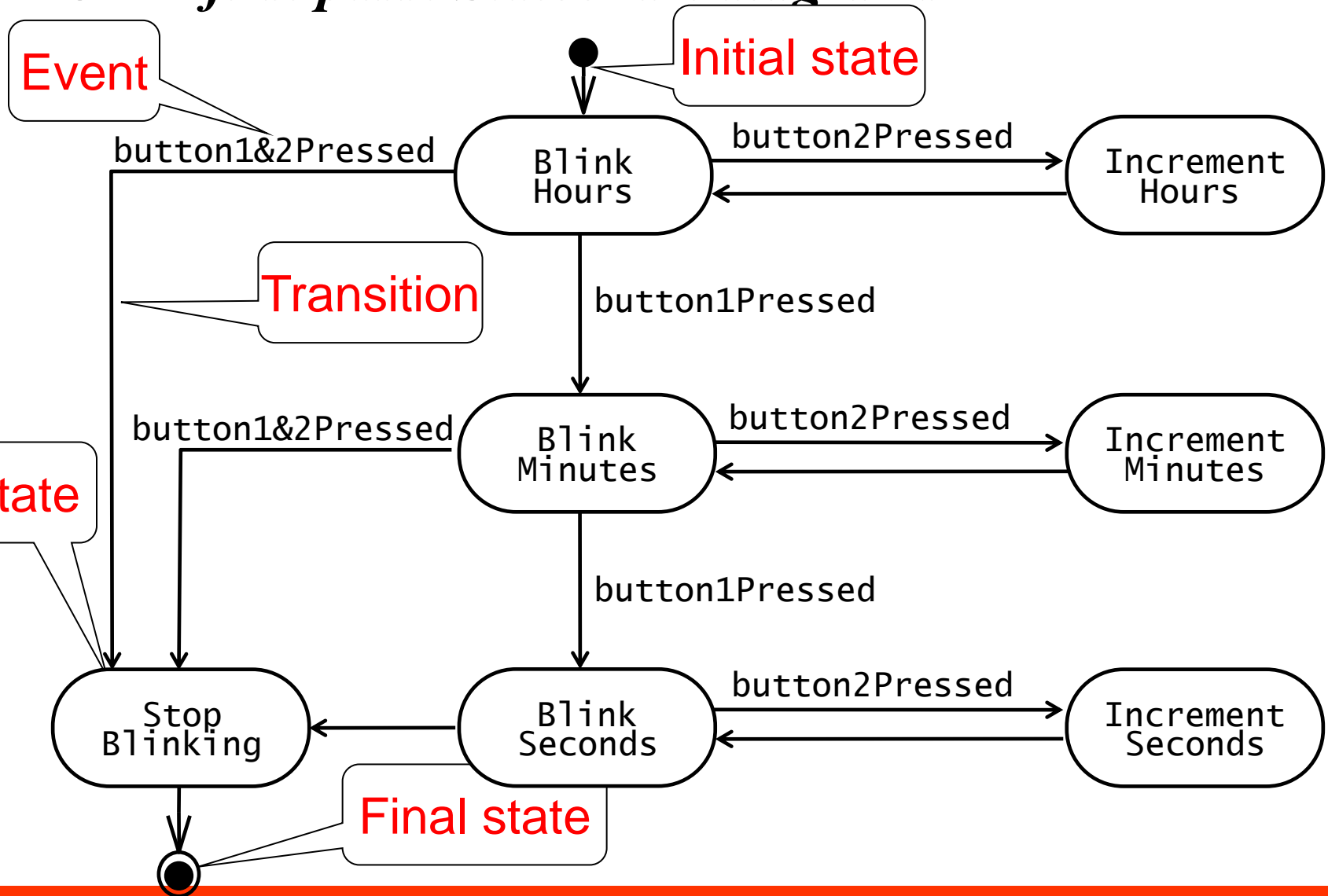


UML first pass: Sequence diagram



Sequence diagrams represent the behavior of a system as messages (“interactions”) between *different objects*

UML first pass: Statechart diagrams



Represent behavior of *a single object* with interesting dynamic behavior.

UML Second Pass

- ◆ **Use case diagrams**
 - ◆ **Describe the functional behavior of the system as seen by the user**
- ◆ **Class diagrams**
 - ◆ **Describe the static structure of the system: Objects, attributes, associations**
- ◆ **Sequence diagrams**
 - ◆ **Describe the dynamic behavior between objects of the system**
- ◆ **State diagrams**
 - ◆ **Describe the dynamic behavior of an individual object**
- ◆ **Activity diagrams**
 - ◆ **Describe the dynamic behavior of a system, in particular the workflow.**

UML Use Case Diagrams

UML Use Case Diagrams

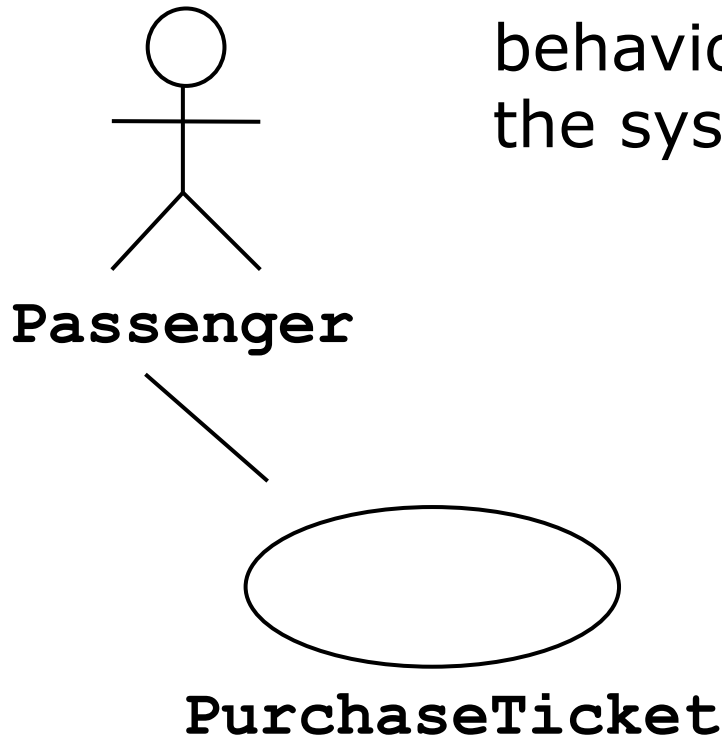
Used during requirements elicitation and analysis to represent external behavior ("visible from the outside of the system")

An *Actor* represents a role, that is, a type of user of the system

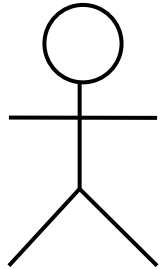
A ***use case*** represents a class of functionality provided by the system

Use case model:

The set of all use cases that completely describe the functionality of the system.



Actors



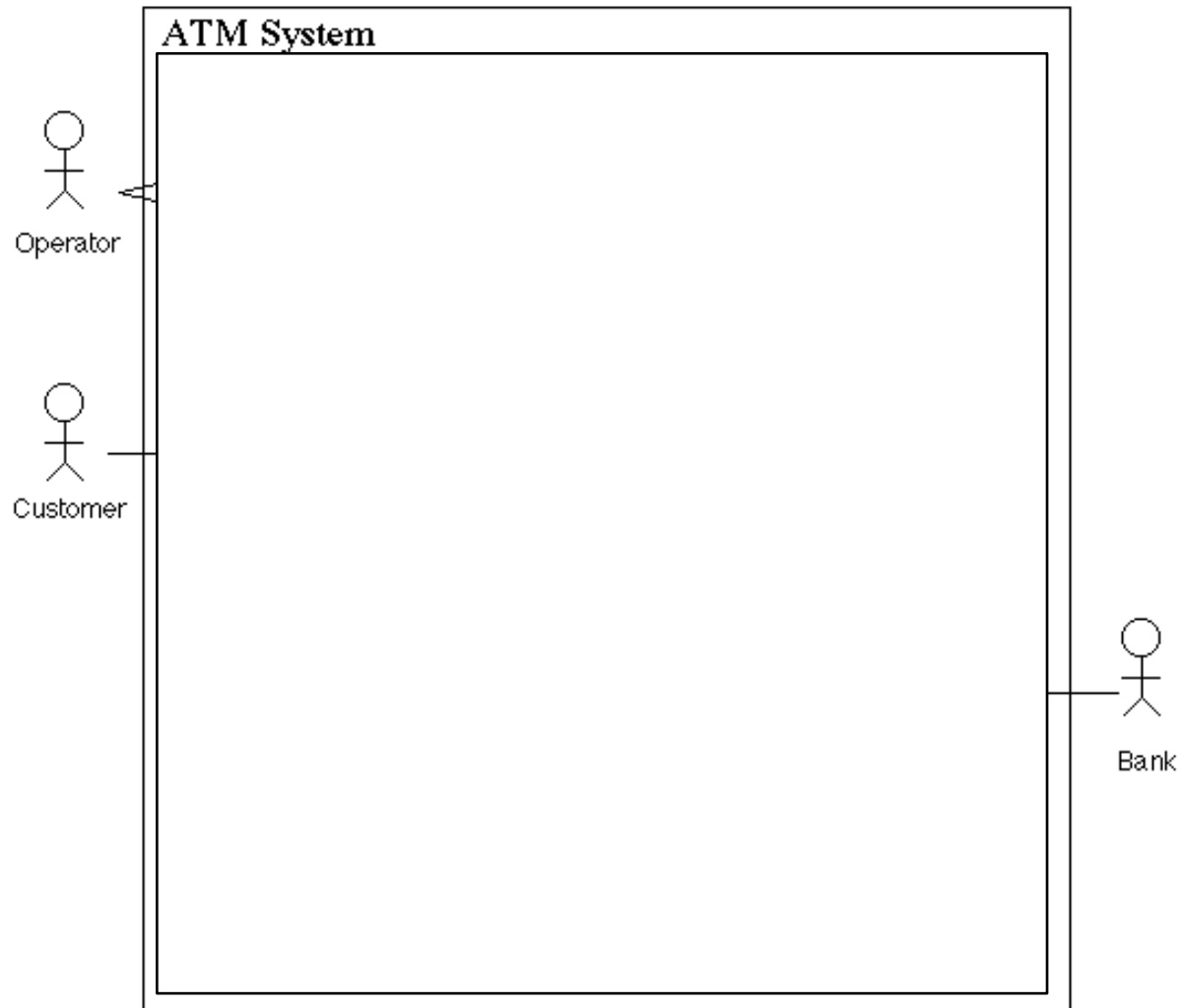
Passenger

- ◆ An actor is a model for an external entity which interacts (communicates) with the system:
 - ◆ **User**
 - ◆ **External system (Another system)**
 - ◆ **Physical environment (e.g. Weather)**
- ◆ An actor has a unique name and an optional description
- ◆ Examples:
 - ◆ **Passenger: A person in the train**
 - ◆ **GPS satellite: An external system that provides the system with GPS coordinates.**

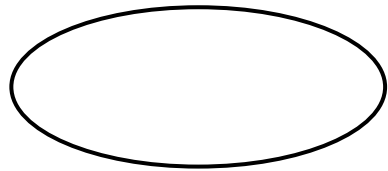
Name

**Optional
Description**

ATM System



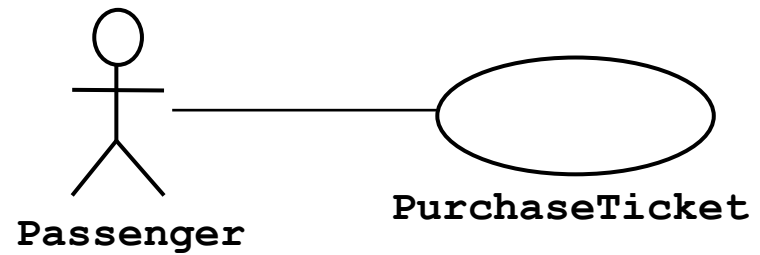
Use Case



PurchaseTicket

- A use case represents a class of functionality provided by the system
- Use cases can be described textually, with a focus on the event flow between actor and system
- The textual use case description consists of 6 parts:
 1. **Unique name**
 2. **Participating actors**
 3. **Entry conditions**
 4. **Exit conditions**
 5. **Flow of events**
 6. **Special requirements.**

Textual Use Case Description Example



1. Name: Purchase ticket

2. Participating actor: Passenger

3. Entry condition:

- ♦ Passenger stands in front of ticket distributor
- ♦ Passenger has sufficient money to purchase ticket

4. Exit condition:

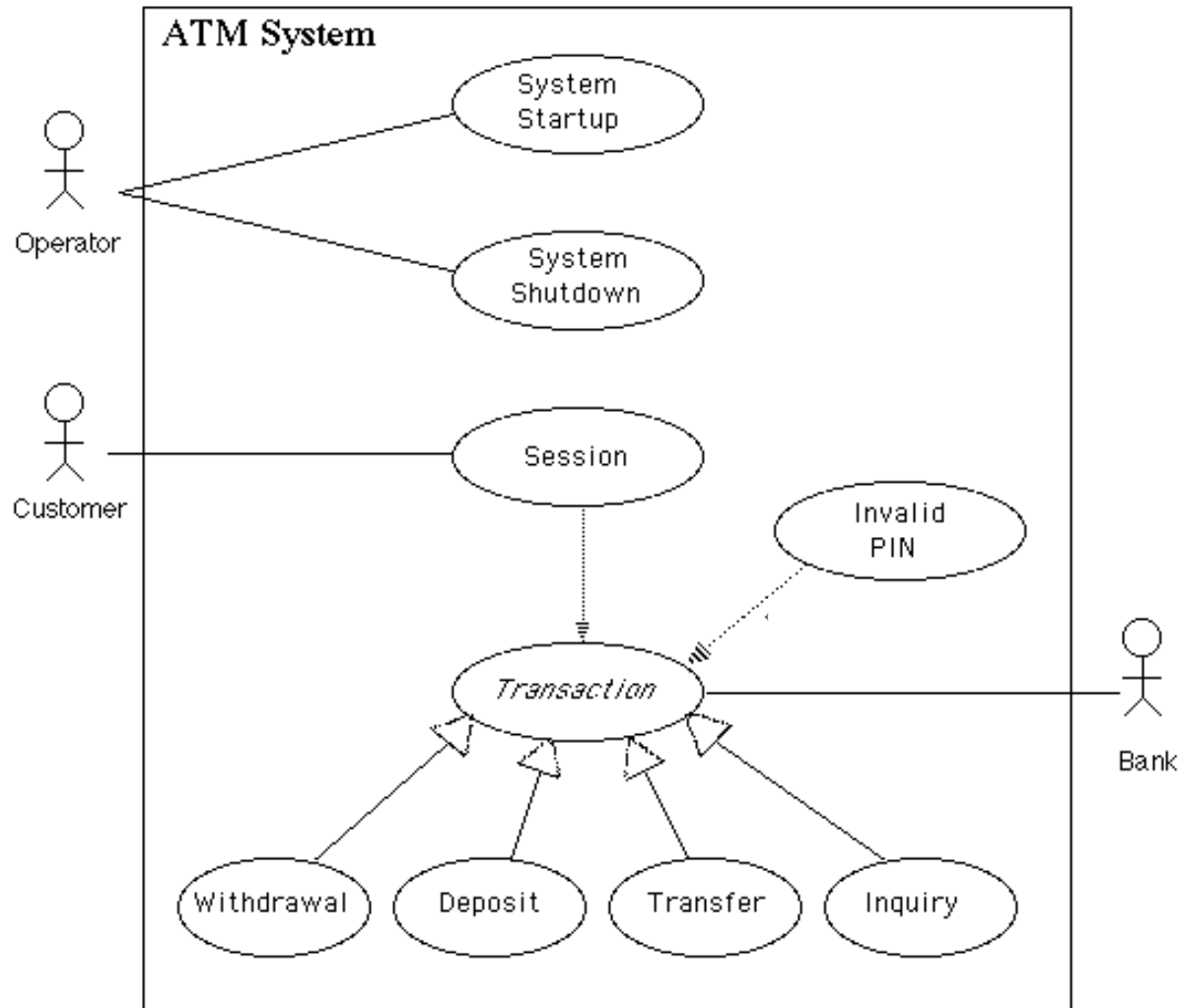
- ♦ Passenger has ticket

5. Flow of events:

- 1. Passenger** selects the number of zones to be traveled
- 2. Ticket Distributor** displays the amount due
- 3. Passenger** inserts money, at least the amount due
- 4. Ticket Distributor** returns change
- 5. Ticket Distributor** issues ticket

6. Special requirements: None.

ATM System



Uses Cases can be related

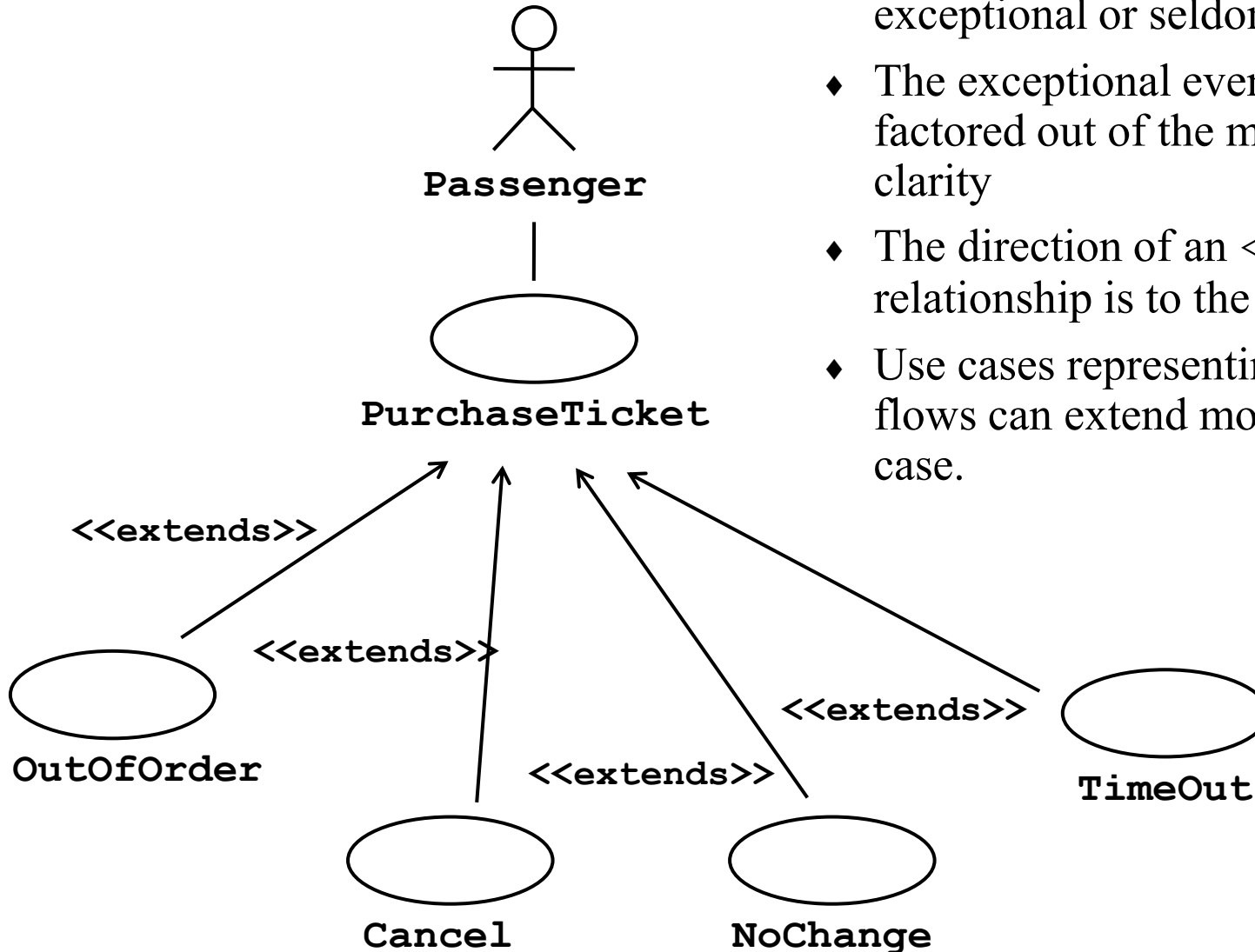
- ◆ **Extends Relationship**

- ◆ **To represent seldom invoked use cases or exceptional functionality**

- ◆ **Includes Relationship**

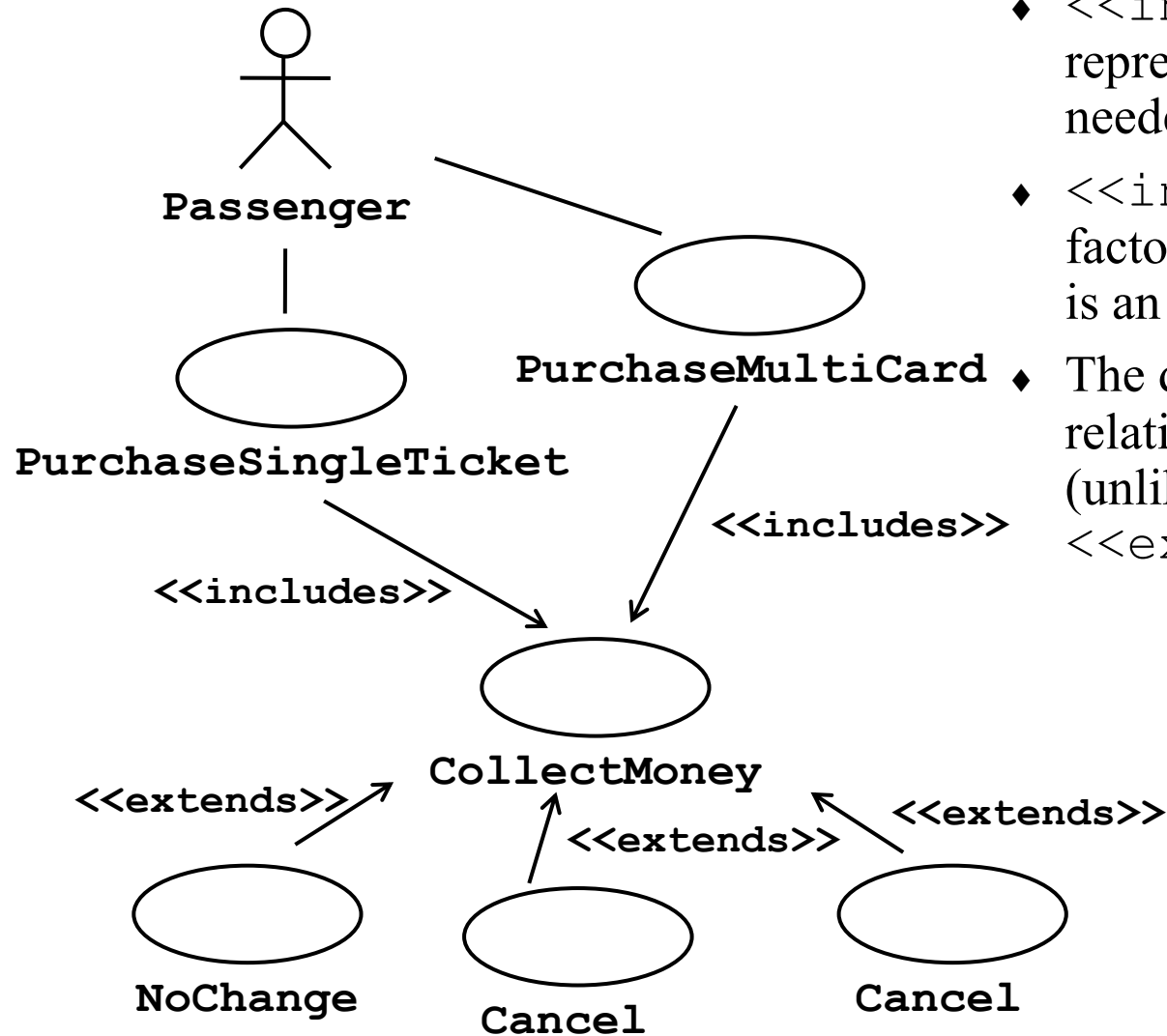
- ◆ **To represent functional behavior common to more than one use case.**

The <<extends>> Relationship



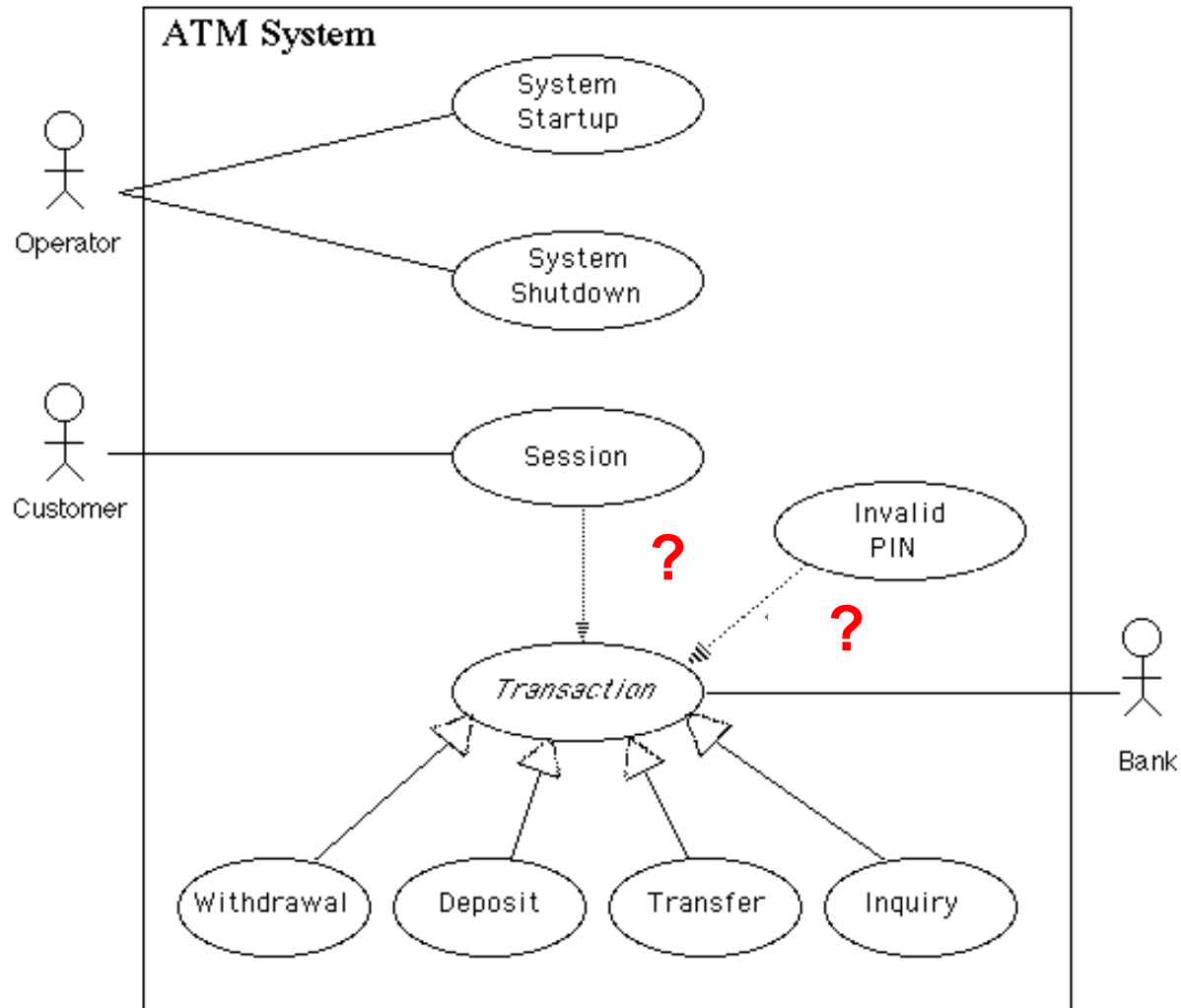
- ♦ <<extends>> relationships model exceptional or seldom invoked cases
- ♦ The exceptional event flows are factored out of the main event flow for clarity
- ♦ The direction of an <<extends>> relationship is to the extended use case
- ♦ Use cases representing exceptional flows can extend more than one use case.

The <<includes>> Relationship

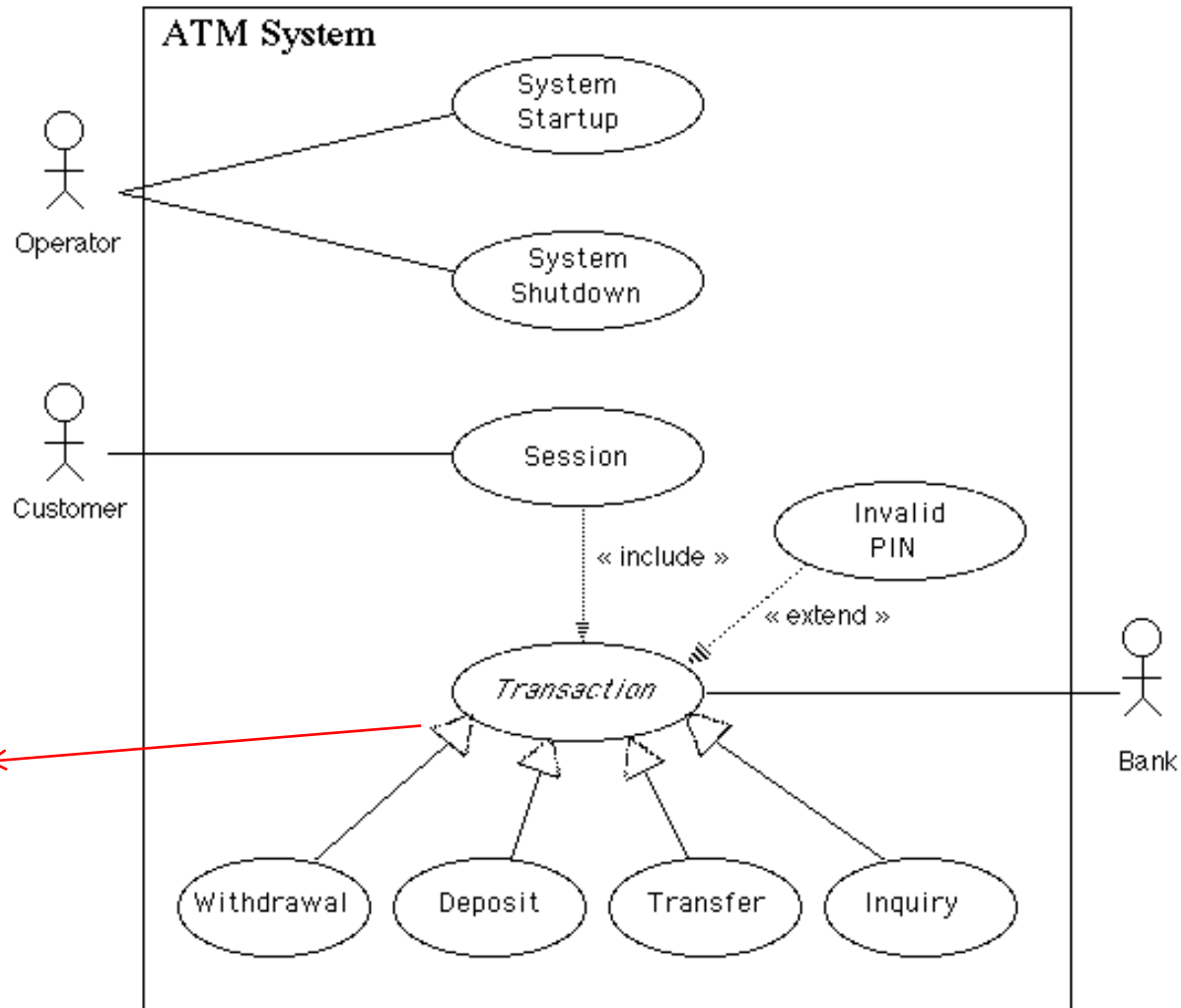


- ♦ <<includes>> relationship represents common functionality needed in more than one use case
- ♦ <<includes>> behavior is factored out for reuse, not because it is an exception
- ♦ The direction of a <<includes>> relationship is to the using use case (unlike the direction of the <<extends>> relationship).

ATM System



ATM System



Hierarchy
type?

Homework:

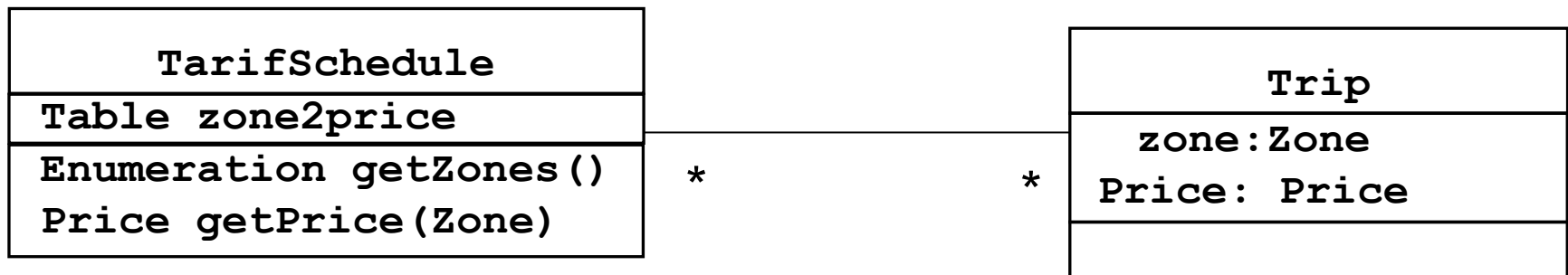
1 - What is the purpose of modeling? (One or two sentences)

2 - Draw a use case diagram for a ticket distributor for a train system. The system includes two actors: a traveler, who purchases different types of tickets, and a central computer system, which maintains a reference database for the tariff. Use cases should include: *BuyOneWayTicket*, *BuyWeeklyCard*, *BuyMonthlyCard*, *UpdateTariff*. Also include the following exceptional cases: *Time-Out* (i.e., traveler took too long to insert the right amount), *TransactionAborted* (i.e., traveler selected the cancel button without completing the transaction), *DistributorOutOfChange*, and *DistributorOutOfPaper*.

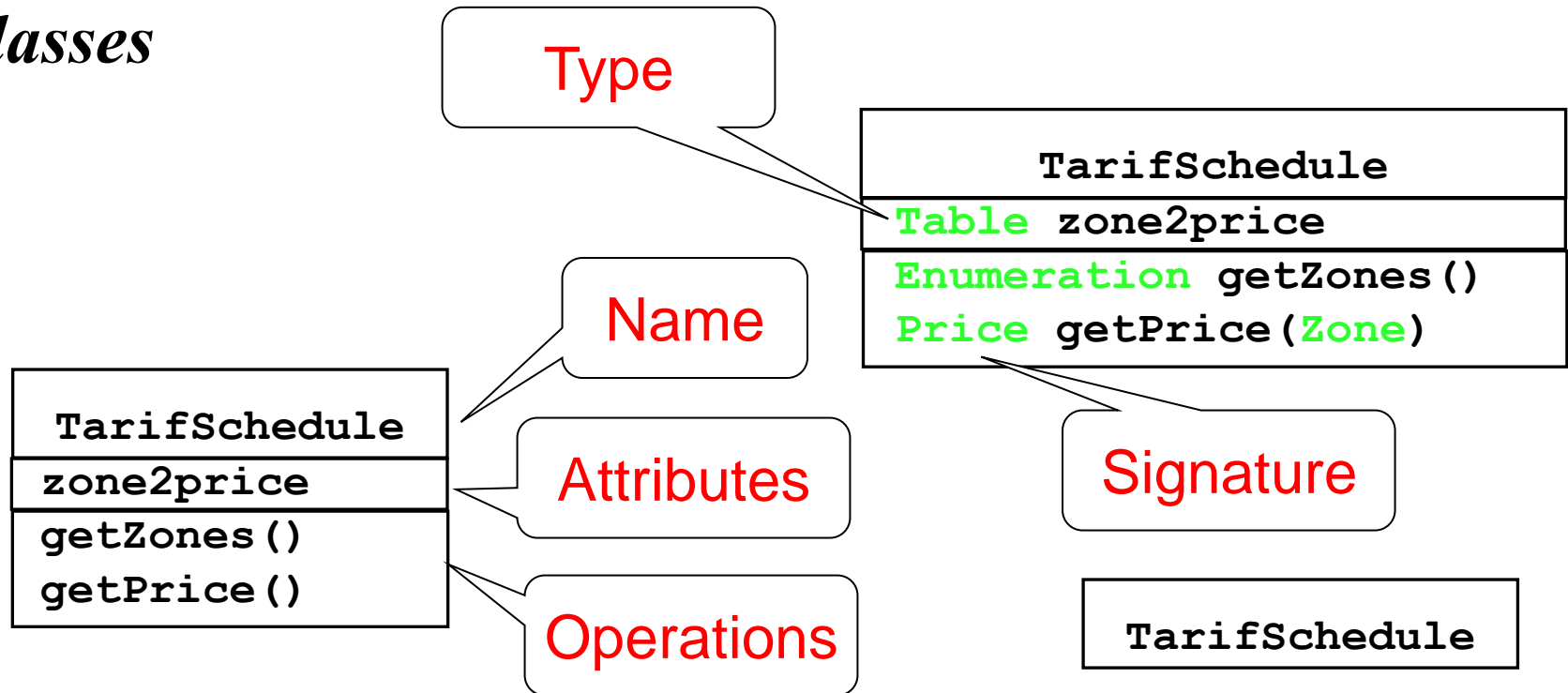
Class Diagrams

Class Diagrams

- ♦ Class diagrams represent the structure of the system
- ♦ Used
 - ♦ during requirements analysis to model application domain concepts
 - ♦ during system design to model subsystems
 - ♦ during object design to specify the detailed behavior and attributes of classes.



Classes



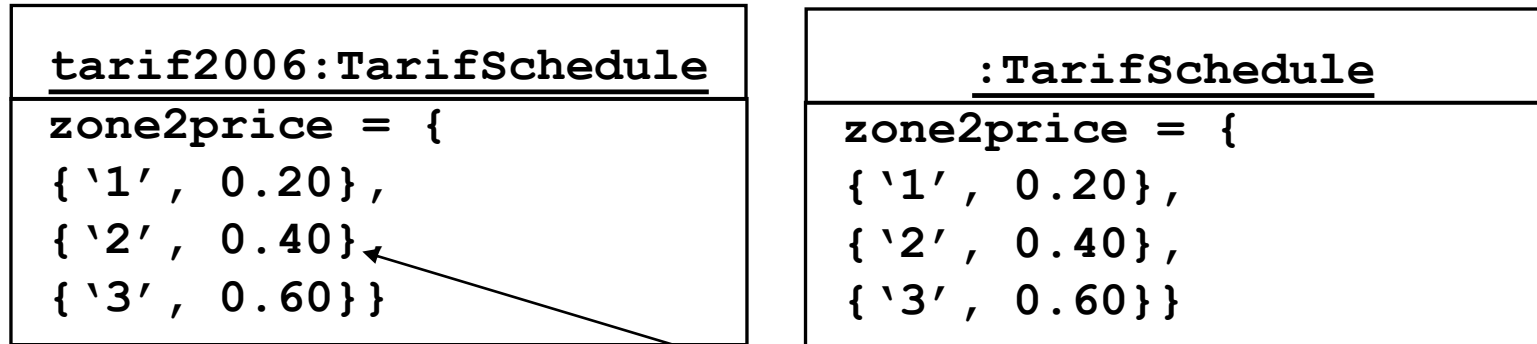
- ♦ A *class* represents a concept
- ♦ A class encapsulates state (*attributes*) and behavior (*operations*)

Each attribute has a ***type***

Each operation has a ***signature***

The class name is the only mandatory information

Instances



- ♦ An *instance* represents The attributes are represented with their *values*
- ♦ The name of an instance is underlined
- ♦ The name can contain only the class name of the instance (anonymous instance)

Actor vs Class vs Object

◆ **Actor**

- ◆ **An entity outside the system to be modeled, interacting with the system (“Passenger”)**

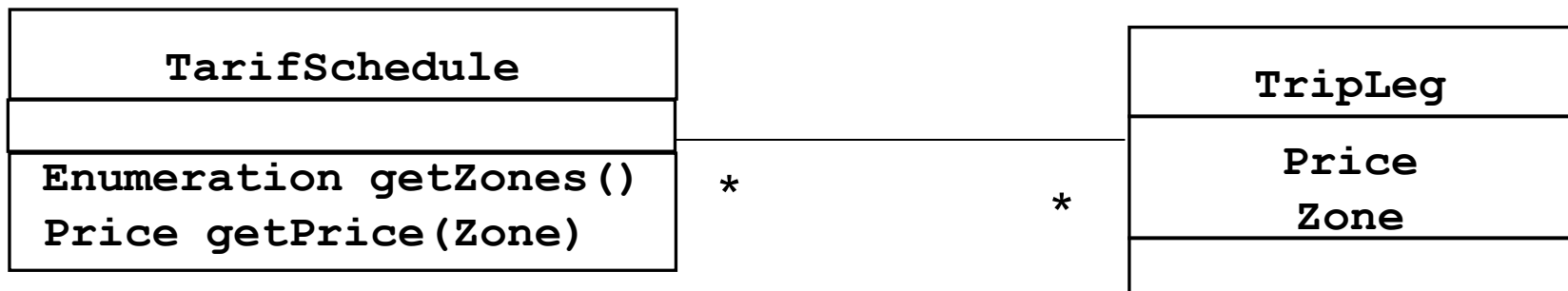
◆ **Class**

- ◆ **An abstraction modeling an entity in the application or solution domain**
- ◆ **The class is part of the system model (“User”, “Ticket distributor”, “Server”)**

◆ **Object**

- ◆ **A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).**

Associations



Associations denote relationships between classes

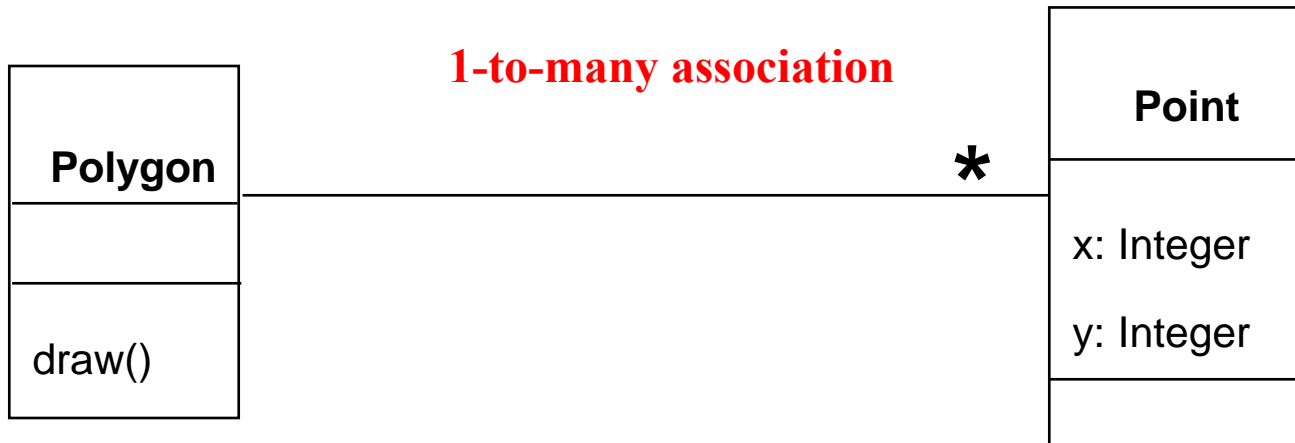
The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

1-to-1 and 1-to-many Associations

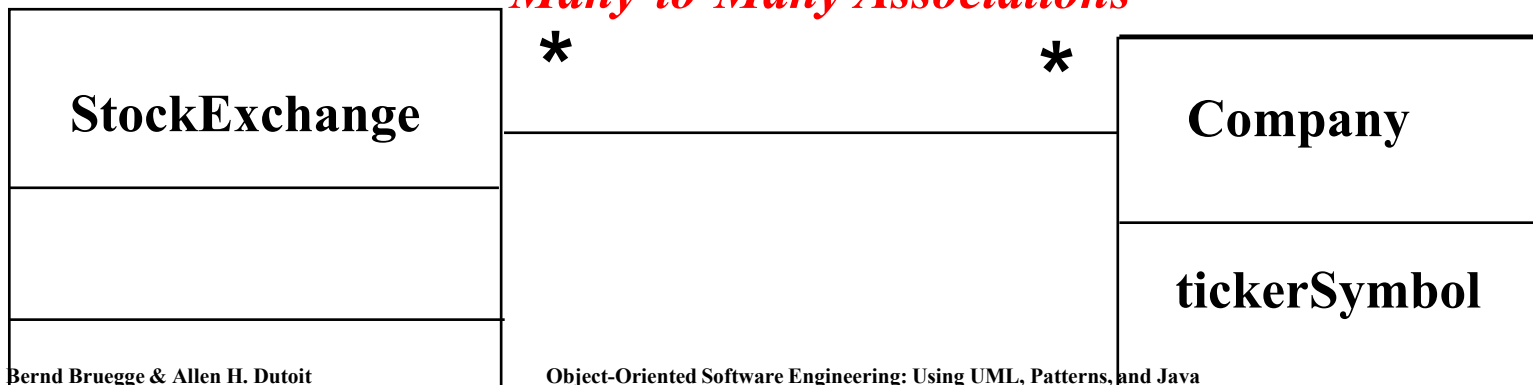
1-to-1 association



1-to-many association

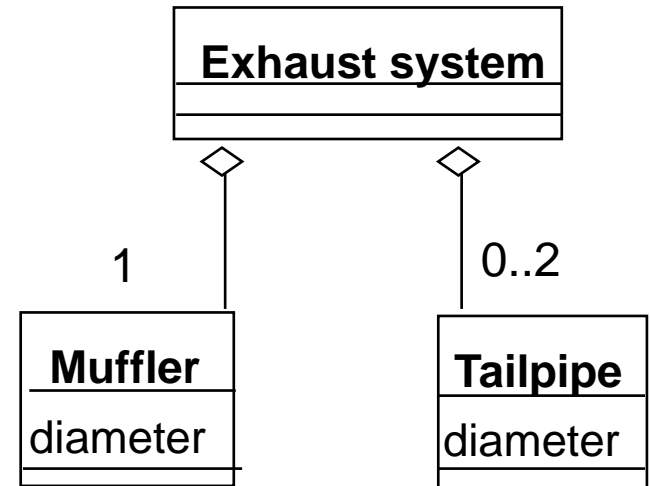


Many-to-Many Associations

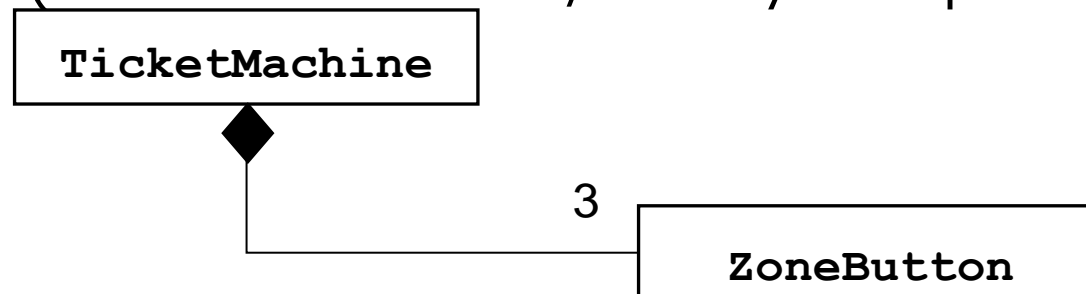


Aggregation

- ♦ An *aggregation* is a special case of association denoting a “part-of” hierarchy
- ♦ The *aggregate* is the parent class, the components are the children classes

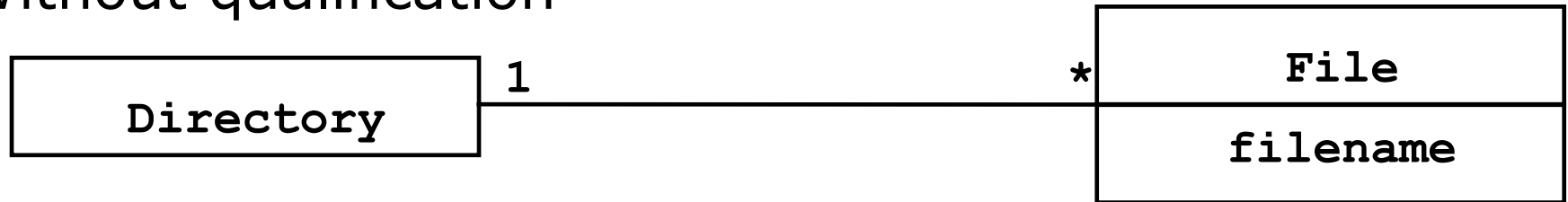


A solid diamond denotes *composition*: A strong form of aggregation where the *life time of the component instances* is controlled by the aggregate. That is, the parts don't exist on their own (“the whole controls/destroys the parts”)



Qualifiers

Without qualification

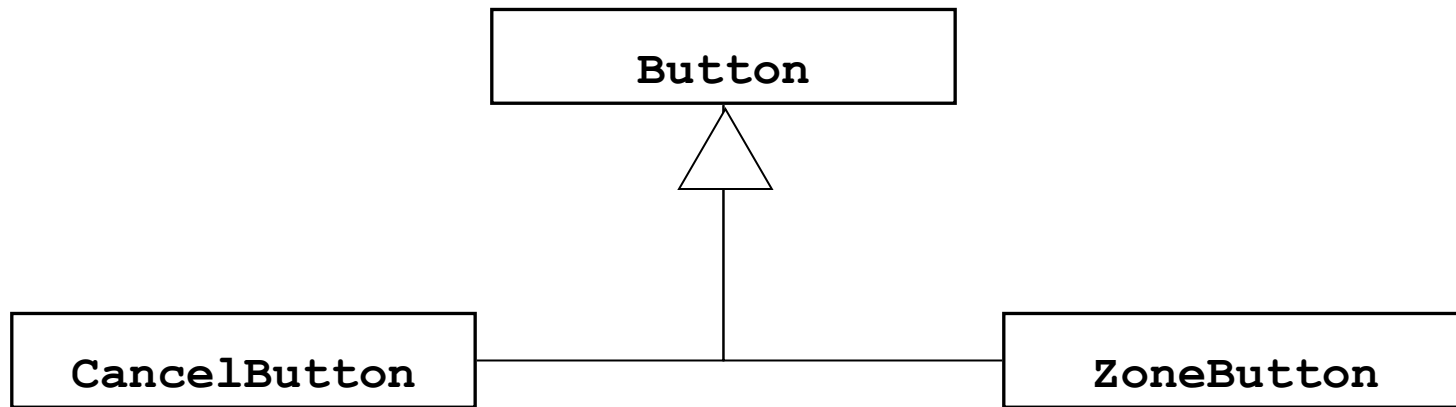


With qualification



- ◆ Qualifiers identify subsets of related instances in association navigations; they provide a model of indices or keys for association ends.
- ◆ Qualifiers can be used to reduce the multiplicity of an association

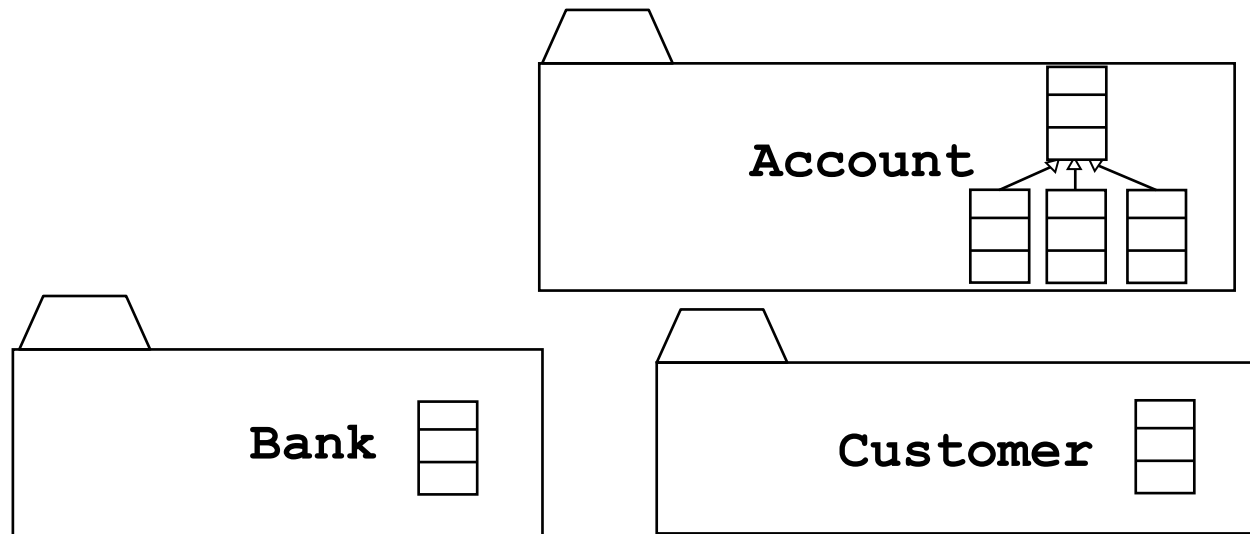
Inheritance



- ♦ *Inheritance* is another special case of an association denoting a “kind-of” hierarchy
- ♦ Inheritance simplifies the analysis model by introducing a taxonomy
- ♦ The **children classes** inherit the attributes and operations of the **parent class**.

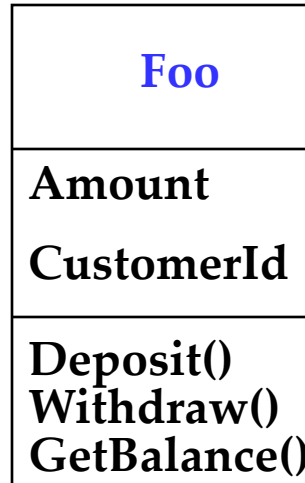
Packages

- ♦ Packages help you to organize UML models to increase their readability
- ♦ We can use the UML package mechanism to organize classes into subsystems



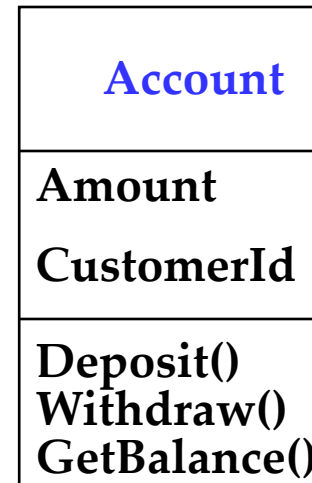
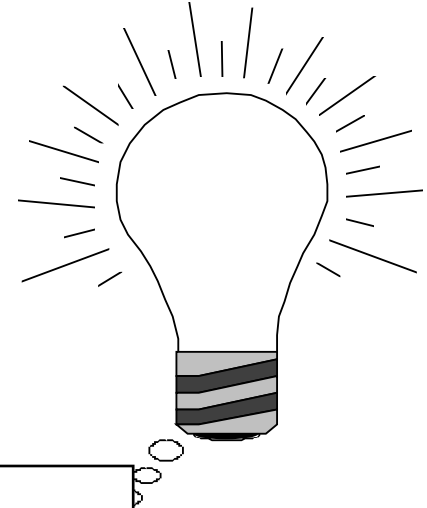
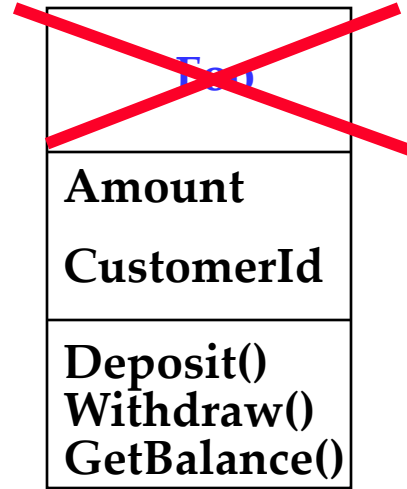
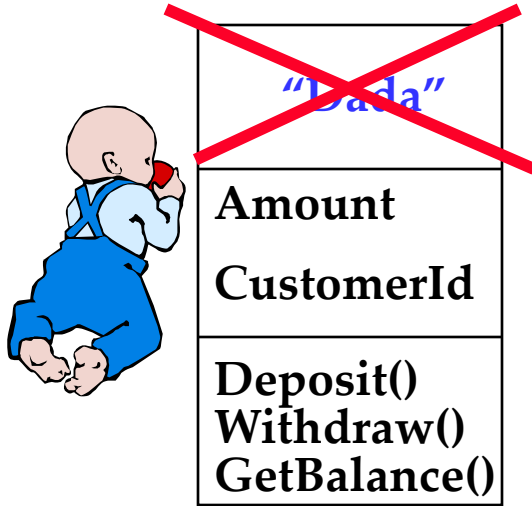
- ♦ Any complex system can be decomposed into subsystems, where each subsystem is modeled as a package.

Object Modeling in Practice



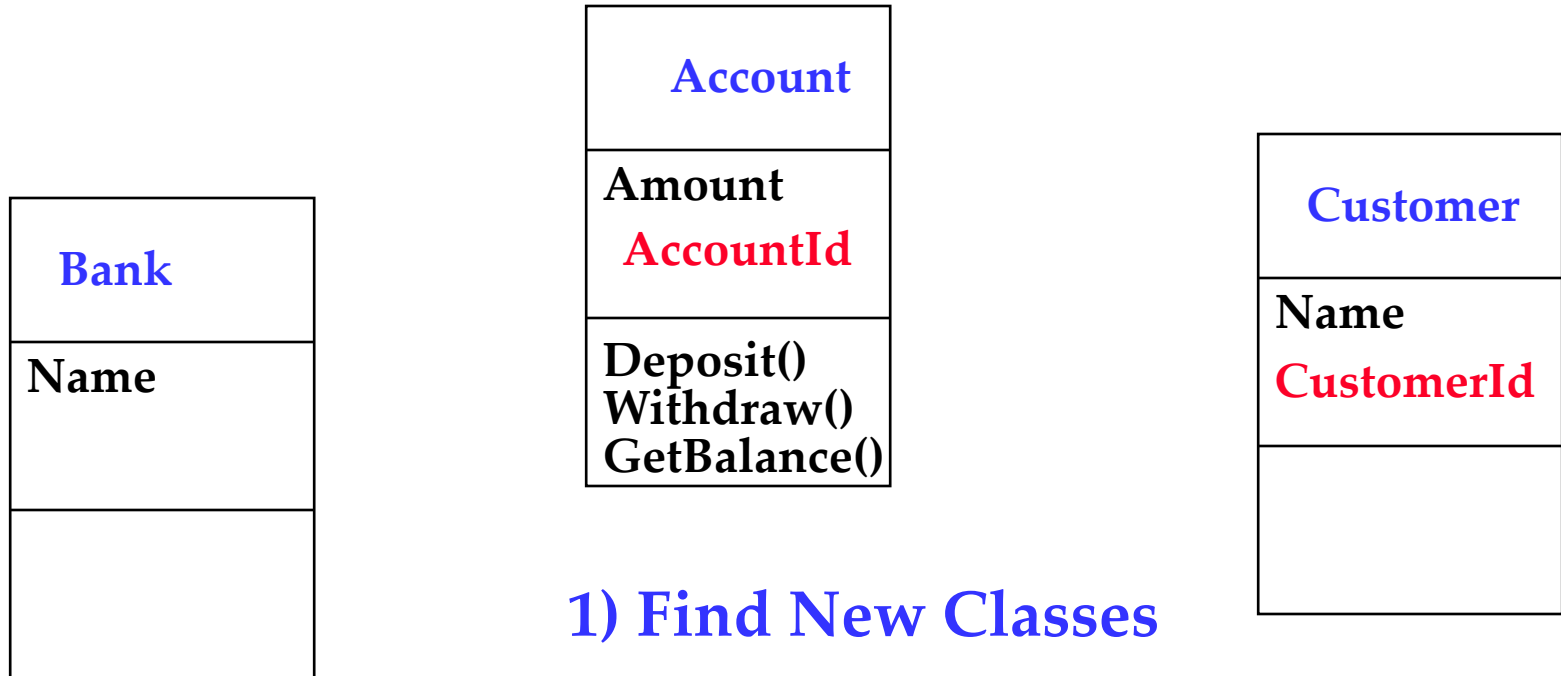
Class Identification: Name of Class, Attributes and Methods
Is **Foo the right name?**

Object Modeling in Practice: Brainstorming



Is **Foo** the right name?

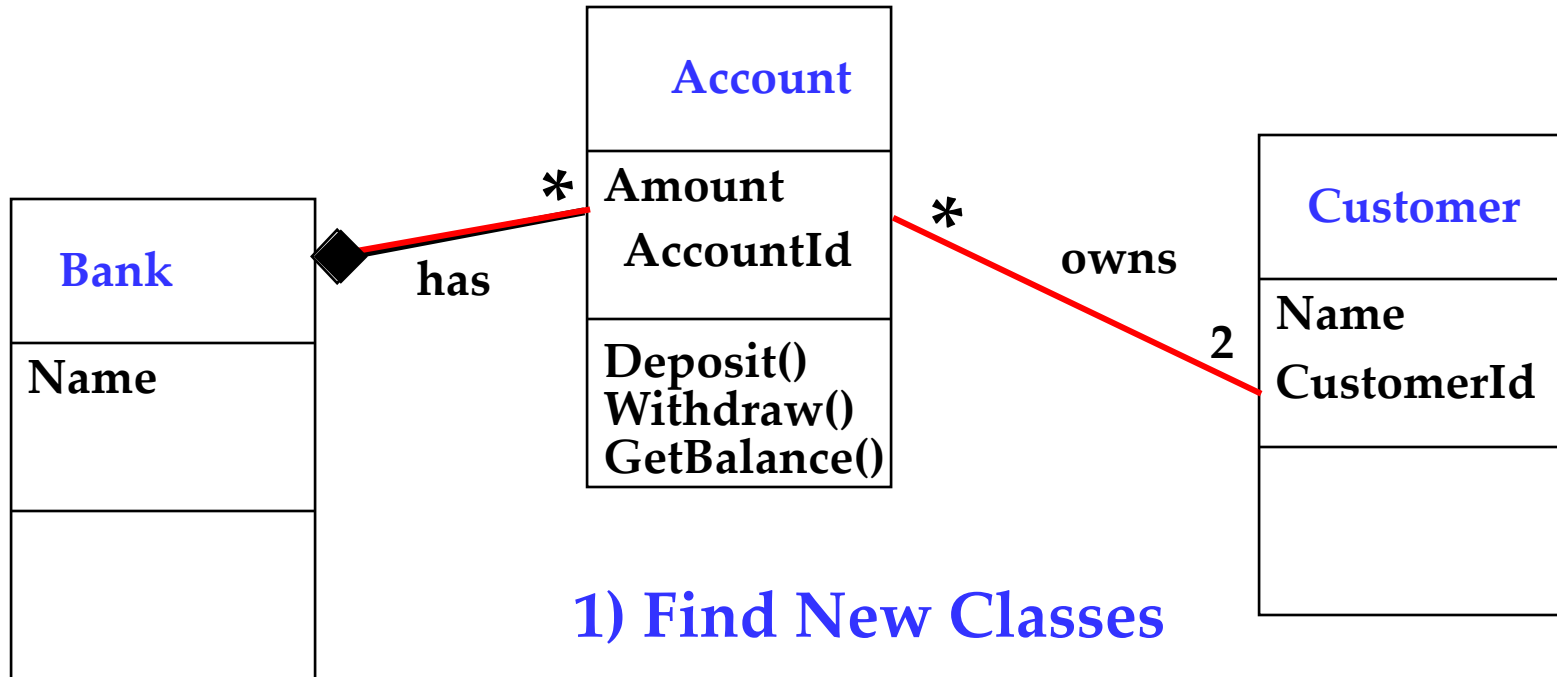
Object Modeling in Practice: More classes



1) Find New Classes

2) Review Names, Attributes and Methods

Object Modeling in Practice: Associations



1) Find New Classes

2) Review Names, Attributes and Methods

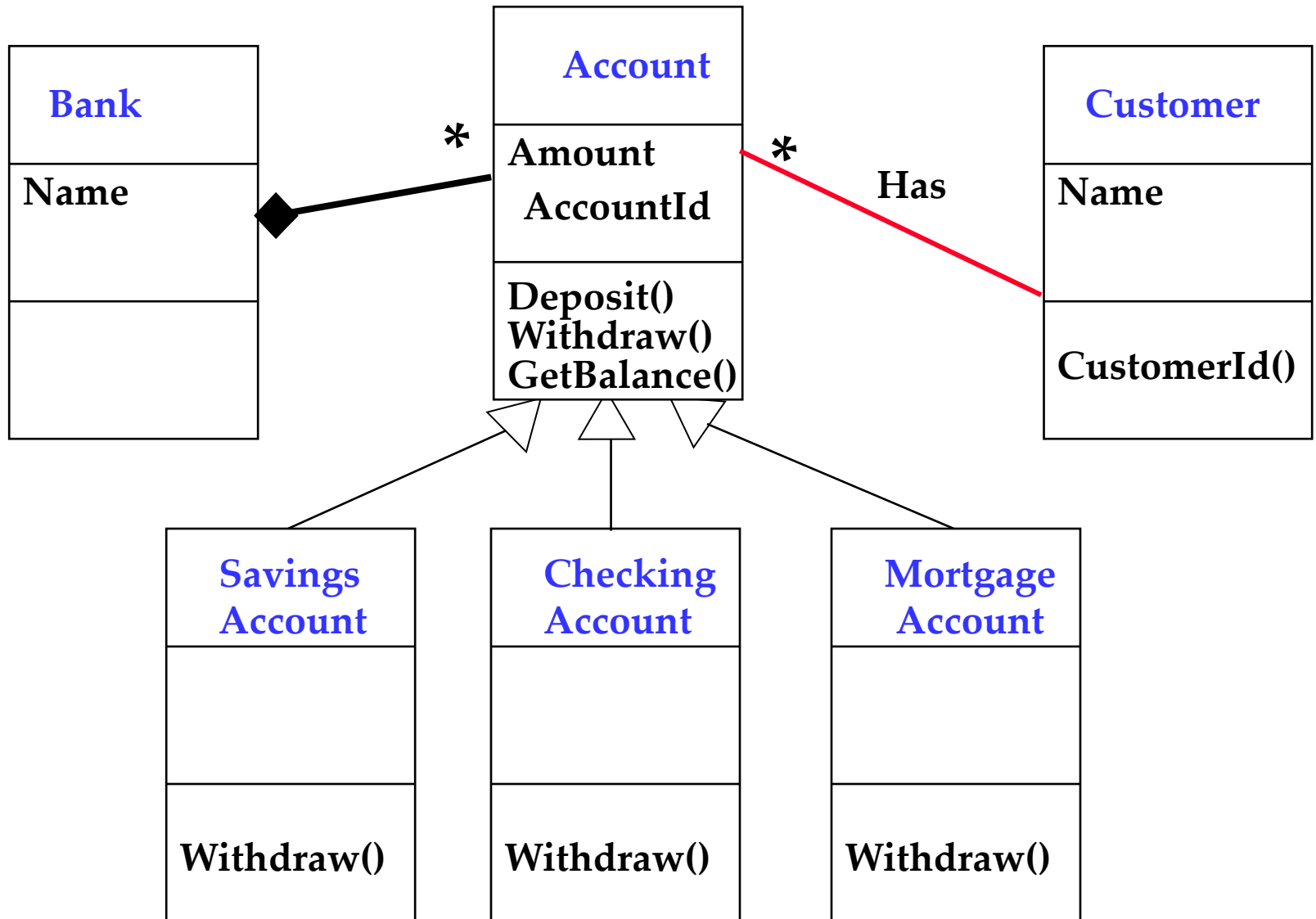
3) Find Associations between Classes

4) Label the generic associations

5) Determine the multiplicity of the associations

6) Review associations

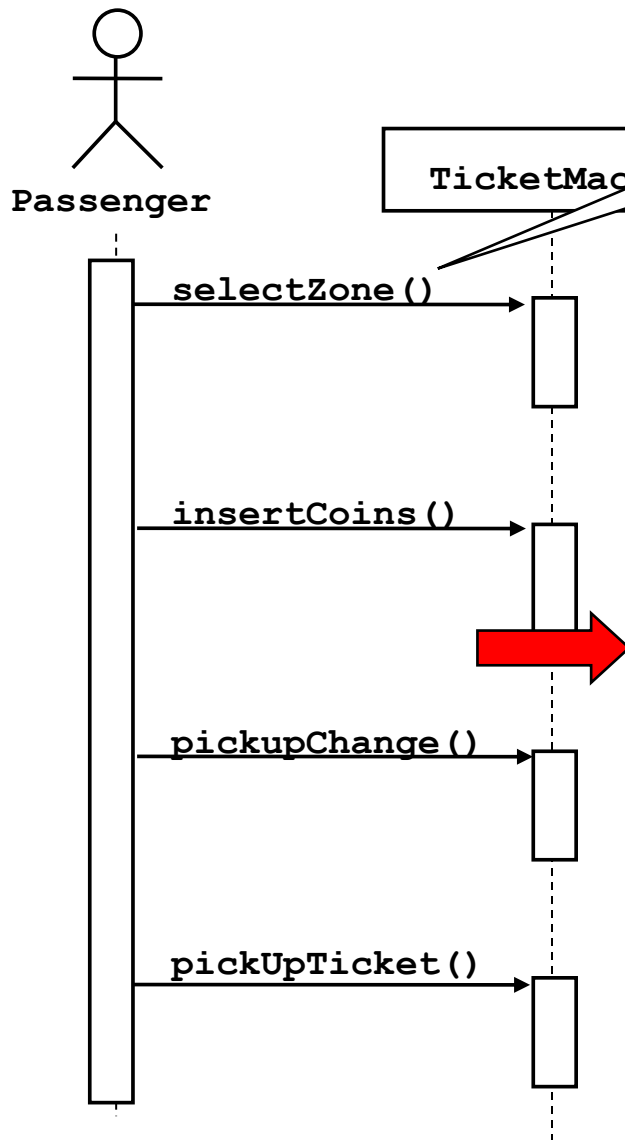
Practice Object Modeling: Find Taxonomies



Sequence Diagrams

Sequence Diagrams

**Focus on
control flow**



Used during analysis

- ♦ To refine use case descriptions
- ♦ to find additional objects (“participating objects”)

Used during system design

fine system interfaces

**Messages ->
Operations on
participating Object**

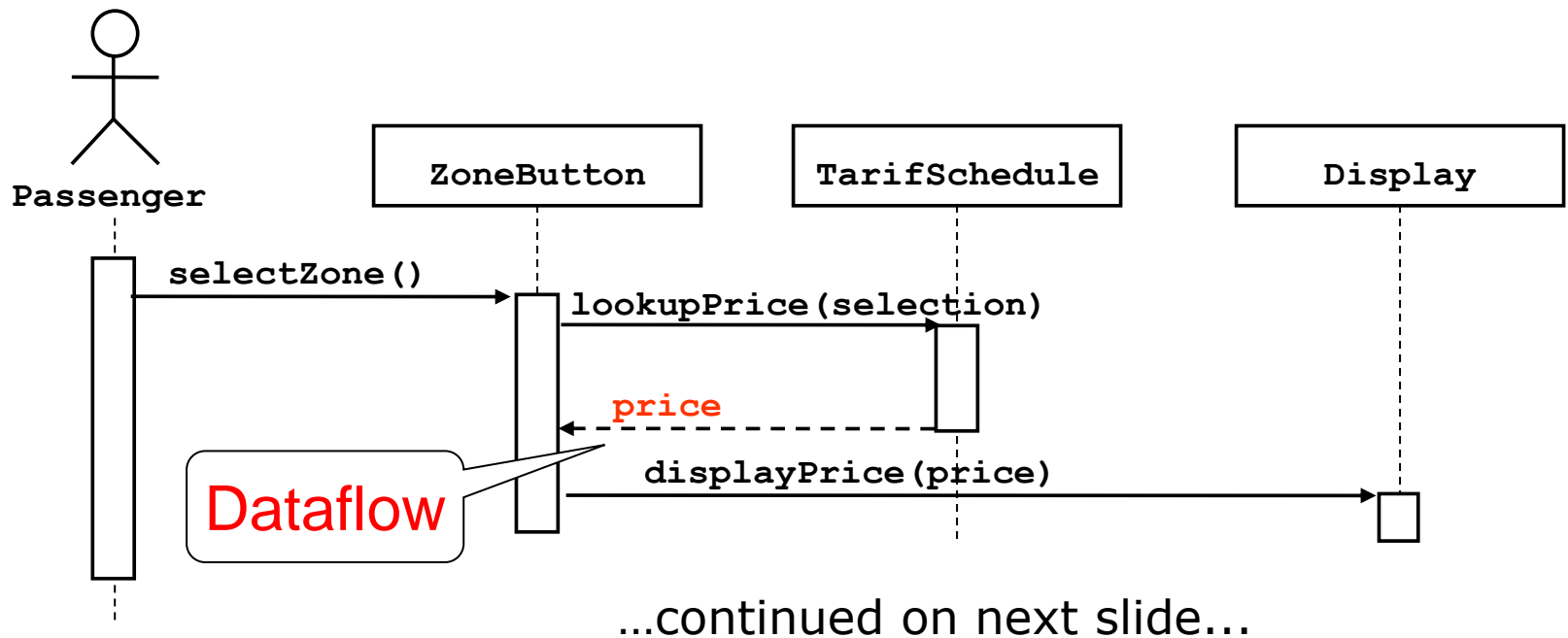
TicketMachine

`selectZone()`
`insertCoins()`
`pickupChange()`
`pickUpTicket()`

Messages are represented by arrows

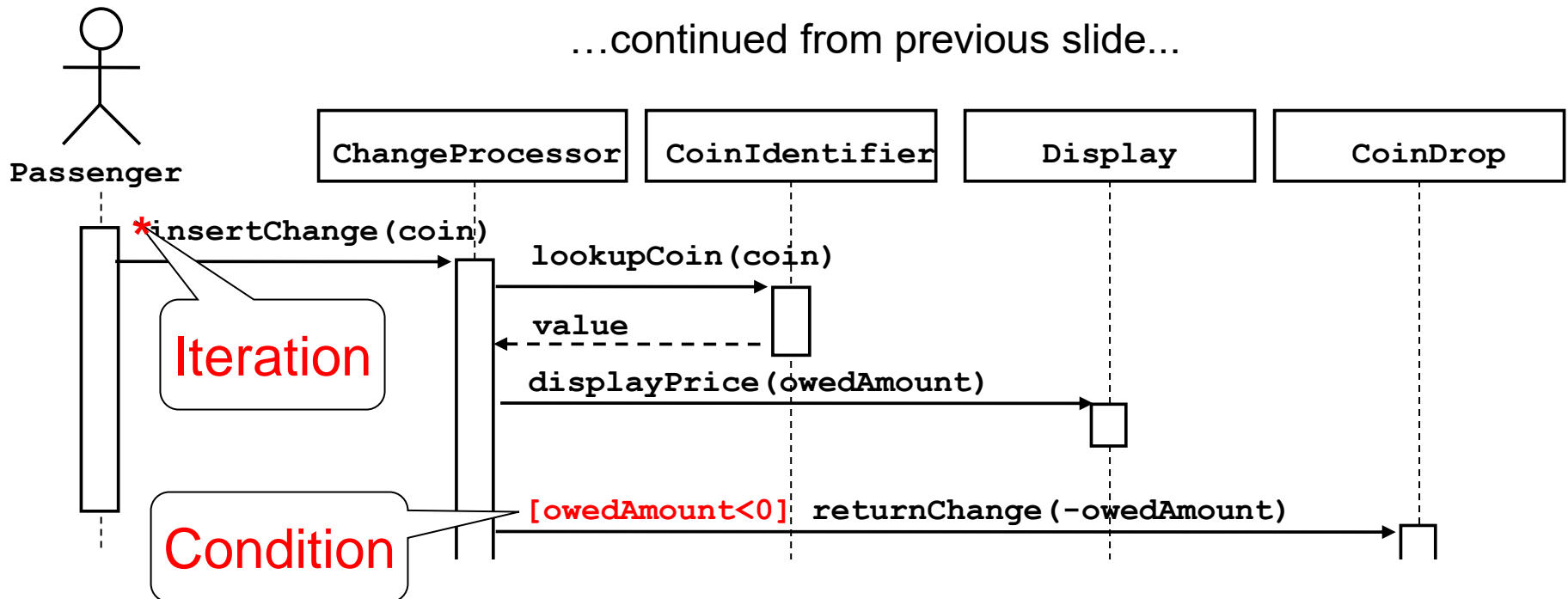
- ♦ *Activations* are represented by narrow rectangles.

Sequence Diagrams can also model the Flow of Data



- ♦ The source of an arrow indicates the activation which sent the message
- ♦ **Horizontal dashed arrows indicate data flow**, for example return results from a message

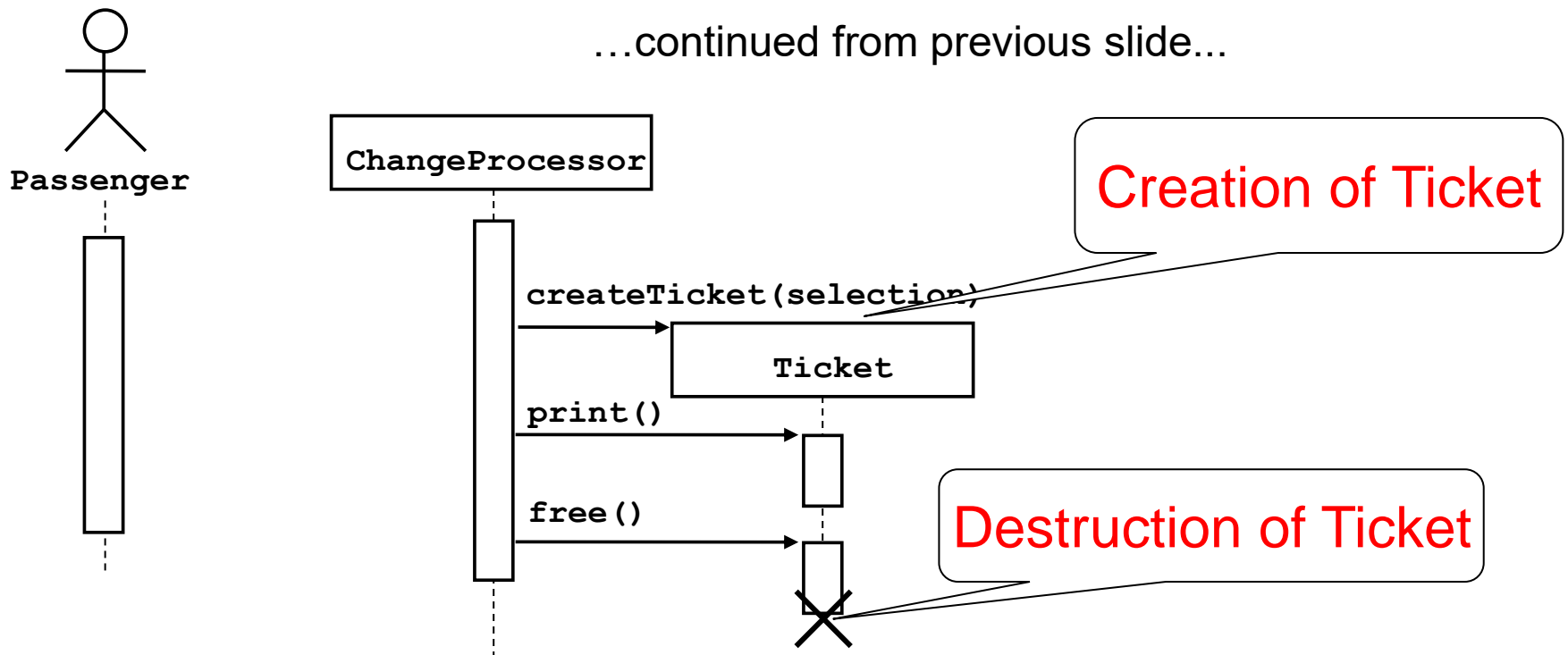
Sequence Diagrams: Iteration & Condition



...continued on next slide...

- ♦ Iteration is denoted by a * preceding the message name
- ♦ Condition is denoted by boolean expression in [] before the message name

Creation and destruction



- ♦ Creation is denoted by a message arrow pointing to the object
- ♦ Destruction is denoted by an X mark at the end of the destruction activation
 - ♦ **In garbage collection environments, destruction can be used to denote the end of the useful life of an object.**

Sequence Diagram Properties

- ◆ UML sequence diagram represent *behavior in terms of interactions*
- ◆ Useful to identify or find missing objects
- ◆ Time consuming to build, but worth the investment
- ◆ Complement the class diagrams (which represent structure).

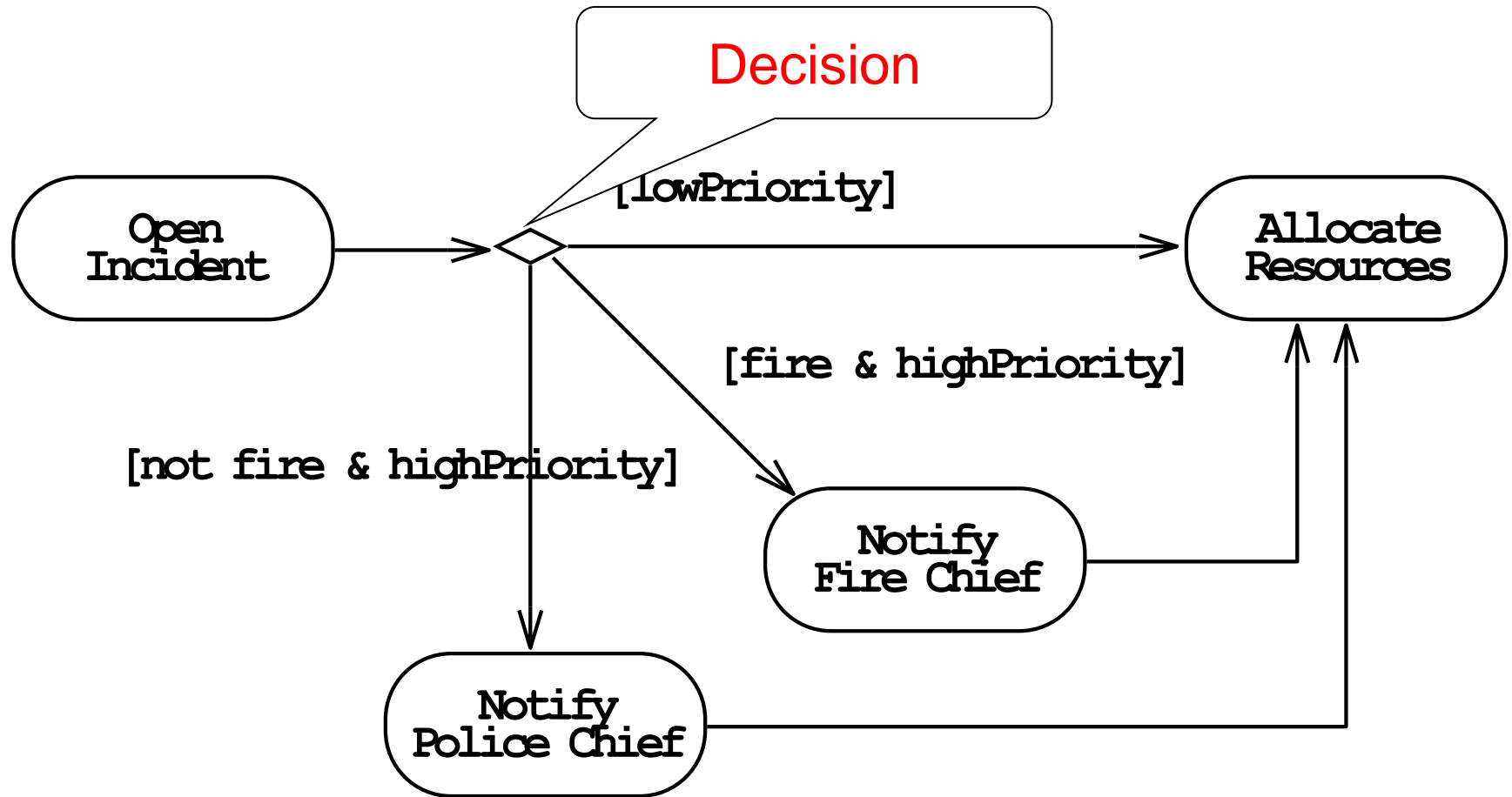
Activity Diagrams

Activity Diagrams

- ♦ An activity diagram is a special case of a state chart diagram
- ♦ The states are activities (“functions”)
- ♦ An activity diagram is useful to depict the workflow in a system

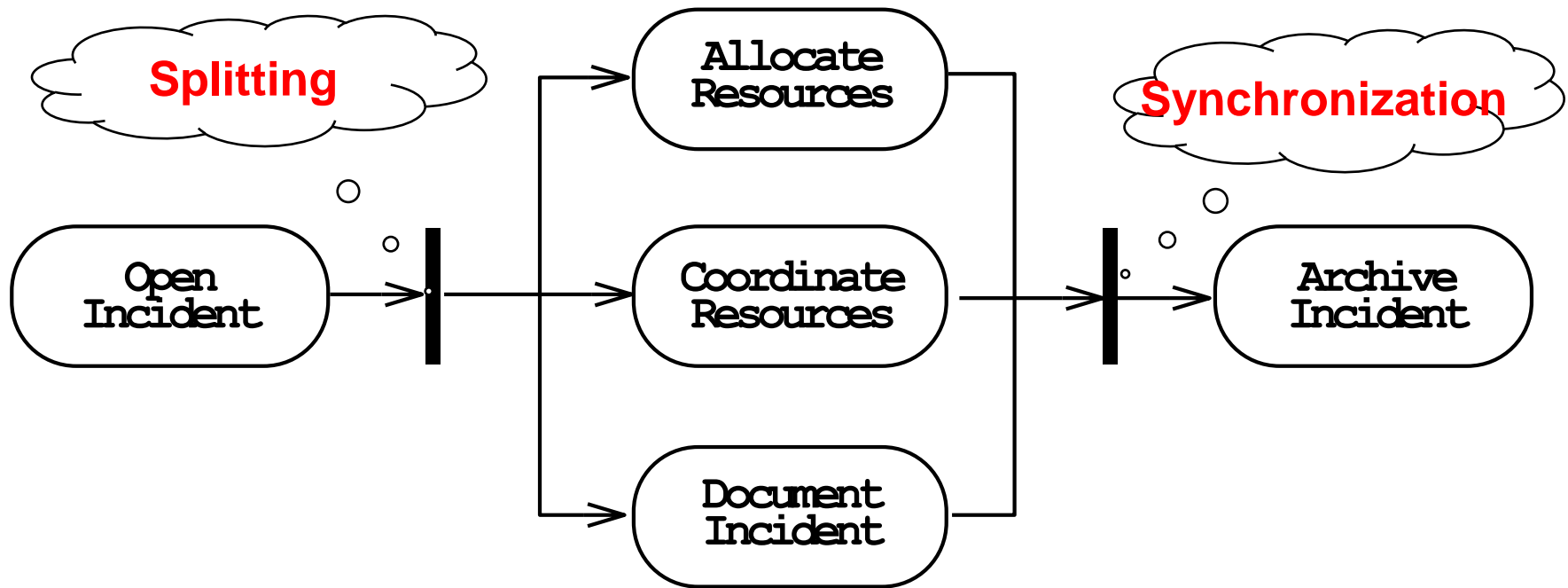


Activity Diagrams allow to model Decisions



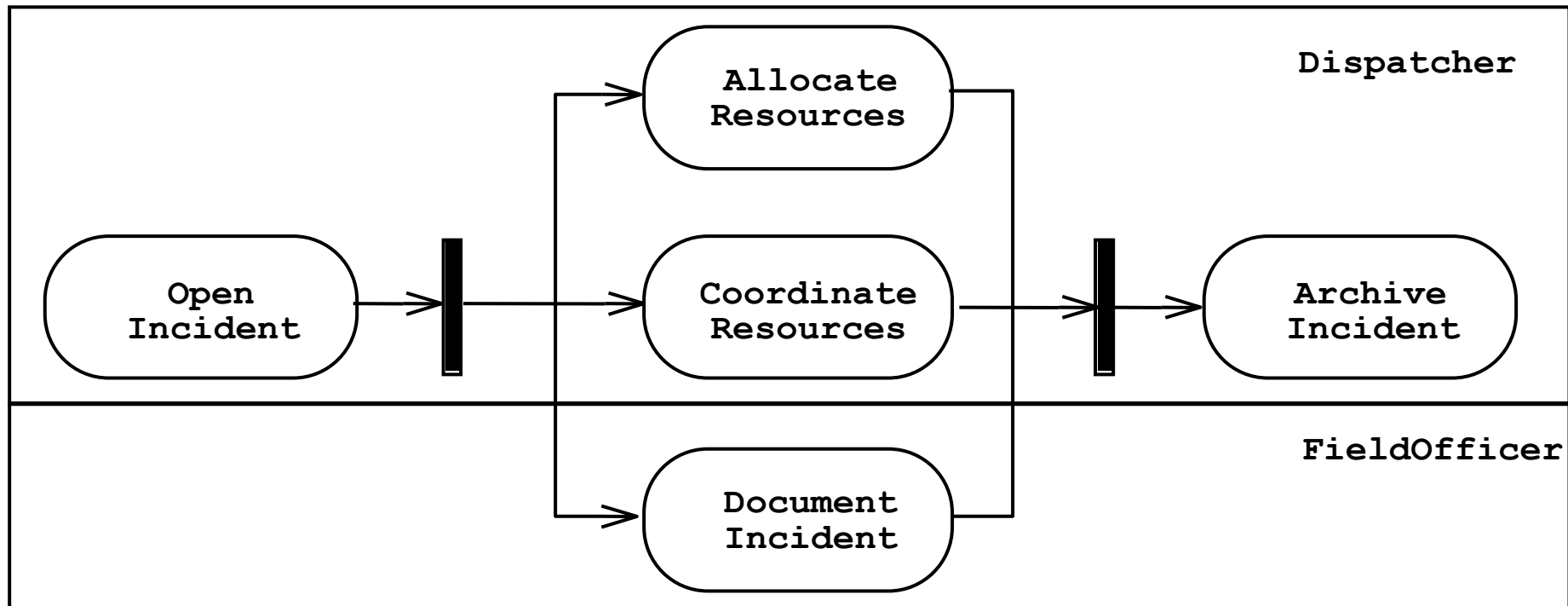
Activity Diagrams can model Concurrency

- ◆ Synchronization of multiple activities
- ◆ Splitting the flow of control into multiple threads



Activity Diagrams: Grouping of Activities

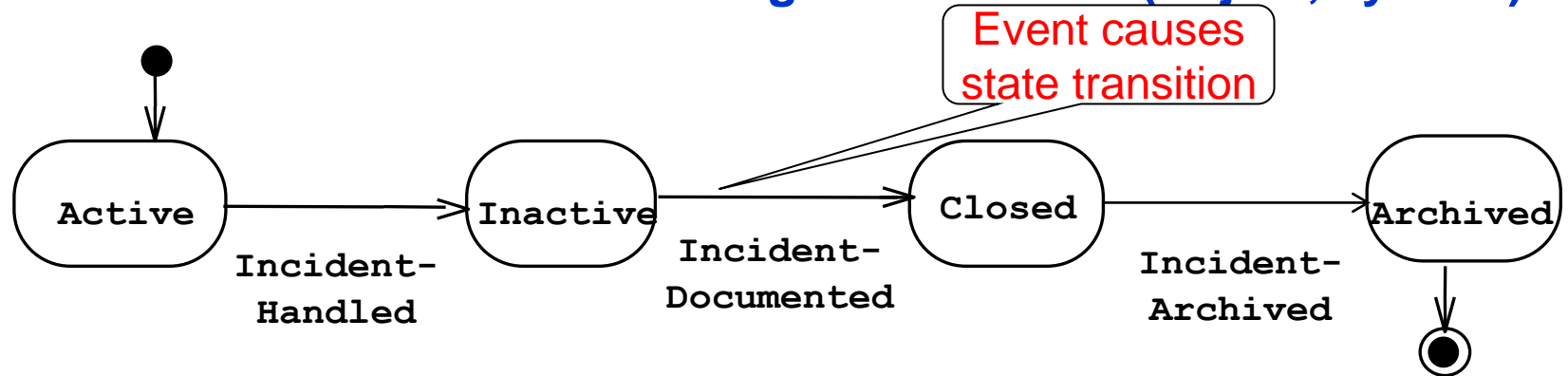
- ♦ Activities may be grouped into **swimlanes** to denote the object or subsystem that implements the activities.



Activity Diagram vs. Statechart Diagram

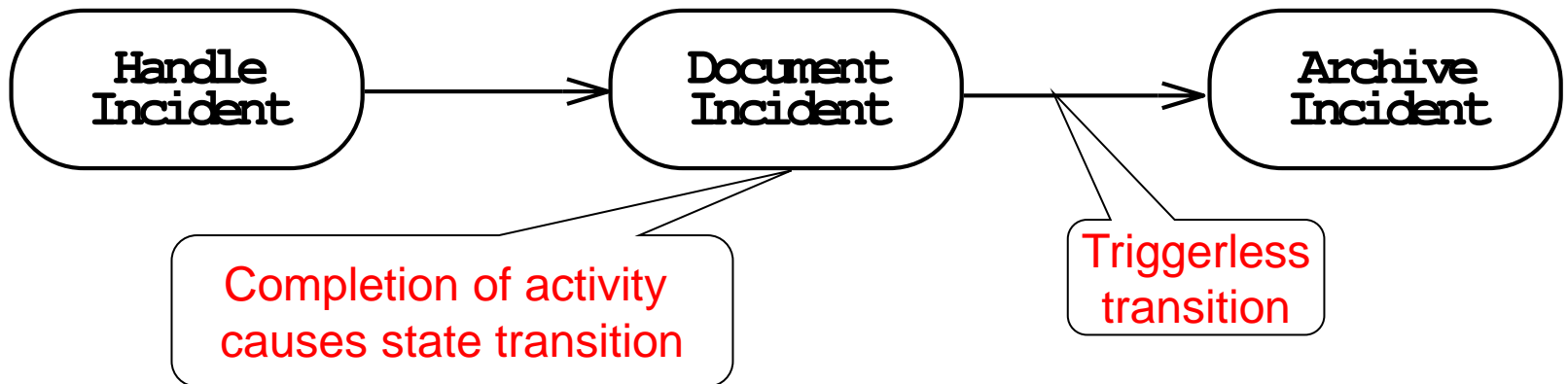
Statechart Diagram for Incident

Focus on the set of attributes of a single abstraction (object, system)



Activity Diagram for Incident

(Focus on dataflow in a system)



UML Summary

- ♦ UML provides a wide variety of notations for representing many aspects of software development
 - ♦ **Powerful, but complex**
- ♦ UML is a programming language
 - ♦ **Can be misused to generate unreadable models**
 - ♦ **Can be misunderstood when using too many exotic features**
- ♦ We concentrated on a few notations:
 - ♦ **Functional model: Use case diagram**
 - ♦ **Object model: class diagram**
 - ♦ **Dynamic model: sequence diagrams, statechart and activity diagrams**