Modeling with UML

Lecture 3

Modeling with UML

Abstraction allows us to ignore unessential details

Thought process where ideas are distanced from object and can be expressed by models

A ***model*** is an abstraction describing a system or a subsystem

UML provides a wide variety of notations for modeling many aspects of software systems

- Functional model: Use case diagram
- Object model: Class diagram
- Dynamic model: Sequence diagrams, statechart

UML First 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

Activity diagrams

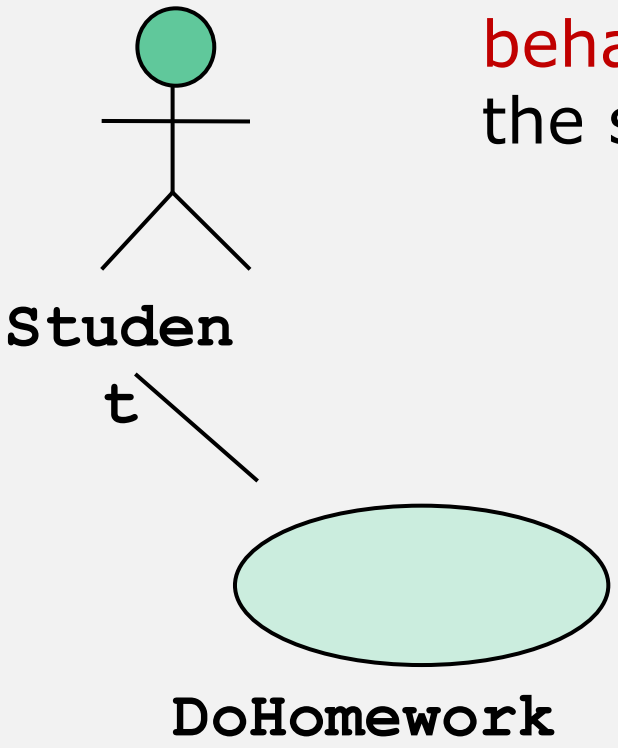
- Describe the dynamic behavior of a system, in particular the workflow.

Statechart diagrams

- Describe the dynamic behavior of an individual object

UML Use Case Diagrams

3 Important Terms



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

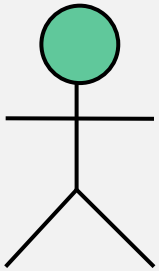
An ***Actor*** represents **a role**, that is, a type of user of the system

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



Student

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:

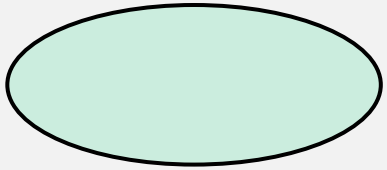
Student: A studying person

Teaching Assistant: Member of teaching staff who supports the instructor.

Optional Description

Name

Use Case



DoHomework

- 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

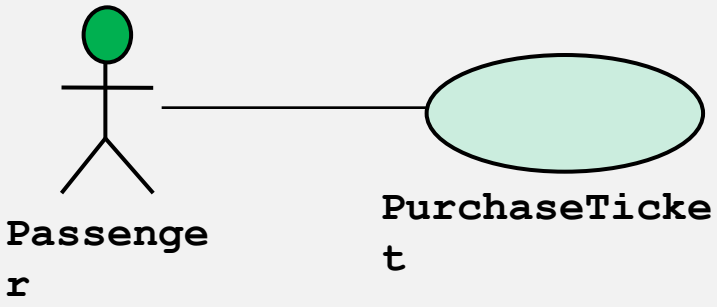
1. *Name:* `Purchase ticket`
2. *Participating actor:* `Passenger`
3. *Entry condition:*

`Passenger` stands in front of ticket distributor

`Passenger` has sufficient money to purchase ticket

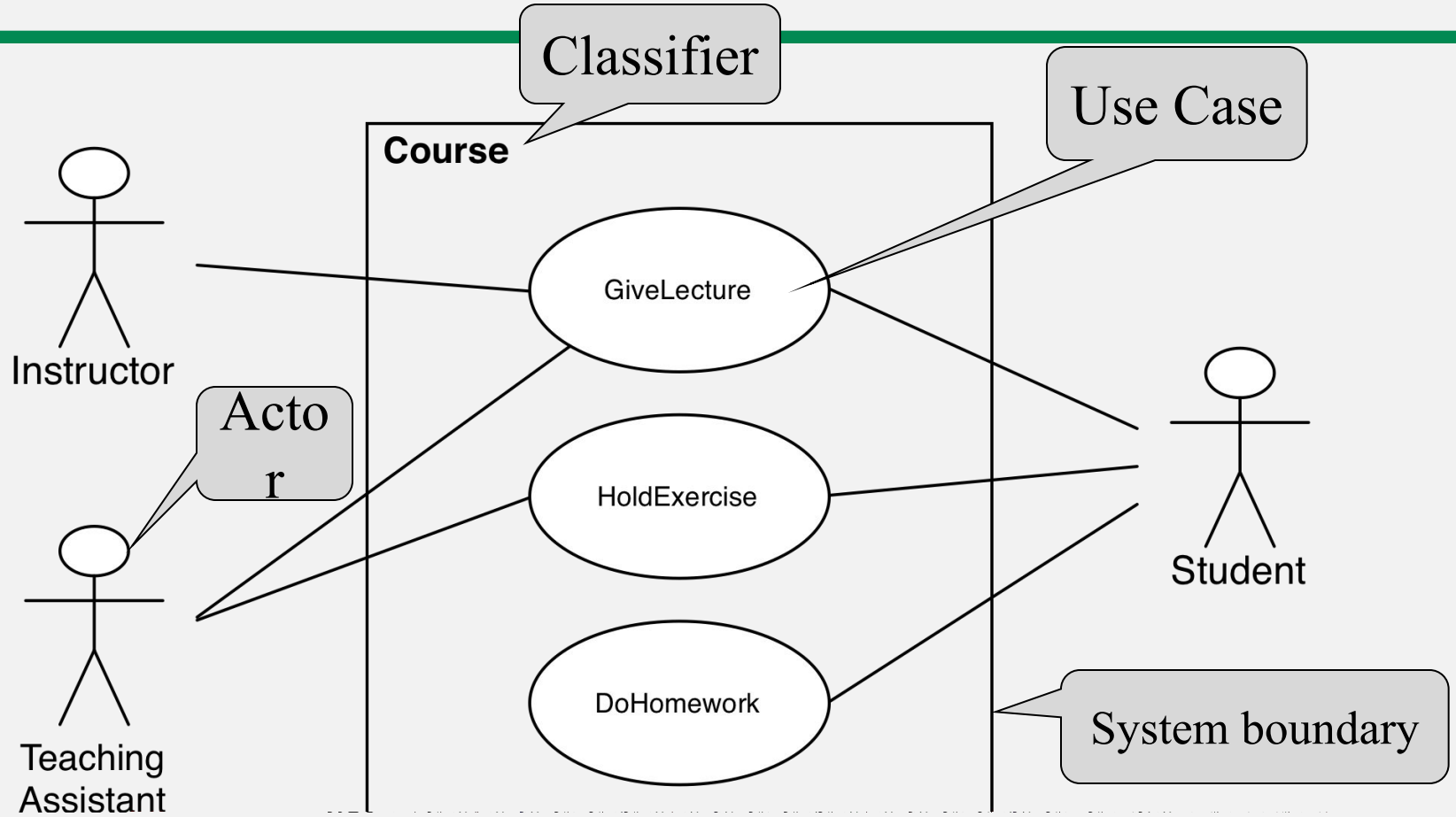
4. *Exit condition:*

`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.*

Use Case Model



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

Uses Cases can be related

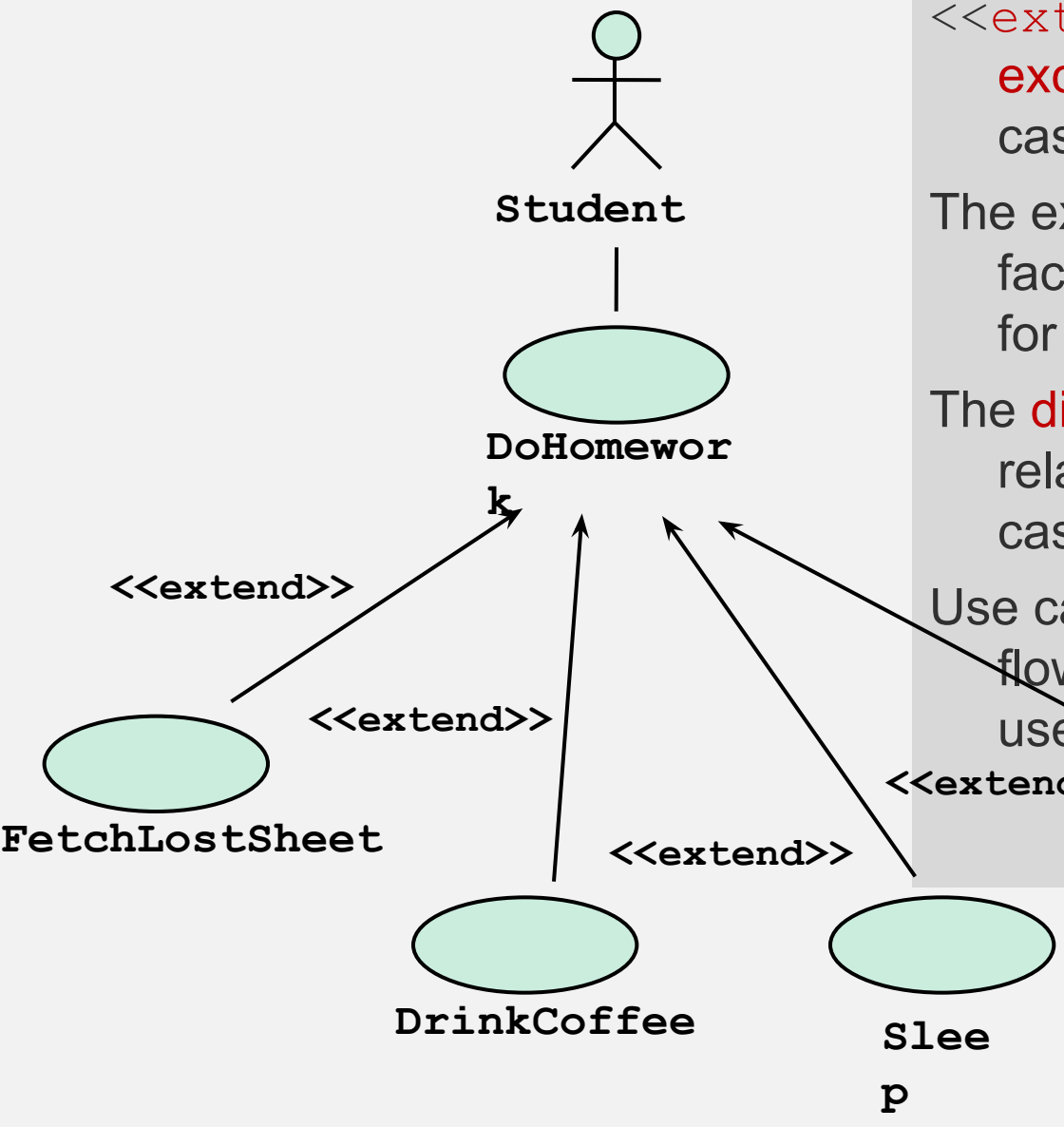
Extends Relationship

To represent seldom invoked use cases or exceptional functionality

Includes Relationship

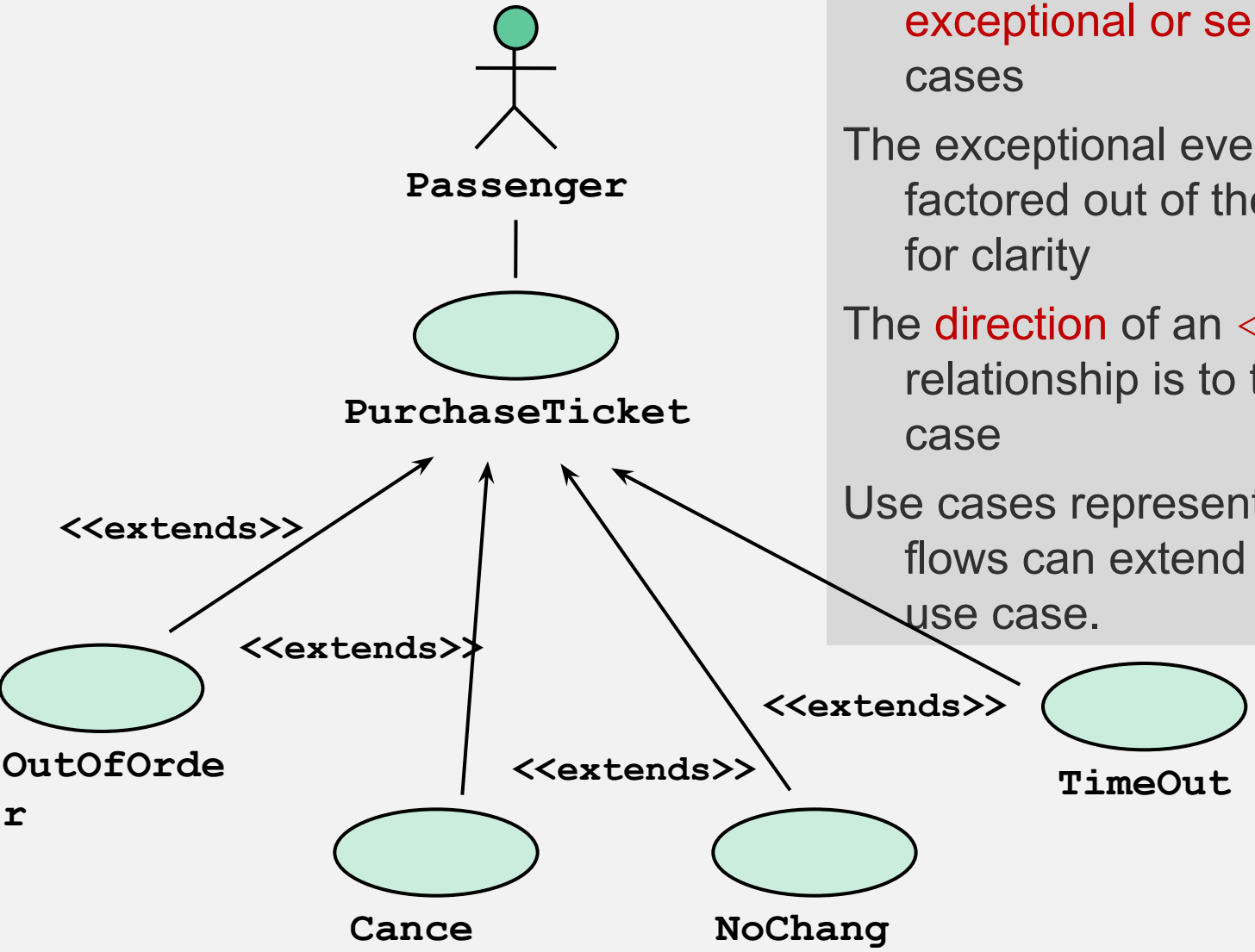
To represent functional behavior common to more than one use case.

The <<extend>> Relationship



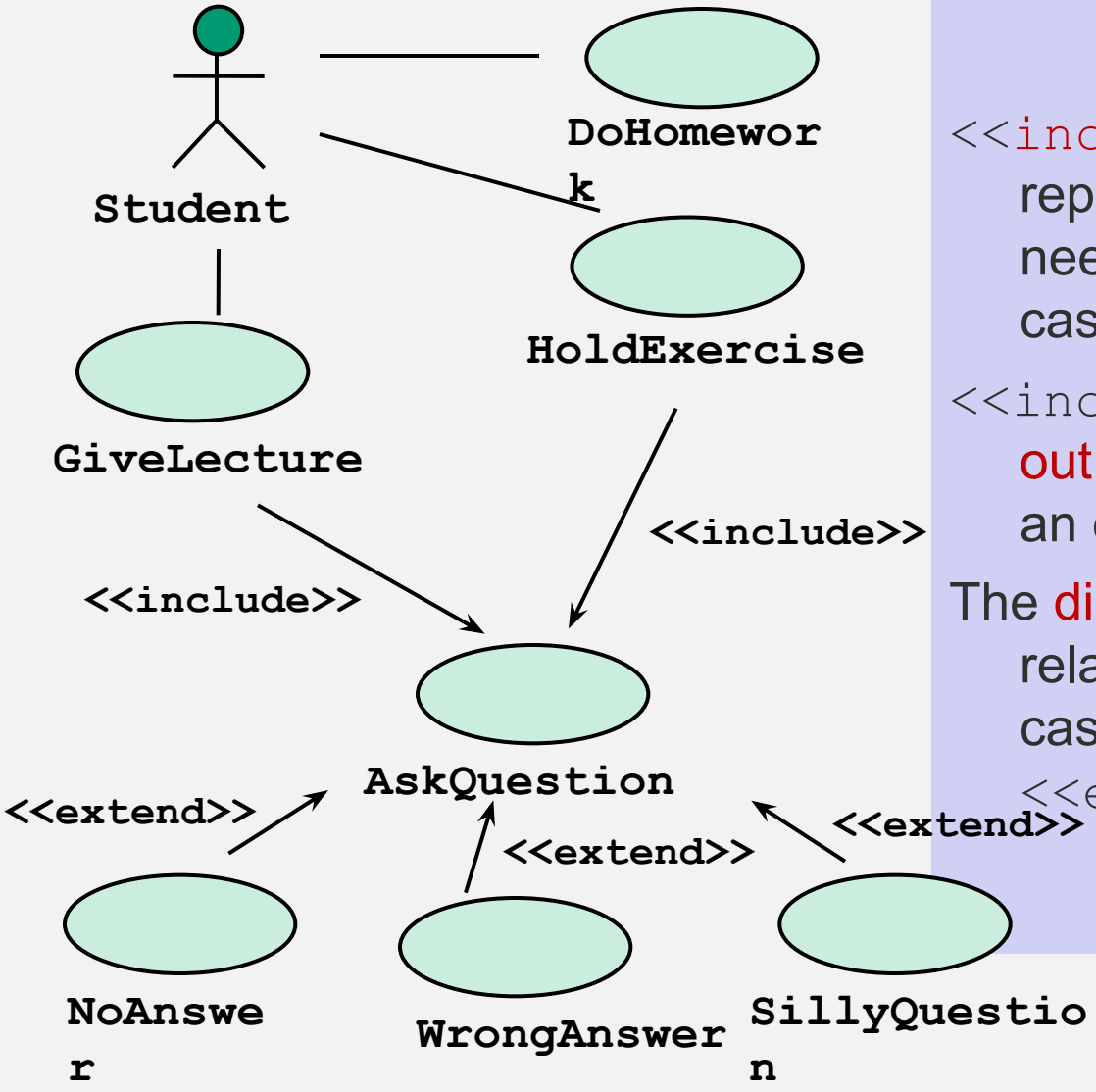
- <<extend>> relationships model exceptional or seldom invoked cases
- The exceptional event flows are factored out of the main event flow for clarity
- The direction of an <<extend>> relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use case.

The <<*extends*>> Relationship



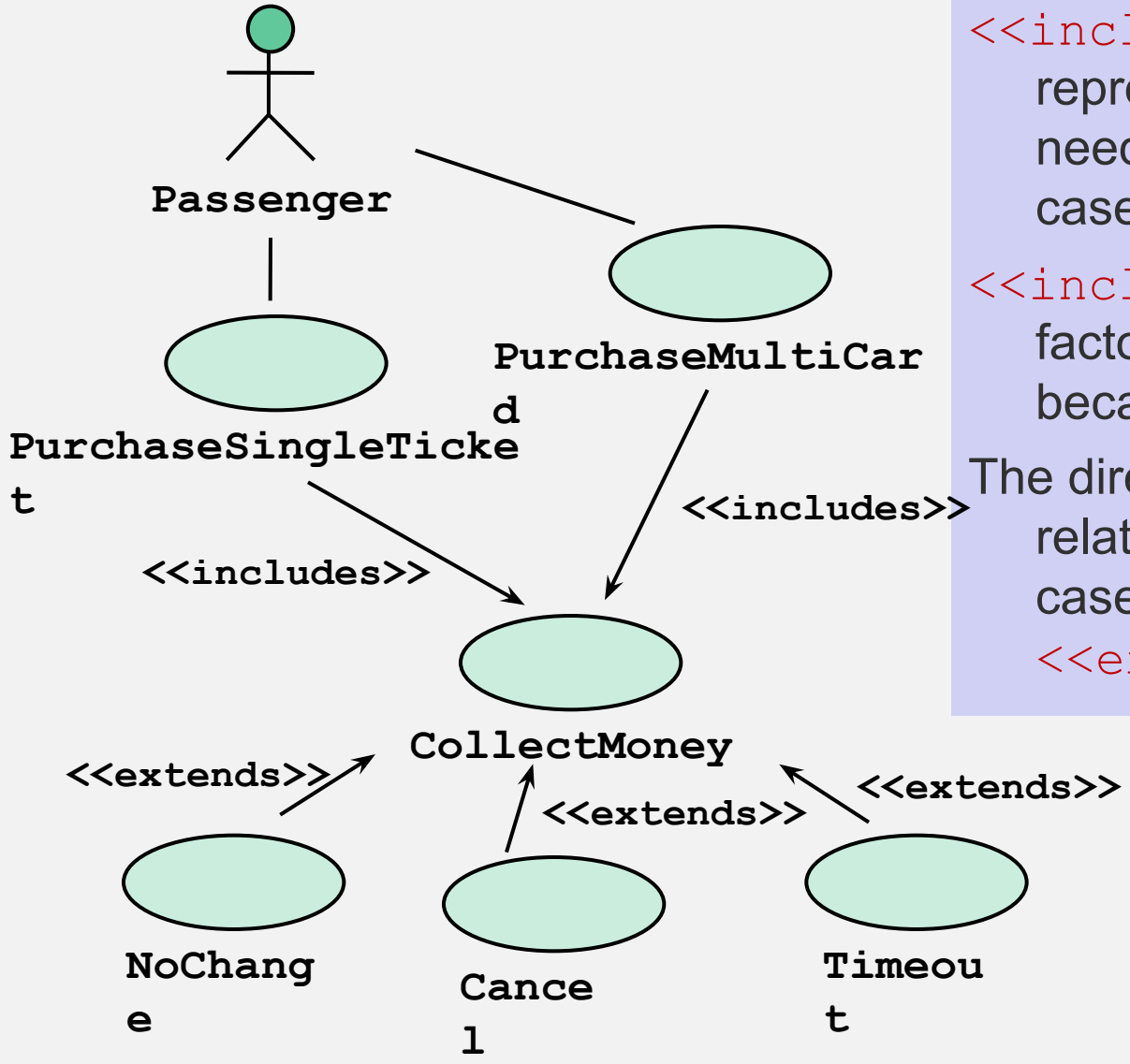
- <<extends>> relationships model exceptional or seldom invoked cases
- The exceptional event flows are factored out of the main event flow for clarity
- The direction of an <<extends>> relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use case.

The <<include>> Relationship



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

The <<includes>> Relationship



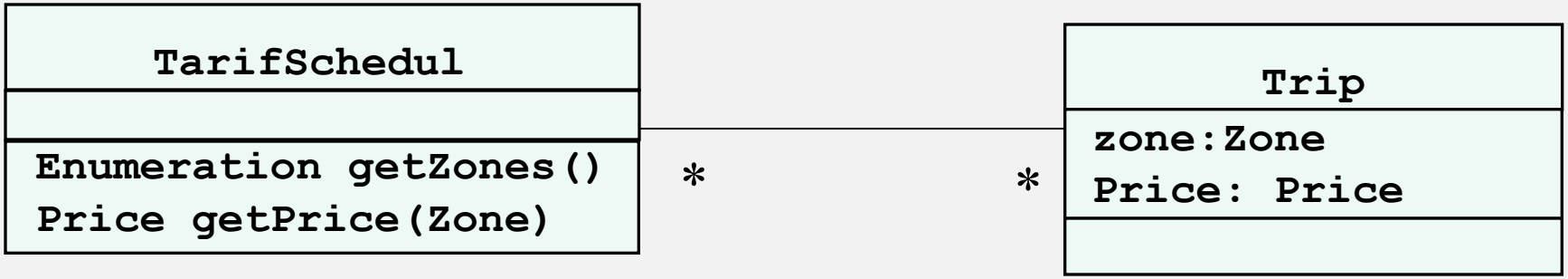
- <<includes>> relationship represents common functionality needed in more than one use case
- <<includes>> behavior is factored out for reuse, not because it is an exception
- The direction of a <<includes>> relationship is to the using use case (unlike the direction of the <<extends>> relationship).

Class Diagrams

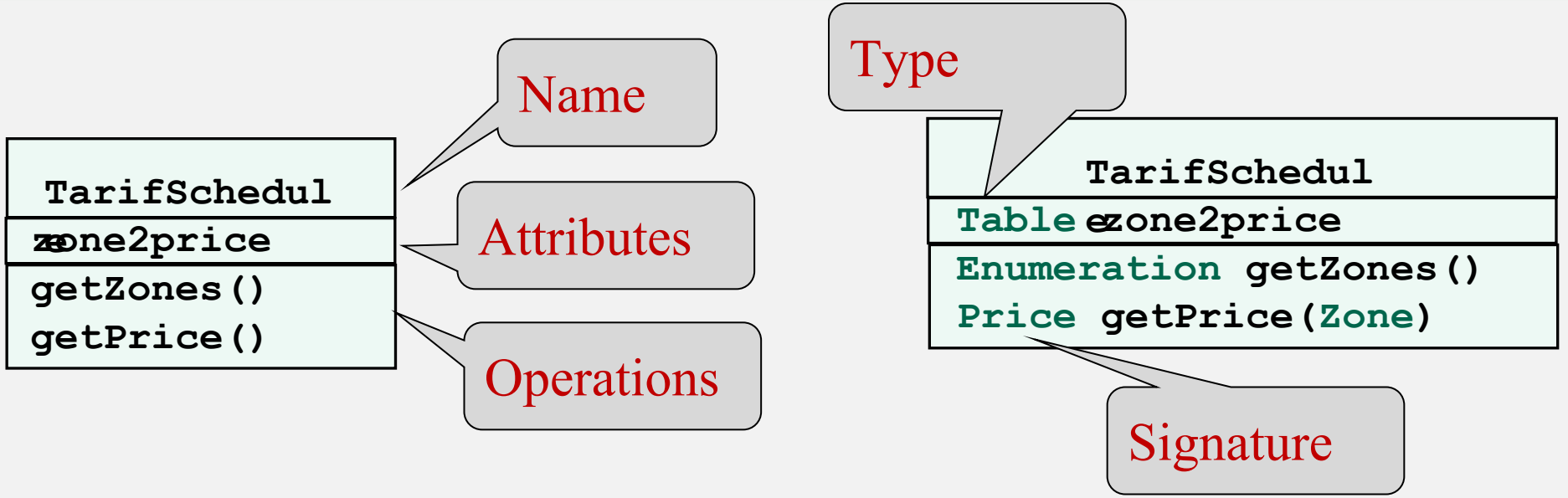
Class diagrams represent the structure of the system

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**

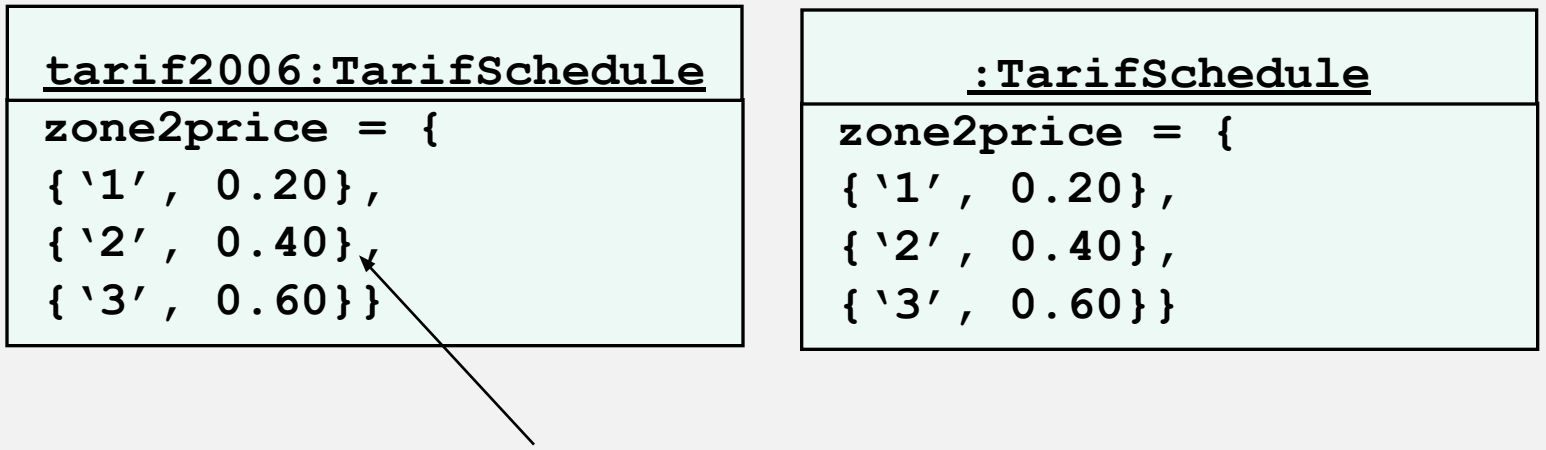
A class encapsulates state ***(attributes)*** and behavior ***(operations)***

Each attribute has a ***type***

Each operation has a ***signature***

The class name is the only mandatory information

Instances



An ***instance*** represents a phenomenon

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

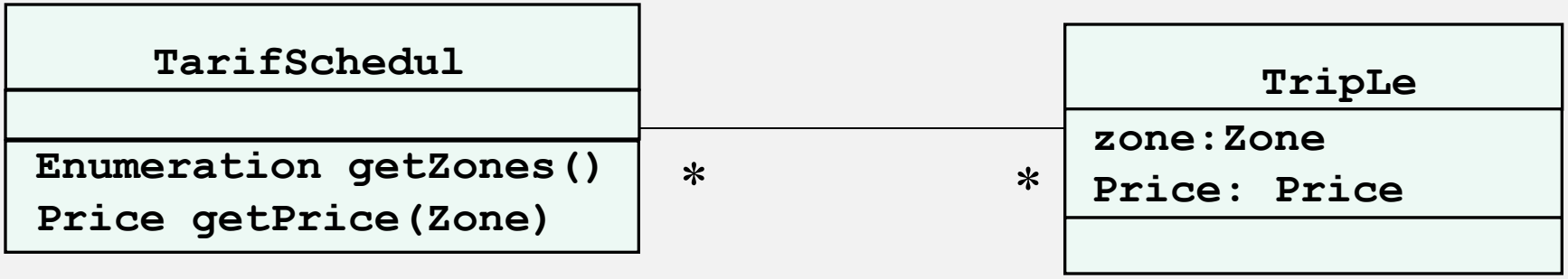
An abstraction modeling an entity in the application or solution domain

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

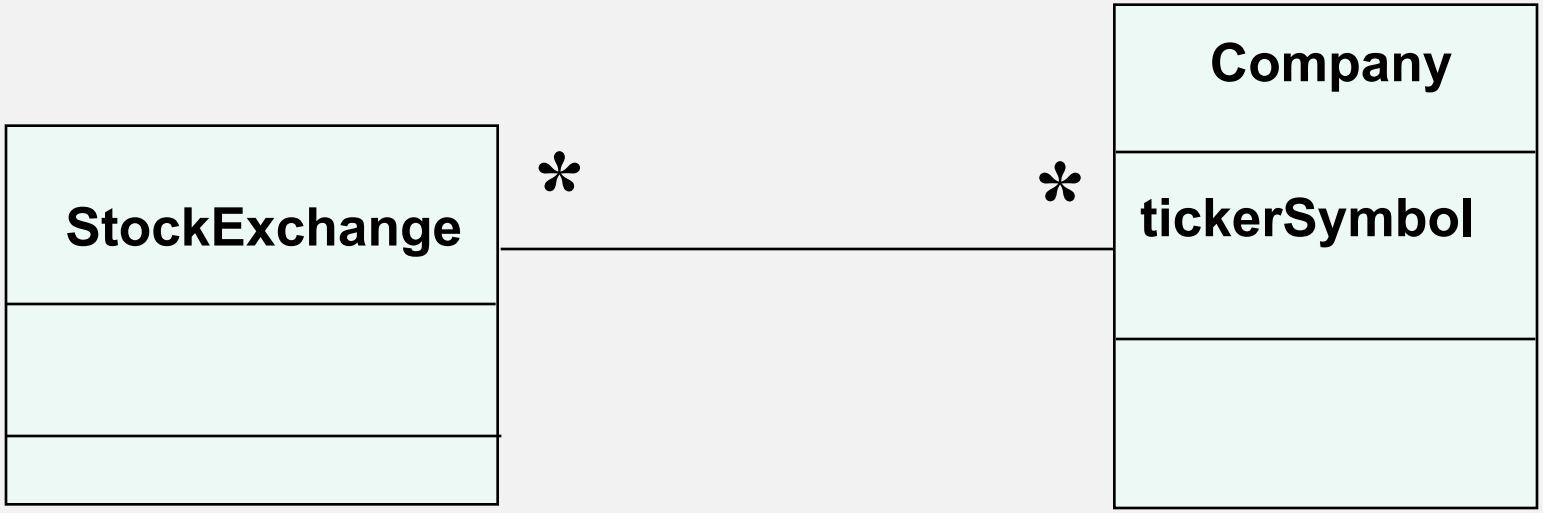
The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

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

1-to-1 association

1-to-many association

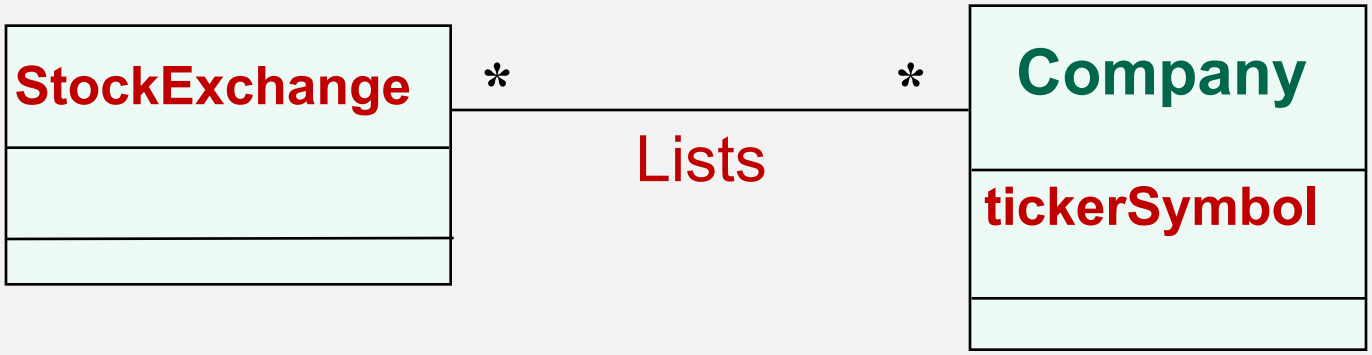
Many-to-Many Associations



From Problem Statement To Object Model

Problem Statement: A **stock exchange lists** *many* **companies**. *Each company is uniquely identified by a* **ticker symbol**

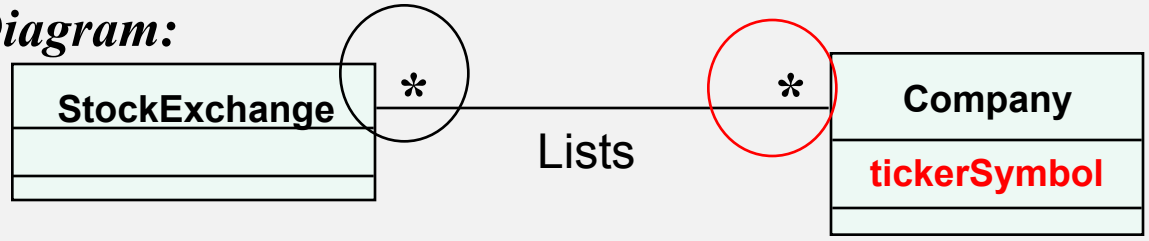
Class Diagram:



From Problem Statement to Code

Problem Statement: A stock exchange lists many companies. Each company is identified by a ticker symbol

Class Diagram:



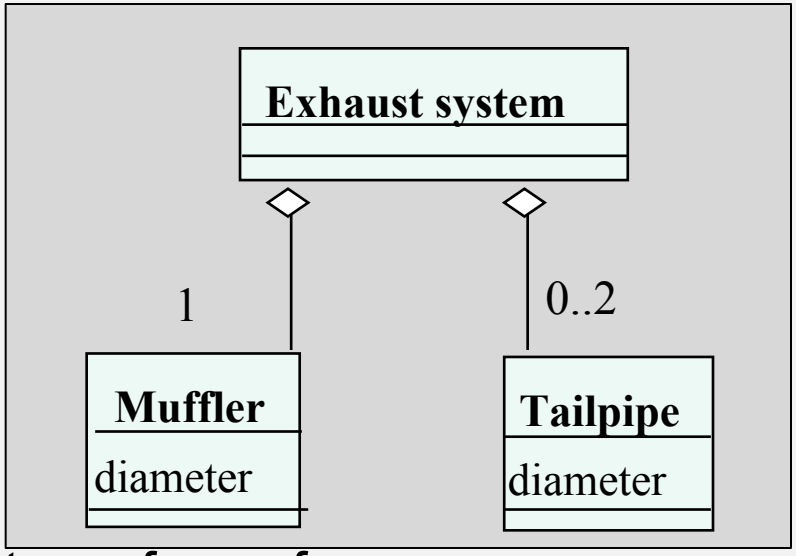

```
public class StockExchange
{
private Vector m_Company = new Vector();
};

public class Company
{
public int m_tickerSymbol;
private Vector m_StockExchange = new Vector();


}
```

Associations are mapped to Attributes!

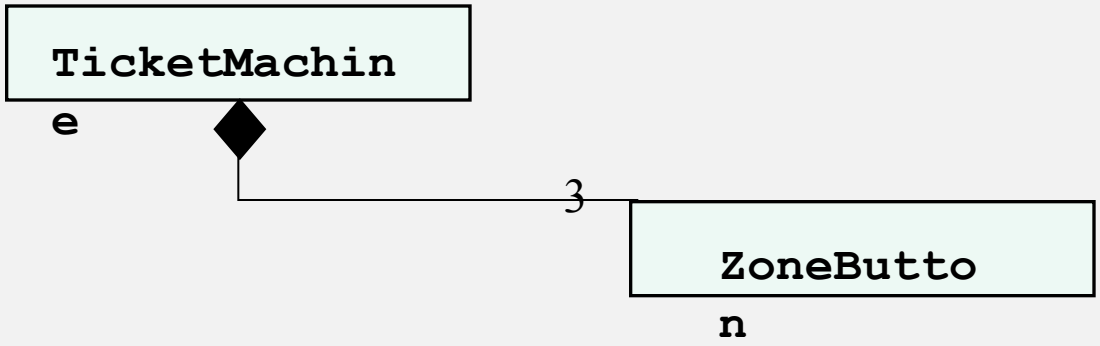
Aggregation

An **aggregation** is a special case of association denoting a "consists-of" hierarchy

The ***aggregate*** is the parent class, the components are the children classes

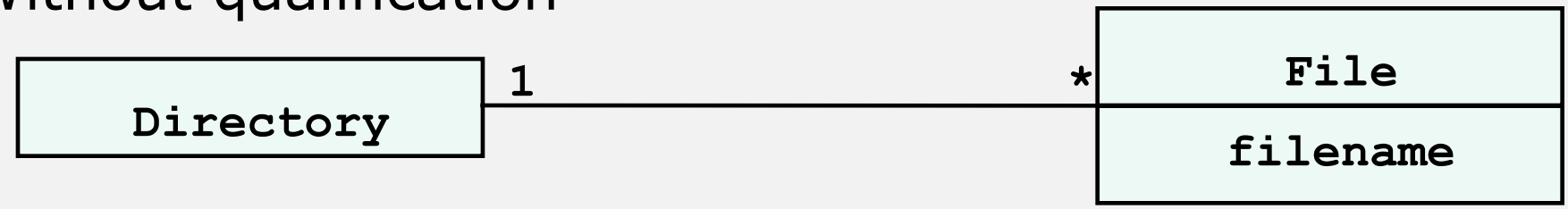


A solid diamond denotes ***composition***: A strong form of aggregation where the *life time of the component instances* is controlled by the aggregate. That is, the parts don't exist on their won ("the whole controls/destroys the parts")

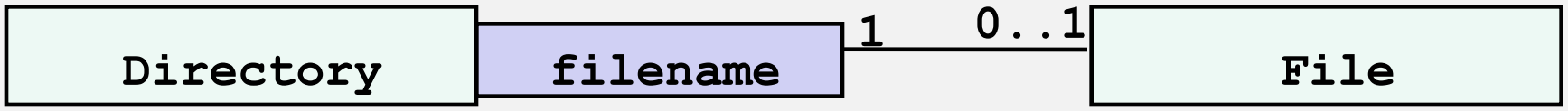


Qualifiers

Without qualification

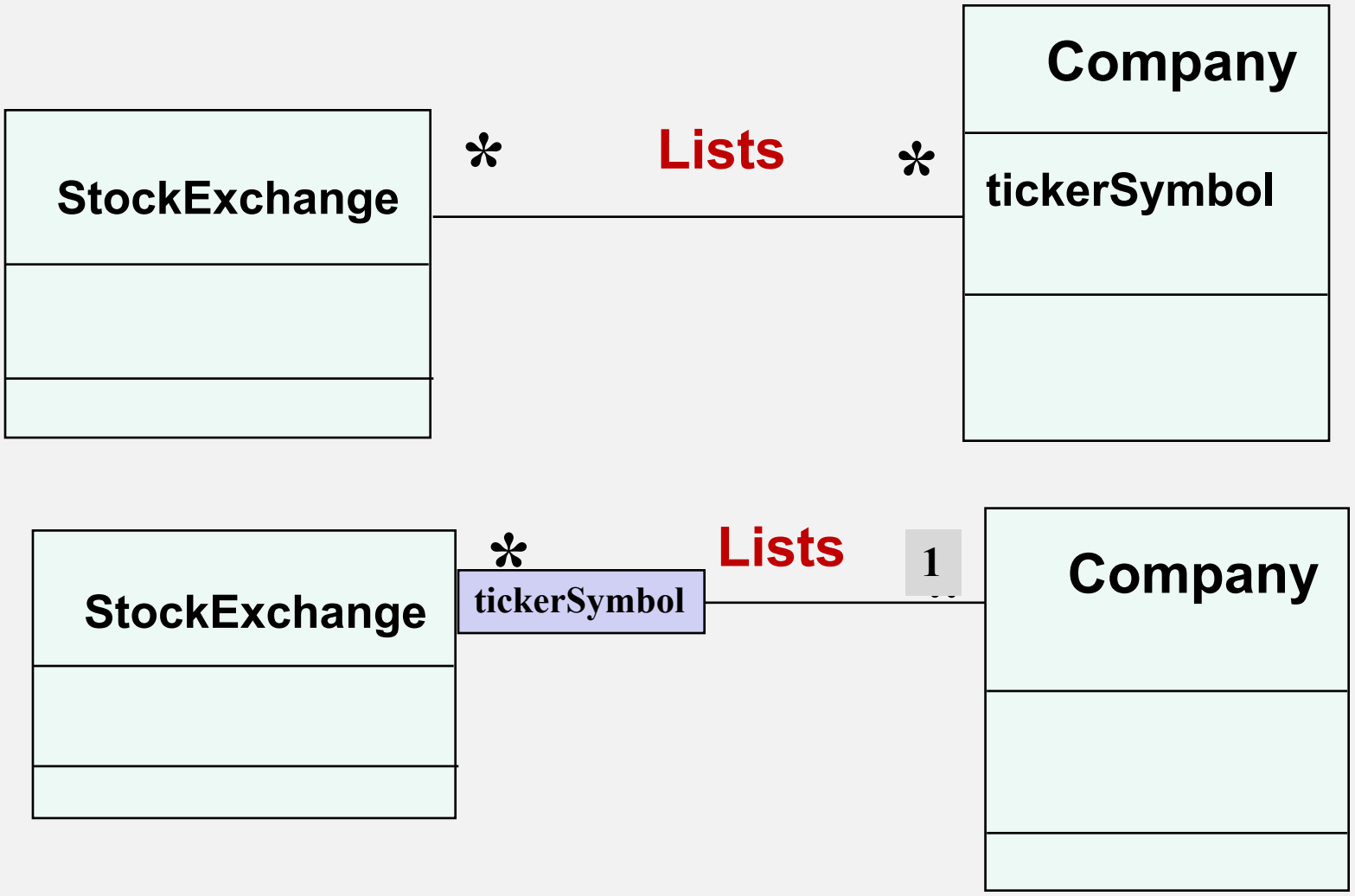


With qualification

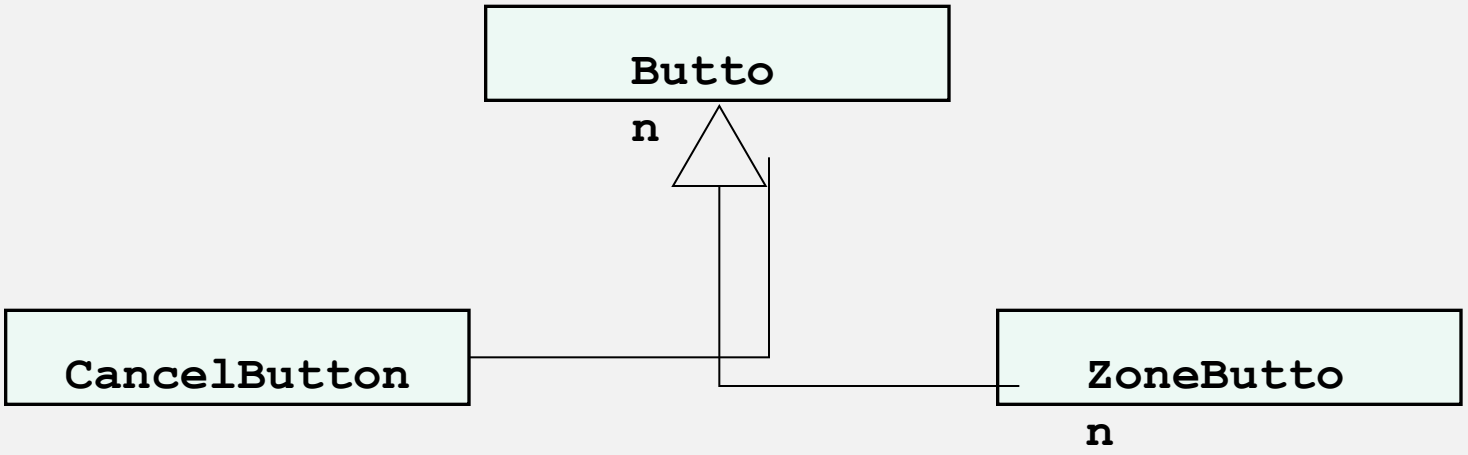


Qualifiers can be used to reduce the multiplicity of an association

Qualification: Another Example



Inheritance



Inheritance is another special case of an association denoting a "kind-of" hierarchy

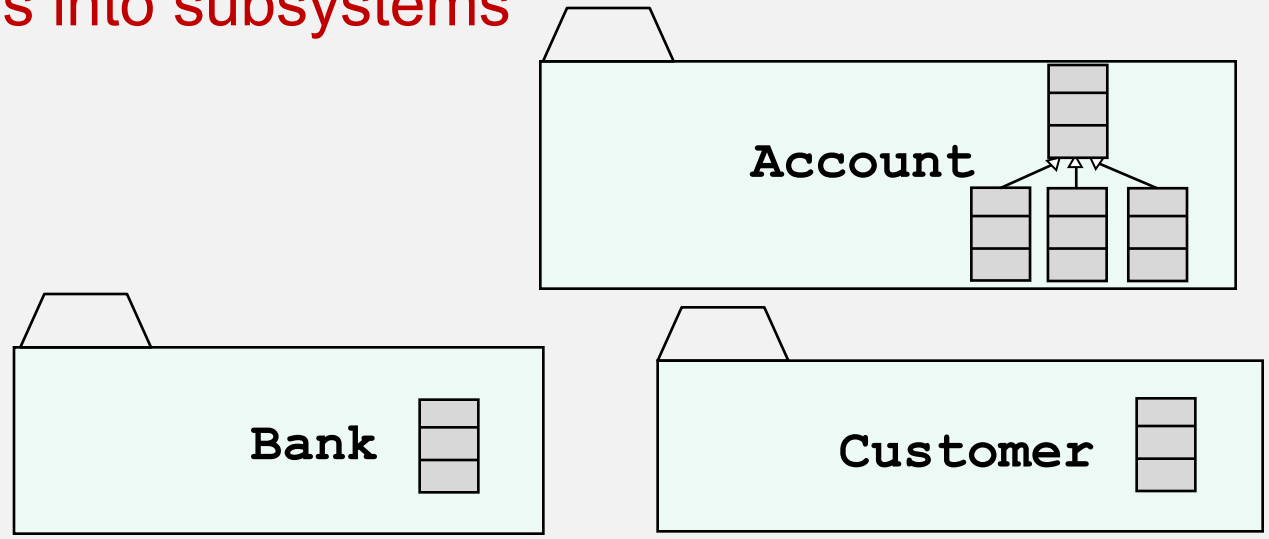
Inheritance simplifies the analysis model by introducing a taxonomy

The **children classes** inherit the attributes and operations of the **parent class**.

Packages

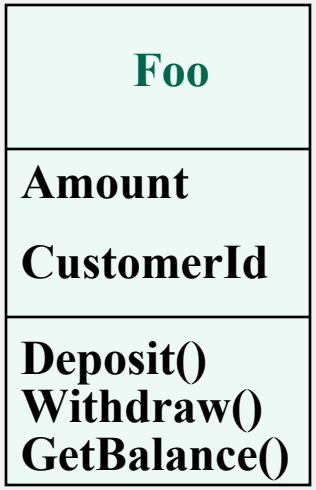
Packages help you to organize UML models to increase their readability

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**.

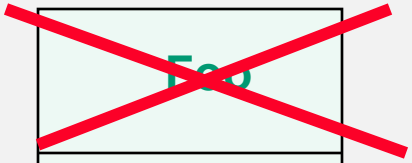
Object Modeling in Practice

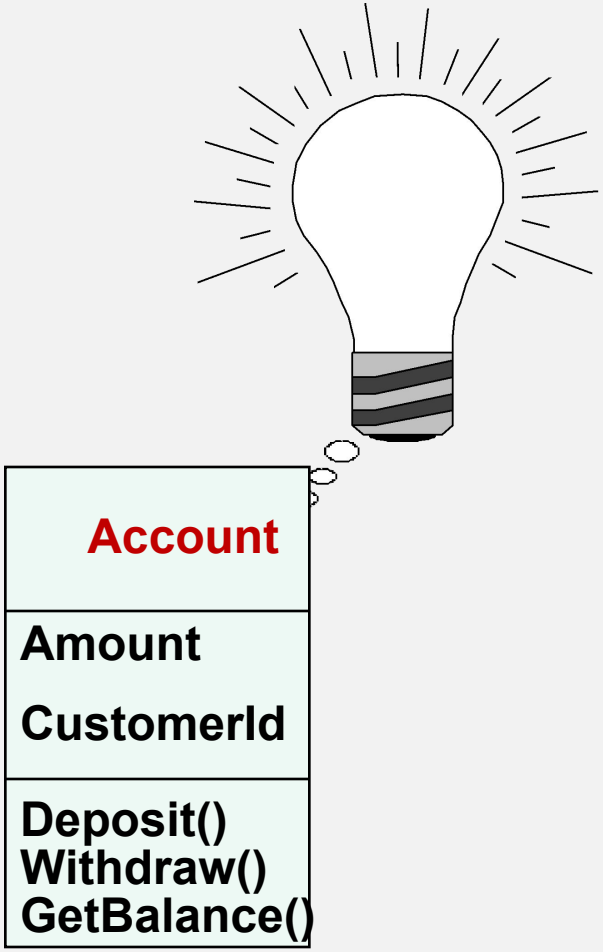


Class Identification: Name of Class, Attributes and Methods

Is Foo the right name?

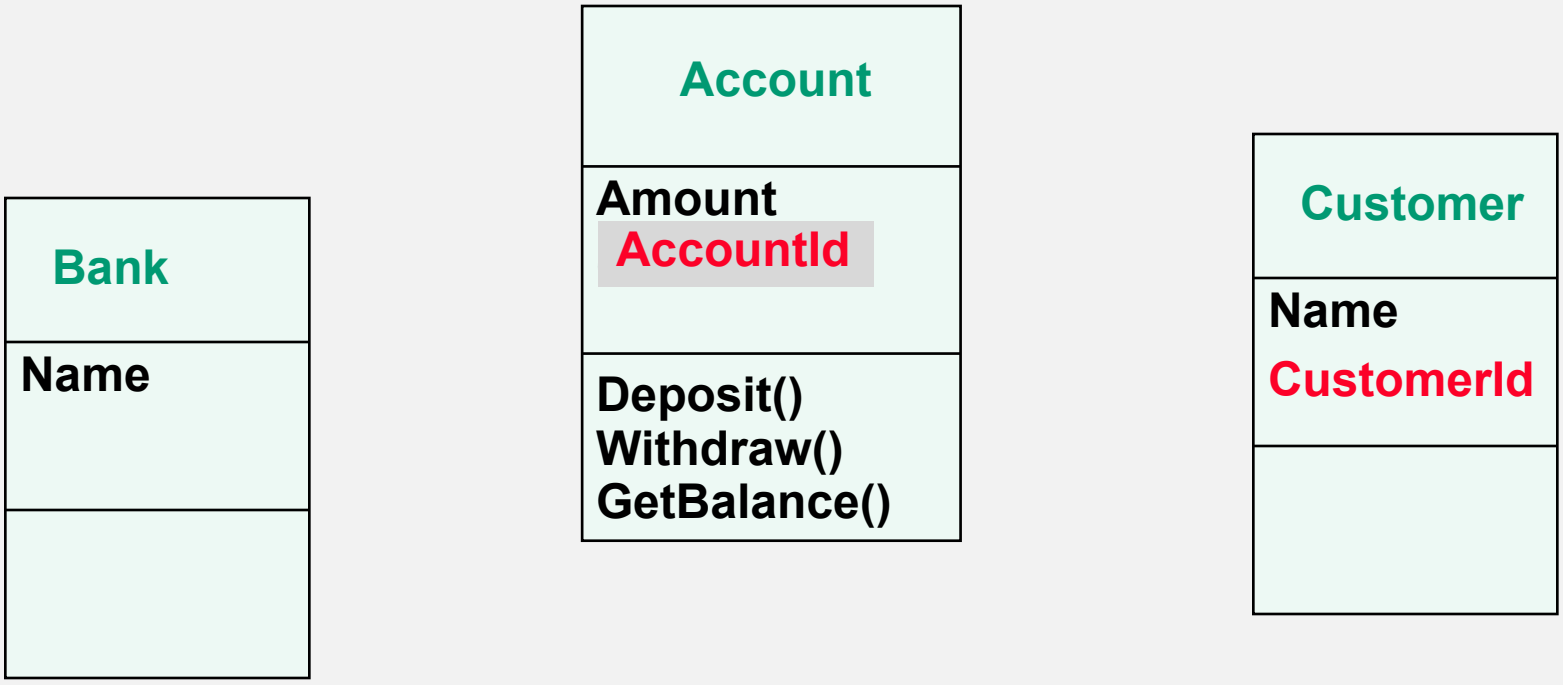
Object Modeling in Practice: Brainstorming





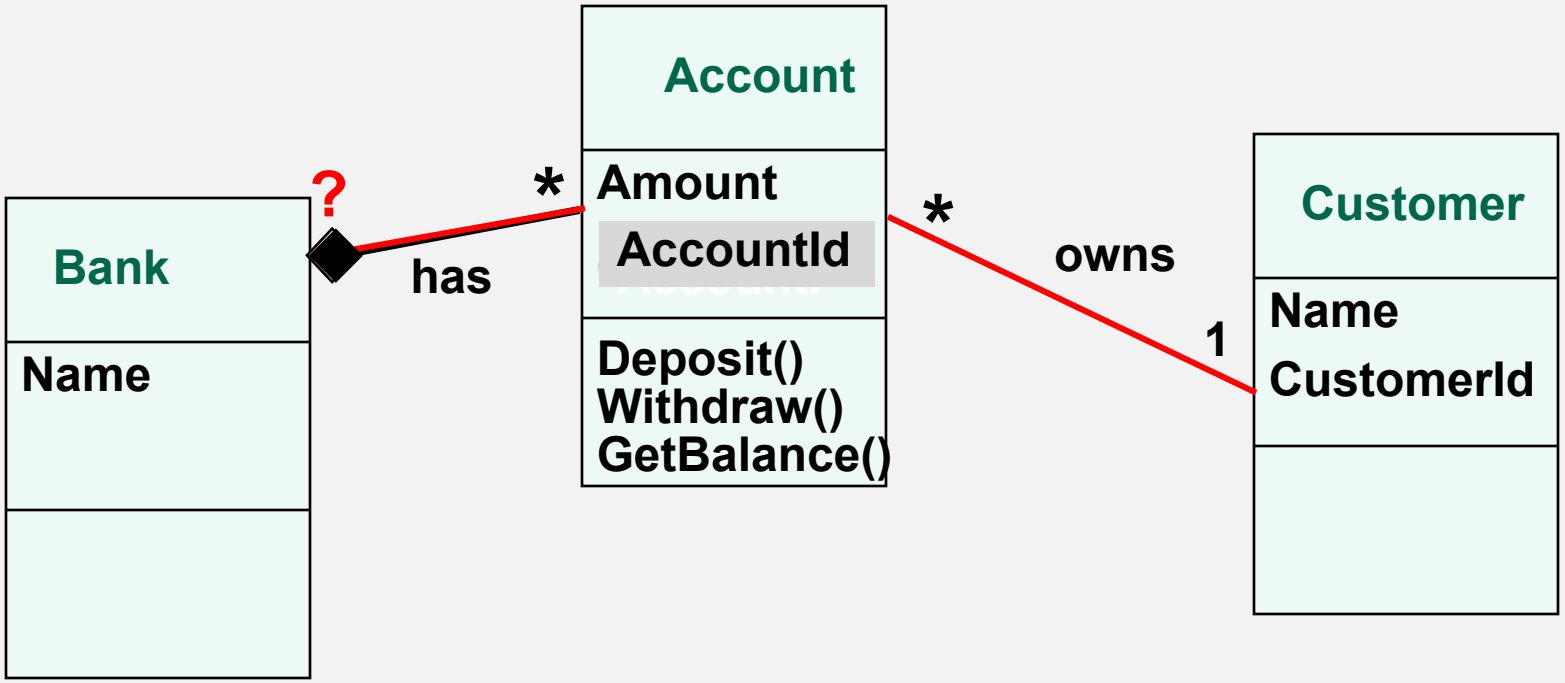
Is Foo the right name?

Object Modeling in Practice: More classes



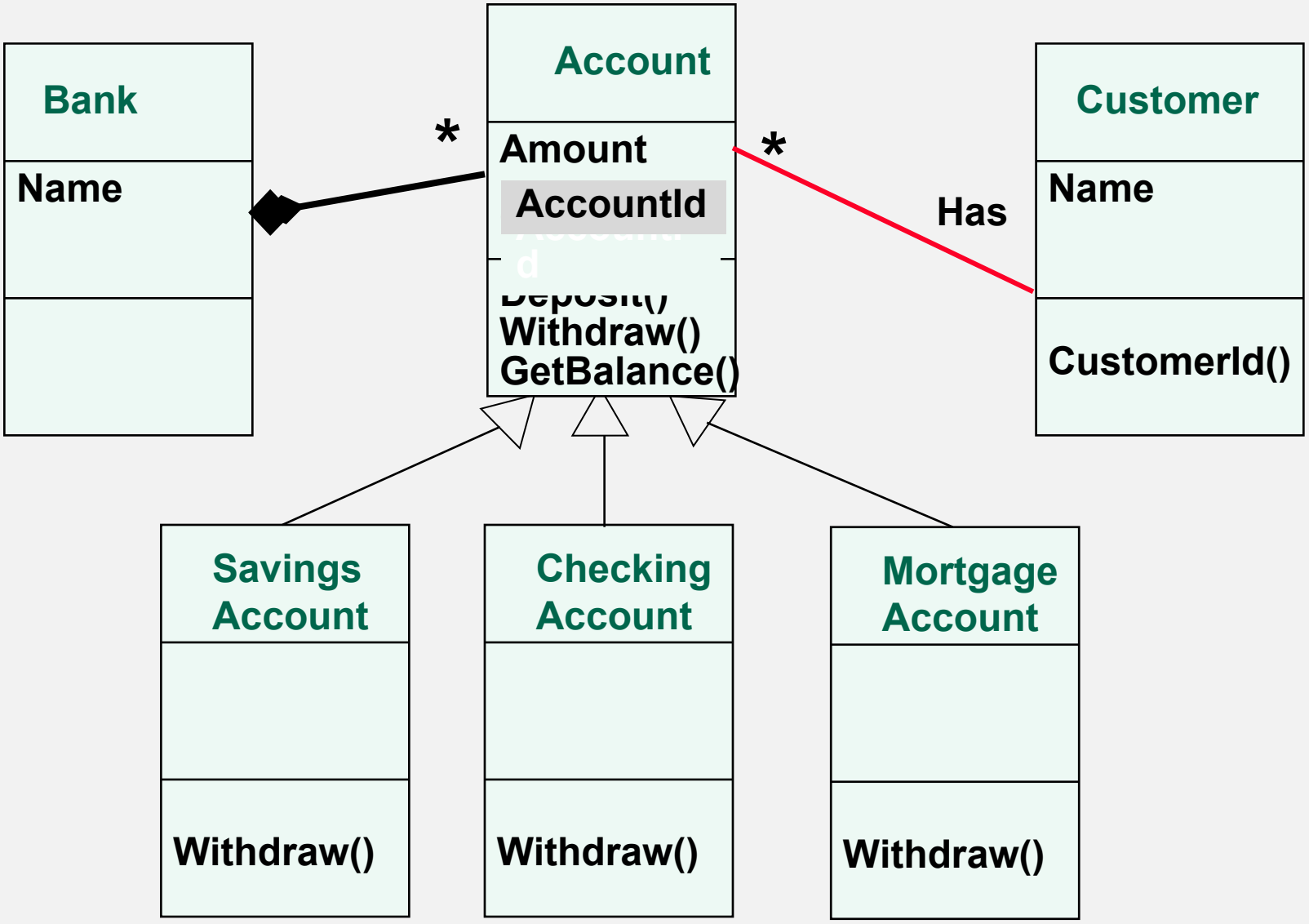
1) Find New Classes
2) Review Names, Attributes and Methods

Object Modeling in Practice: Associations

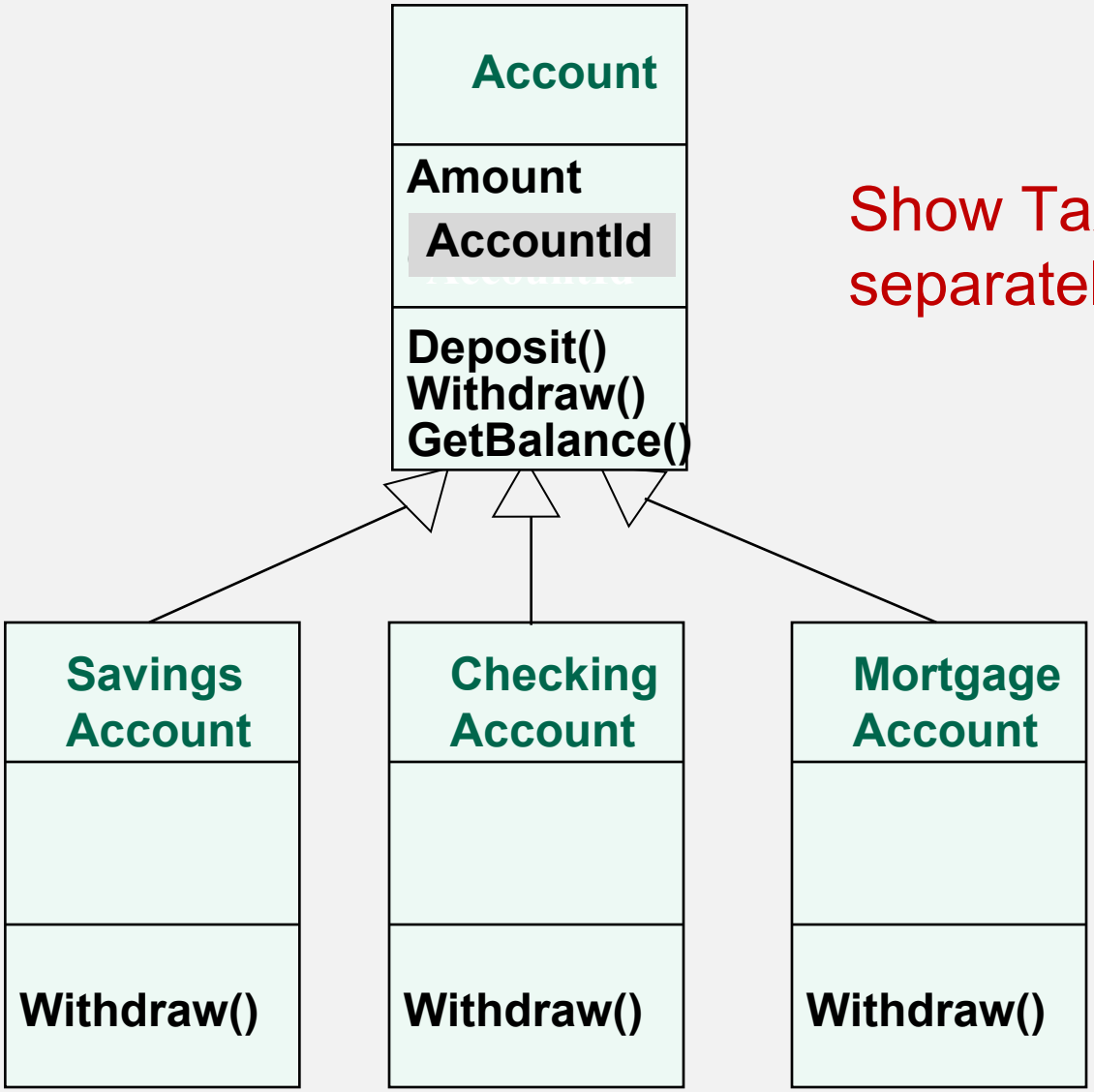


1) Find New Classes
2) Review Names, Attributes and Methods
3) Find Associations between Classes
4) Label the generic associations
5) Determine the multiplicity of the associations
6) Review associations

Practice Object Modeling: Find connection

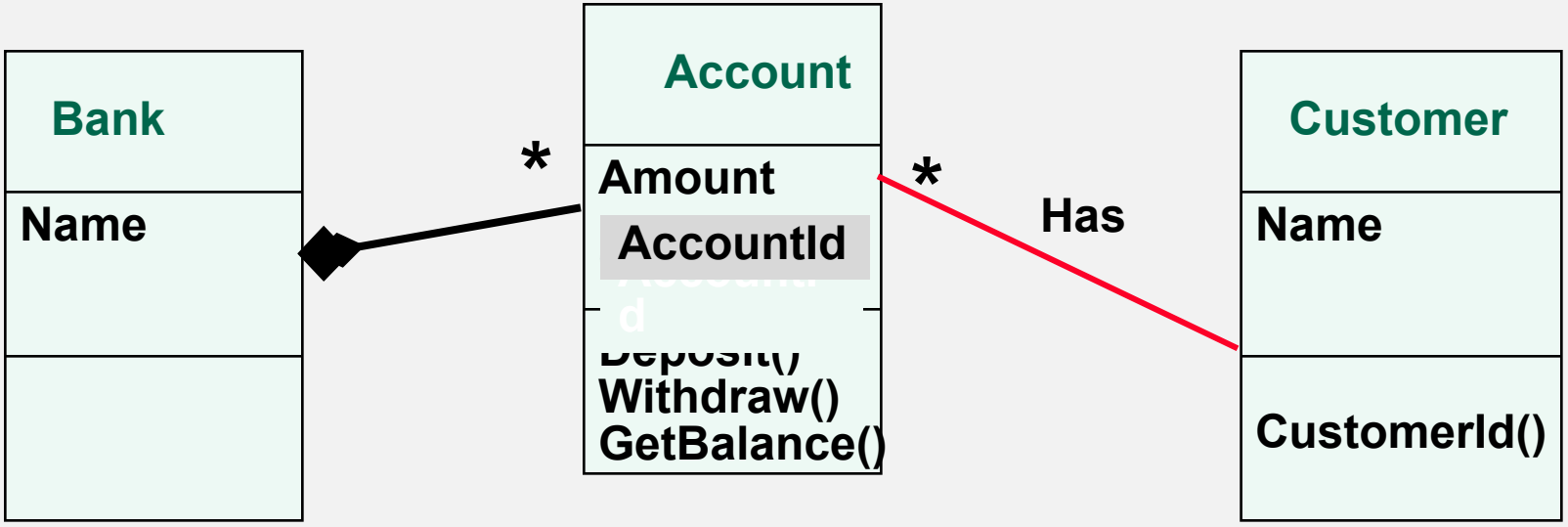


Practice Object Modeling: Simplify, Organize



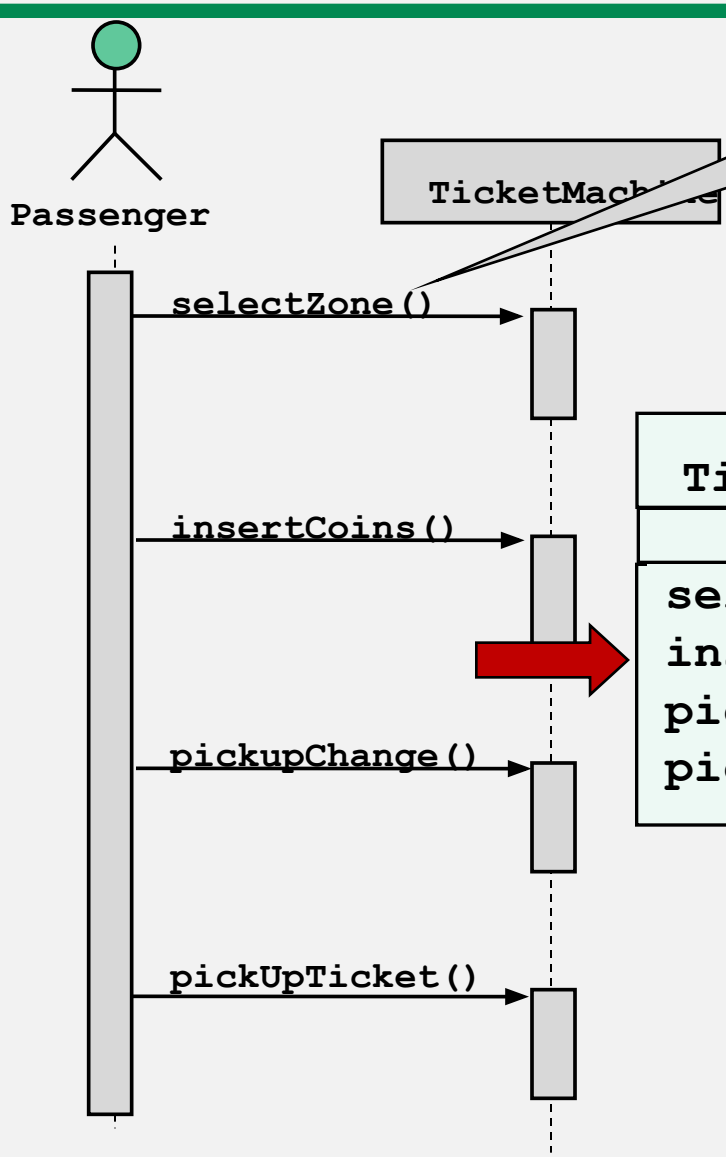
Show Taxonomies separately

Practice Object Modeling: Simplify, Organize



Use the 7±2 heuristics or better 5±2!

Sequence



Focus on Controlflow

Used during analysis

To refine use case descriptions

to find additional objects ("participating objects")

g system design

e subsys

are repre

les. ***Acto***

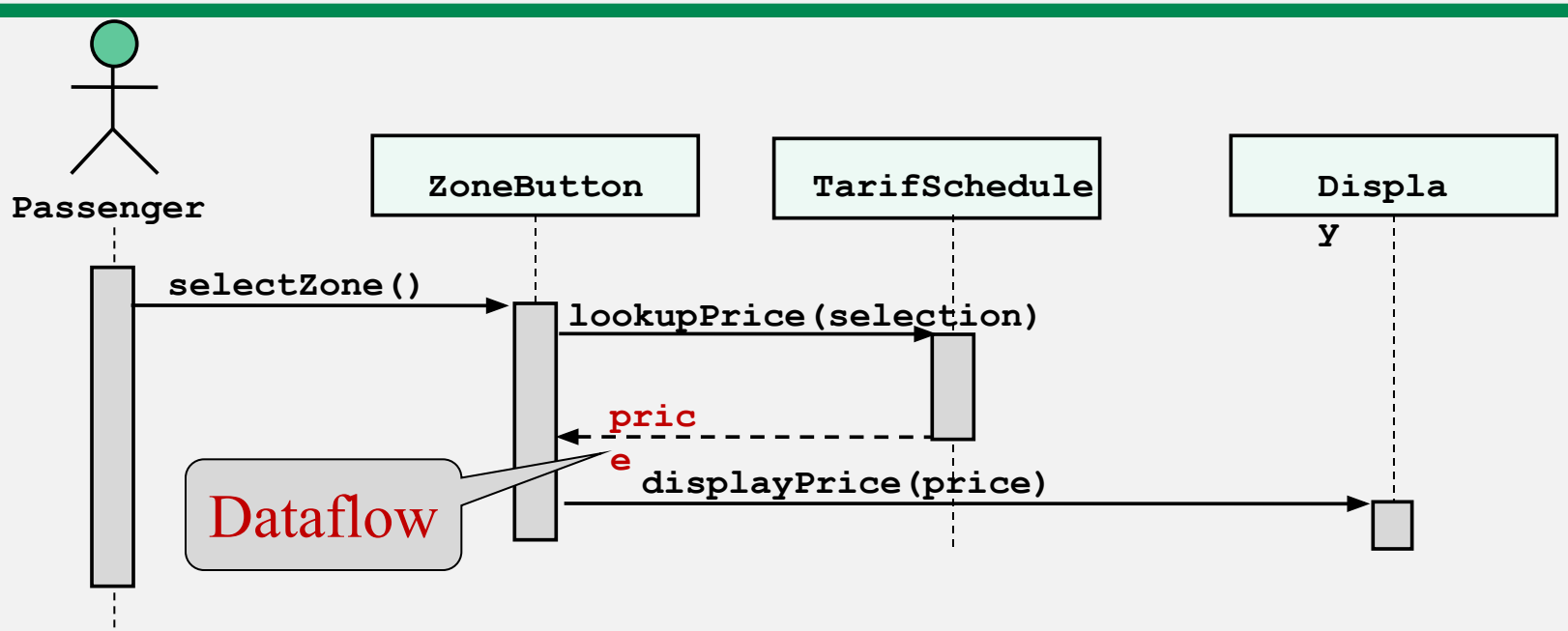
are represented by dashed

Messages are represented by arrows

Activations are represented by narrow rectangles.

Messages -> Operations on participating Object

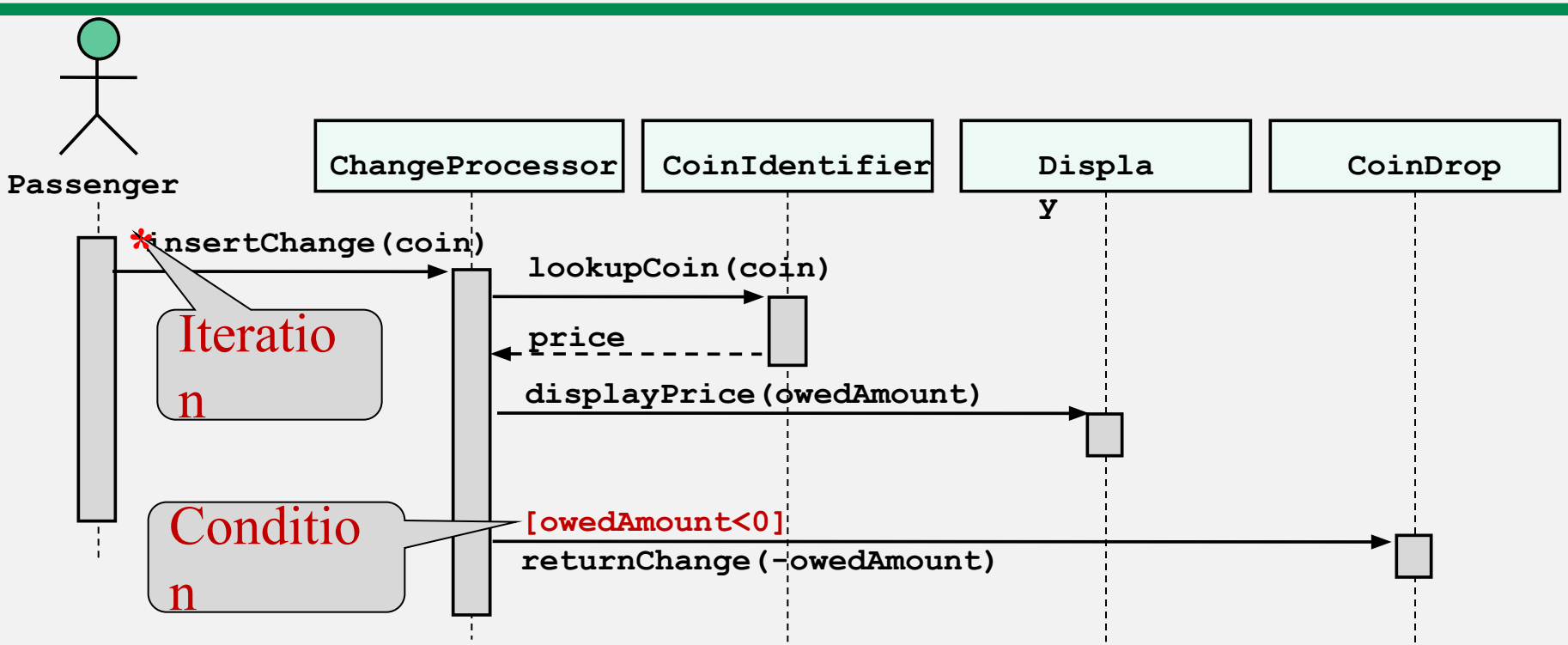
Sequence Diagrams can also model the Flow of Data



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

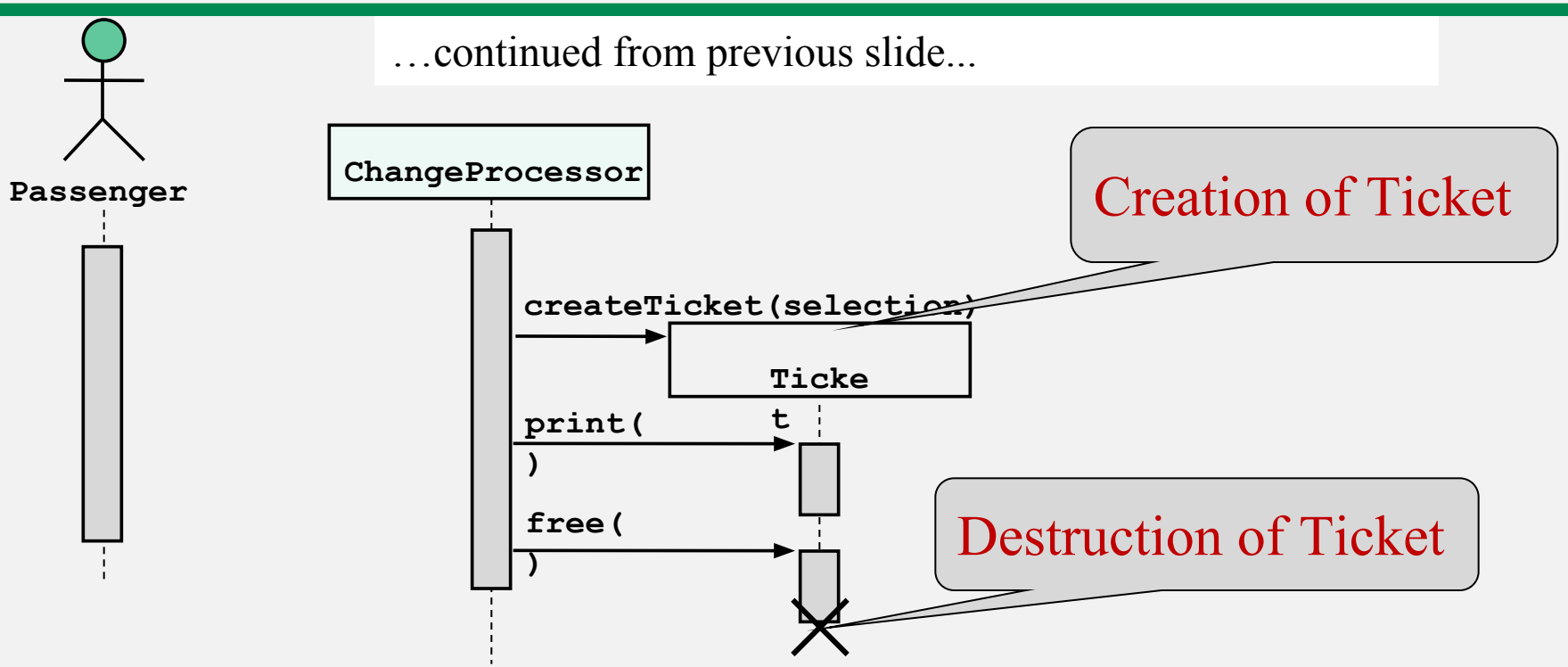
Sequence Diagrams: Iteration & Condition



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

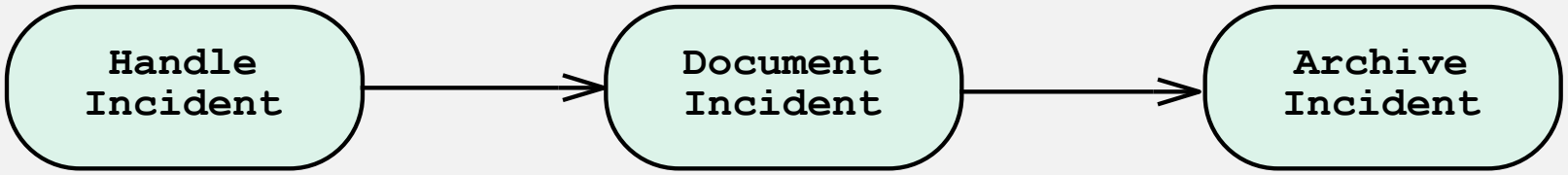
Complement the class diagrams (which represent structure).

Activity Diagrams

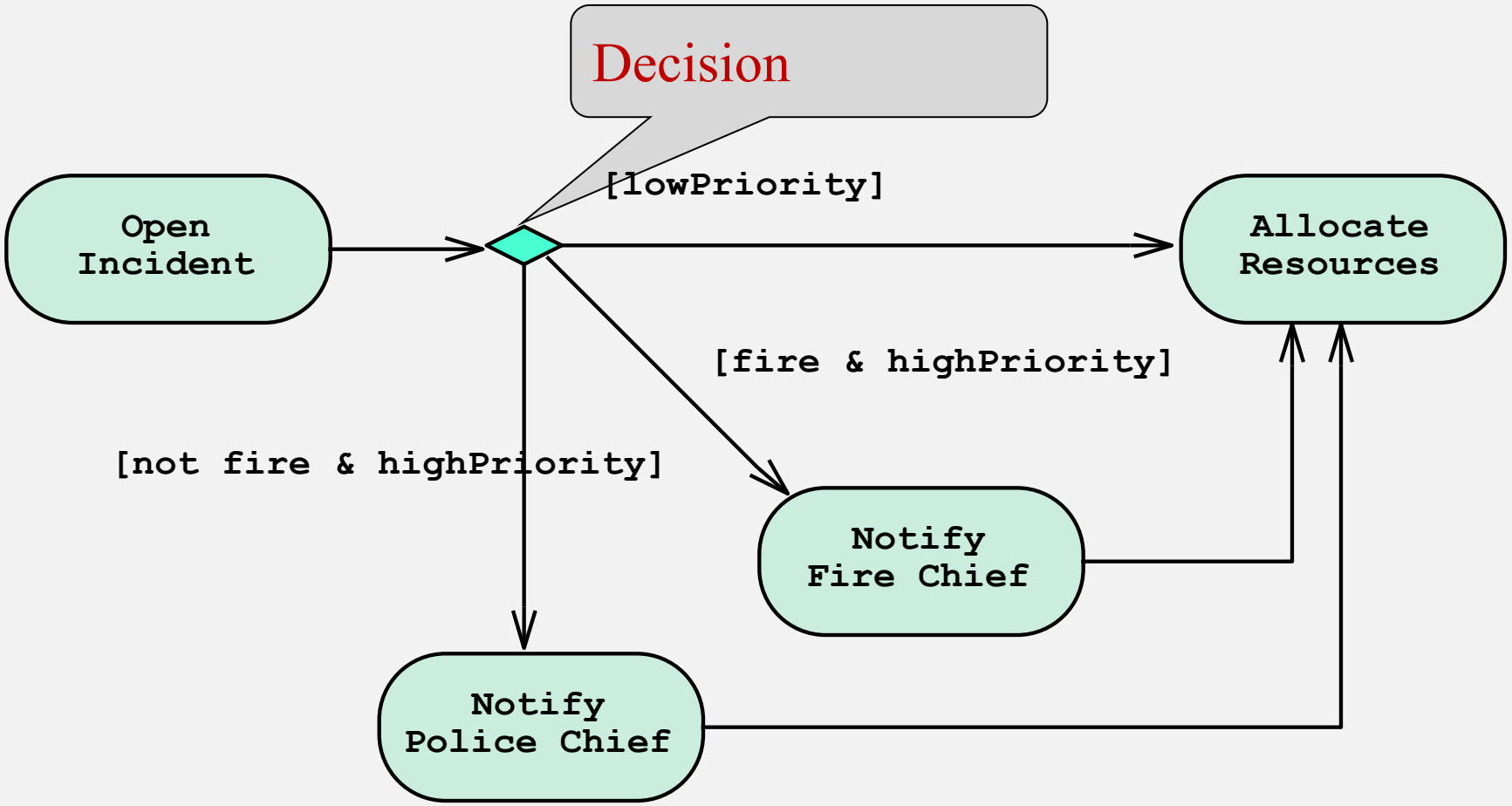
An activity diagram is a special case of a state chart diagram

The states are activities (“functions”)

An activity diagram is useful to depict the **workflow** in a system



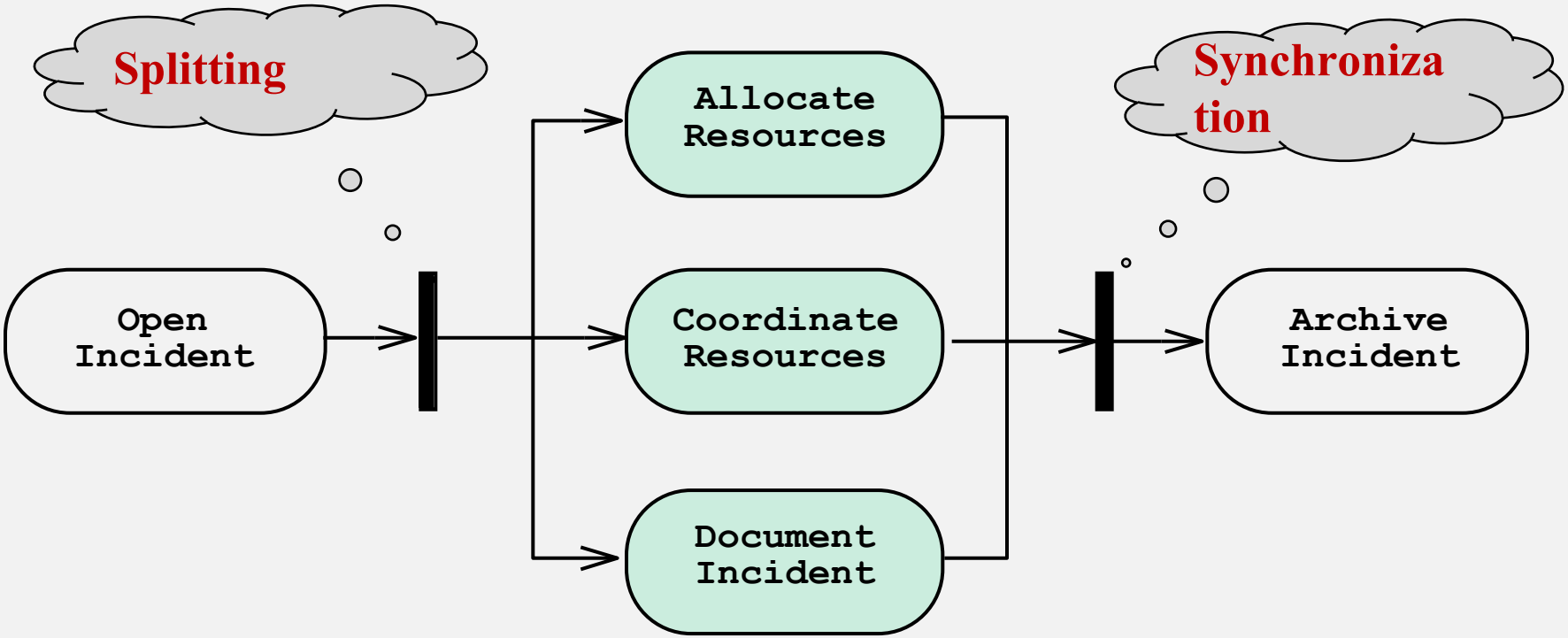
Activity Diagrams allow to model Decisions



Activity Diagrams can model Concurrency

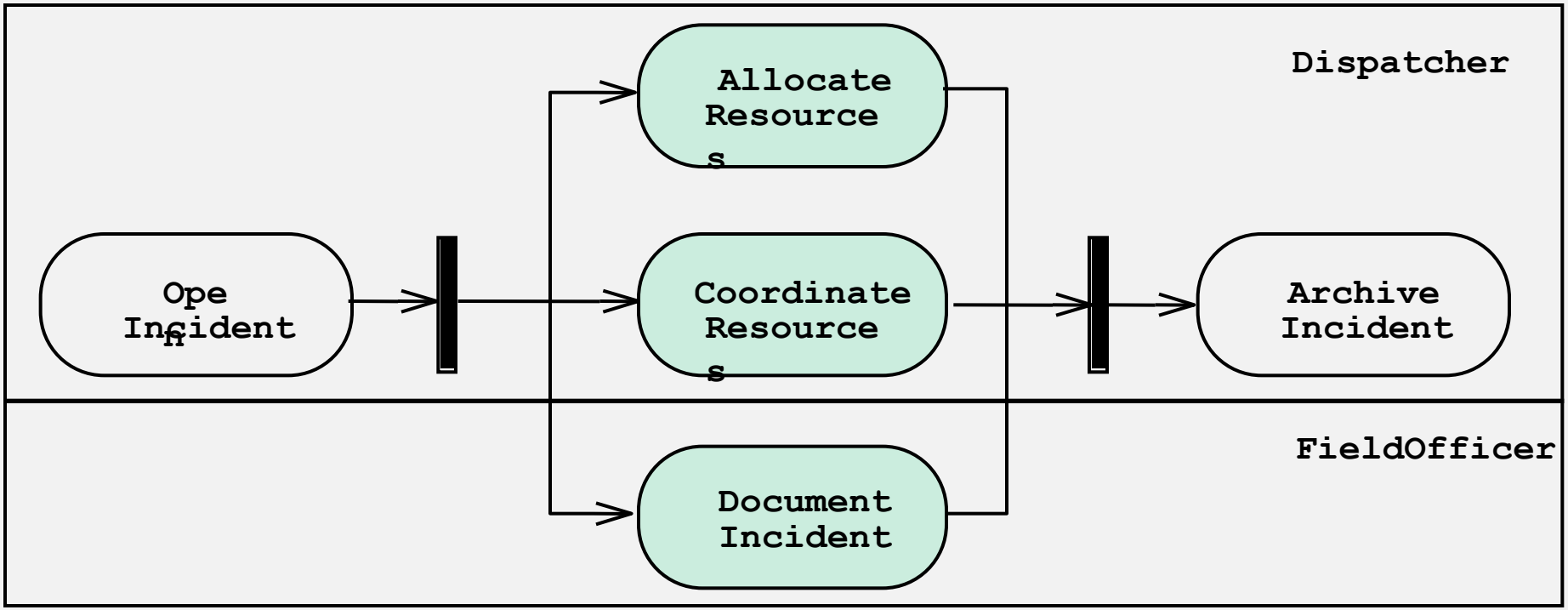
Synchronization of multiple activities

Splitting the flow of control into multiple threads



Activity Diagrams: Grouping of Activities

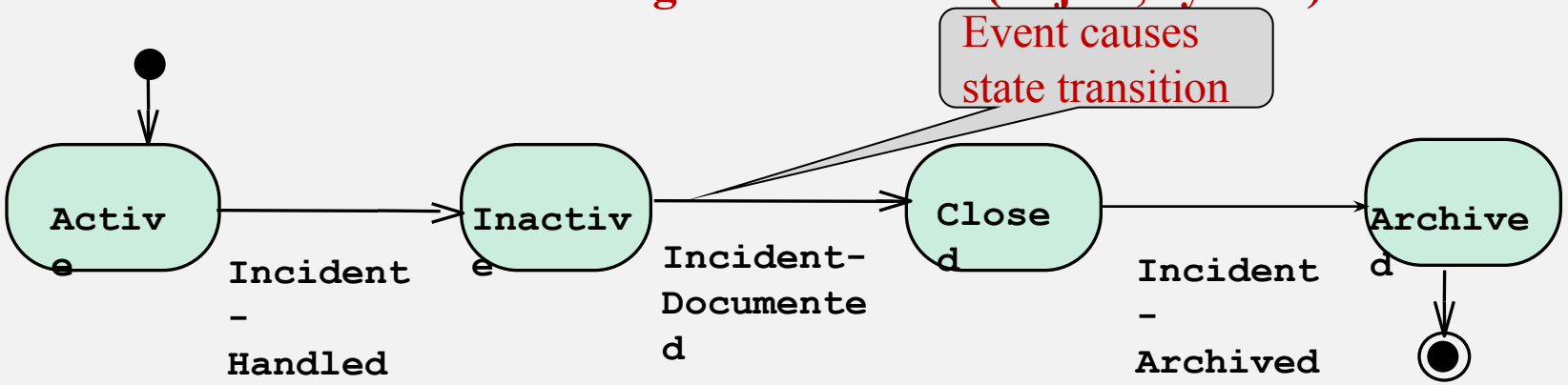
Activities may be grouped into swimlanes to denote the object or subsystem that implements the activities.



Activity Diagram vs. Statechart Diagram

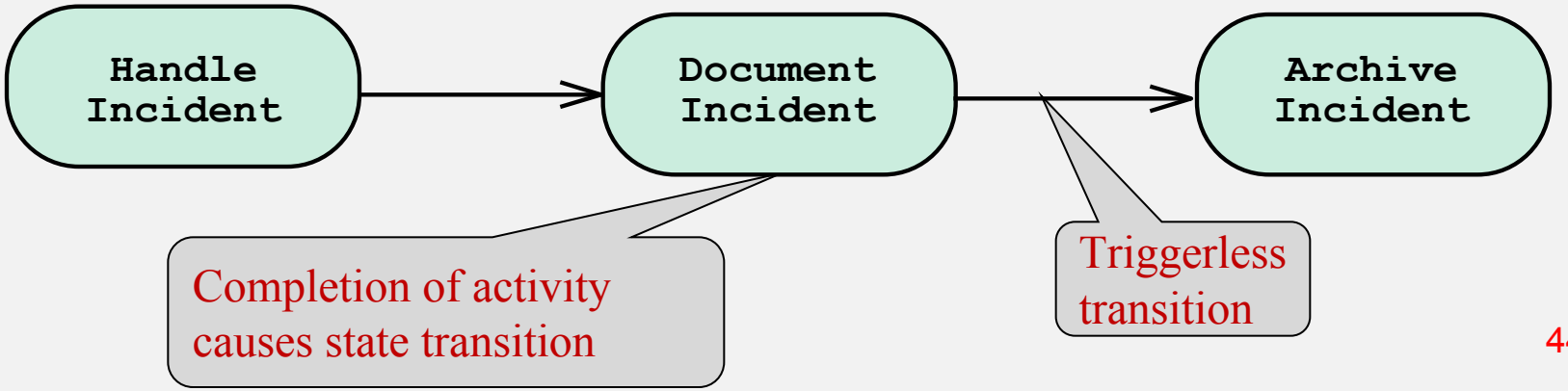
Statechart Diagram for Incident

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

Thank You