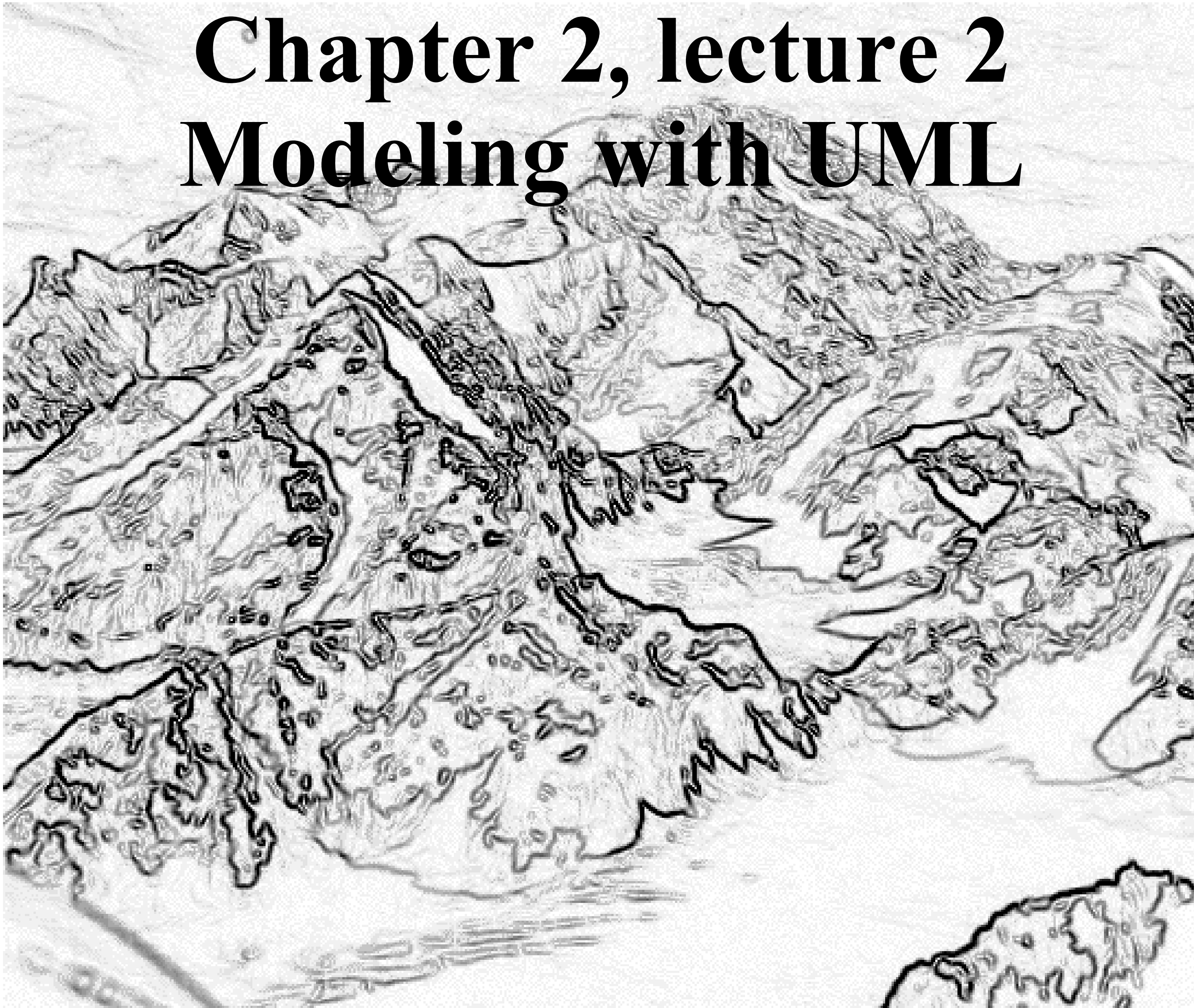
Chapter 2, lecture 2
Modeling with UML



Object-Oriented Software Engineering
Using UML, Patterns, and Java

Overview: More detail on modeling with UML

- Use case diagrams
- Class diagrams
- Sequence diagrams
- Activity diagrams

Other UML Notations

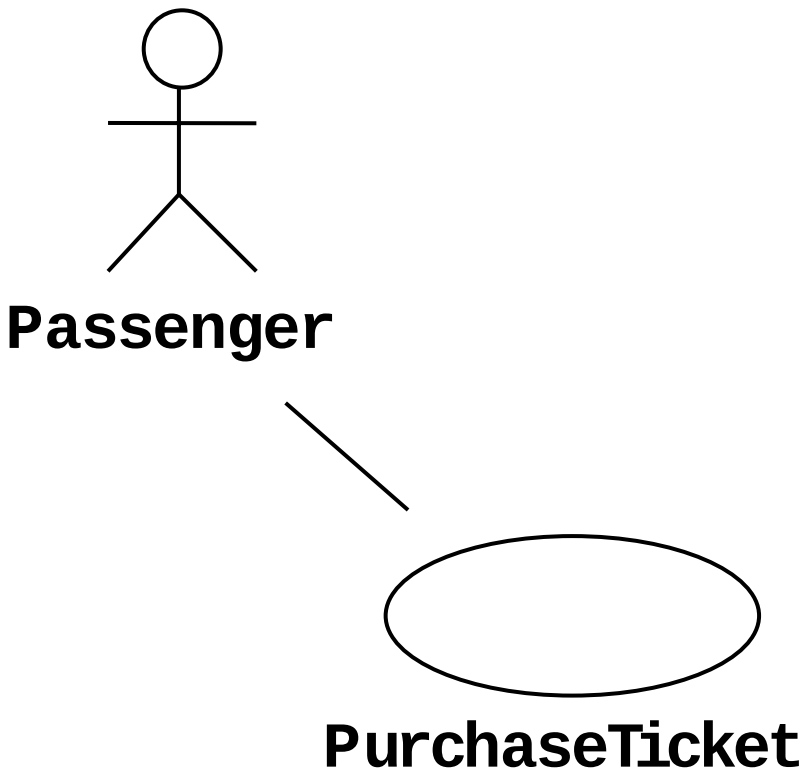
UML provide other notations that we will be introduced in subsequent lectures, as needed.

- Implementation diagrams
  - **Component diagrams**
  - **Deployment diagrams**
  - **Introduced in lecture on System Design**
- Object constraint language
  - **Introduced in lecture on Object Design**

UML Core Conventions

- Rectangles are classes or instances
- Ovals are functions or use cases
- Instances are denoted with an underlined names
  - **myWatch:SimpleWatch**
  - **Joe:Firefighter**
- Types are denoted with non underlined names
  - **SimpleWatch**
  - **Firefighter**
- Diagrams are graphs
  - **Nodes are entities**
  - **Arcs are relationships between entities**

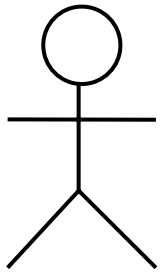
Use Case Diagrams



- Used during requirements elicitation to represent external behavior

- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

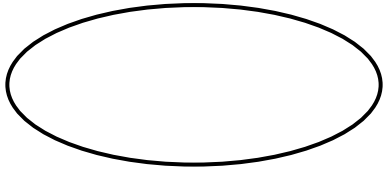
Actors



Passenger

- An actor models an external entity which communicates with the system:
  - **User**
  - **External system**
  - **Physical environment**
- An actor has a unique name and an optional description.
- Examples:
  - **Passenger: A person in the train**
  - **GPS satellite: Provides the system with GPS coordinates**

Use Case



PurchaseTicket

A use case represents a class of functionality provided by the system as an event flow.

A use case consists of:

- Unique name
- Participating actors
- Entry conditions
- Flow of events
- Exit conditions
- Special requirements

Use Case Diagram: Example

Name: Purchase ticket

Participating actor: Passenger

Entry condition:

- Passenger standing in front of ticket distributor.
- Passenger has sufficient money to purchase ticket.

Exit condition:

- Passenger has ticket.

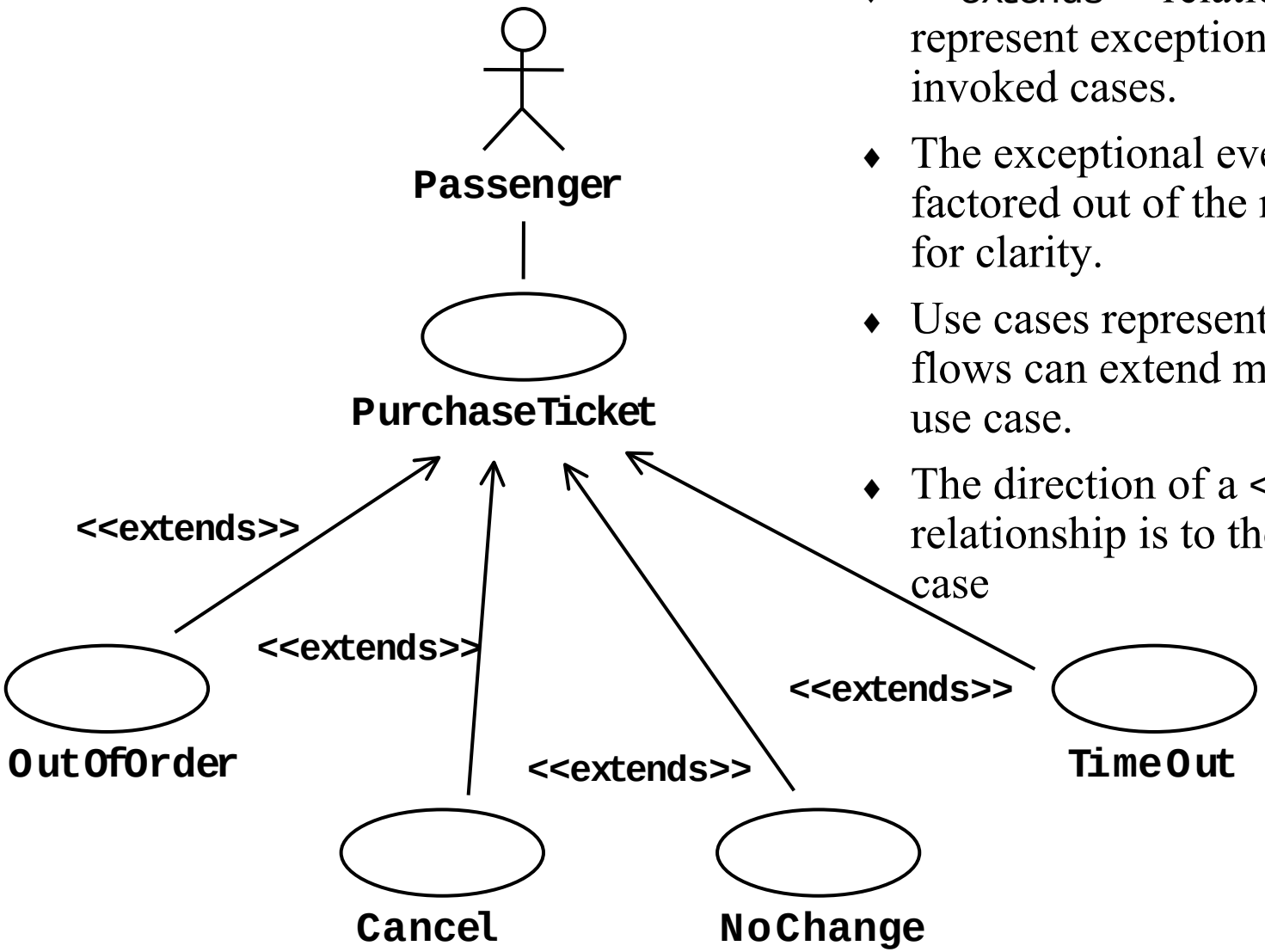
Event flow:

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

Anything missing?

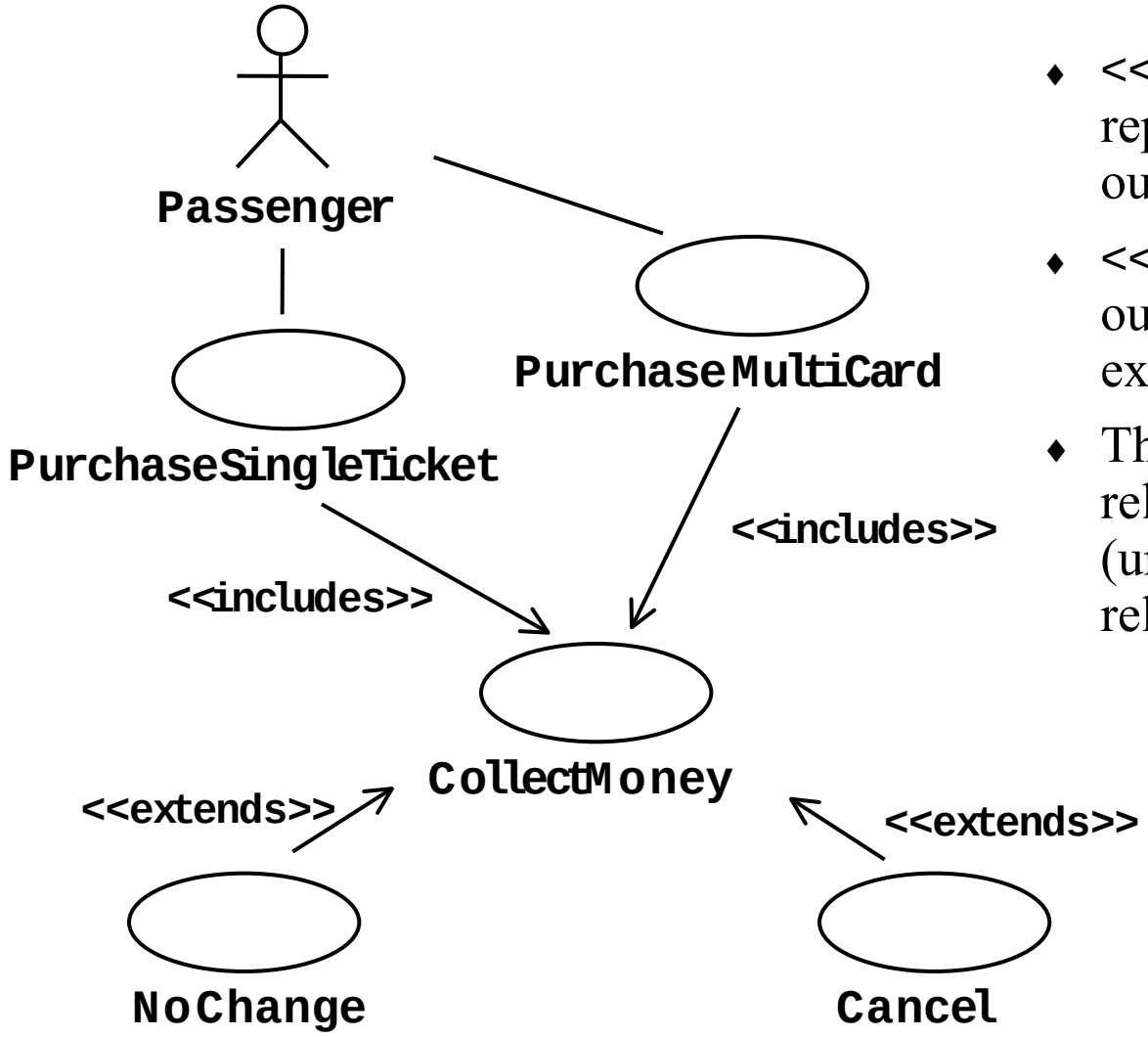
Exceptional cases!

The <<extends>> *Relationship*



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

The <<includes>> *Relationship*

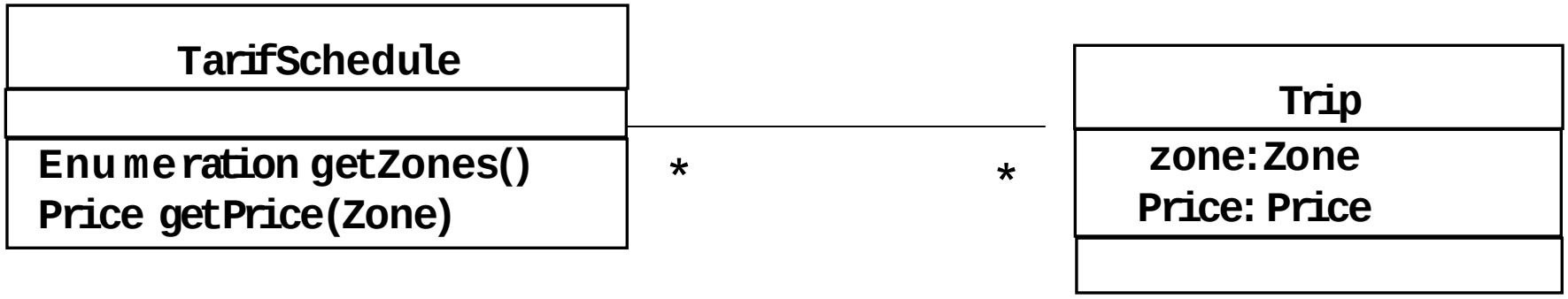


- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

Use Case Diagrams: Summary

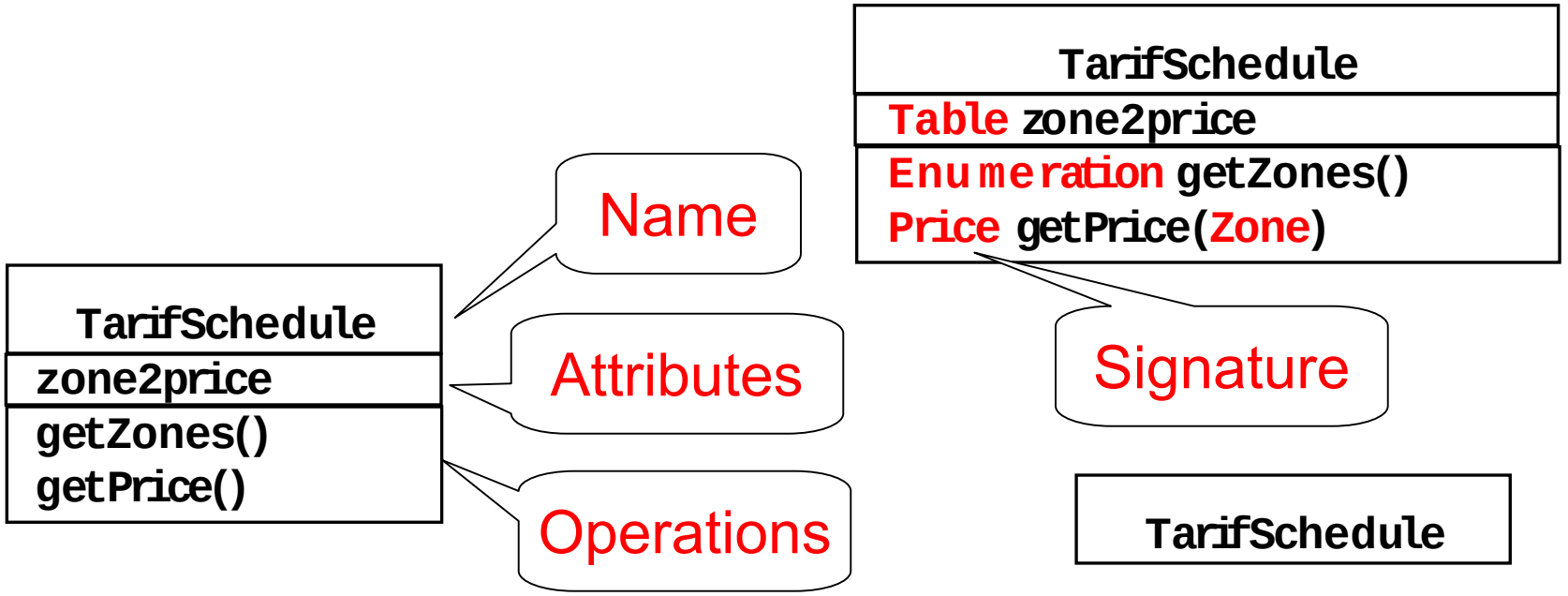
- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

Class Diagrams



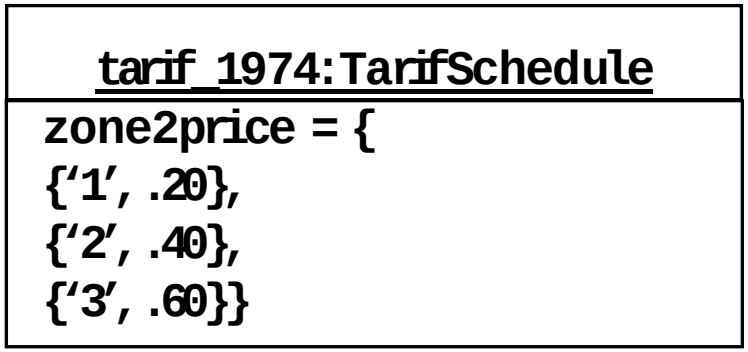
- Class diagrams represent the structure of the system.
- Used
  - **during requirements analysis to model problem domain concepts**
  - **during system design to model subsystems and interfaces**
  - **during object design to model classes.**

Classes



- A ***class*** represent 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

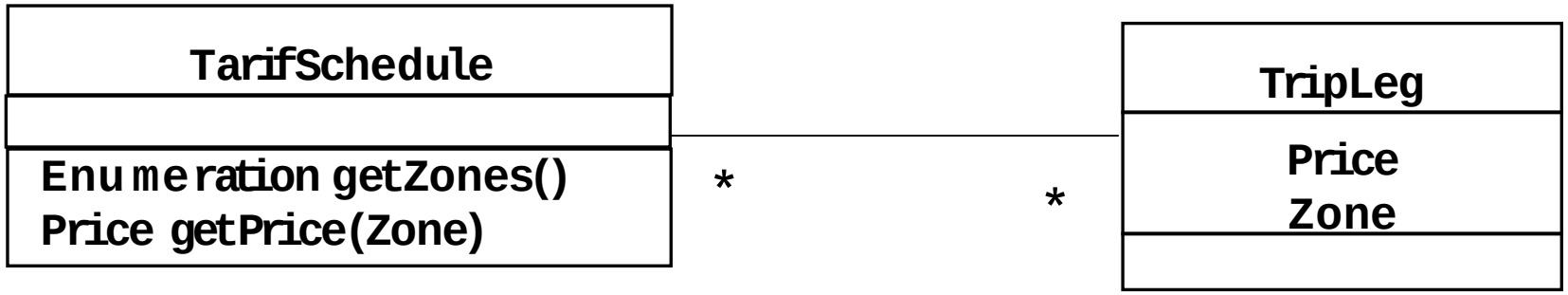


- An ***instance*** represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their ***values***.

Actor vs Instances

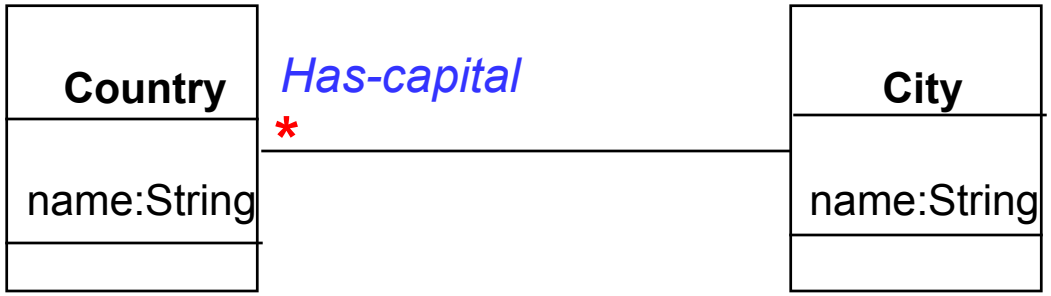
- What is the difference between an *actor* , a *class* and an *instance*?
- Actor:
  - **An entity outside the system to be modeled, interacting with the system ("Passenger")**
- Class:
  - **An abstraction modeling an entity in the problem domain, must be modeled inside the system ("User")**
- Object:
  - **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 source object can legitimately reference.

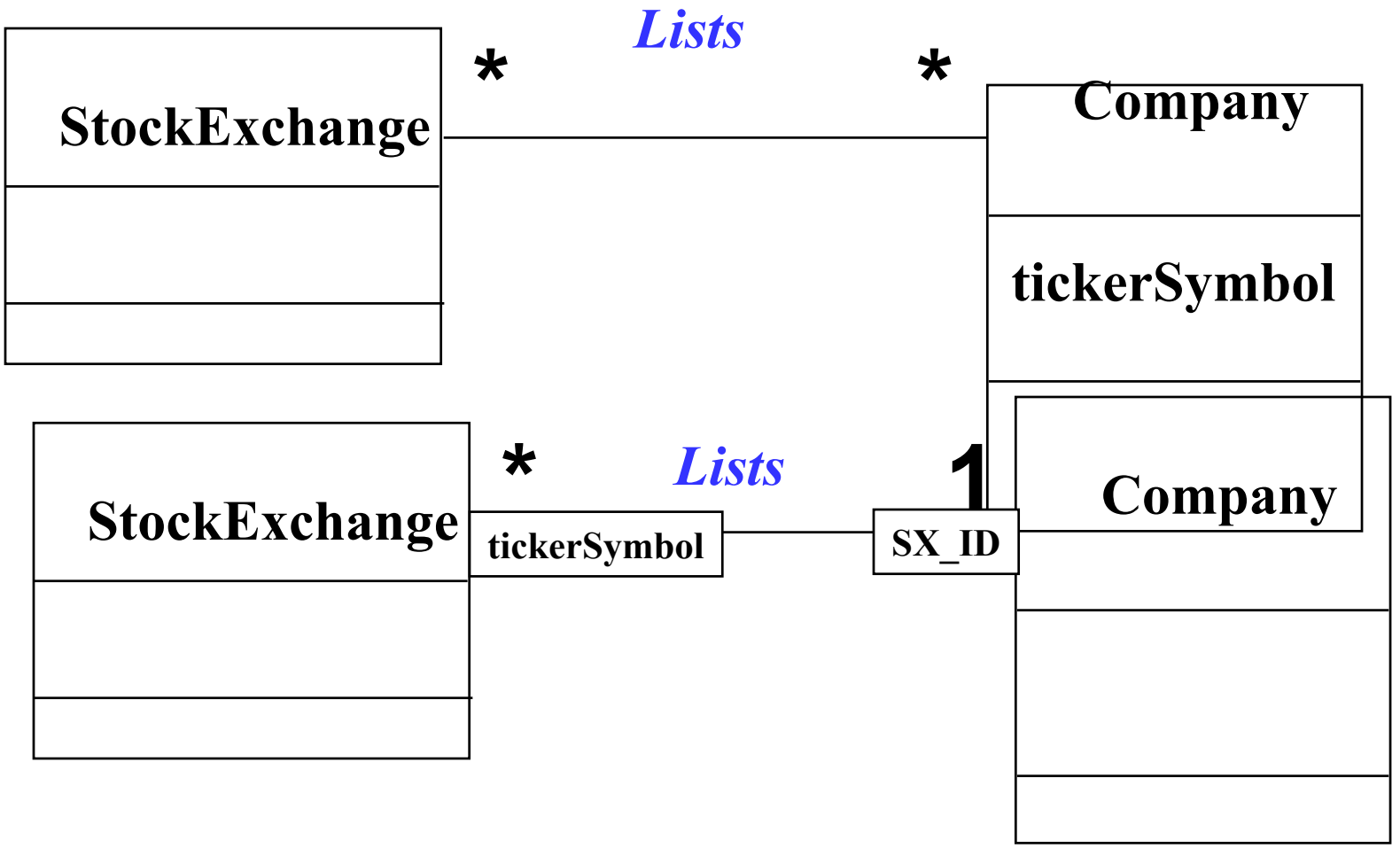
1-to-1 and 1-to-many Associations



One-to-one association

One-to-many association

Many-to-Many Associations



From Problem Statement To Object Model

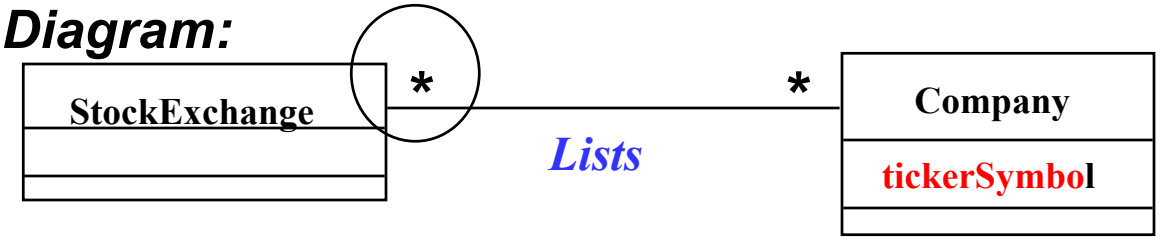
Problem Statement: A stock exchange lists many companies. Each company is uniquely identified by a ticker symbol

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:

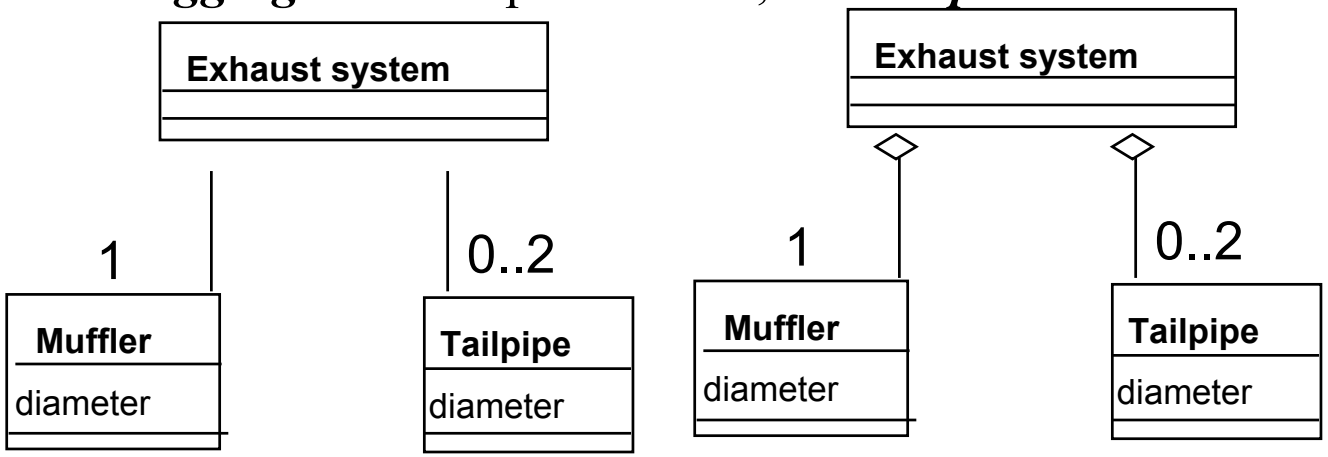


Java Code

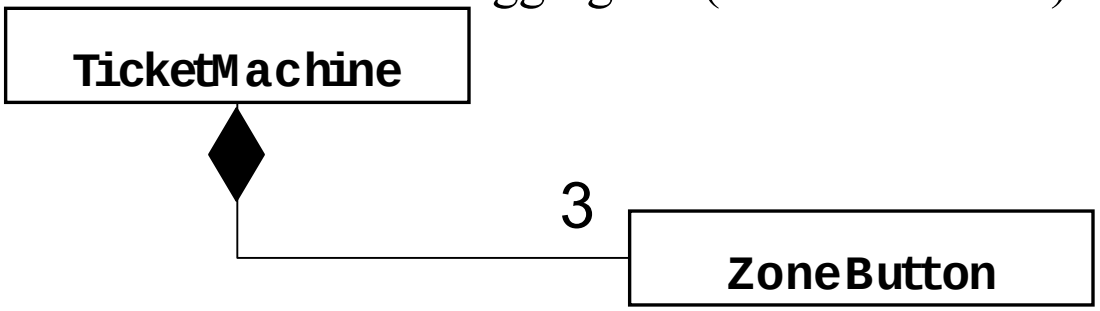
```
public class StockExchange
{
 private Vector m_Company = new Vector();
};

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

Aggregation

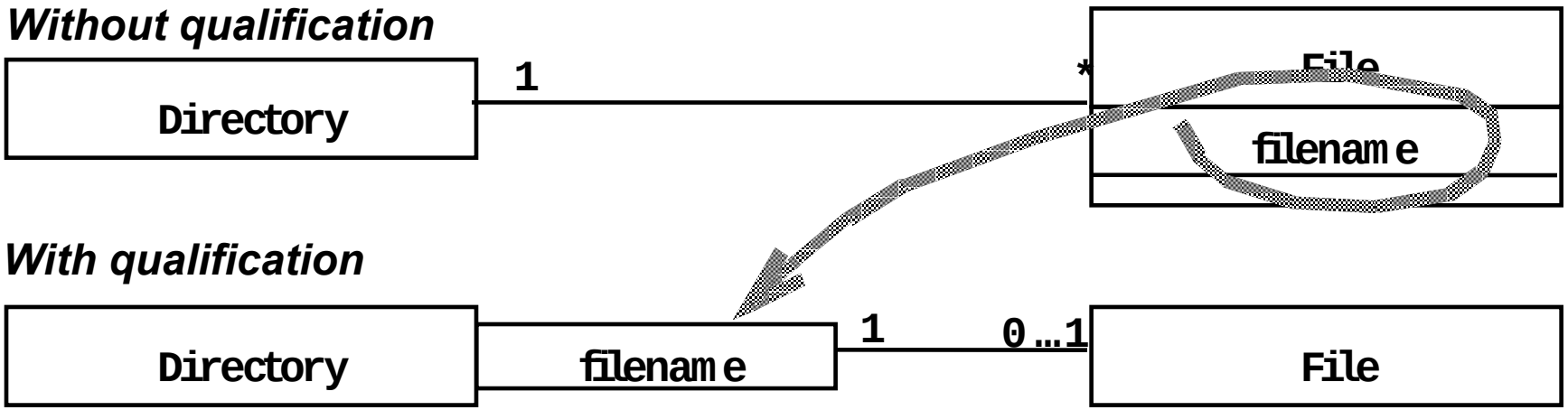
- An ***aggregation*** is a special case of association denoting a "consists of" hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children class.



- A solid diamond denotes ***composition***, a strong form of aggregation where components cannot exist without the aggregate. (Bill of Material)

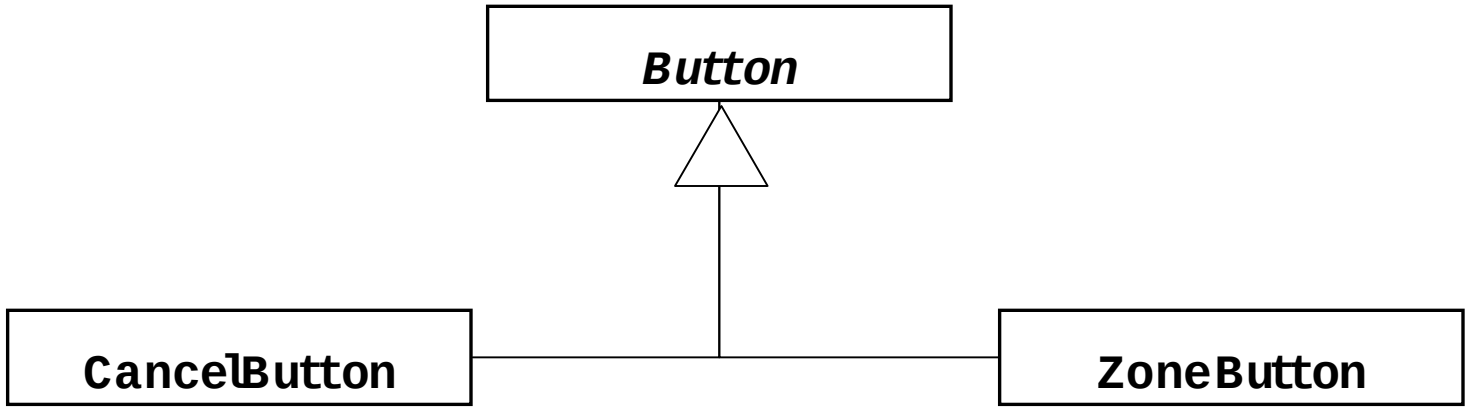


Qualifiers



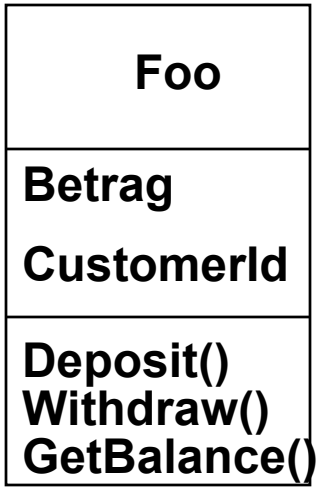
- Qualifiers can be used to reduce the multiplicity of an association.

Inheritance



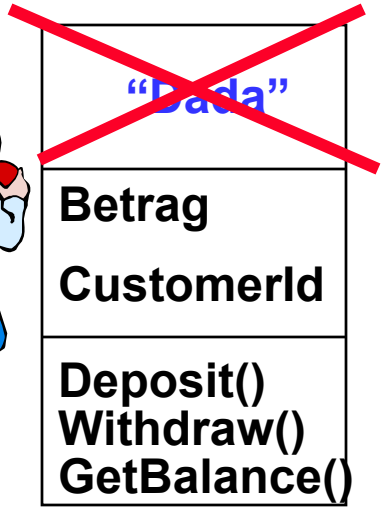
- The **children classes** inherit the attributes and operations of the **parent class**.
- Inheritance simplifies the model by eliminating redundancy.

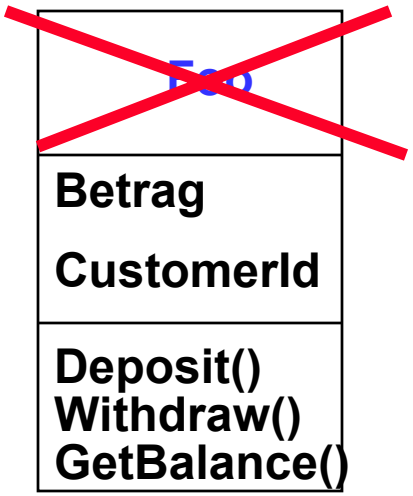
Object Modeling in Practice: Class Identification

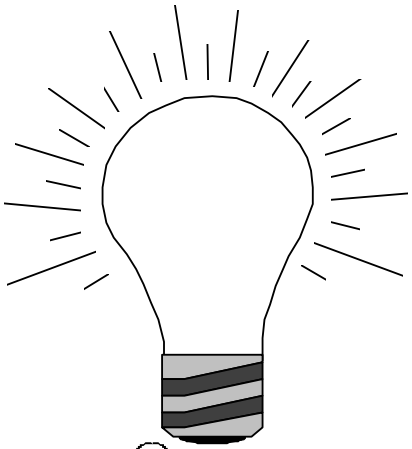


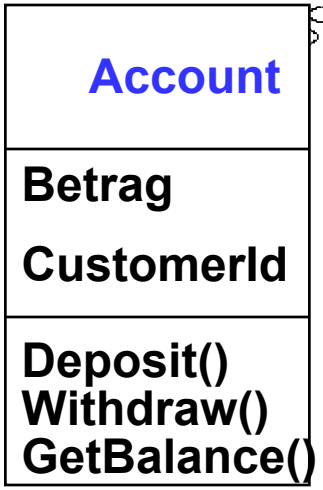
Class Identification: Name of Class, Attributes and Methods

Object Modeling in Practice: Encourage Brainstorming



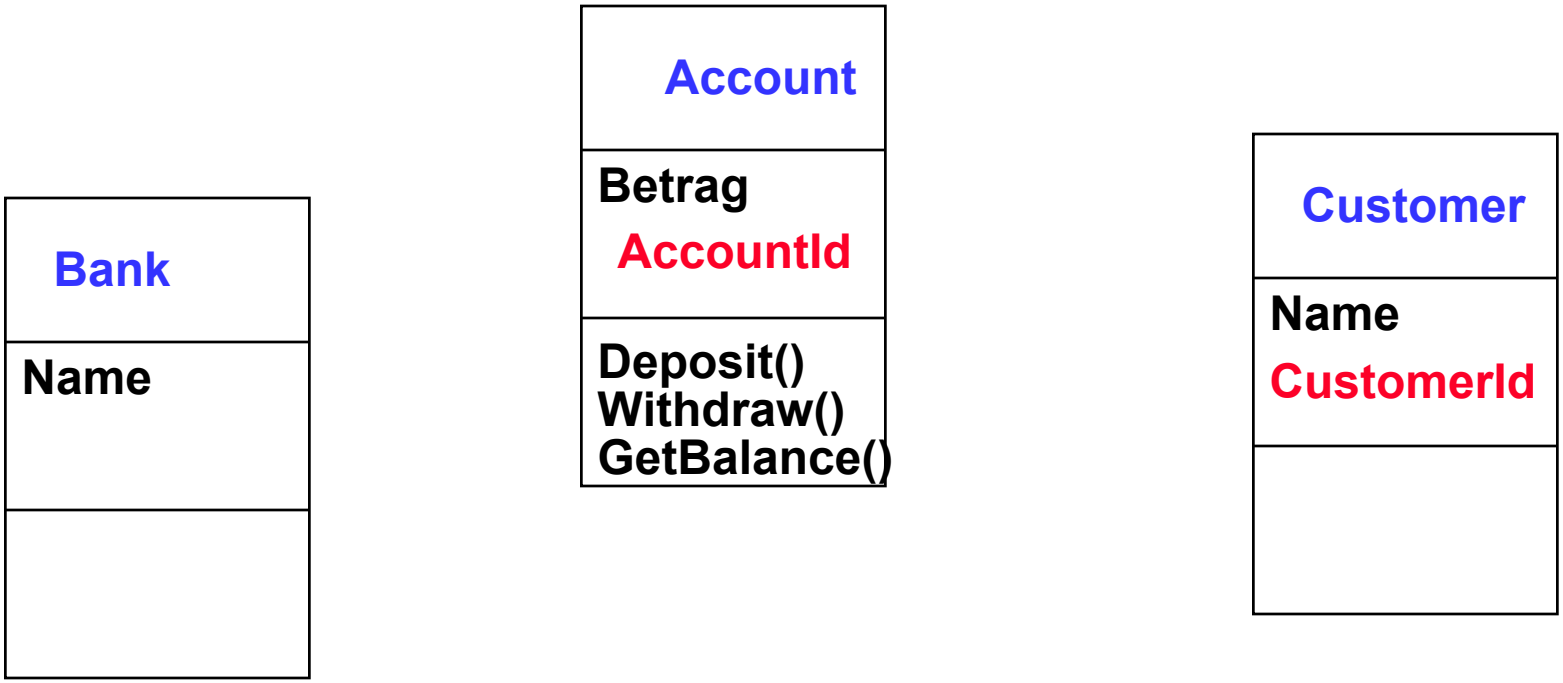






Naming is important!
Is Foo the right name?

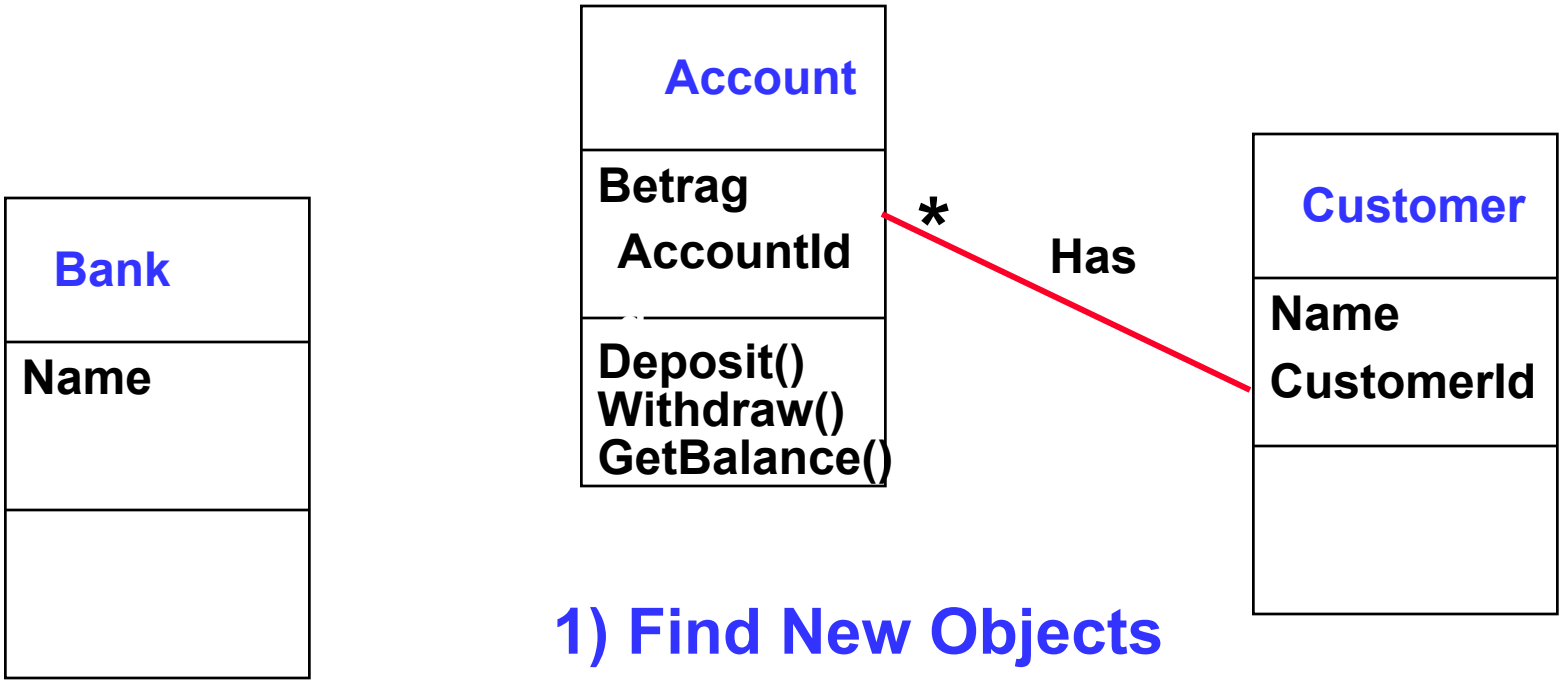
Object Modeling in Practice ctd



1) Find New Objects

2) Iterate on Names, Attributes and Methods

Object Modeling in Practice: A Banking System



1) Find New Objects

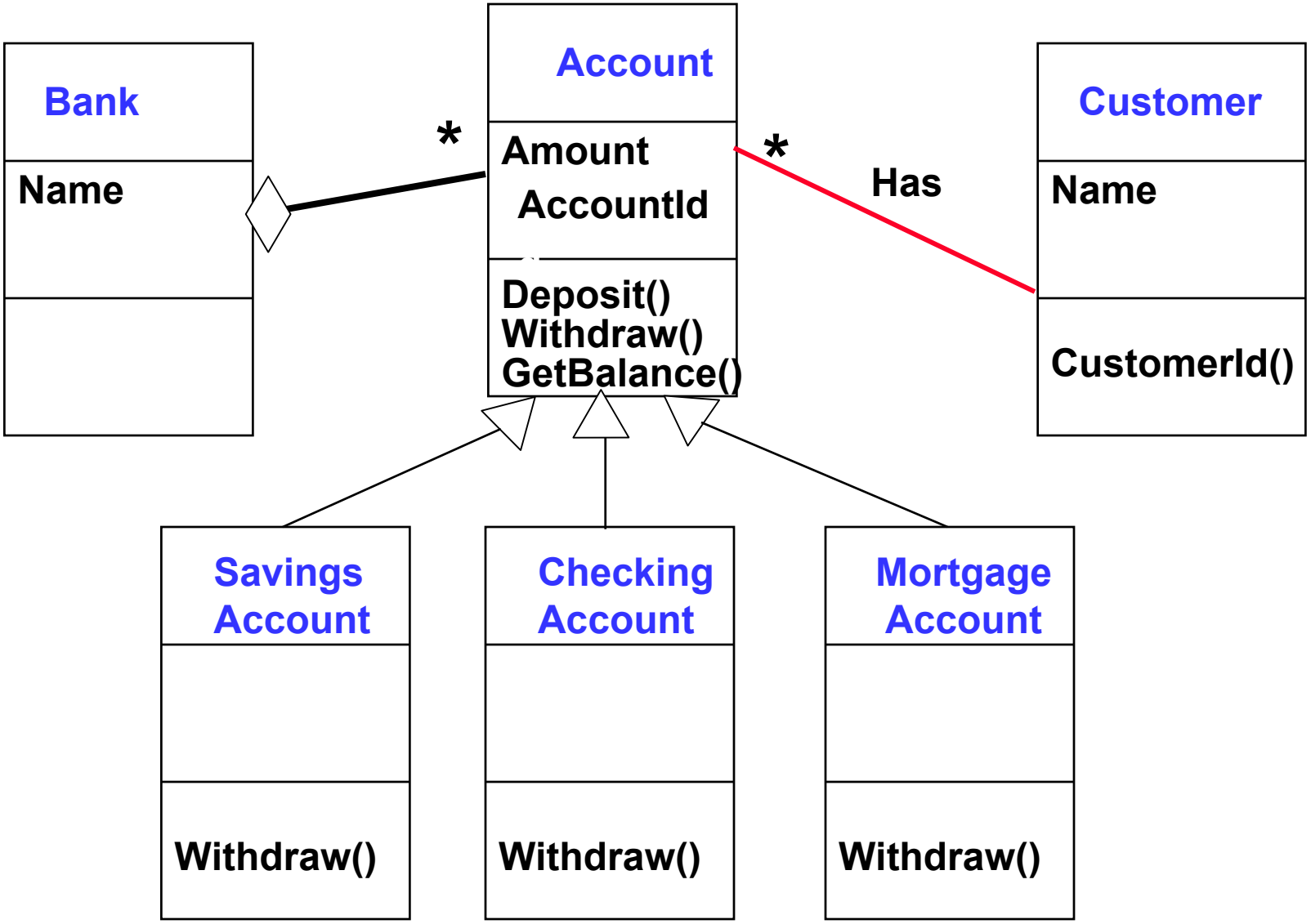
2) Iterate on Names, Attributes and Methods

3) Find Associations between Objects

4) Label the assocations

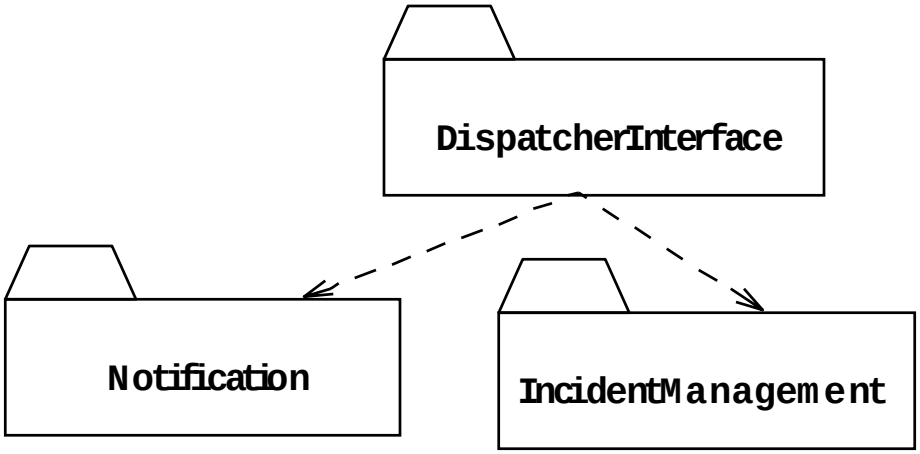
5) Determine the multiplicity of the assocations

Practice Object Modeling: Iterate, Categorize!



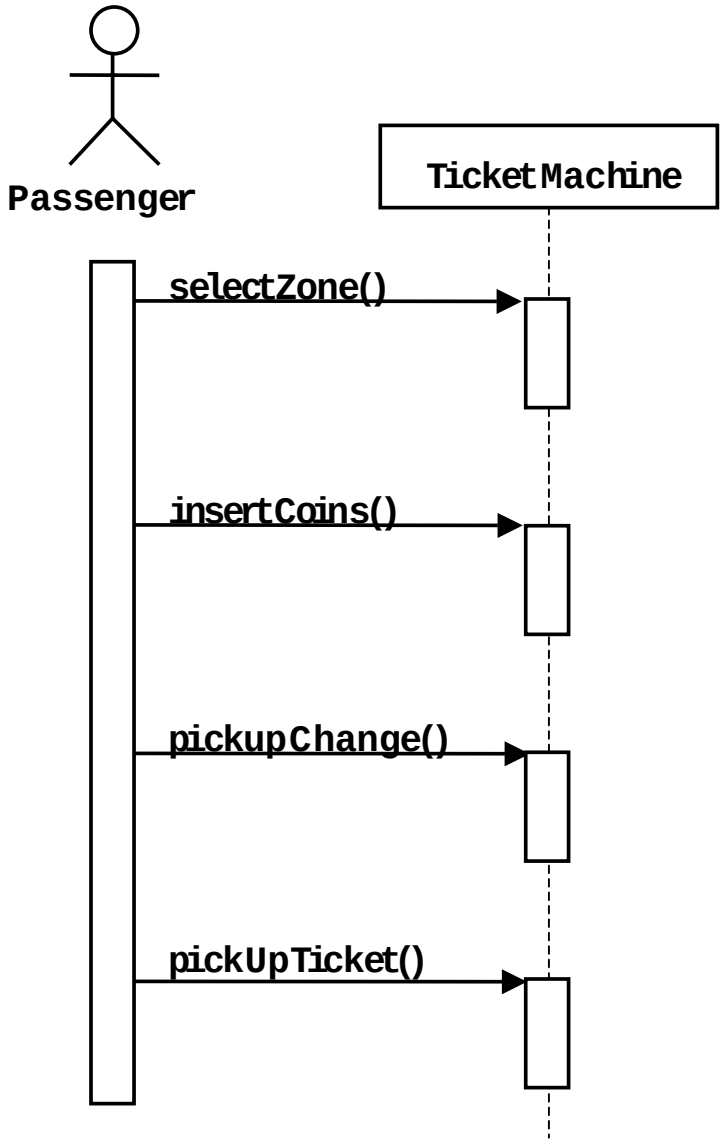
Packages

- A package is a UML mechanism for organizing elements into groups (usually not an application domain concept)
- Packages are the basic grouping construct with which you may organize UML models to increase their readability.



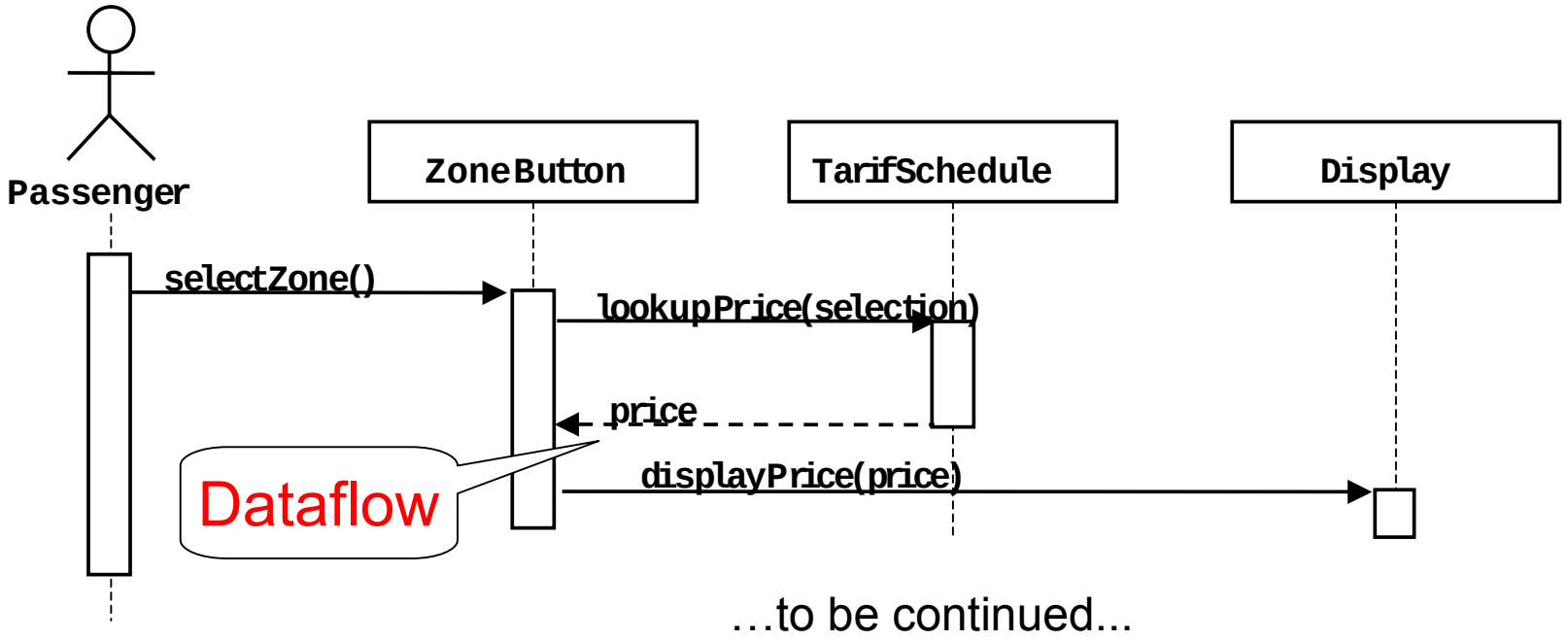
- A complex system can be decomposed into subsystems, where each subsystem is modeled as a package

UML sequence diagrams



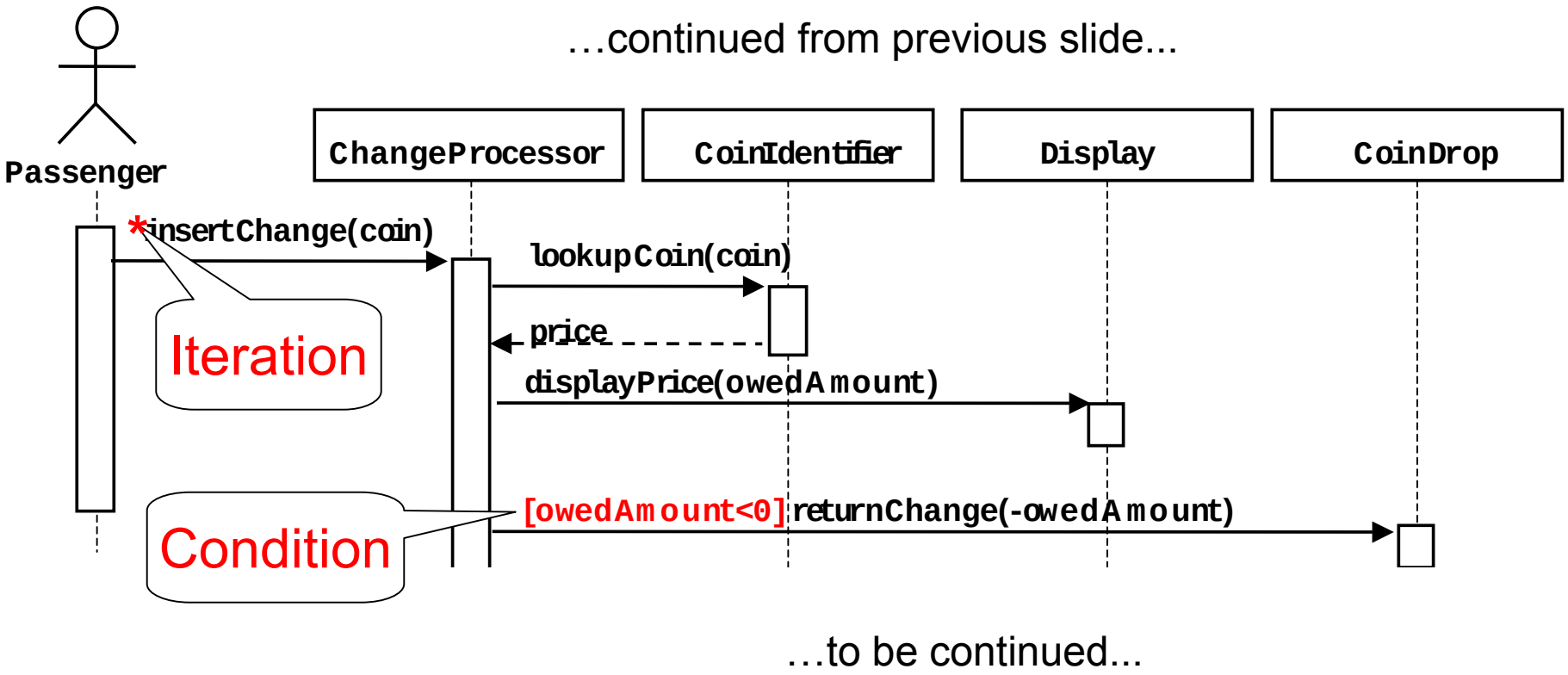
- Used during requirements analysis
  - **To refine use case descriptions**
  - **to find additional objects ("participating objects")**
- Used during system design
  - **to refine subsystem interfaces**
- ***Classes*** are represented by columns
- ***Messages*** are represented by arrows
- ***Activations*** are represented by narrow rectangles
- ***Lifelines*** are represented by dashed lines

Nested messages



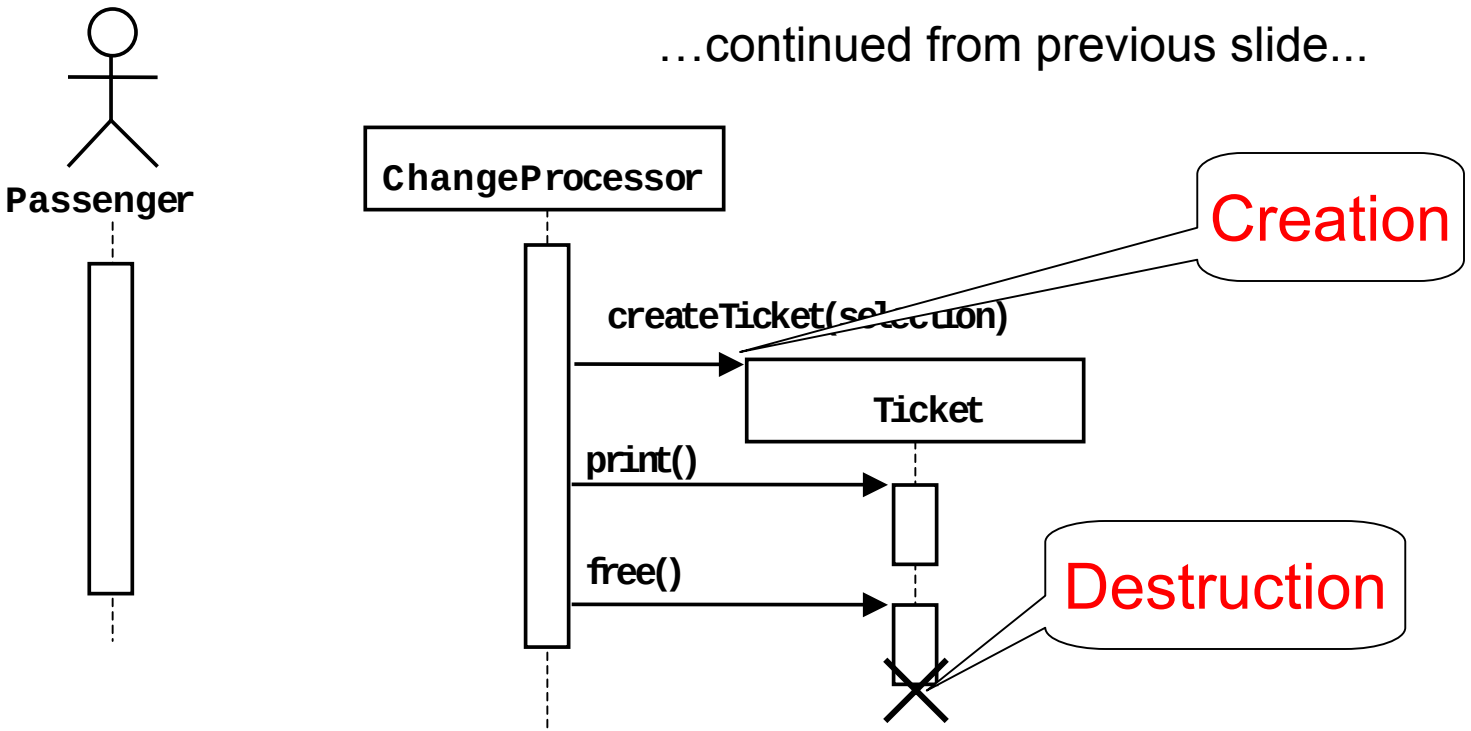
- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations
- Horizontal dashed arrows indicate data flow
- Vertical dashed lines indicate lifelines

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

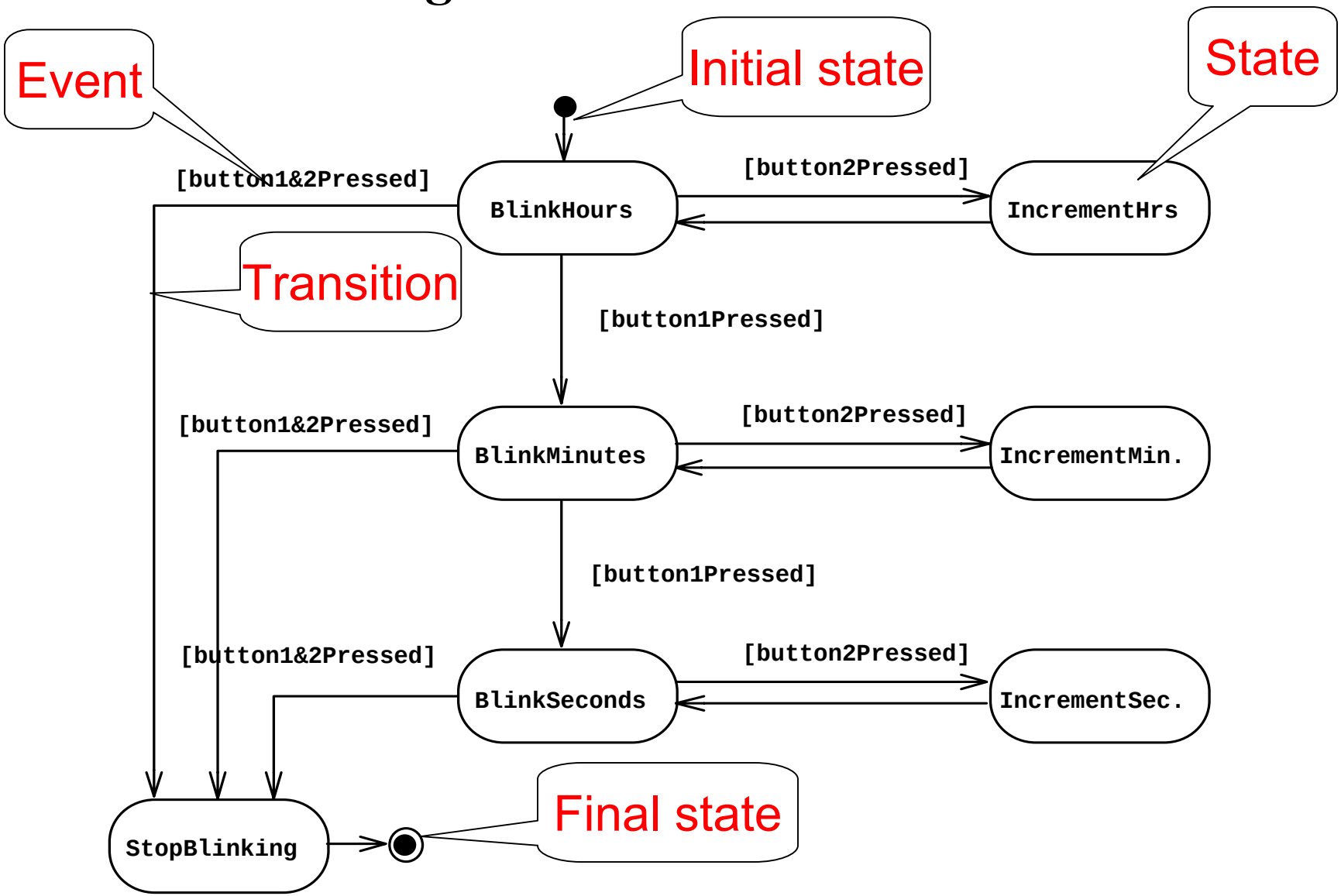


- Creation is denoted by a message arrow pointing to the object.
- Destruction is denoted by an X mark at the end of the destruction activation.
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

Sequence Diagram Summary

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

State Chart Diagrams



Represent behavior as states and transitions

Activity Diagrams

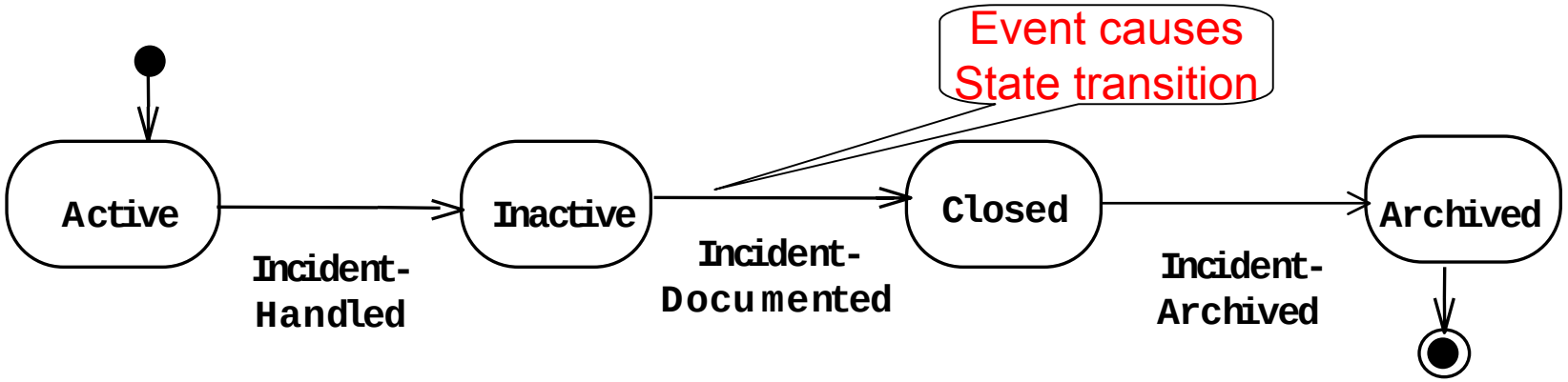
- An activity diagram shows flow control within a system

- An activity diagram is a special case of a state chart diagram in which states are activities ("functions")
- Two types of states:
  - ***Action state:***
    - **Cannot be decomposed any further**
    - **Happens "instantaneously" with respect to the level of abstraction used in the model**
  - ***Activity state:***
    - **Can be decomposed further**
    - **The activity is modeled by another activity diagram**

Statechart Diagram vs. Activity Diagram

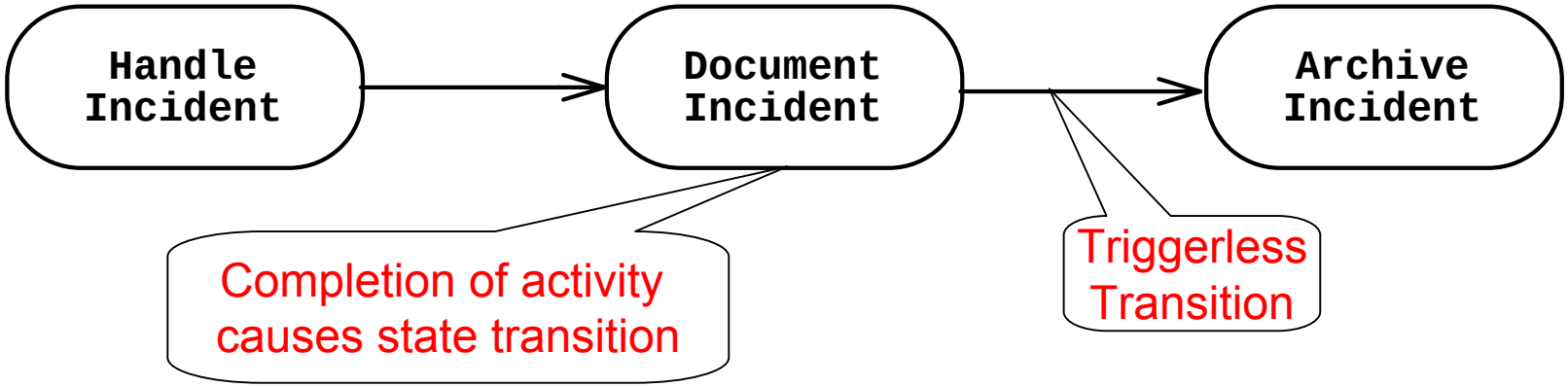
Statechart Diagram for Incident (similar to Mealy Automaton)

(State: Attribute or Collection of Attributes of object of type Incident)

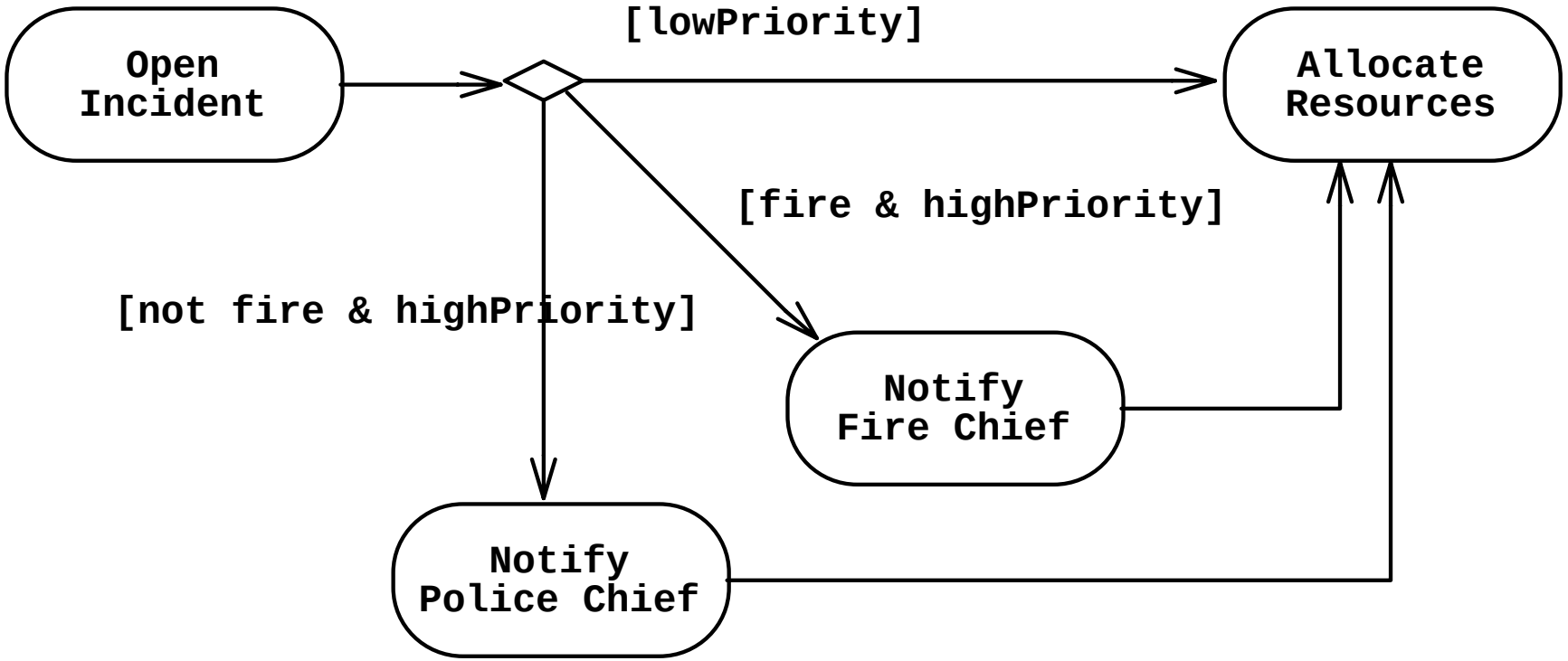


Activity Diagram for Incident (similar to Moore

(State: Operation or Collection of Operations)

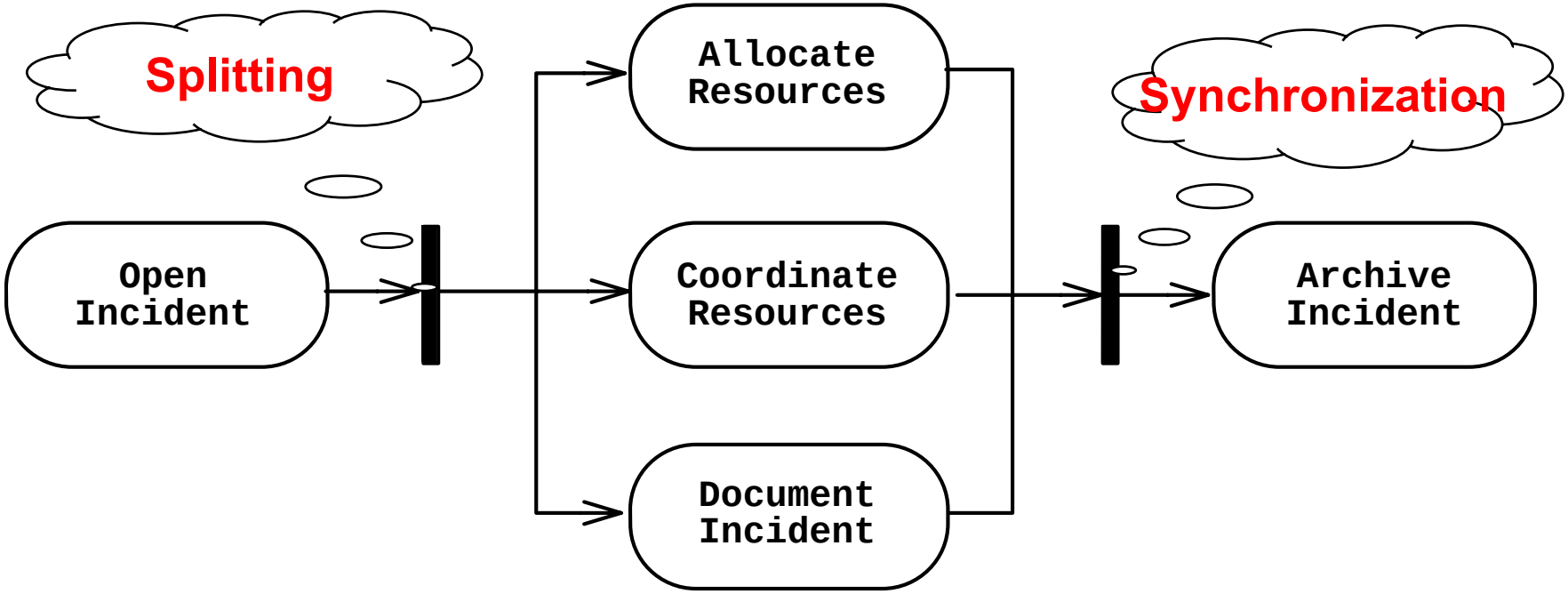


Activity Diagram: Modeling Decisions



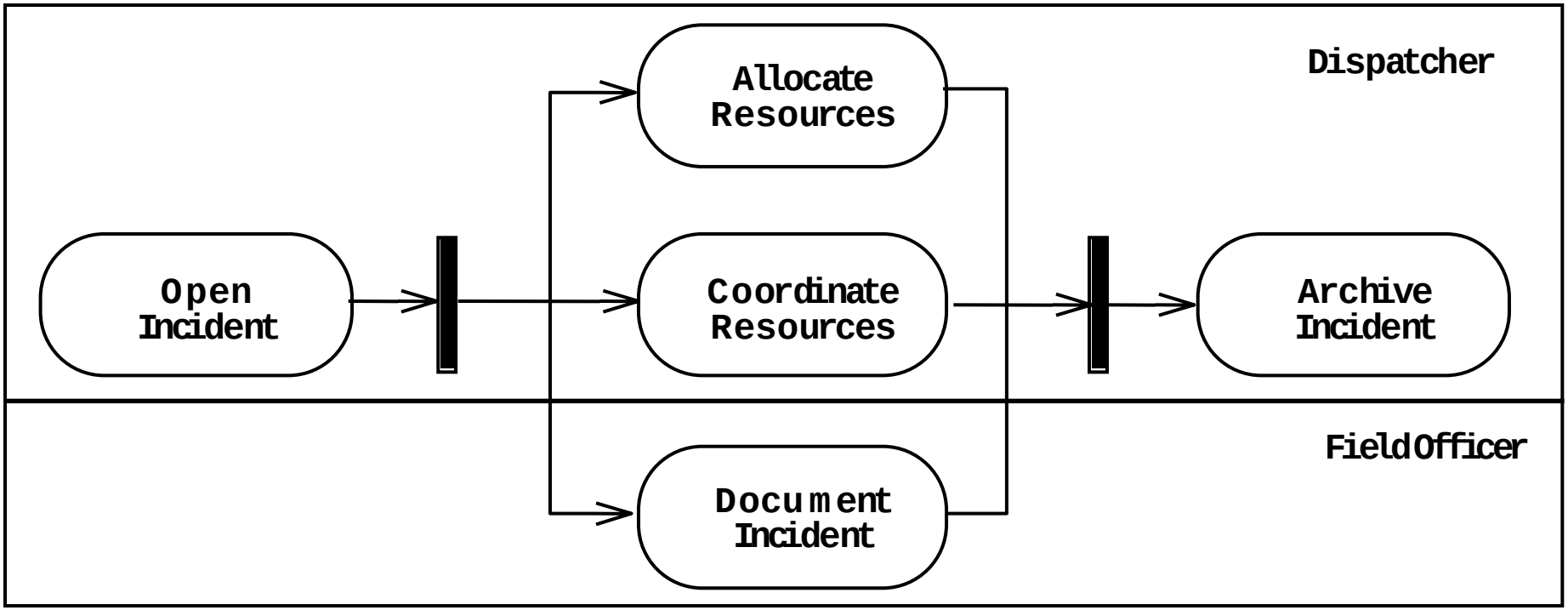
Activity Diagrams: Modeling Concurrency

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



Activity Diagrams: Swimlanes

- Actions may be grouped into swimlanes to denote the object or subsystem that implements the actions.



What should be done first? Coding or Modeling?

- It all depends….

- Forward Engineering:
  - **Creation of code from a model**
  - **Greenfield projects**
- Reverse Engineering:
  - **Creation of a model from code**
  - **Interface or reengineering projects**
- Roundtrip Engineering:
  - **Move constantly between forward and reverse engineering**
  - **Useful when requirements, technology and schedule are changing frequently**

UML Summary

- UML provides a wide variety of notations for representing many aspects of software development
  - **Powerful, but complex language**
  - **Can be misused to generate unreadable models**
  - **Can be misunderstood when using too many exotic features**

- For now we concentrate on a few notations:
  - **Functional model: Use case diagram**
  - **Object model: class diagram**
  - **Dynamic model: sequence diagrams, statechart and activity diagrams**