Object-Oriented Software Engineering

Using UML, Patterns, and Java

Chapter 2, Modeling with UML

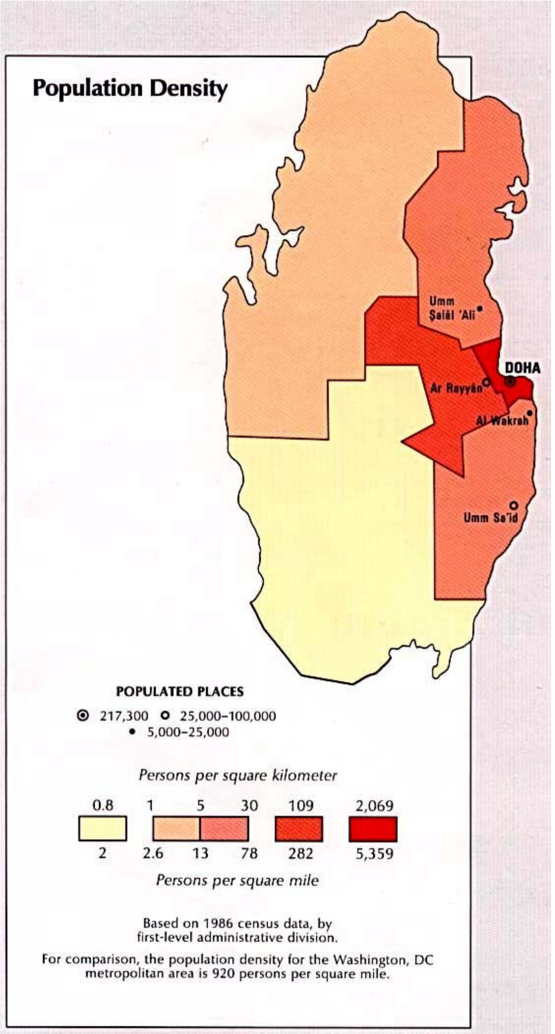
Overview: modeling with UML

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

What is modeling?

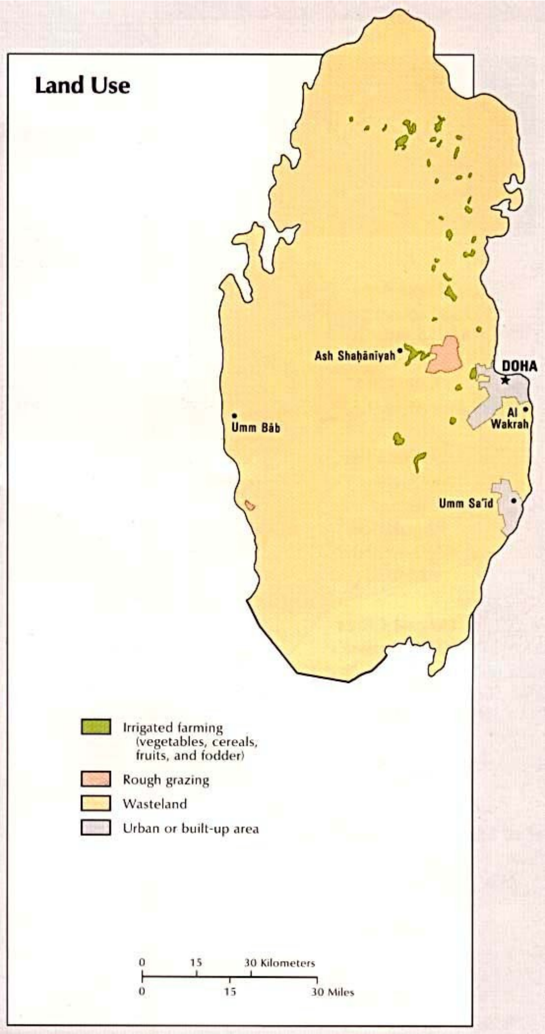
- Modeling consists of building an abstraction of reality
- Abstractions are simplifications because:
  - **They ignore irrelevant details and**
  - **They only represent the relevant details**
- What is *relevant* or *irrelevant* depends on the purpose of the model

Example: Qatar Maps



Based on 1986 census data, by first-level administrative division.

For comparison, the population density for the Washington, DC metropolitan area is 920 persons per square mile.



Why model software?

Why model software?

- Software is getting increasingly more complex
- Modeling is a means for dealing with complexity

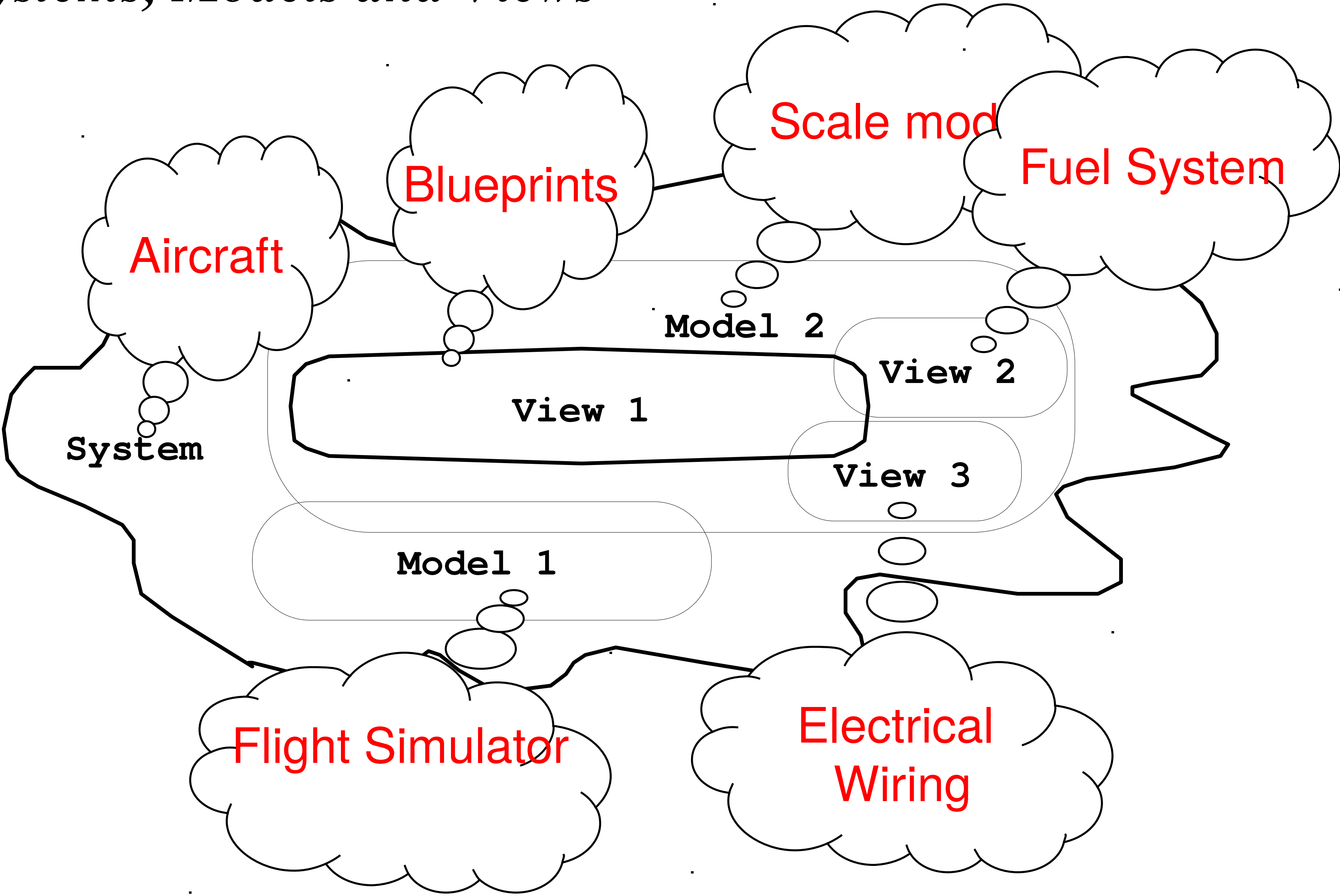
Systems, Models and Views

- A ***model*** is an abstraction describing a subset of a system
- A ***view*** depicts selected aspects of a model
  - **Simplifies a complex model – subset of a model to make it understandable**
- 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: electrical wiring, fuel system (views of the scale model)

Systems, Models and Views

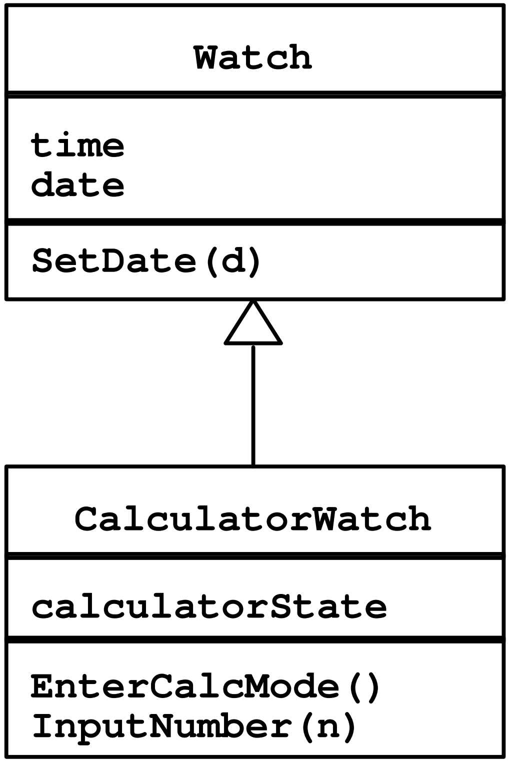


Concepts in software: Data Type and Instance

- Data Type
  - **An abstraction in the context of programming languages**
  - **Has *name*, *members*, and valid *operations***
    - **Name: `int`**
    - **Members: 0, -1, 1, 2, -2, . . . (all signed integers between -2^{32} and 2^{32})**
    - **Operations: +, -, *, integer /, mod, . . .**
- Instance
  - **Member of a specific type**
- The type of a variable represents all possible instances the variable can take

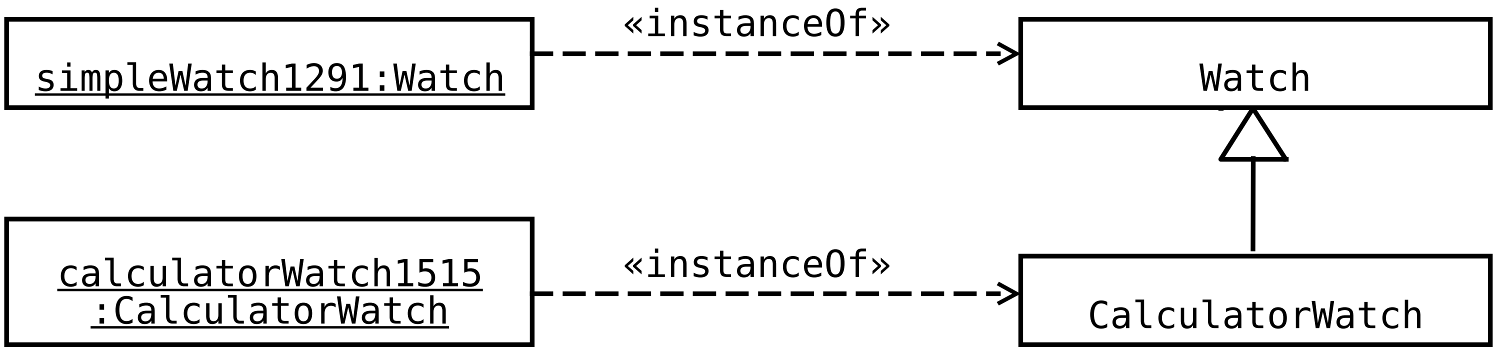
Abstract Data Types & Classes

- Abstract data type
  - **Special type whose implementation is hidden from the rest of the system.**
  - **E.g. `set, stack`, etc.**
- Class
  - **An abstraction in the context of object-oriented languages**
  - **Like an abstract data type, a class encapsulates both state (variables) and behavior (methods)**
  - **Unlike abstract data types, classes can be defined in terms of other classes using inheritance**



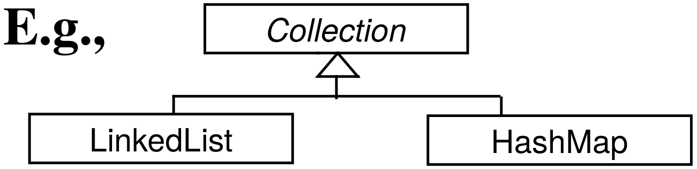
Objects

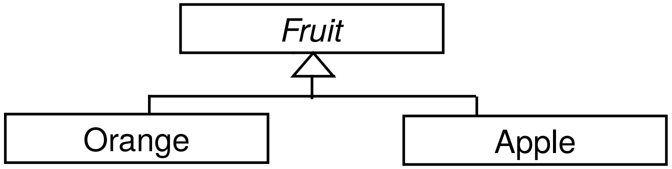
- Object
  - **Instance of a class**
  - **Represented by a rectangle with name underlined**
  - **Notice: `CalculatorWach1515` is a watch but it is not an instance of `Watch`**



Subclasses and Superclasses

- Superclass
  - **Generalization class**
- Subclass
  - **Specialization of its superclass**
  - **Refines the superclass by defining new attributes and operations**
- Abstract class
  - **Generalized concepts that model shared attributes and operations**
  - **Can not be instantiated**
  - **Name italicized**
  - **E.g.,**



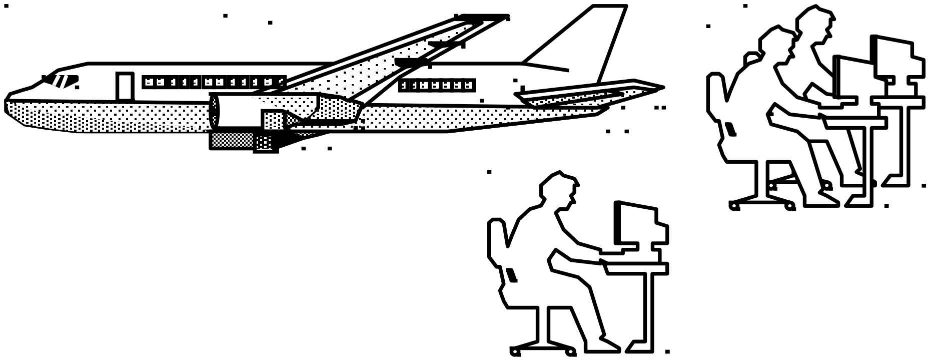


Application and Solution Domain

- Application Domain (Requirements Analysis):
  - **Represents aspects of the user's problem**
    - **Environment of the system, users, work processes, etc.**

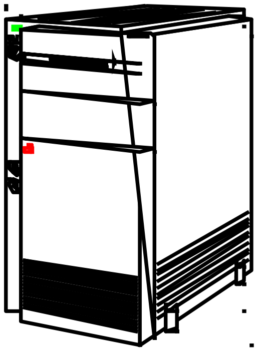
- Solution Domain (System Design, Object Design):
  - **The available technologies to build the system**

Object-oriented modeling



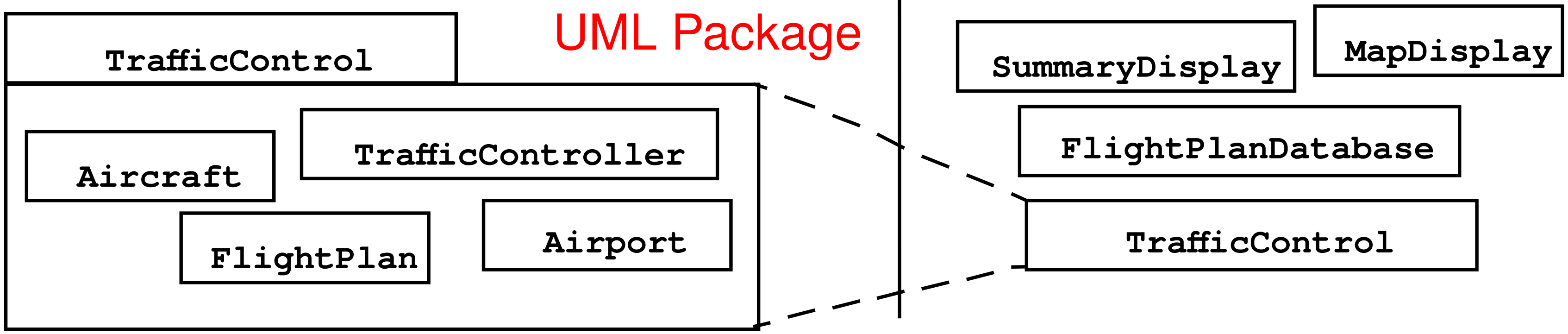
Application Domain

Application Domain Model (OO Analysis)



Solution Domain

System Model (OO Design)



TrafficControl

Aircraft

TrafficController

FlightPlan

Airport

UML Package

SummaryDisplay

MapDisplay

FlightPlanDatabase

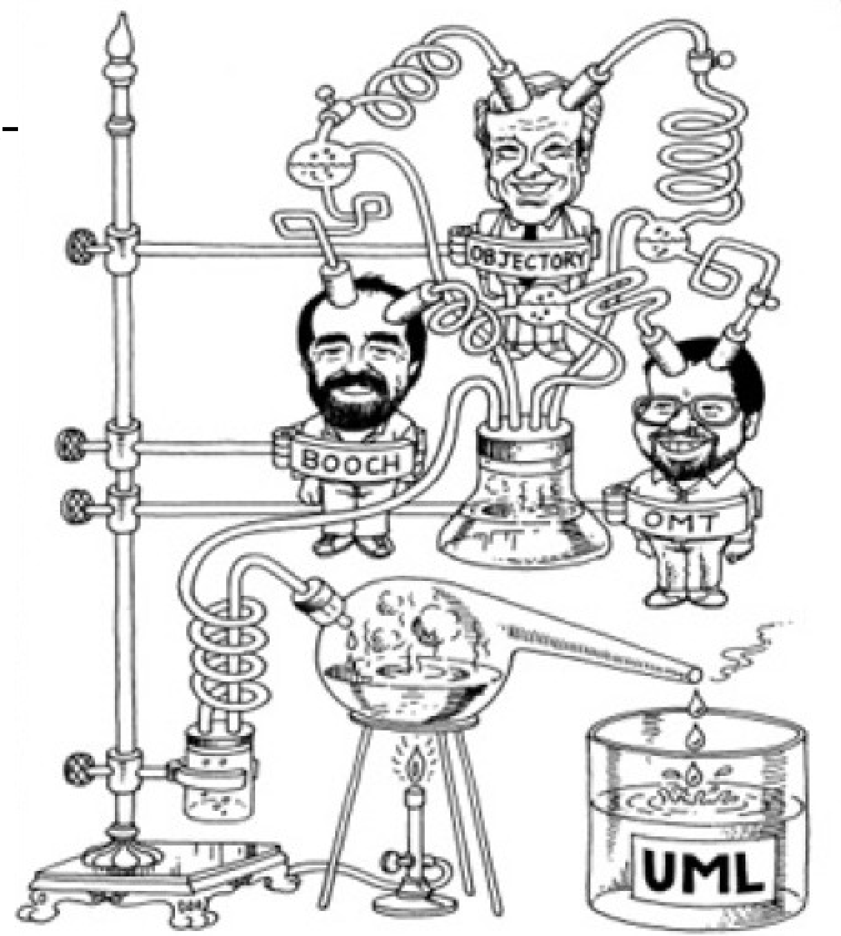
TrafficControl

What is UML?

- *"The UML is the standard language for <u>specifying</u>, <u>visualizing</u>, <u>constructing</u>, and <u>documenting</u> all the artifacts of a software system."*
- An emerging standard notation (by Object Management Group--OMG) for modeling object-oriented software
- Unified Modeling Language
  - **Effective for modeling large, complex software systems**
  - **It can specify systems in an implementation-independent manner**
- It is simple to learn for most developers, but provides advanced features for expert analysts, designers and architects
  - **20% of the constructs are used 80% of the time**

About UML?

- Resulted from the convergence of notations from three leading object-oriented methods (early 90's:
  - **OMT (James Rumbaugh)**
  - **OOSE (Ivar Jacobson)**
  - **Booch (Grady Booch)**
- The above three OO gurus joined forces in one company, Rational (now IBM)
  - **ROSE, Rational Unified Process (RUP), UML**
- Supported by several CASE tools
  - **Rational ROSE**
  - **TogetherJ**
  - **Eclipse UML plugin (in our lab)**



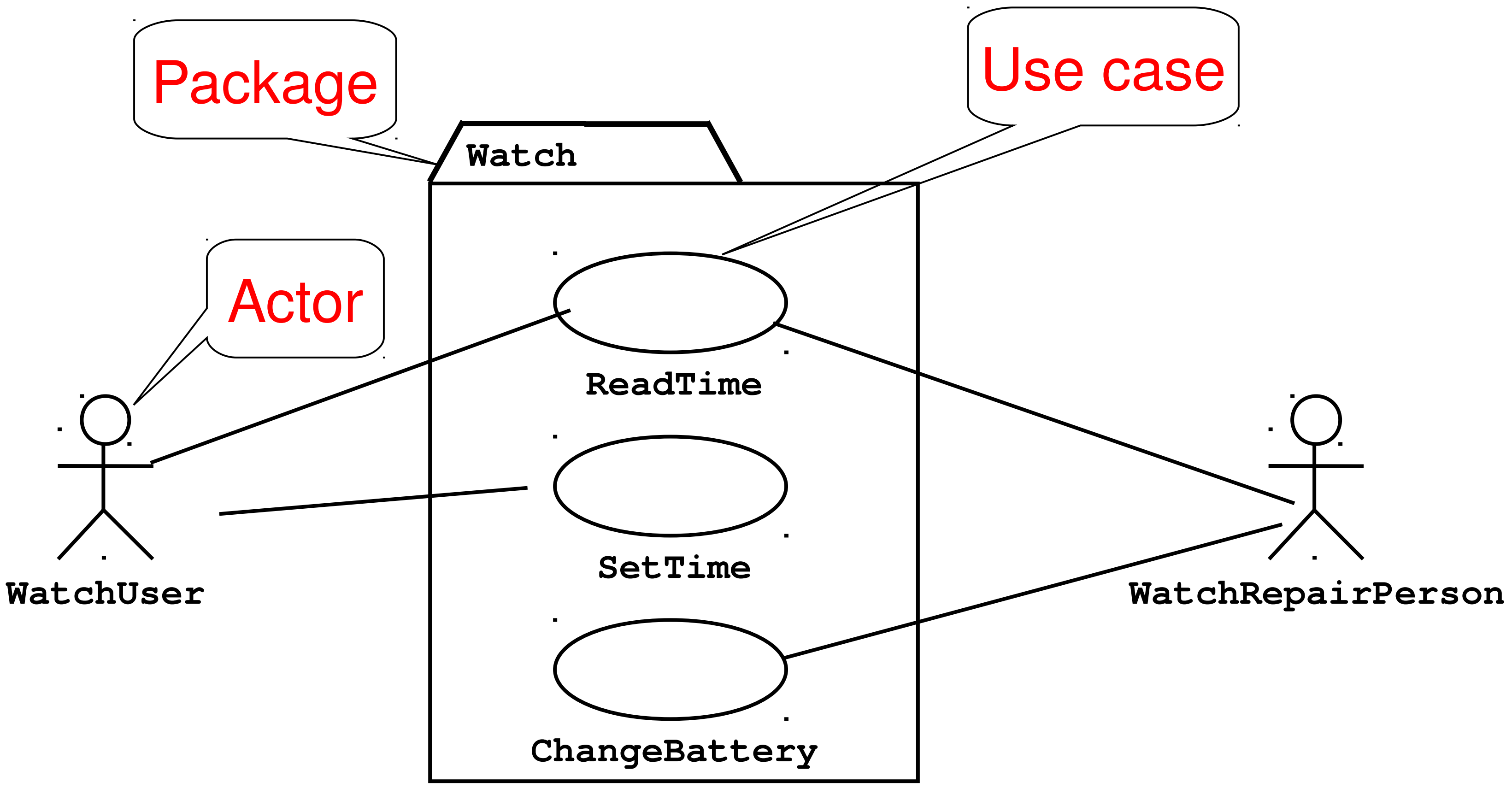
UML: First Pass

- You can model 80% of most problems by using about 20 % UML
- We will see these 20%

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 actors and the system and between objects of the system**
- Statechart diagrams
  - **Describe the dynamic behavior of an individual object (essentially a finite state automaton)**
- Activity Diagrams
  - **Model the dynamic behavior of a system, in particular the data flow or control flow through a system (essentially a flowchart)**

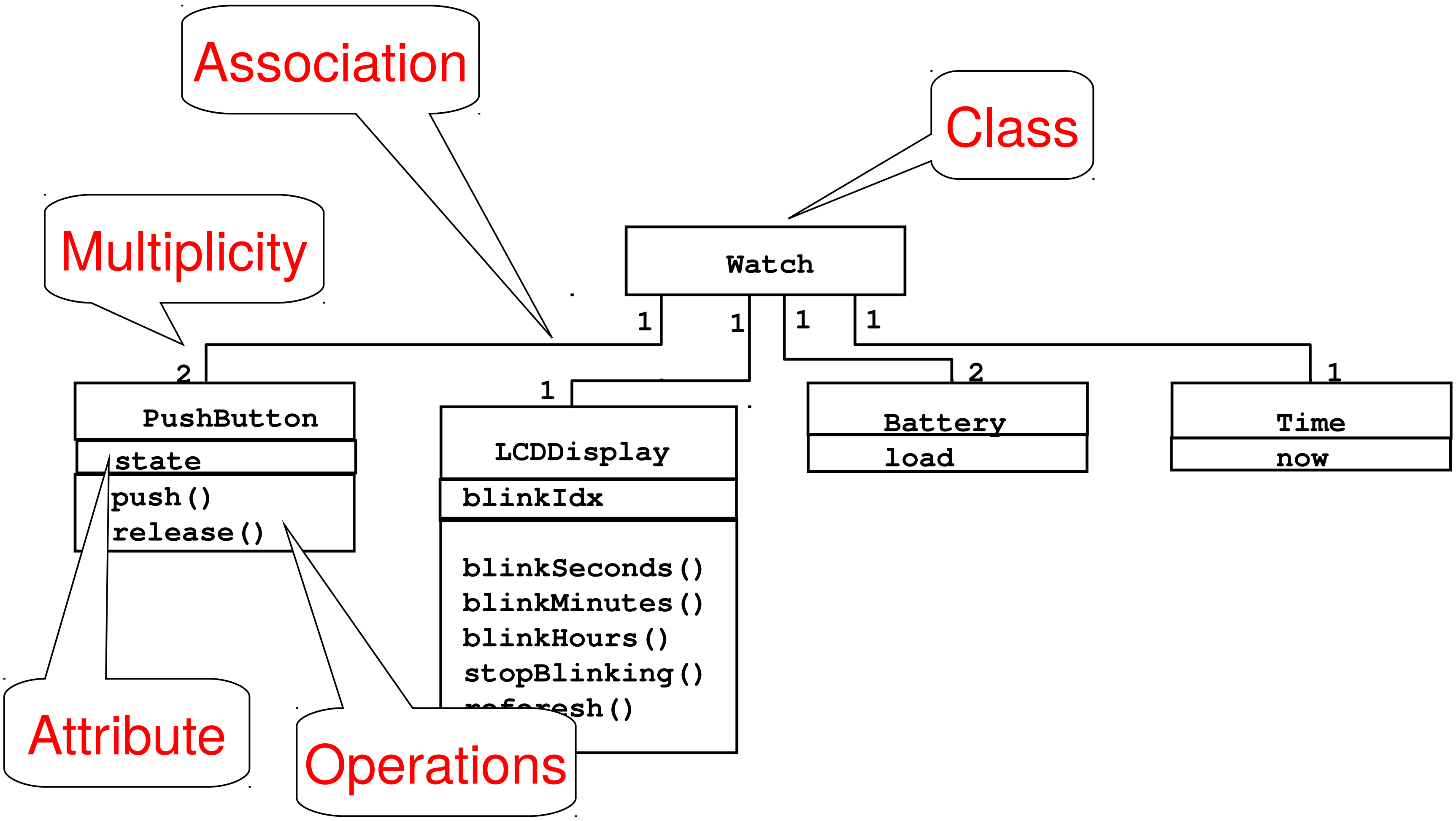
UML first pass: Use case diagrams



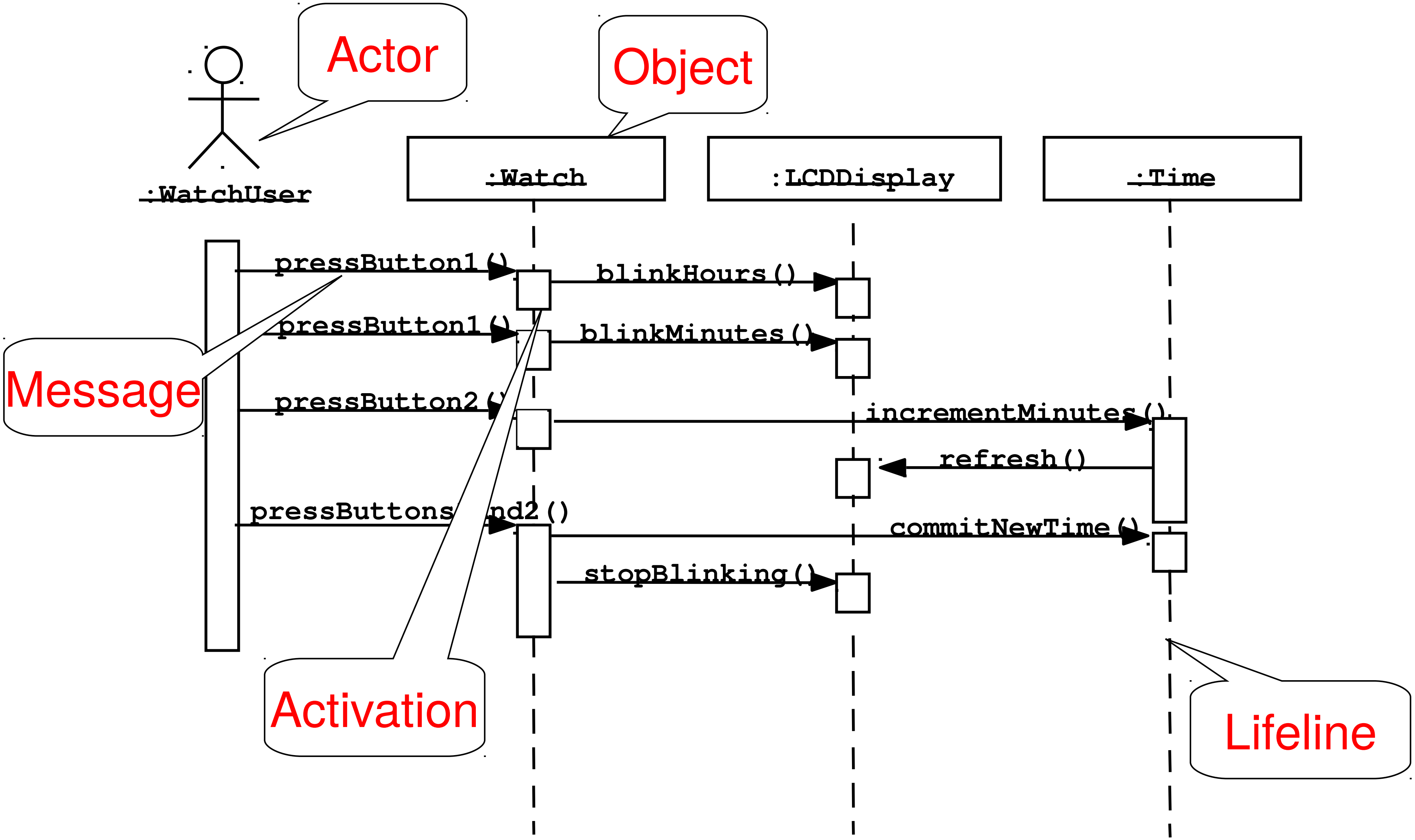
Use case diagrams represent the functionality of the system from user’s point of view

UML first pass: Class diagrams

Class diagrams represent the structure of the system

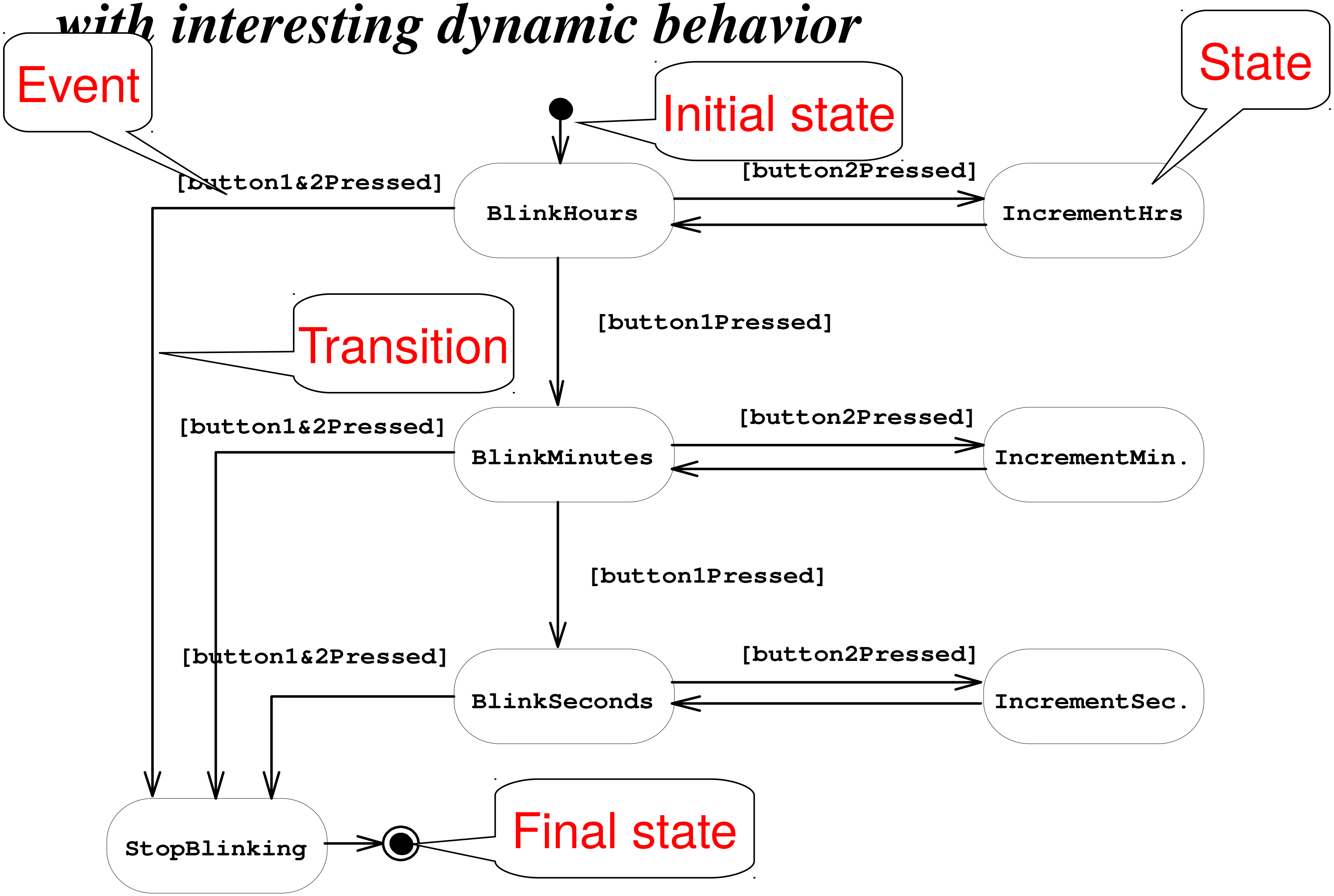


UML first pass: Sequence diagram



Sequence diagrams represent the behavior as interactions

UML first pass: Statechart diagrams for objects with interesting dynamic behavior



Represent behavior as states and transitions

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**

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

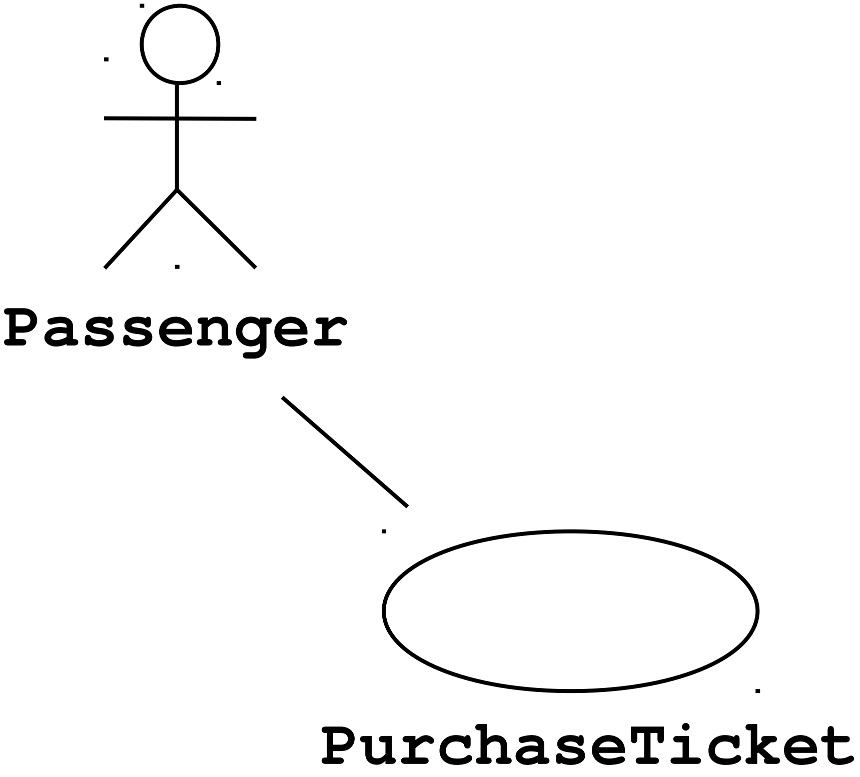
Use Case Diagrams

What is a Use Case Model?

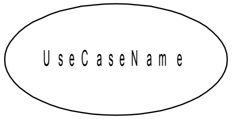
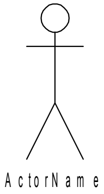
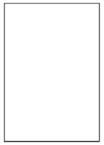
- A view of a system that emphasizes the behavior as it appears to outside users. A use case model partitions system functionality into transactions ('use cases') that are meaningful to users ('actors').
  - *Actors* **represent external entities, E.g.,**
    - **Roles: user, bank teller, customer, system administrator**
    - **Other systems: company DB, Web server, etc.**
  - *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 system functionality.

Use Case Diagram

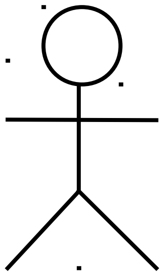
- Shows use cases, actor and their relationships
- Use case internals are typically specified by textual description
- Use case include
  - **use case diagram**
  - **use case description**



Use Case Diagram Elements

Construct	Description	Syntax
use case	A sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system.	
actor	A coherent set of roles that users of use cases play when interacting with these use cases.	
system boundary	Represents the boundary between the physical system and the actors who interact with the physical system.	

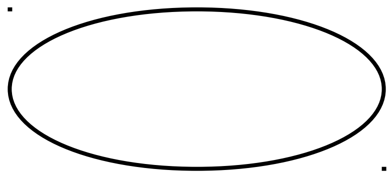
Actors



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.

A use case consists of:

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

Use Case Description: 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.

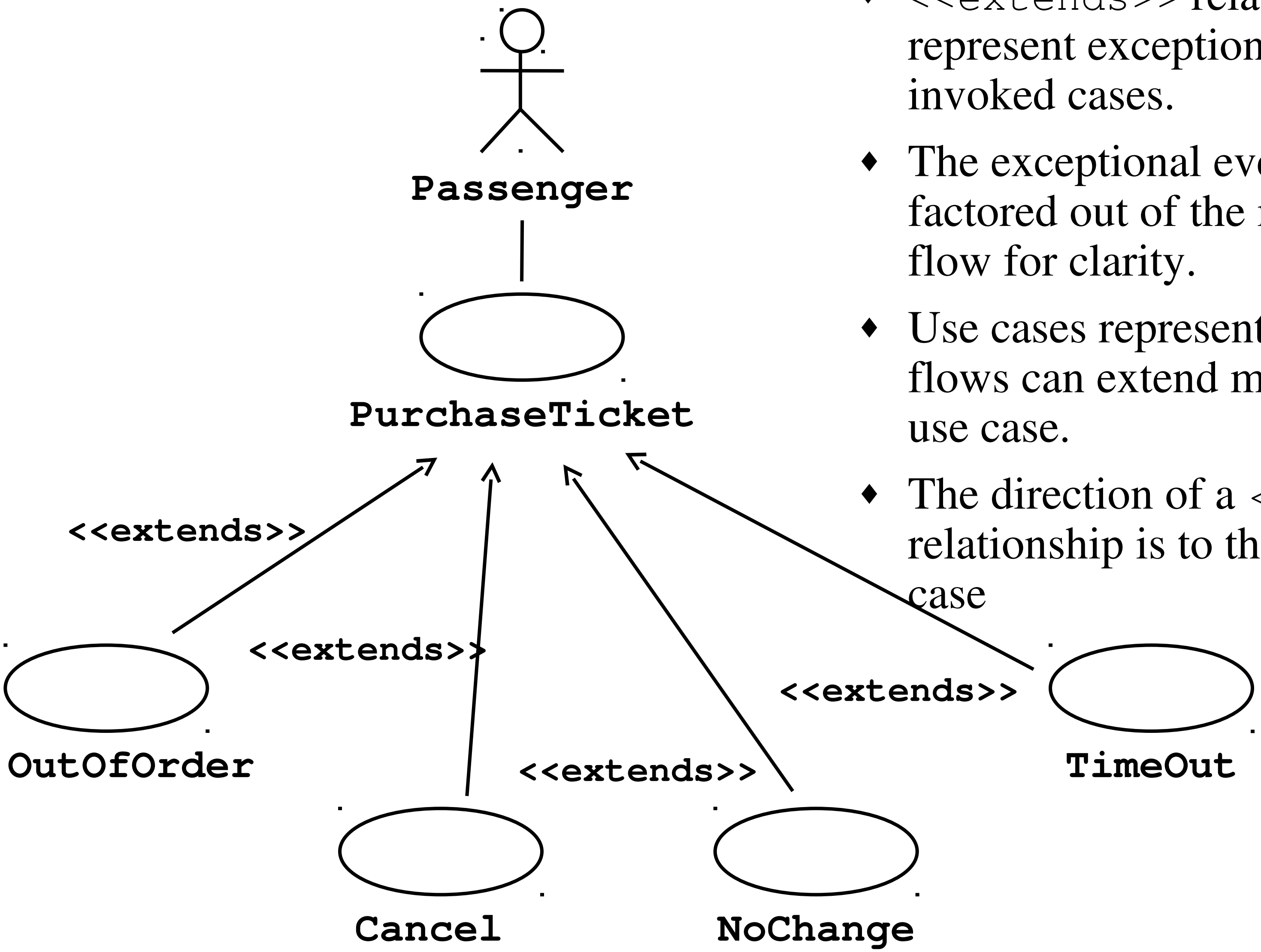
Event flow:

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?

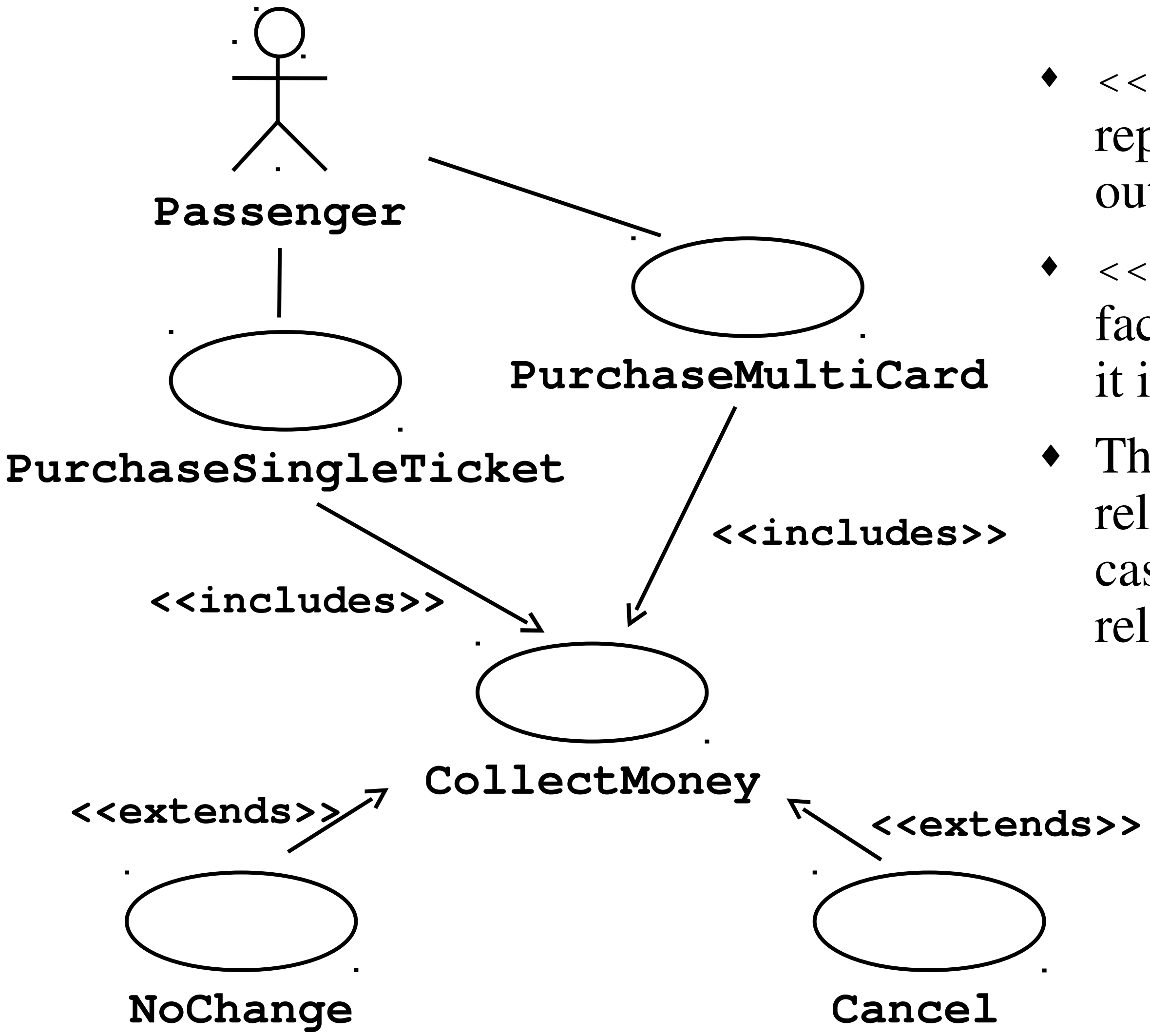
Exceptional cases!

The `<<extends>>` *Relationship*



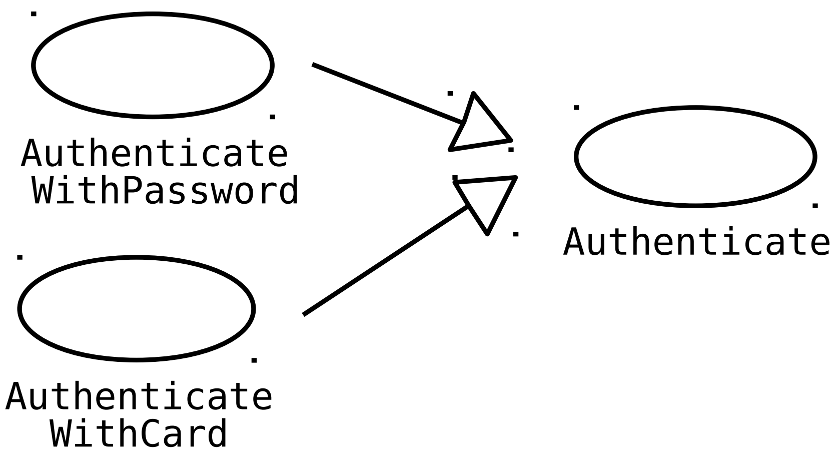
- `<<extends>>` relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a `<<extends>>` relationship is to the extended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

The Inheritance Relationship



- Used when one use case can specialize another more general one by adding more detail
- In the use case description the specialized use case inherits the initiating actor, the entry condition, and the exit condition.

Use Case Relationships Summary

Construct	Description	Syntax
Association/ Communication	The participation of an actor in a use case. I.e., instance of an actor and instances of a use case communicate with each other	――――
Generalization/ Inheritance	A taxonomic relationship between a more general use case and a more specific use case	――――▷
Extend	A relationship from an extension use case to a base use case, specifying how the behavior of the extension use case can be inserted into the base use case	<<extend>> - - - - - - ->
Include	A relationship from an base use case to a inclusion use case, specifying how the behavior of the use case is inserted into the behavior defined for the base use case	<<include>> - - - - - - ->

Use Case Diagram Example

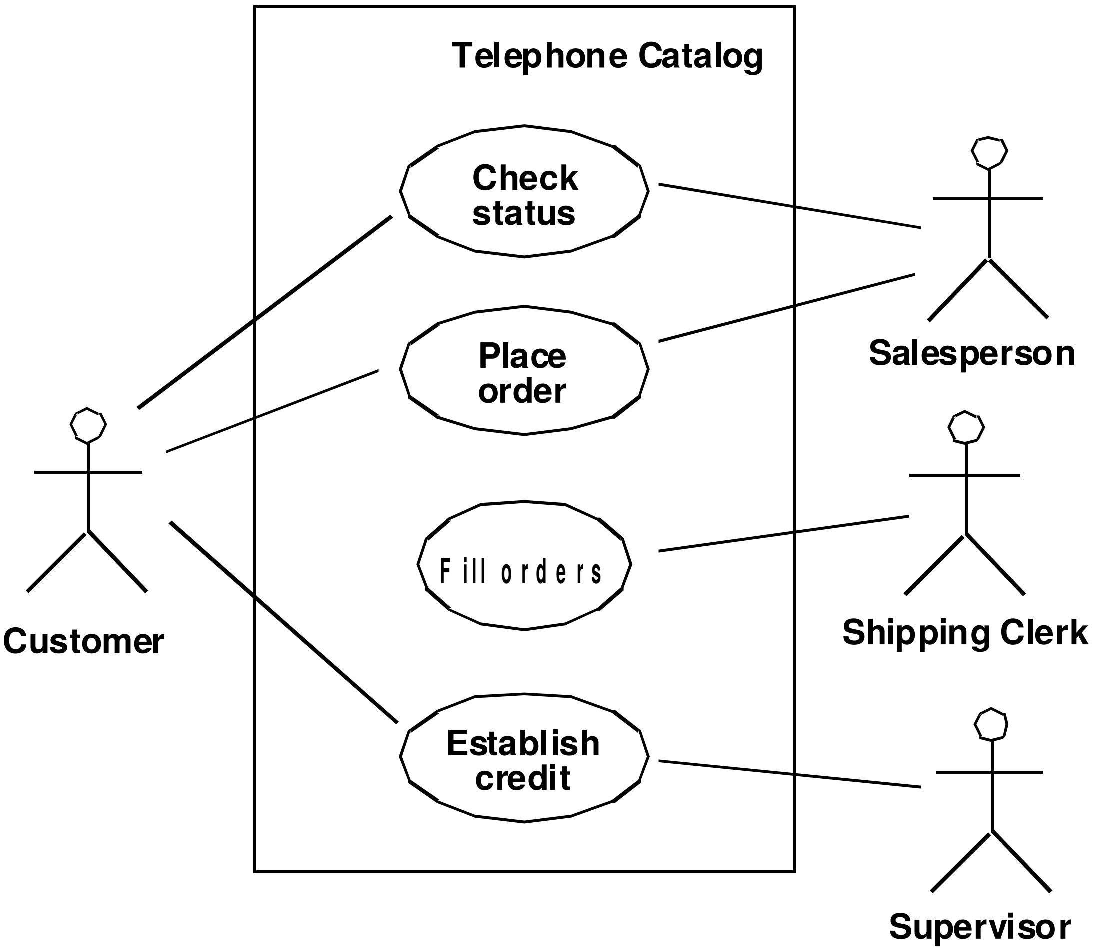
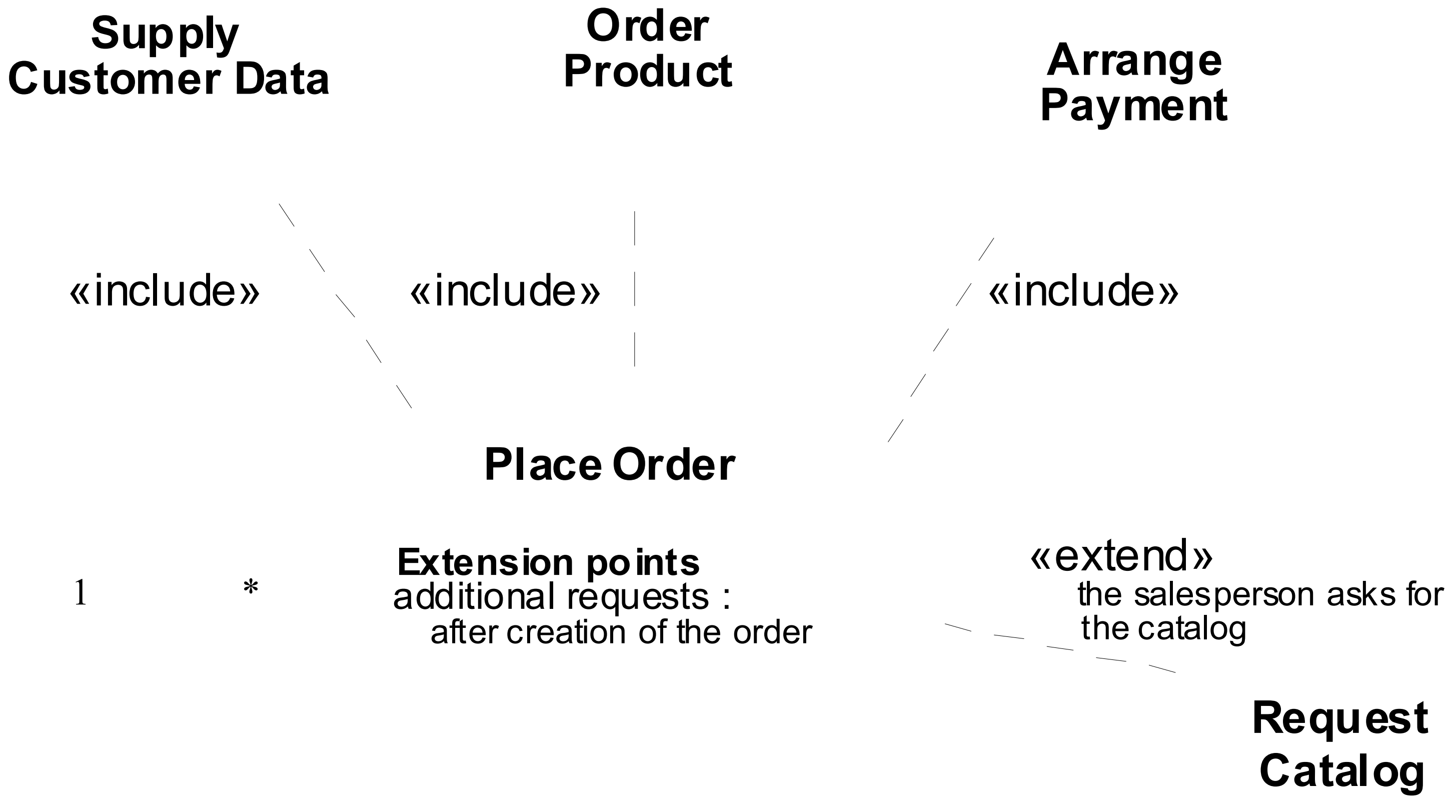


Fig. 3-53, *UML Notation Guide*

Use Case Relationships Example



Supply Customer Data

Order Product

Arrange Payment

«include»

«include»

«include»

Place Order

1

*

Extension points
additional requests :
after creation of the order

«extend»
the salesperson asks for
the catalog

Request Catalog

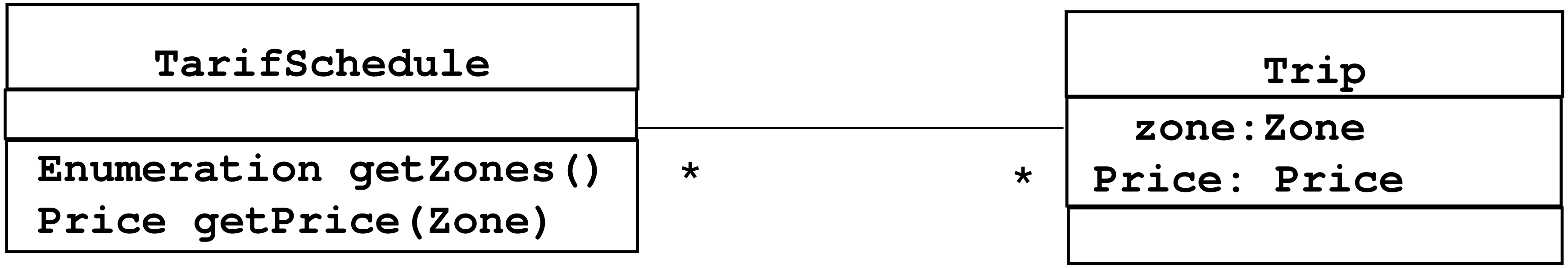
Fig. 3-54, *UML Notation Guide*

Use Case Diagrams: Summary

- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases (descriptions)
- 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.
- Use cases can be used for
  - **Model user requirements**
  - **Model test scenarios**

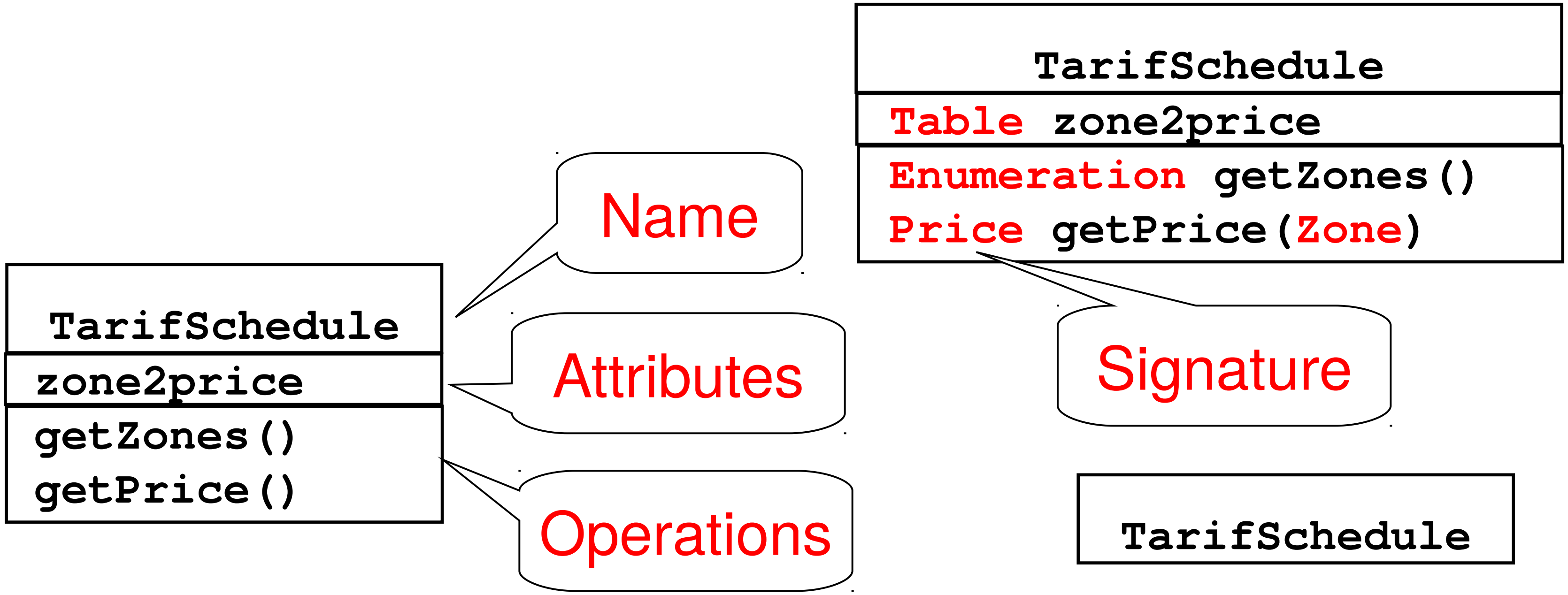
Class Diagrams

Class Diagrams



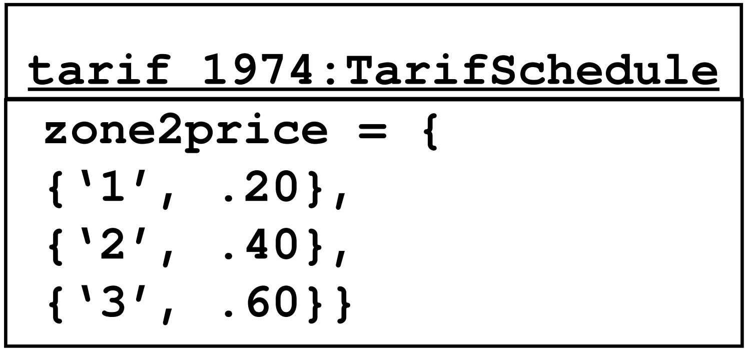
- Class diagrams represent the structure of the system.
- Used
    - **during requirements analysis to model problem domain concepts**
    - **during system design to model subsystems and interfaces**
    - **during object design to model classes.**

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.

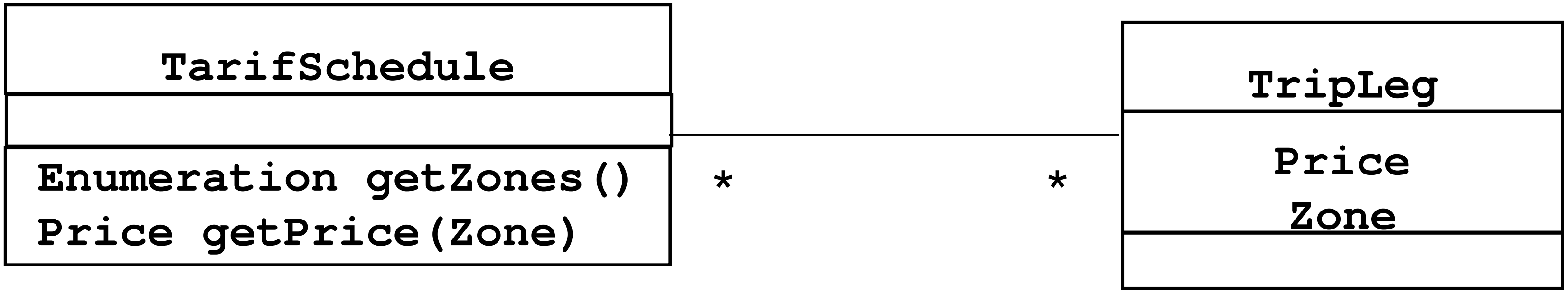
Instances



```
tarif 1974:TarifSchedule
zone2price = {
{‘1’, .20},
{‘2’, .40},
{‘3’, .60}}
```

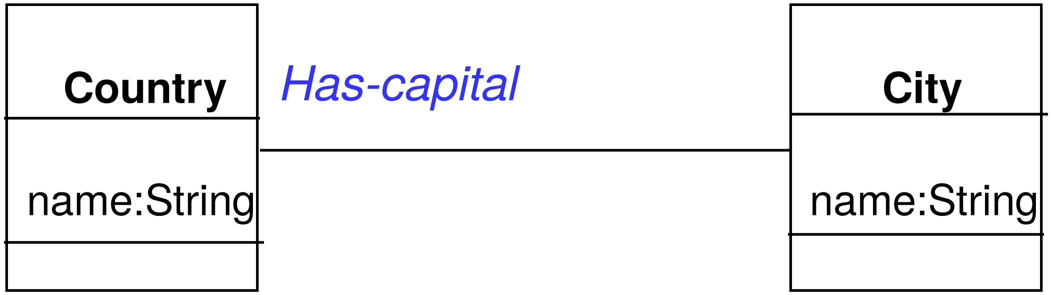
- An ***instance*** represents a phenomenon.
- The name of an instance is <u>underlined</u> and can contain the class of the instance.
- The attributes are represented with their ***values***.

Associations

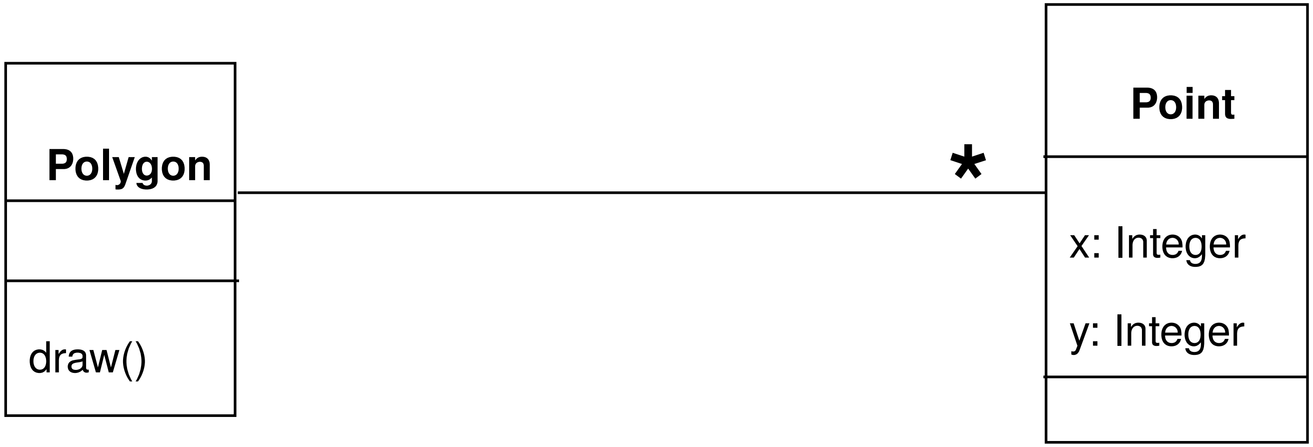


- Associations denote relationships between classes.
- The *multiplicity* of an association end denotes how many objects the source object can legitimately reference.

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

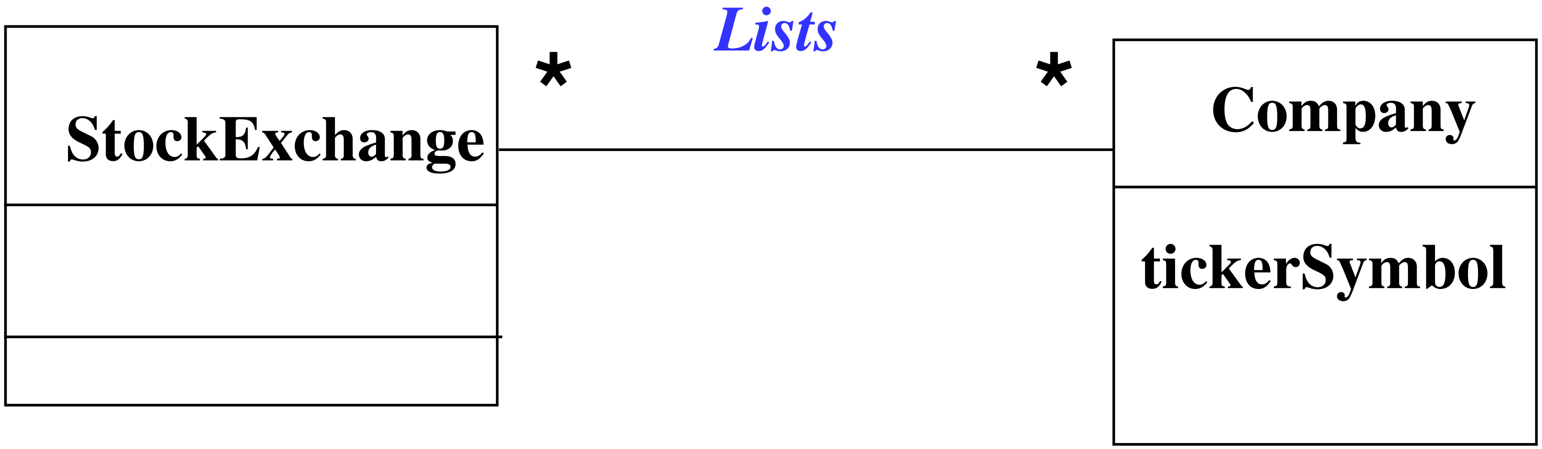


One-to-one association



One-to-many association

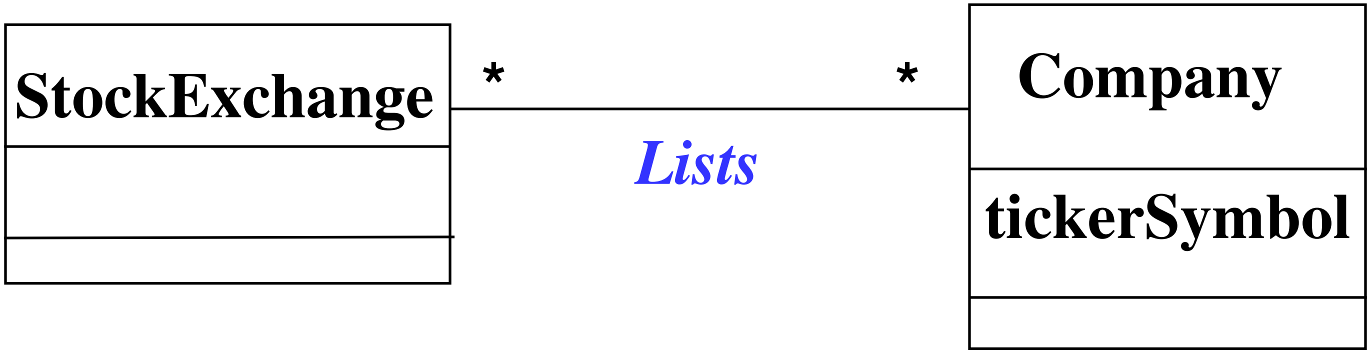
Many-to-Many Associations



From Problem Statement To Object Model

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

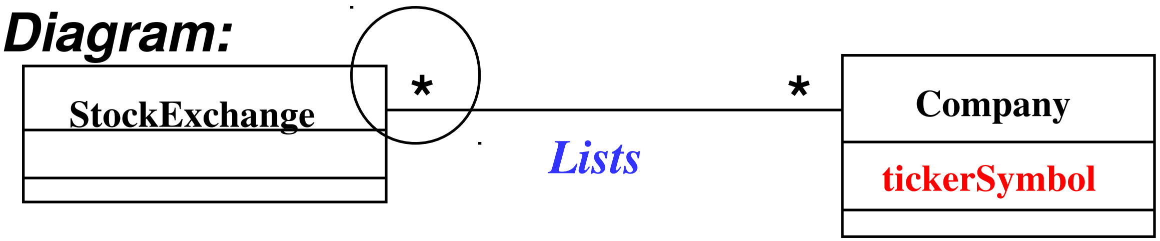
Class Diagram:



From Problem Statement to Code

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

Class Diagram:

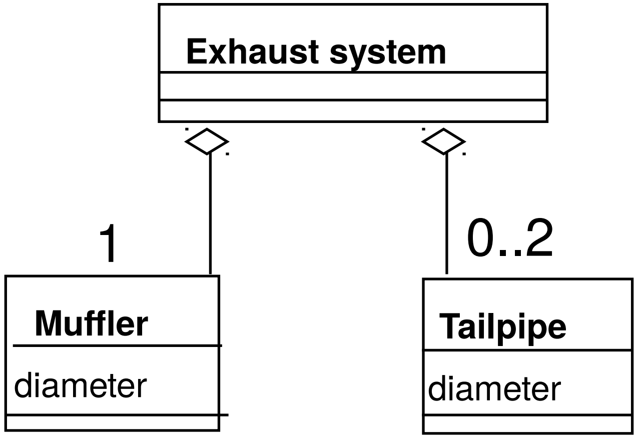


Java Code

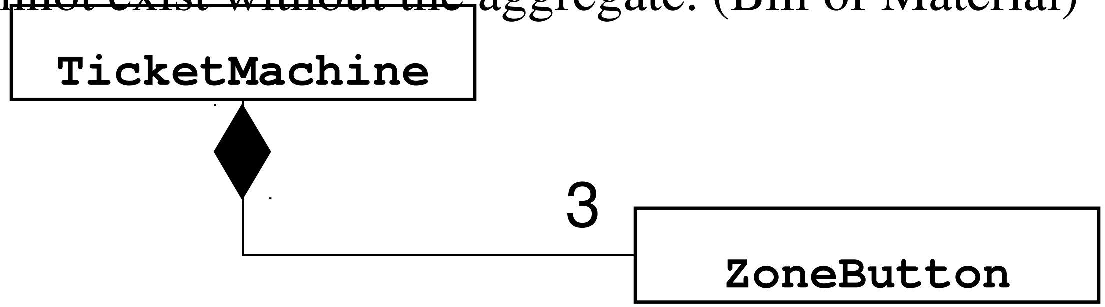
```
public class StockExchange
{
 private Vector m_Company = new Vector();
};

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

Aggregation

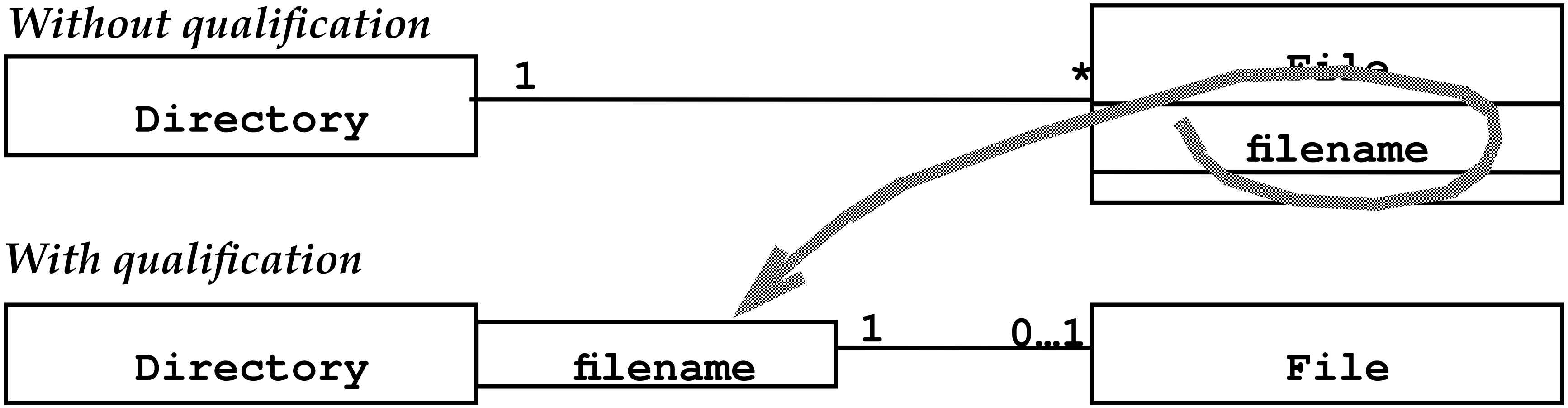
- An ***aggregation*** is a special case of association denoting a “consists of” hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children class.



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

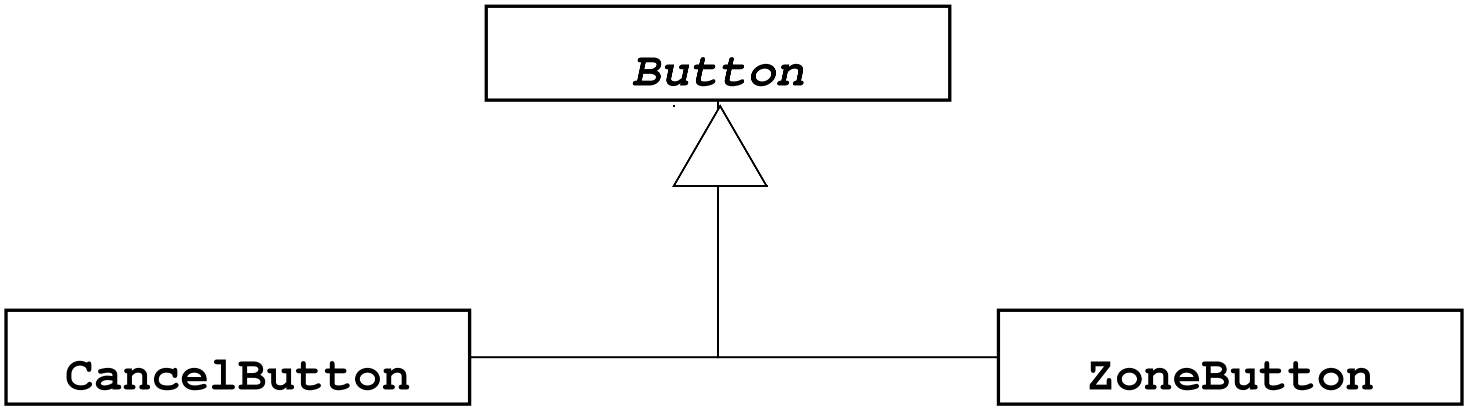


Qualifiers



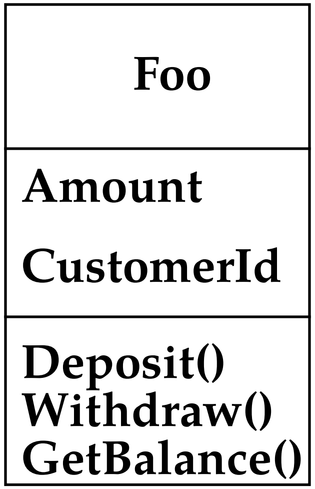
- Qualifiers can be used to reduce the multiplicity of an association.

Inheritance



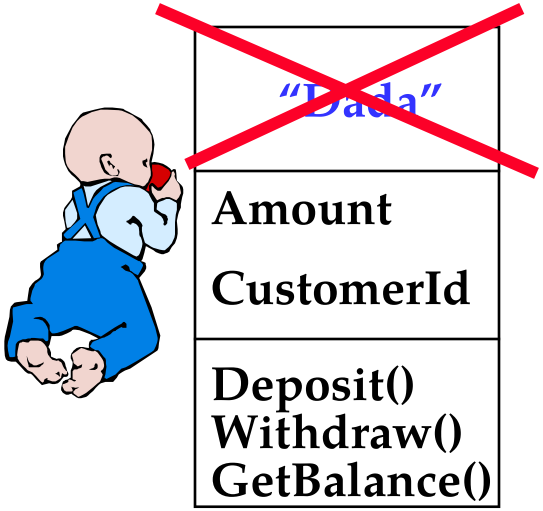
- The **children classes** inherit the attributes and operations of the **parent class**.
- Inheritance simplifies the model by eliminating redundancy.

Object Modeling in Practice: Class Identification



Class Identification: Name of Class, Attributes and Methods

Object Modeling in Practice: Encourage Brainstorming

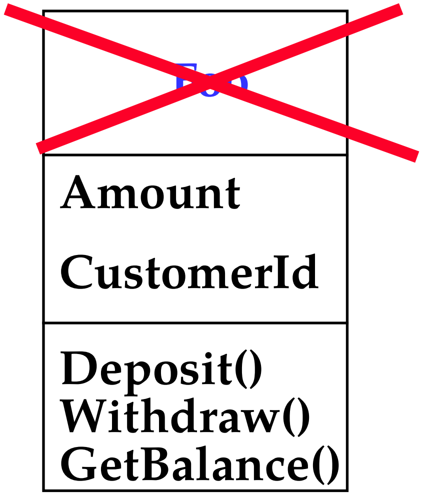


~~“Dada”~~

Amount

CustomerId

Deposit()
Withdraw()
GetBalance()

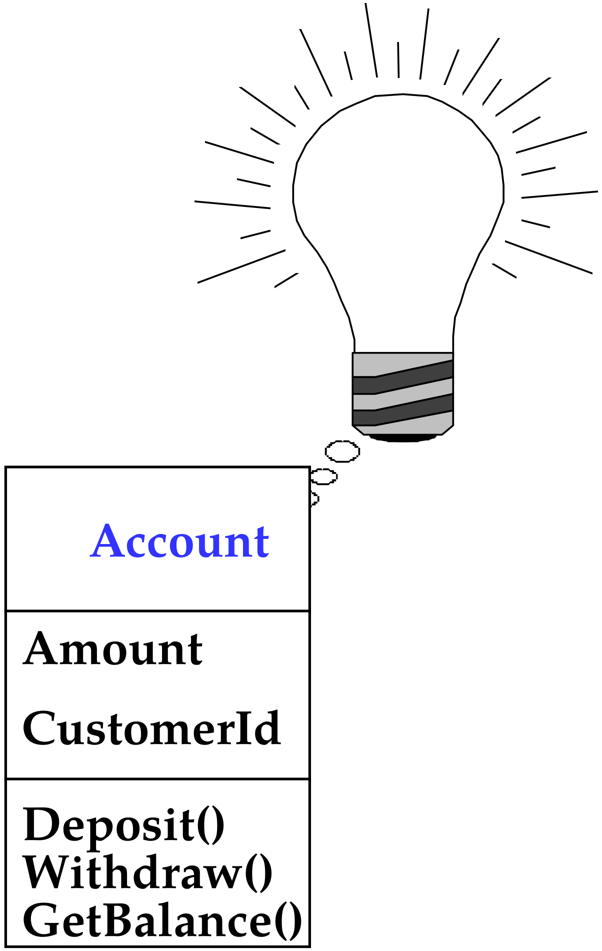


~~Foo~~

Amount

CustomerId

Deposit()
Withdraw()
GetBalance()



Account

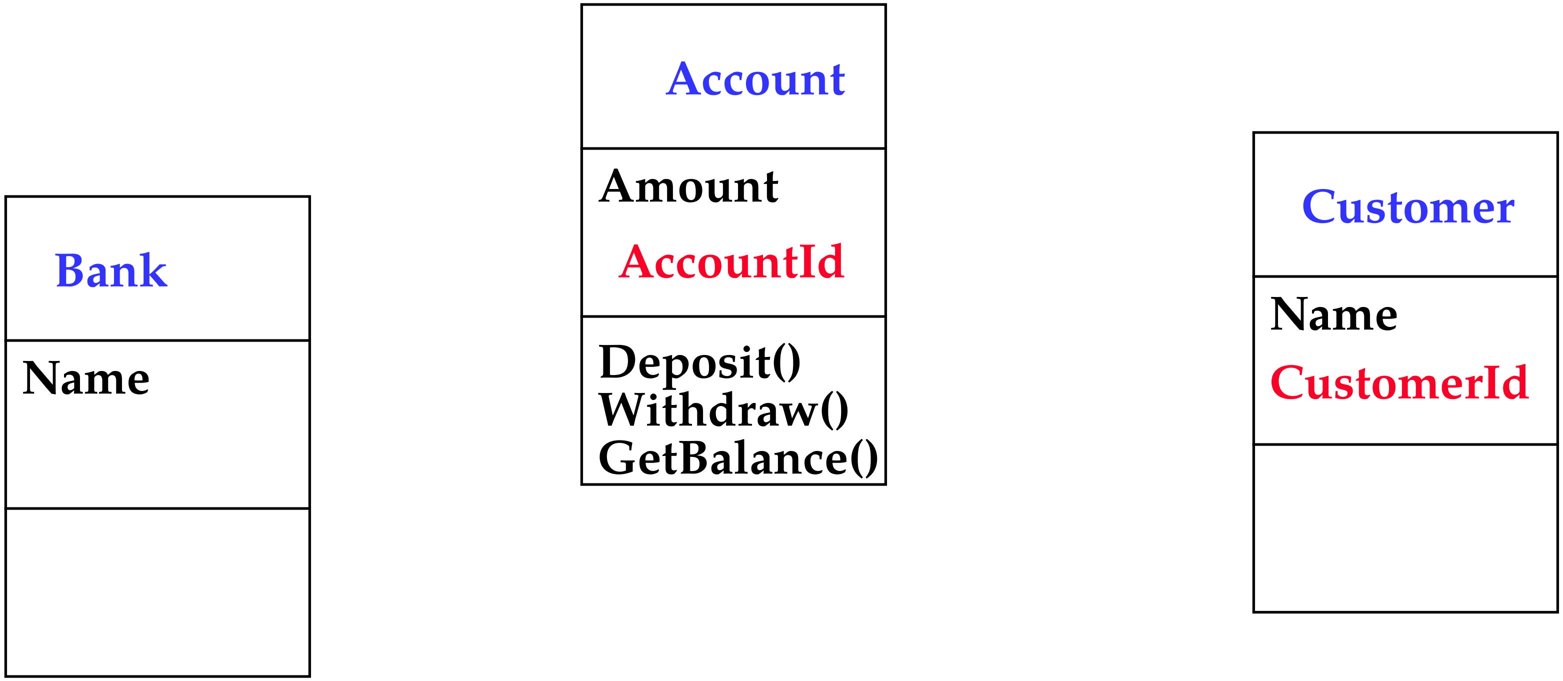
Amount

CustomerId

Deposit()
Withdraw()
GetBalance()

Naming is important!
Is Foo the right name?

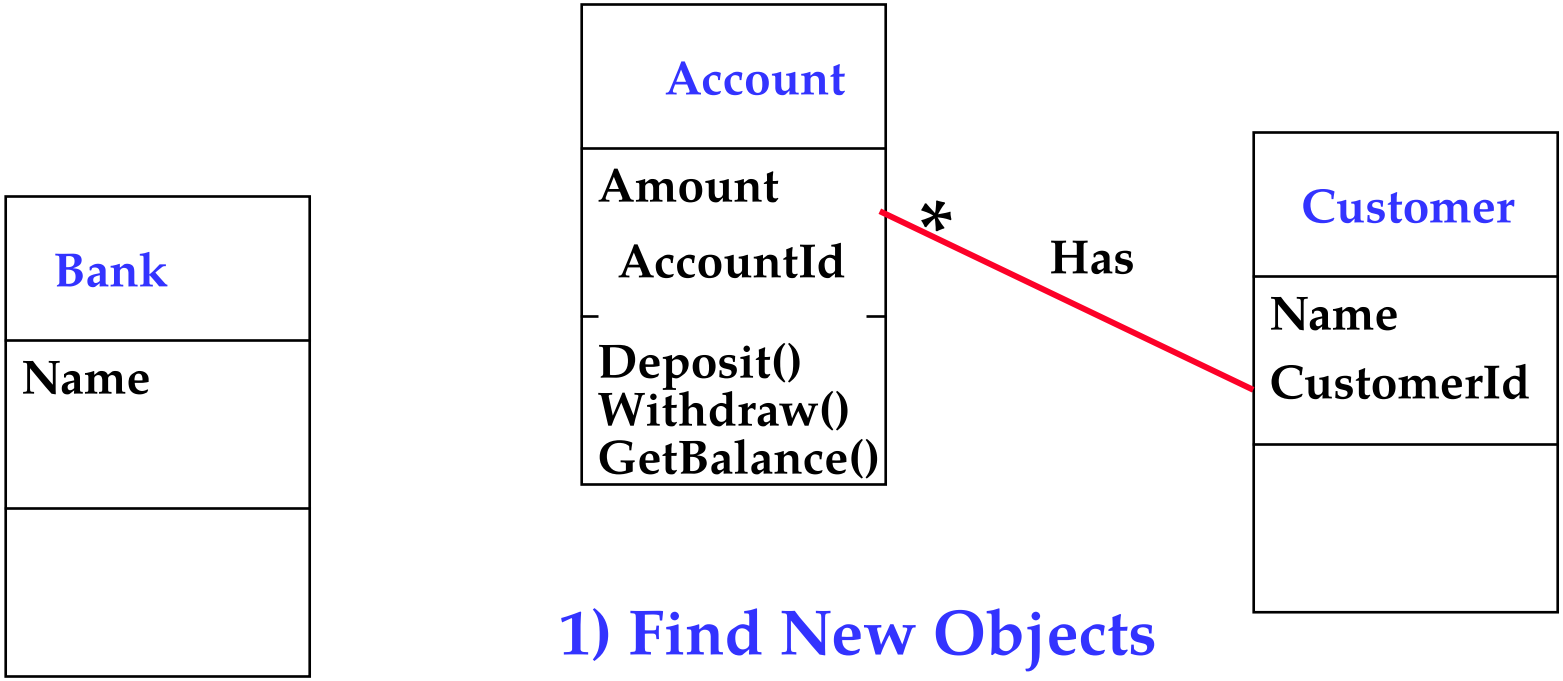
Object Modeling in Practice ctd



1) Find New Objects

2) Iterate on Names, Attributes and Methods

Object Modeling in Practice: A Banking System



1) Find New Objects

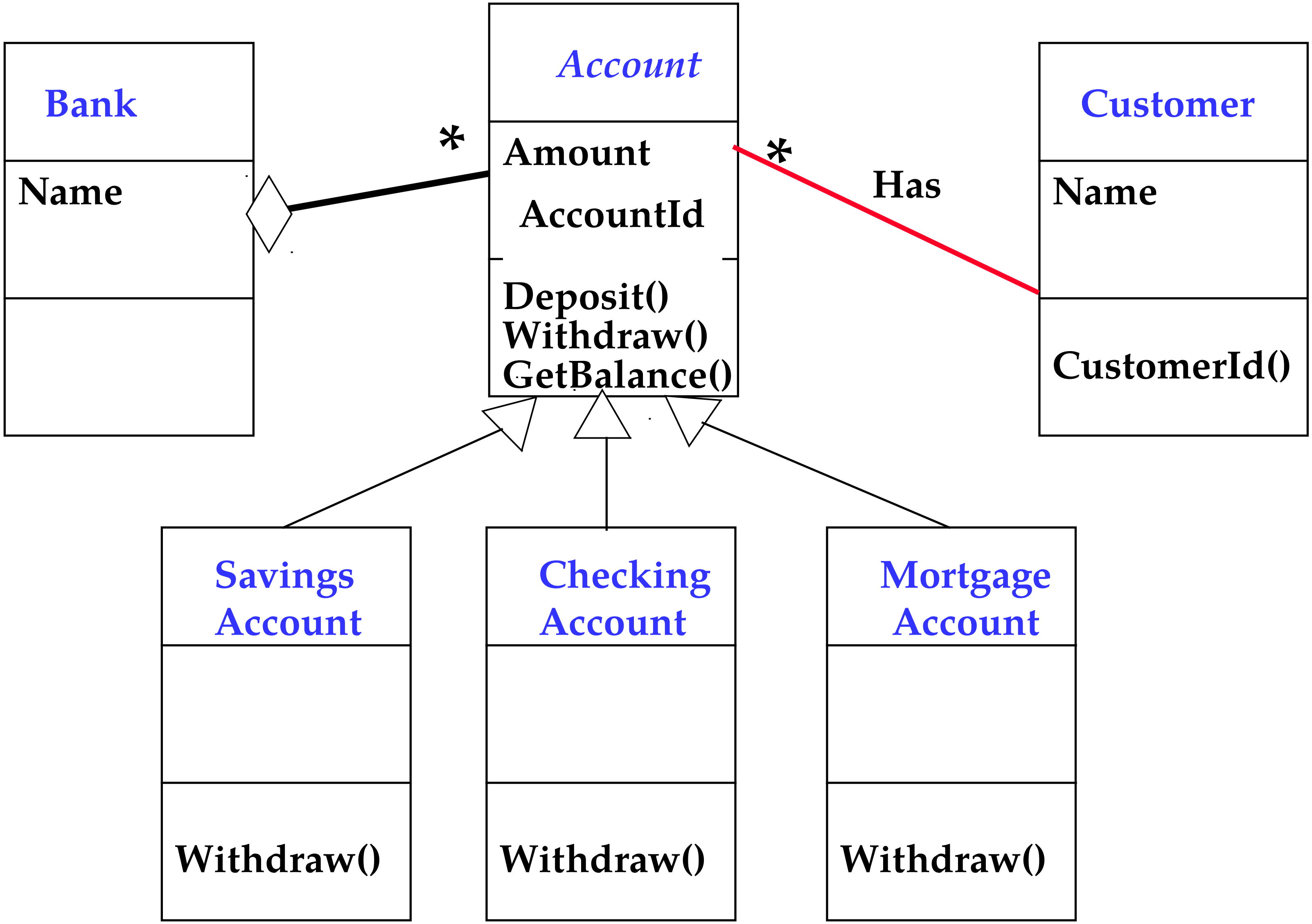
2) Iterate on Names, Attributes and Methods

3) Find Associations between Objects

4) Label the assocations

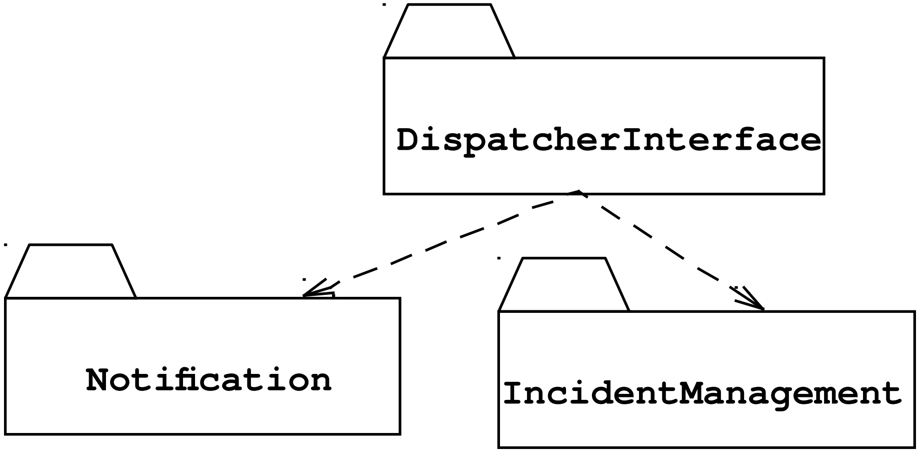
5) Determine the multiplicity of the assocations

Practice Object Modeling: Iterate, Categorize!



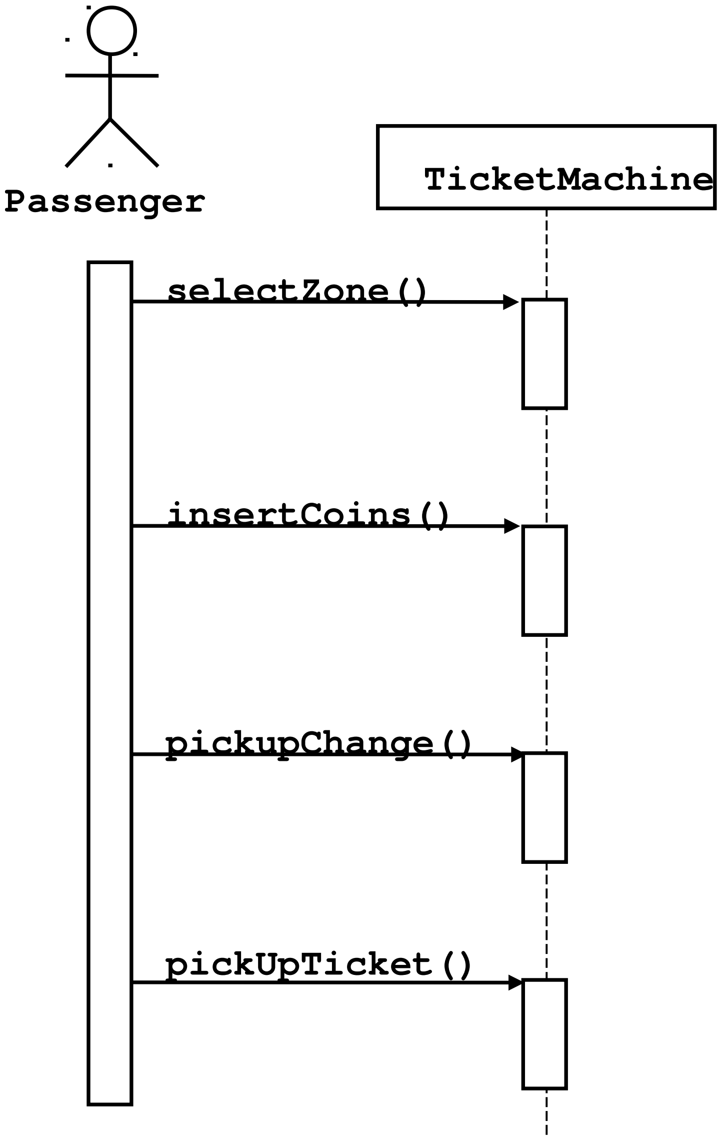
Packages

- 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.



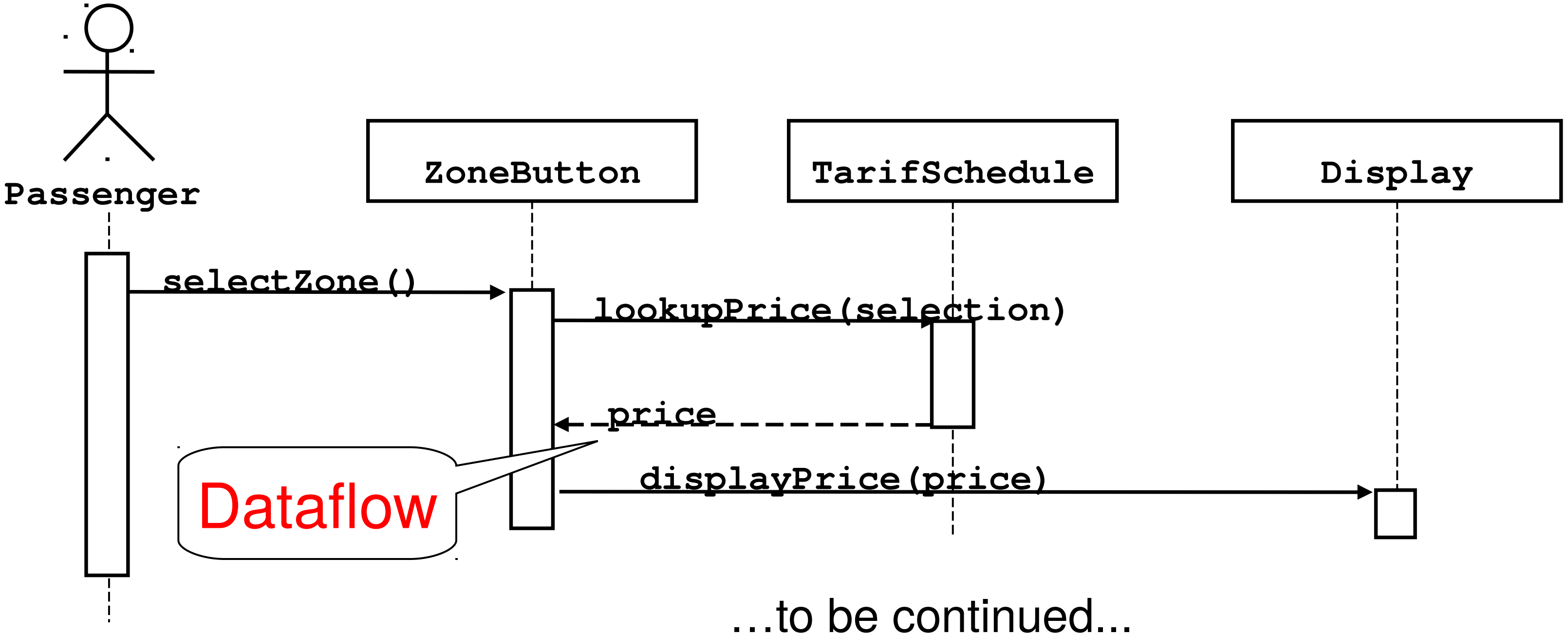
- A complex system can be decomposed into subsystems, where each subsystem is modeled as a package

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

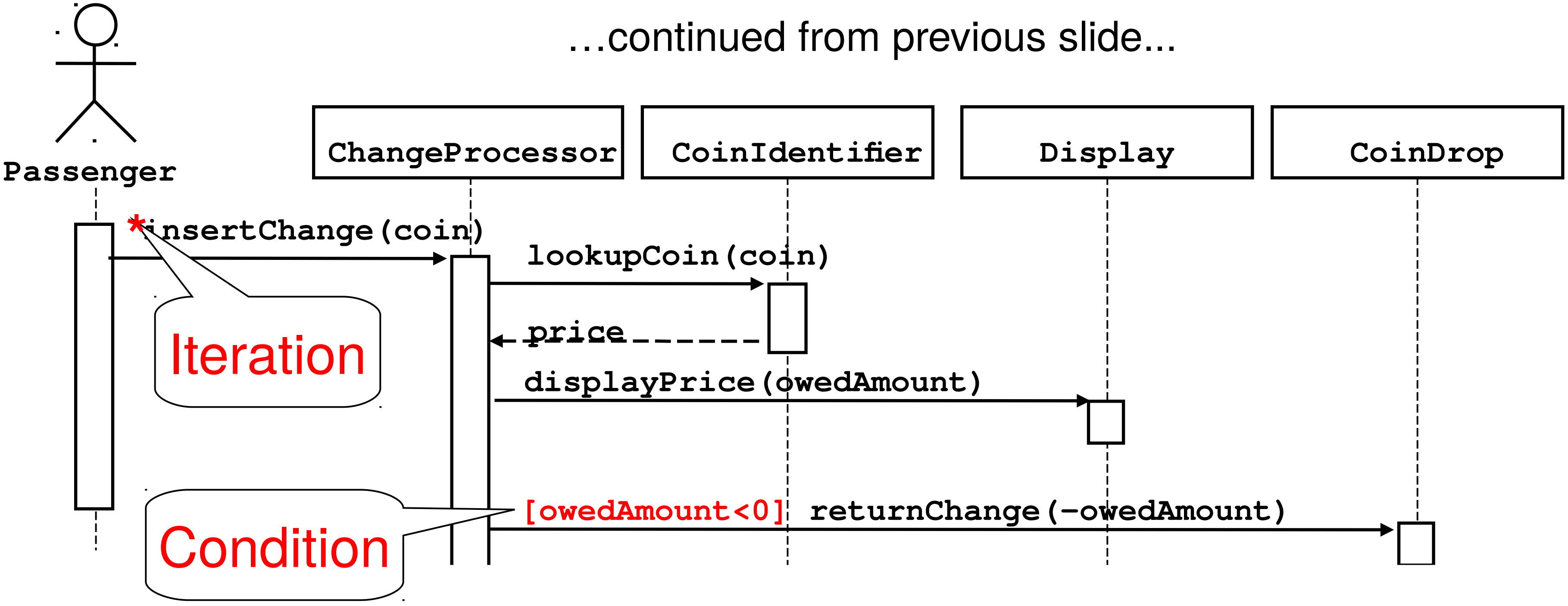


…to be continued...

- 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

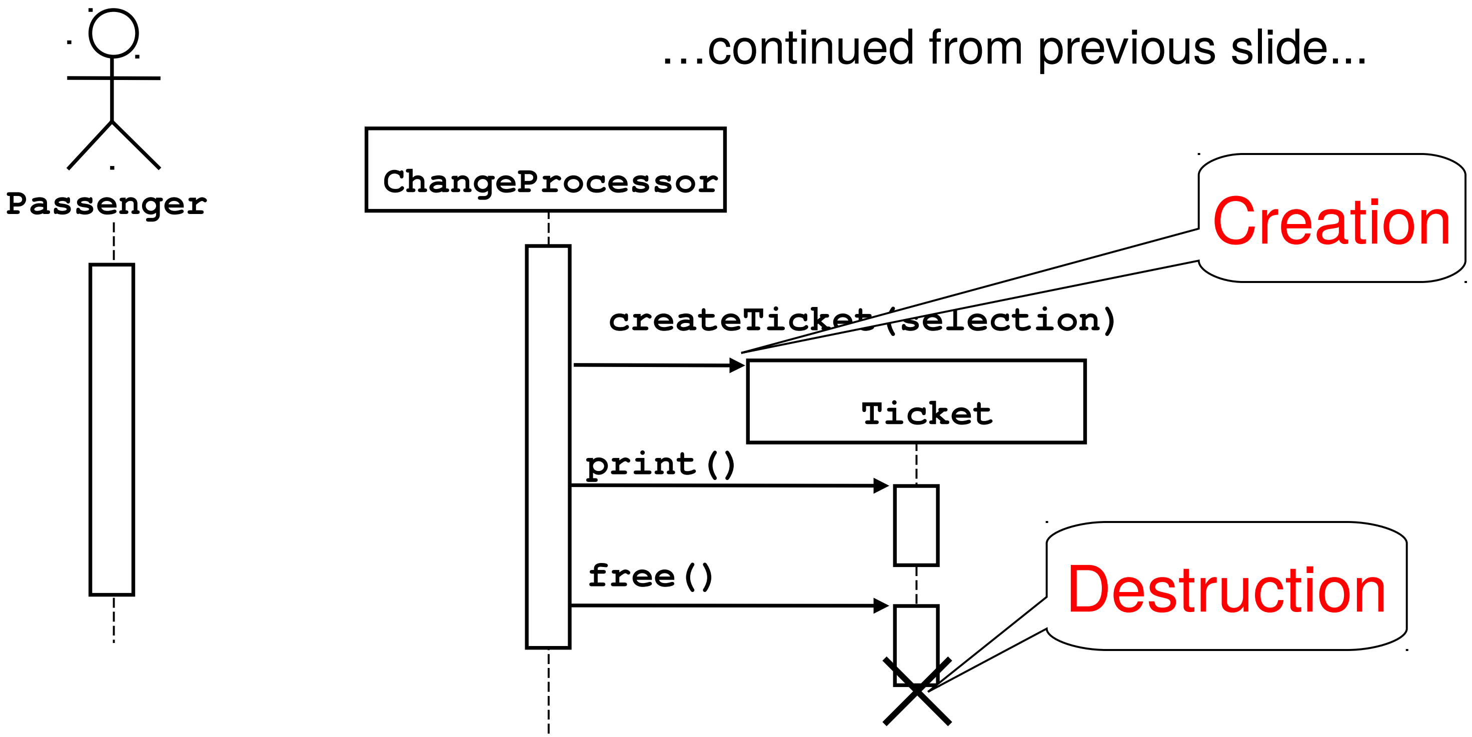
…continued from previous slide...



…to be continued...

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

Creation and destruction

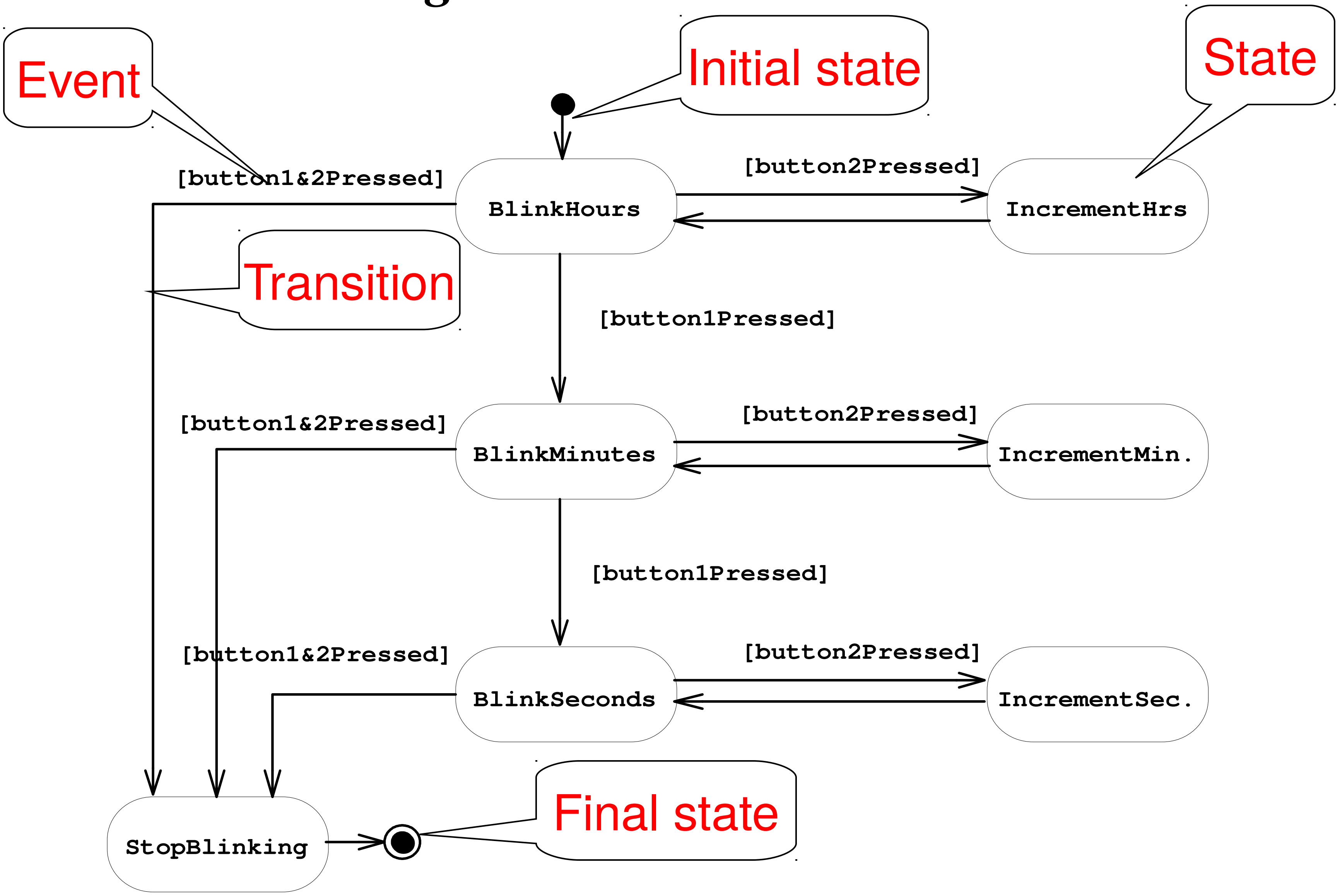


- Creation is denoted by a message arrow pointing to the object.
- Destruction is denoted by an X mark at the end of the destruction activation.
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

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).

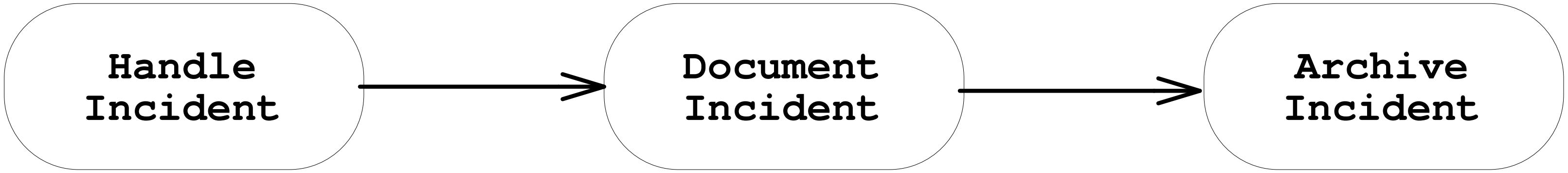
State Chart Diagrams



Represent behavior as states and transitions

Activity Diagrams

- An activity diagram shows flow control within a system

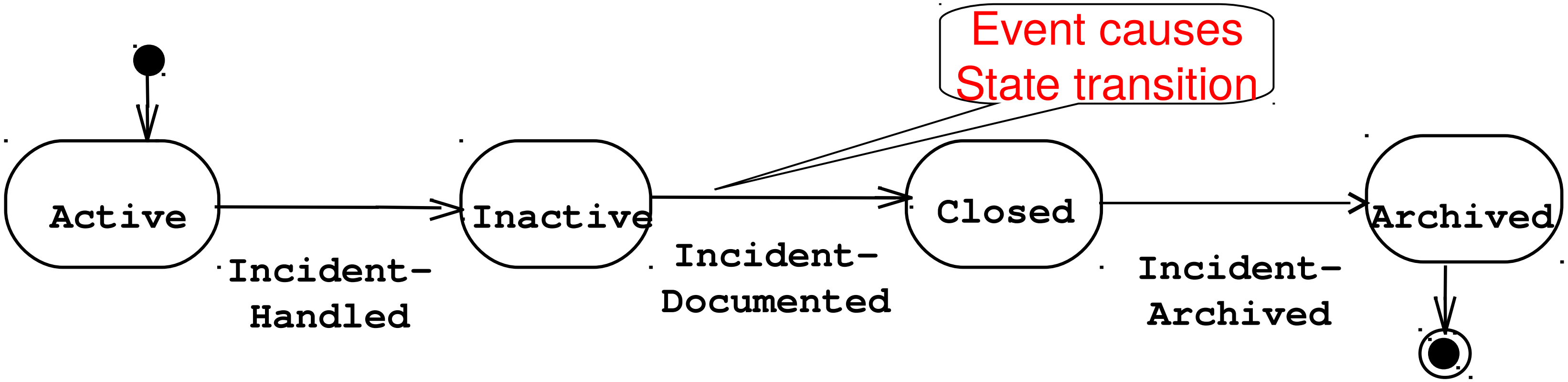


- An activity diagram is a special case of a state chart diagram in which states are activities (“functions”)
- Two types of states:
  - ***Action state:***
    - **Cannot be decomposed any further**
    - **Happens “instantaneously” with respect to the level of abstraction used in the model**
  - ***Activity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

Statechart Diagram vs. Activity Diagram

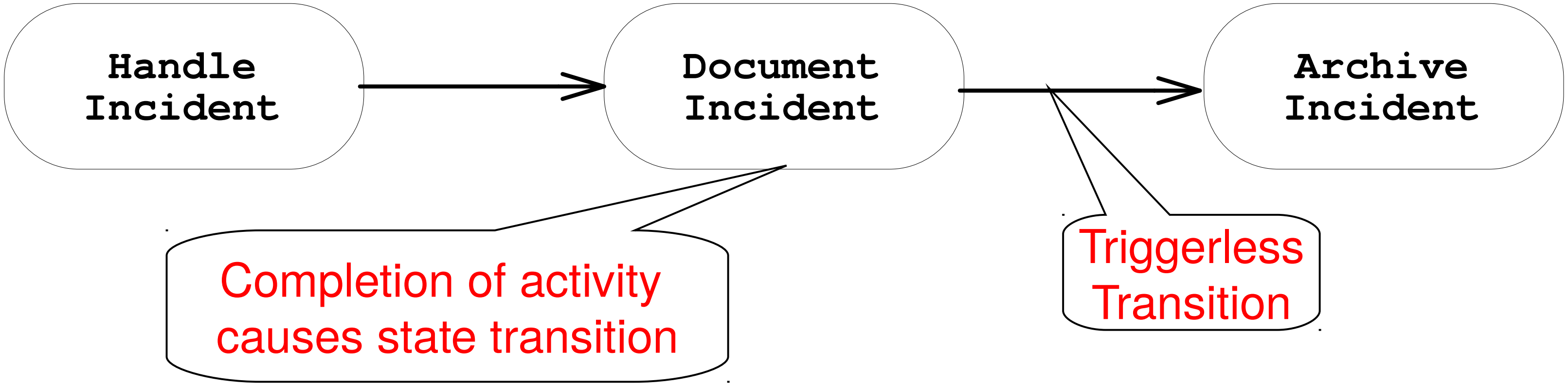
Statechart Diagram for Incident (similar to Mealy Automaton)

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

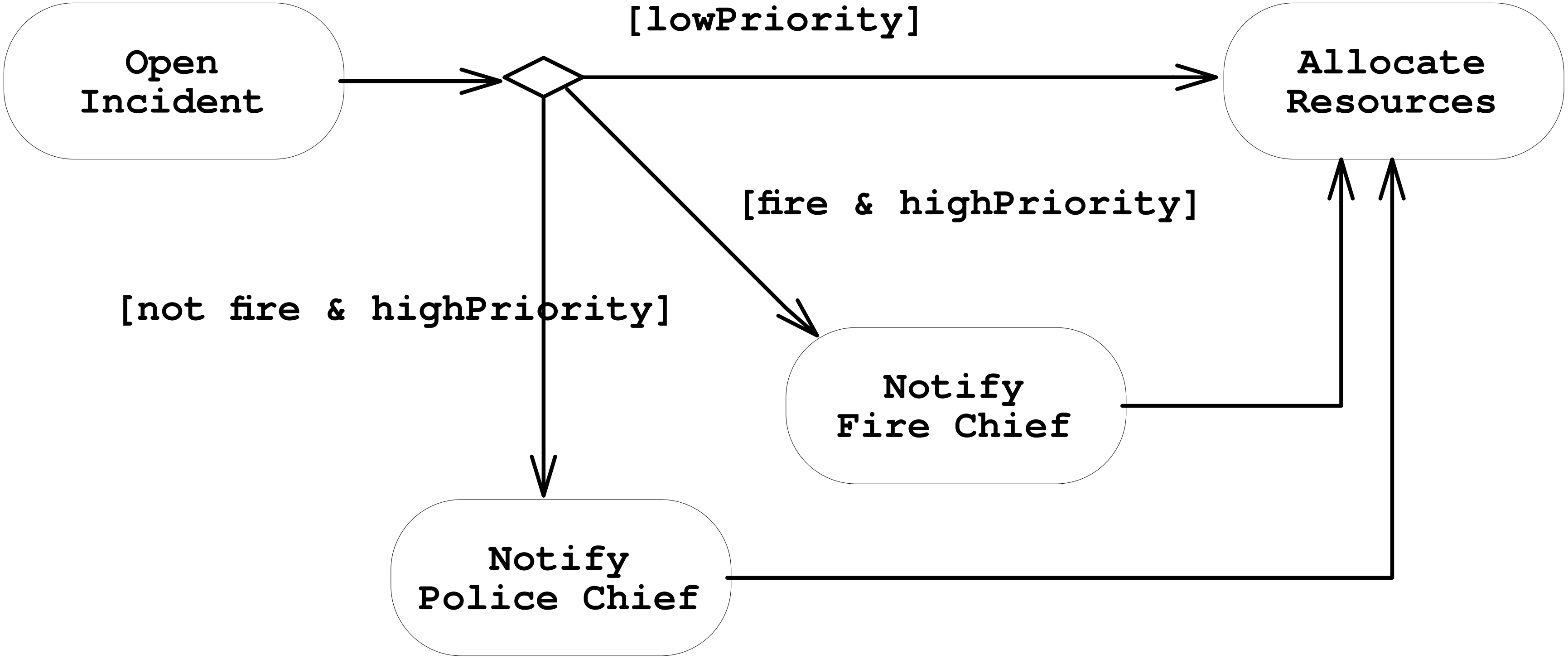


Activity Diagram for Incident (similar to Moore

(State: Operation or Collection of Operations)



Activity Diagram: Modeling Decisions



Open Incident

[lowPriority]

Allocate Resources

[fire & highPriority]

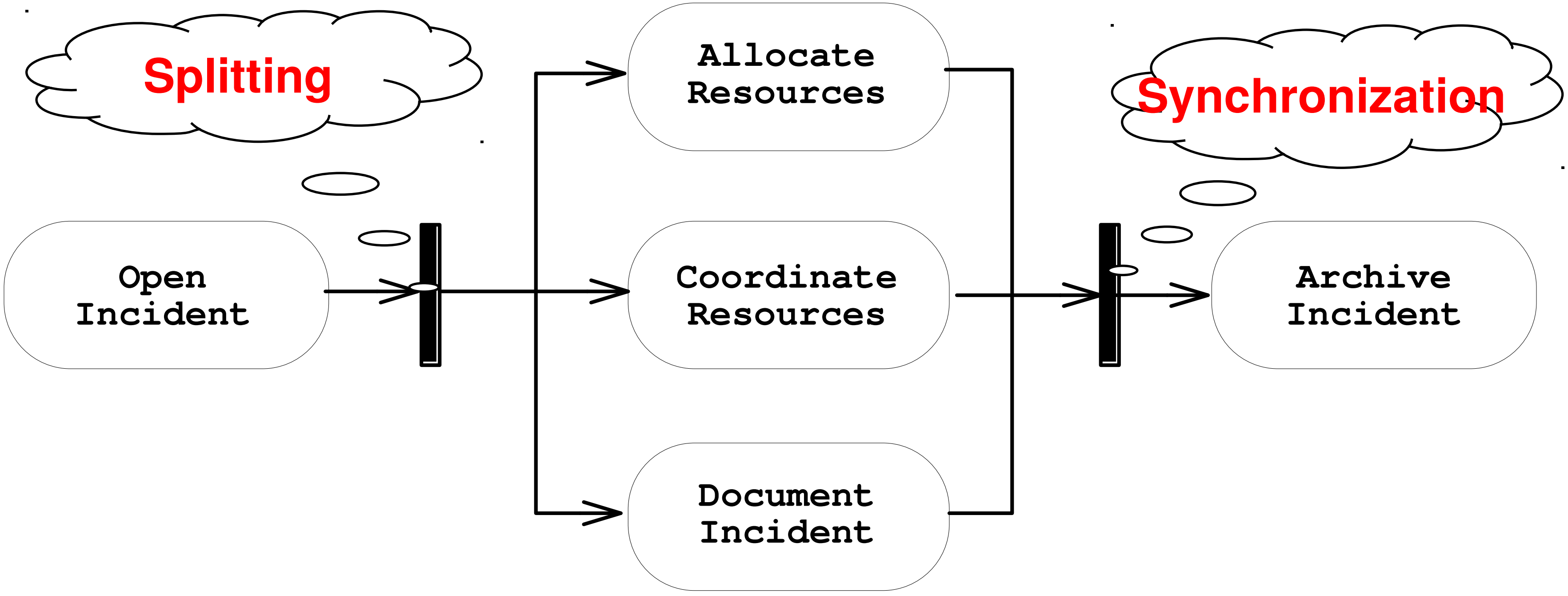
[not fire & highPriority]

Notify Fire Chief

Notify Police Chief

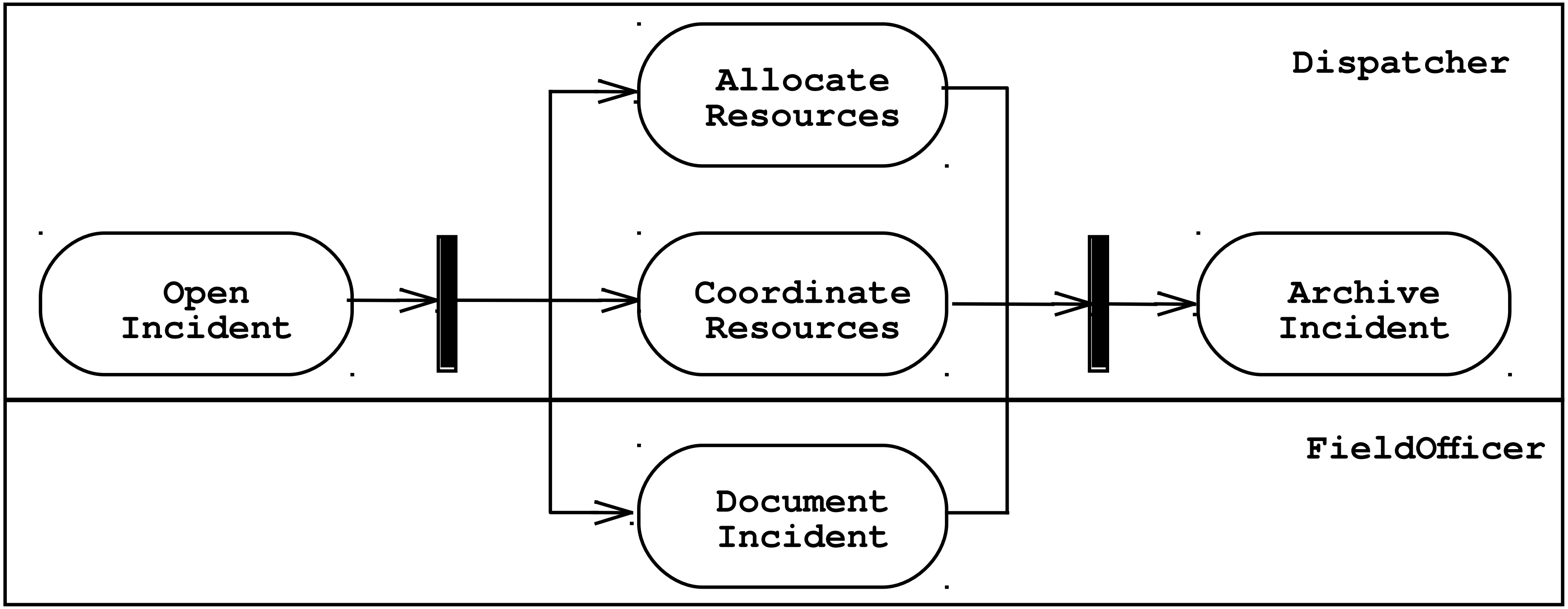
Activity Diagrams: Modeling Concurrency

- 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**

Additional Slides

Models for Plato’s and Aristotle’s Views of Reality

Plato

- Material reality is a second-class subordinate type of reality.
- The first-class type is a “form” Forms lie behind every thing or in the world. Forms can be abstract nouns like “beauty” or “mammal” or concrete nouns like “tree” or “horse”.
- There is an important difference between the world of forms and particulars. Forms are nonmaterial, particulars are material. Forms are permanent and changeless. Particulars are changing.
- Forms can be acquired intellectually through a “dialectic” process that moves toward the highest understanding of reality through the interaction of questions and answers.

Aristotle

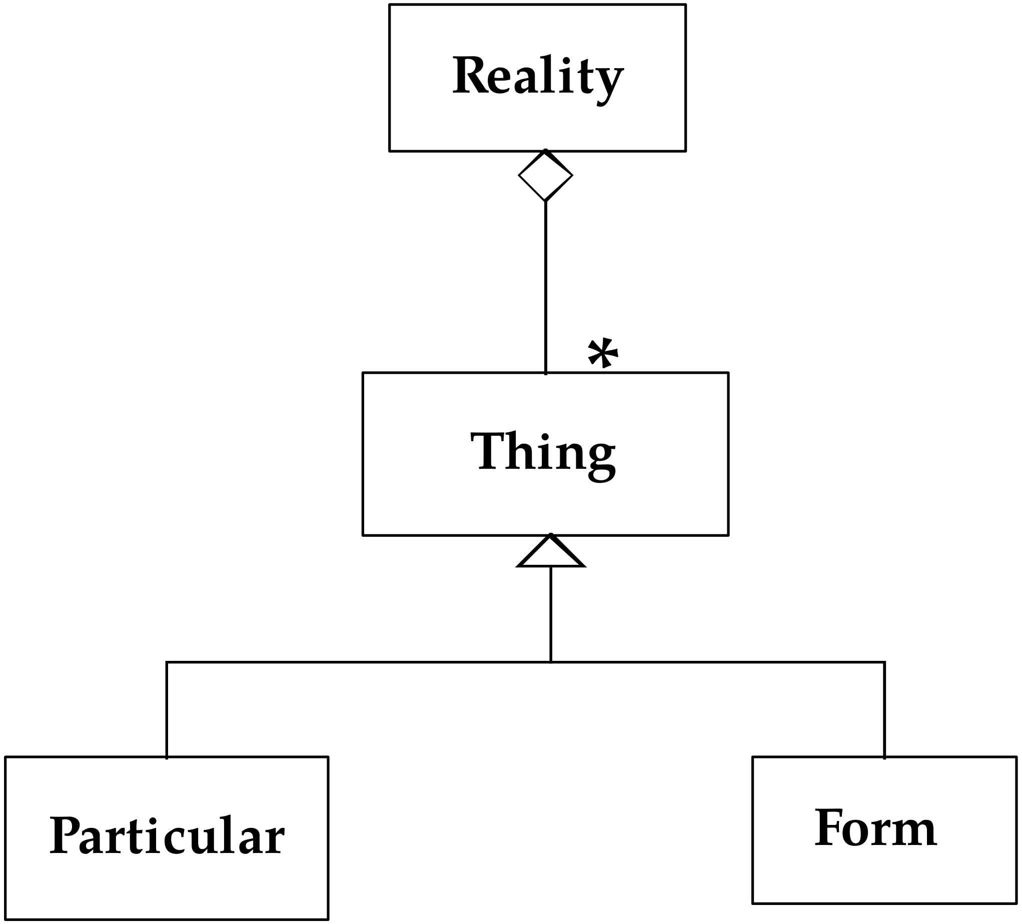
- Aristotle accepted the reality of Forms as nonmaterial entities.
- However, he could not accept Plato’s idea, that these Forms were not real.
- Instead of two separate worlds, one for Forms and one for Particulars, Aristotle had only one world, a world of particular things.
- Particular things according to Aristotle have a certain permance about them, even while they are subject to change: A tree changes colors without ceasing to be a tree. A horse grows in size without ceasing to be a horse.
- What is the root of this permancence? It is the thing’s internal form, which minds detect, when they penetrate beyond the thing’s changing attributes. So for Aristotle, reality is thus made up of particular things that are each composed of form antdn matter..

Using UML, we can illustrate Platon’s and Aristotle’s viewpoints very easily and see their differences as well

Model for Plato's View of Reality

Plato

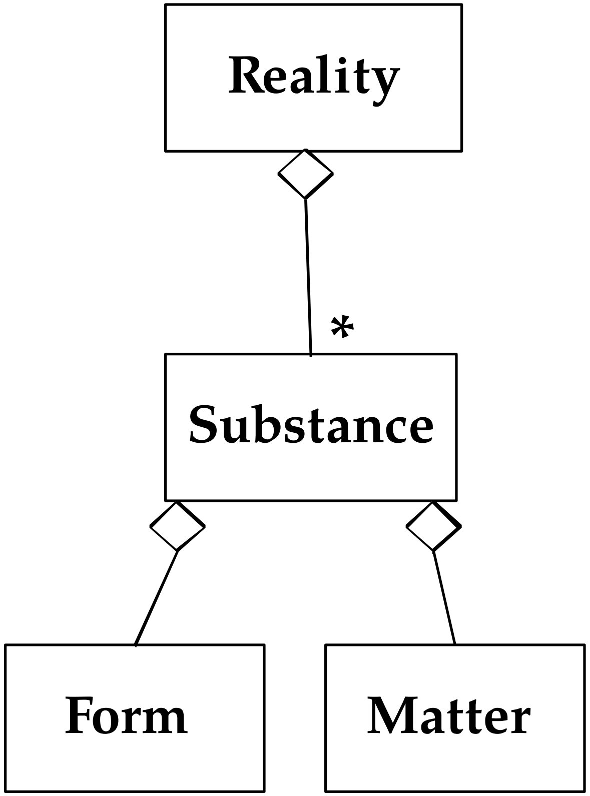
- Material reality is a second-class subordinate type of reality.
- The first-class type is a "form" Forms lie behind every thing or in the world. Forms can be abstract nouns like "beauty" or "mammal" or concrete nouns like "tree" or "horse".
- There is an important difference between the world of forms and particulars. Forms are nonmaterial, particulars are material. Forms are permanent and changeless. Particulars are changing.
- Forms can be acquired intellectually through a "dialectic" process that moves toward the highest understanding of reality through the interaction of questions and answers.



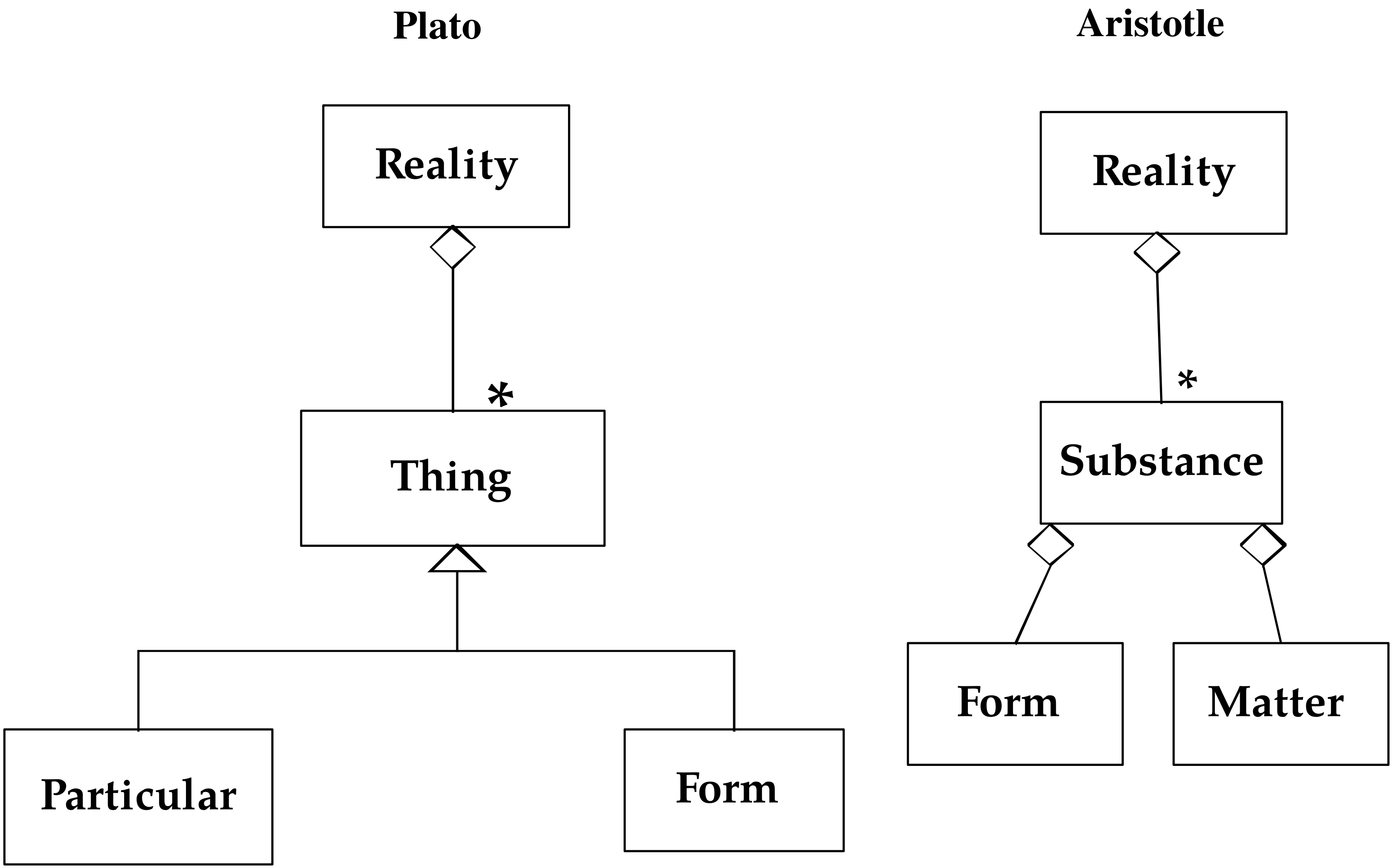
Model Aristotle's Views of Reality

Aristotle

- Aristotle accepted the reality of Forms as nonmaterial entities.
- However, he could not accept Plato's idea, that these Forms were not real.
- Instead of two separate worlds, one for Forms and one for Particulars, Aristotle had only one world, a world of particular things.
- Particular things according to Aristotle have a certain permance about them, even while they are subject to change: A tree changes colors without ceasing to be a tree. A horse grows in size without ceasing to be a horse.
- What is the root of this permancence? It is the thing's internal form, which minds detect, when they penetrate beyond the thing's changing attributes. So for Aristotle, reality is thus made up of particular things that are each composed of form antdn matter..



Comparison of Plato's and Aristotle's Views



Plato

Reality

*

Thing

Particular

Form

Aristotle

Reality

*

Substance

Form

Matter