TUM

Modeling with UML

Bernd Brügge

Technische Universität München

Lehrstuhl für Angewandte Softwaretechnik

http://wwwbruegge.in.tum.de

26 October 2001

Overview: modeling with UML

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

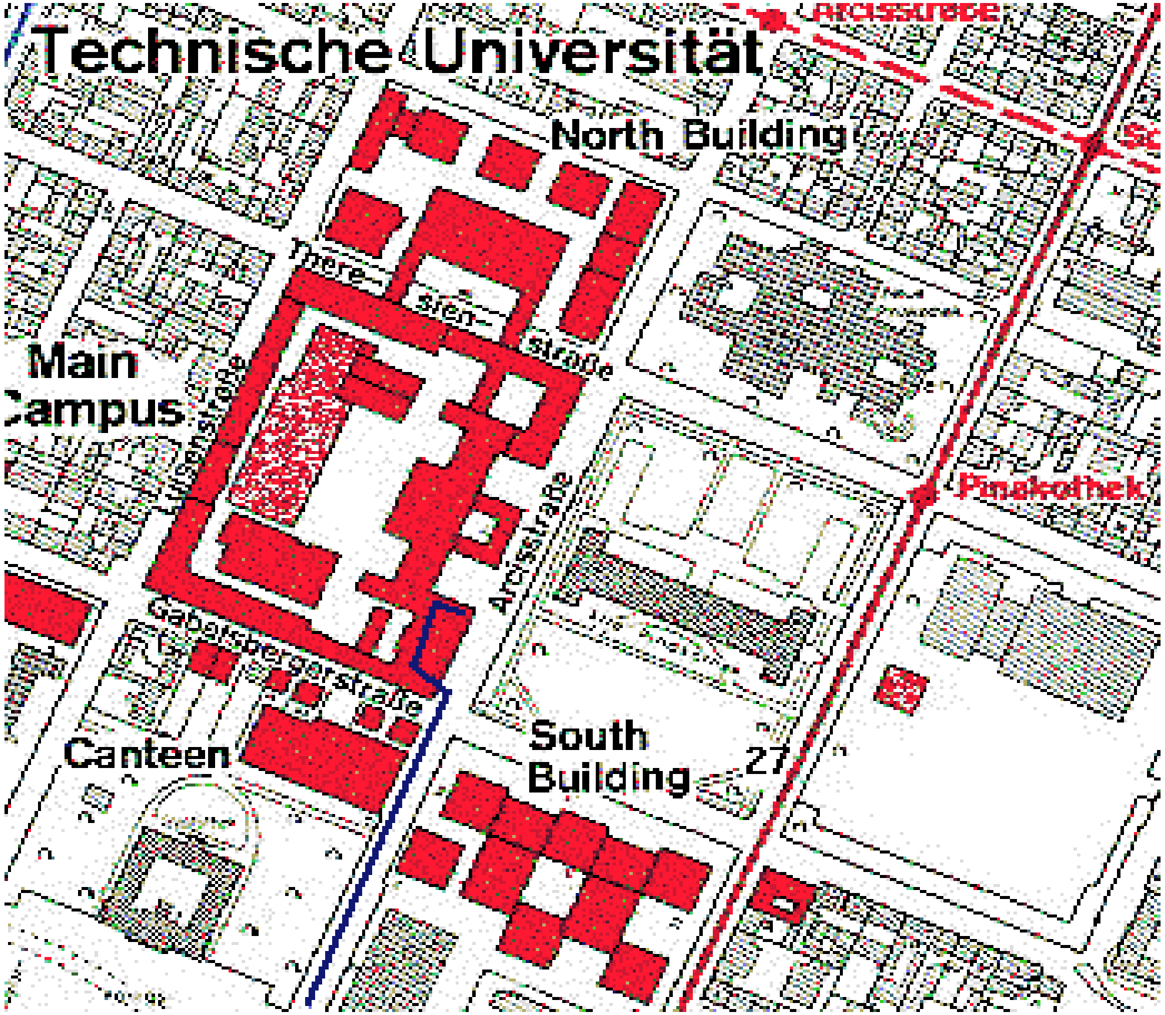
Next time (November 2, 2001):

- Sequence diagrams
- Activity diagrams
- Questions?

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



Why model software?

Why model software?

- Software is getting increasingly more complex
  - **Windows 2000 ~ 40 mio 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 mean for dealing with complexity**

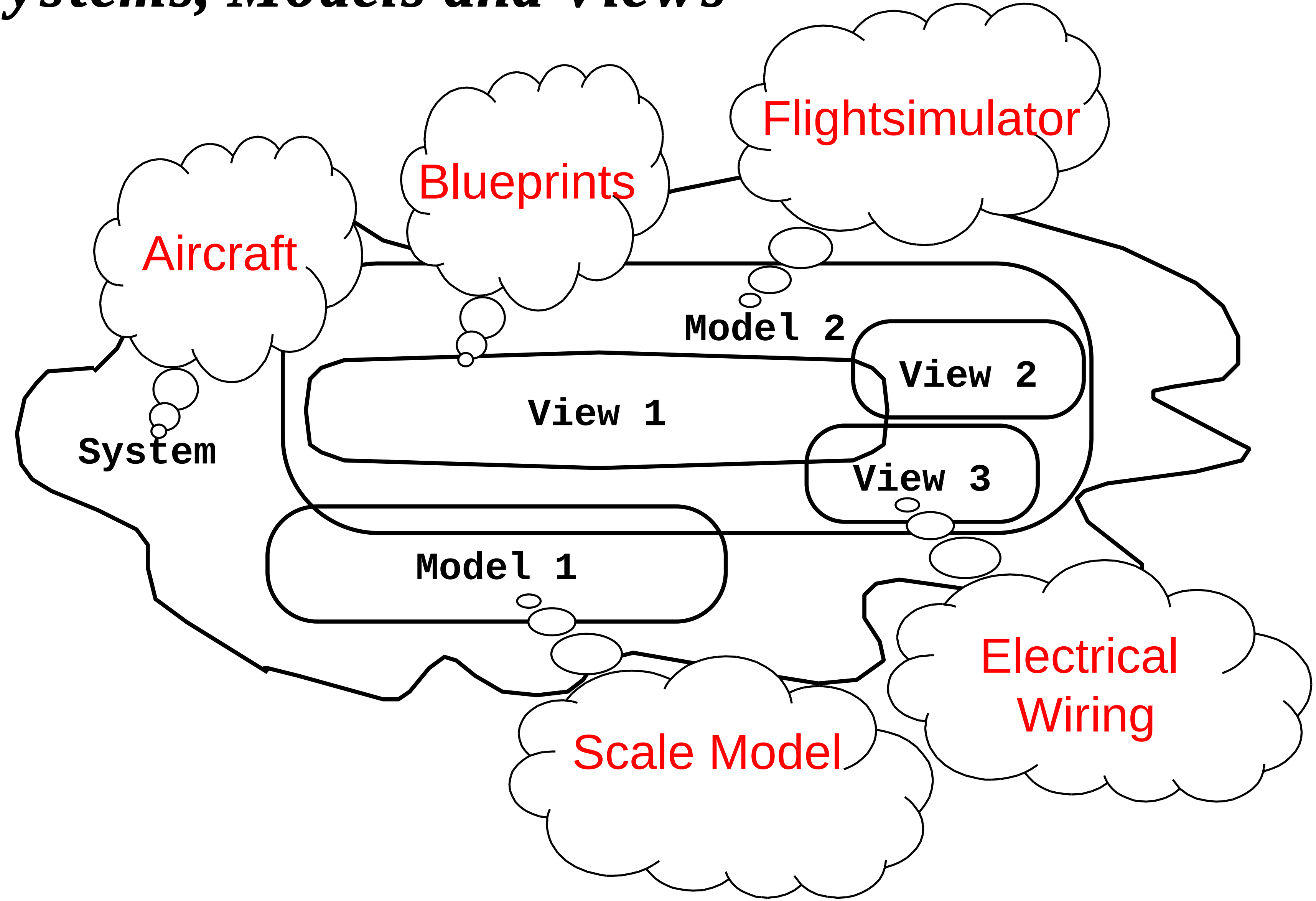
Systems, Models and Views

- 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

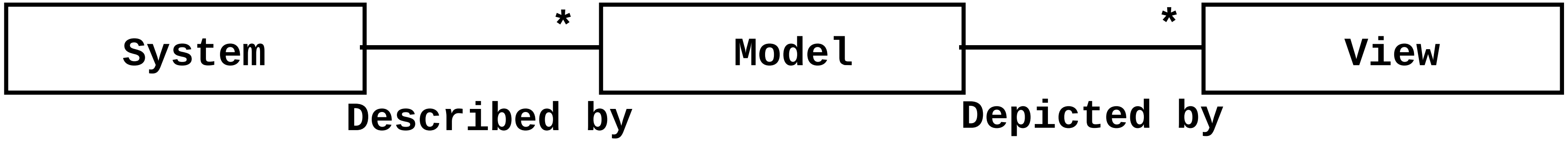
Examples:

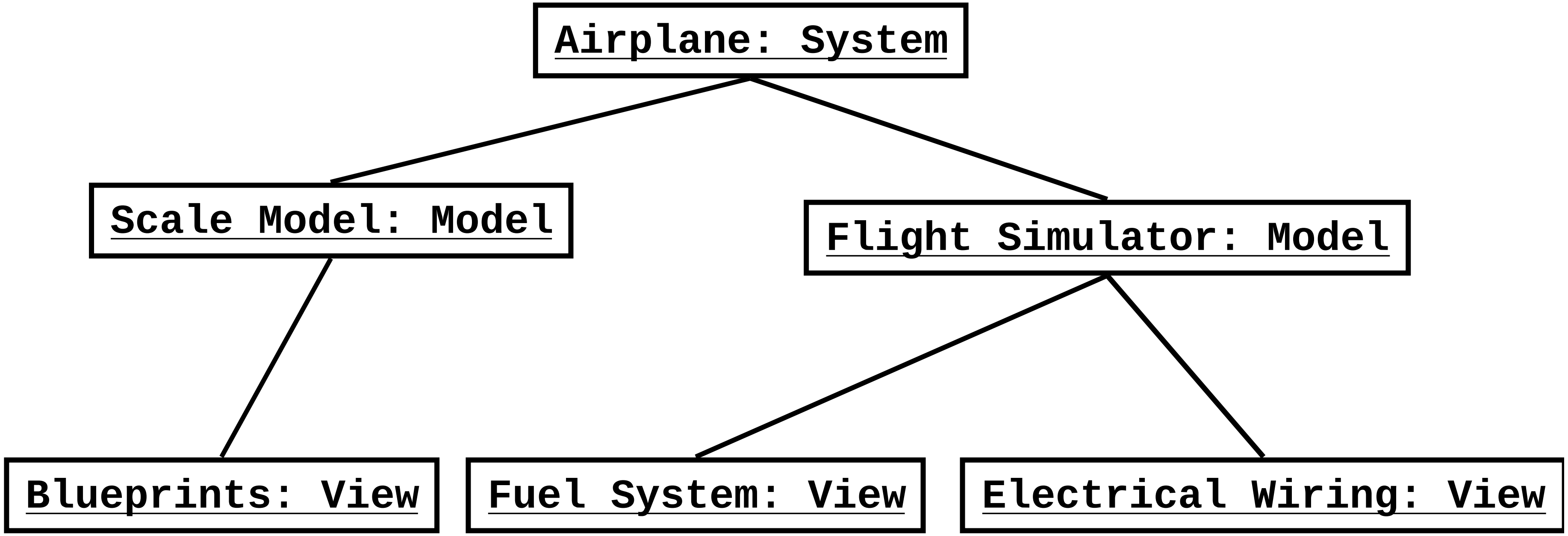
- System: Aircraft
- Models: Flight simulator, scale model
- Views: All blueprints, electrical wiring, fuel system

Systems, Models and Views



Models, Views and Systems (UML)





Concepts and Phenomena

Phenomenon

- **An object in the world of a domain as you perceive it**
- ***Example*: The lecture you are attending**
- ***Example*: My black watch**

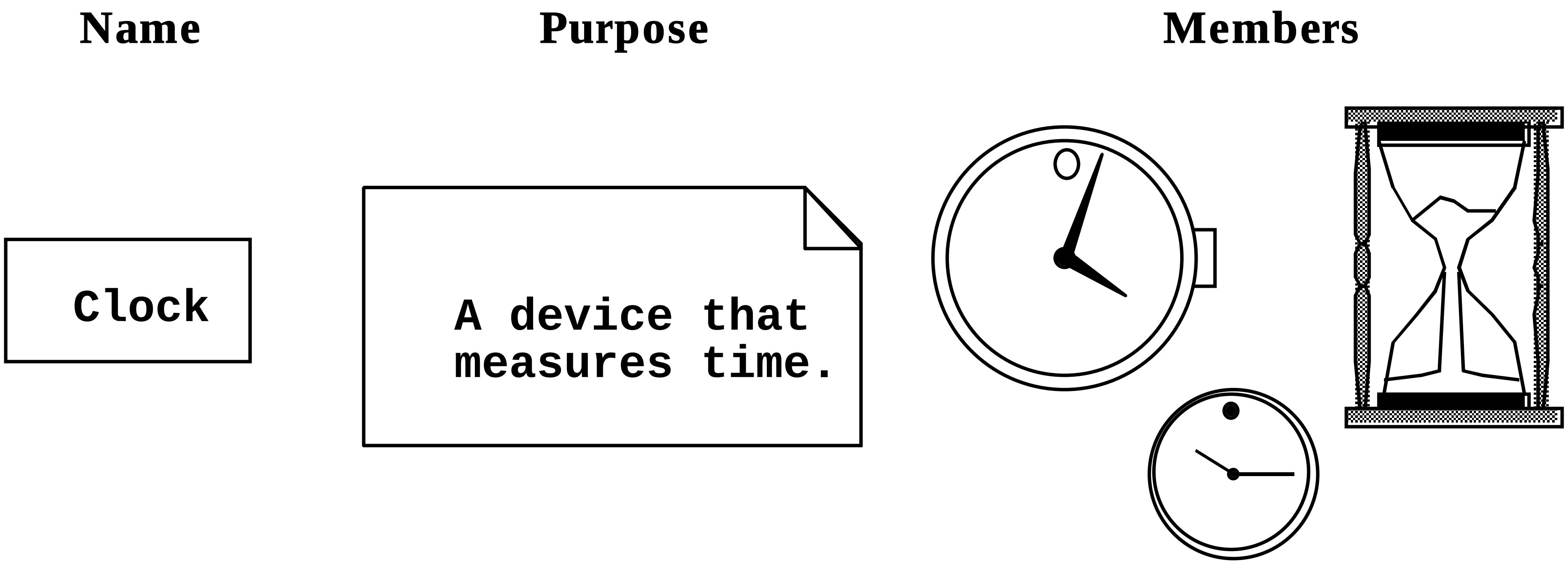
Concept

- **Describes the properties of phenomena that are common.**
- ***Example*: Lectures on software engineering**
- ***Example*: Black watches**

Concept is a 3-tuple:

- **Name (To distinguish it from other concepts)**
- **Purpose (Properties that determine if a phenomenon is a member of a concept)**
- **Members (The set of phenomena which are part of the concept)**

Concepts and phenomena



Name

Purpose

Members

Clock

A device that measures time.

- Abstraction
  - **Classification of phenomena into concepts**
- Modeling
  - **Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.**

Concepts in software: Type and Instance

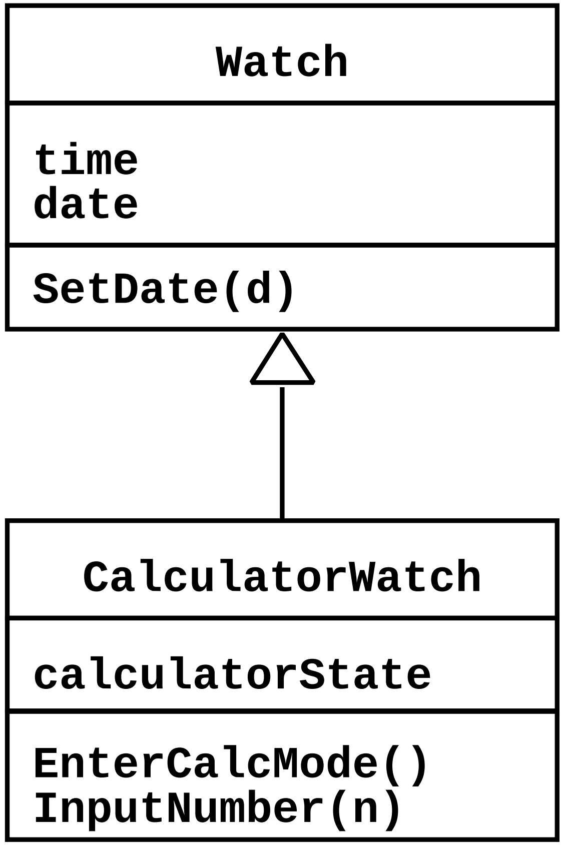
- Type:
  - **An abstraction in the context of programming languages**
  - **Name: int, Purpose: integral number, Members: `0, -1, 1, 2, -2, . . .`**
- Instance:
  - **Member of a specific type**
- The type of a variable represents all possible instances the variable can take

The following relationships are similar:

- **"type" <–> "instance"**
- **"concept" <–> "phenomenon"**

Abstract Data Types & Classes

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

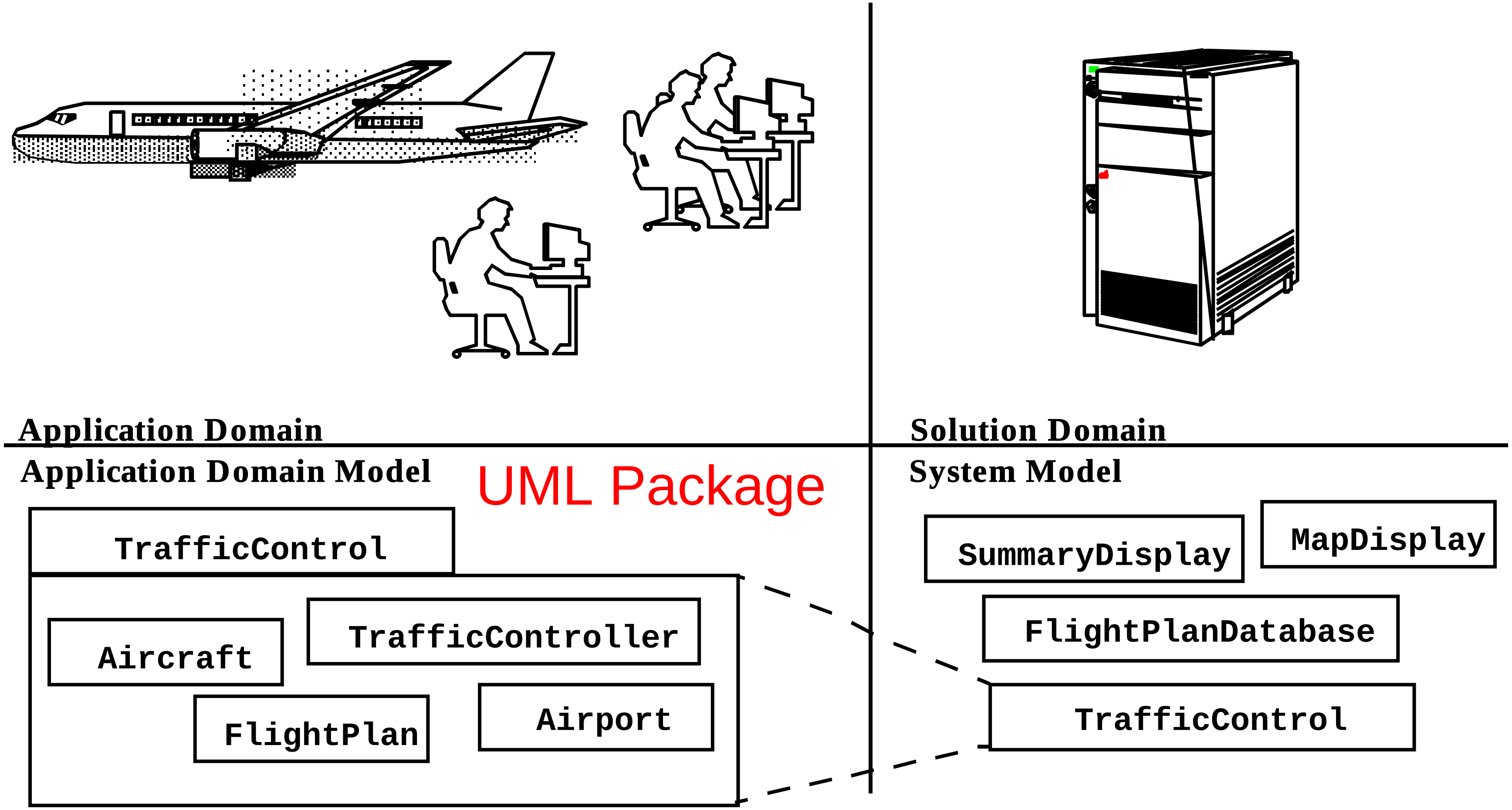


Application and Solution Domain

- Application Domain (Requirements Analysis):
  - **The environment in which the system is operating**

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

Object-oriented modeling



Application Domain

Solution Domain

Application Domain Model

UML Package

System Model

TrafficControl

Aircraft

TrafficController

FlightPlan

Airport

SummaryDisplay

MapDisplay

FlightPlanDatabase

TrafficControl

What is UML?

- UML (Unified Modeling Language)
  - **An emerging standard for modeling object-oriented software.**
  - **Resulted from the convergence of notations from three leading object-oriented methods:**
    - **OMT (James Rumbaugh)**
    - **OOSE (Ivar Jacobson)**
    - **Booch (Grady Booch)**
- Reference: “The Unified Modeling Language User Guide”, Addison Wesley, 1999.
- Supported by several CASE tools
  - **Rational ROSE**
  - **TogetherJ (Lecture on November 16)**

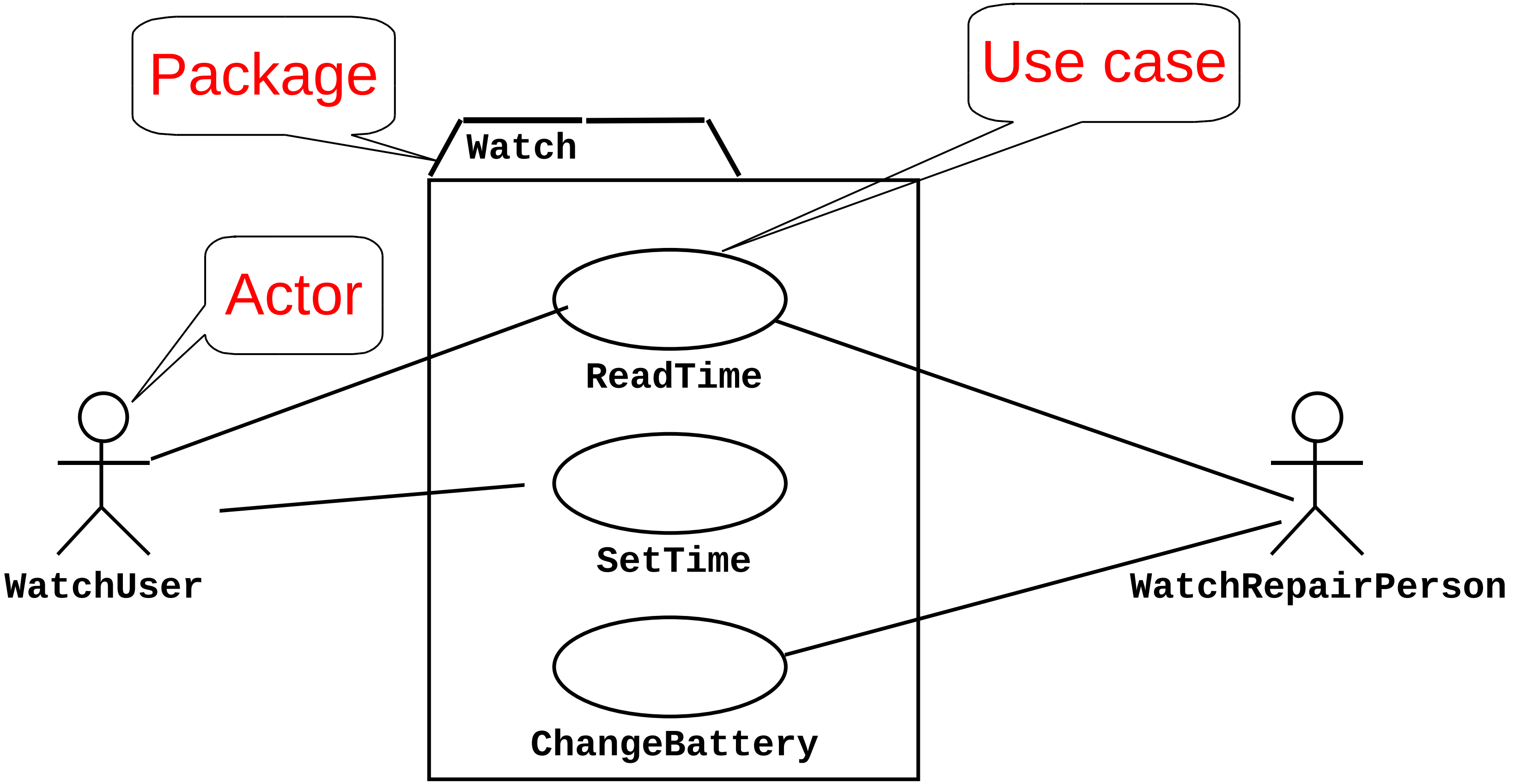
UML: First Pass

- You can model 80% of most problems by using about 20 % UML
- We teach you those 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 workflow (essentially a flowchart)**

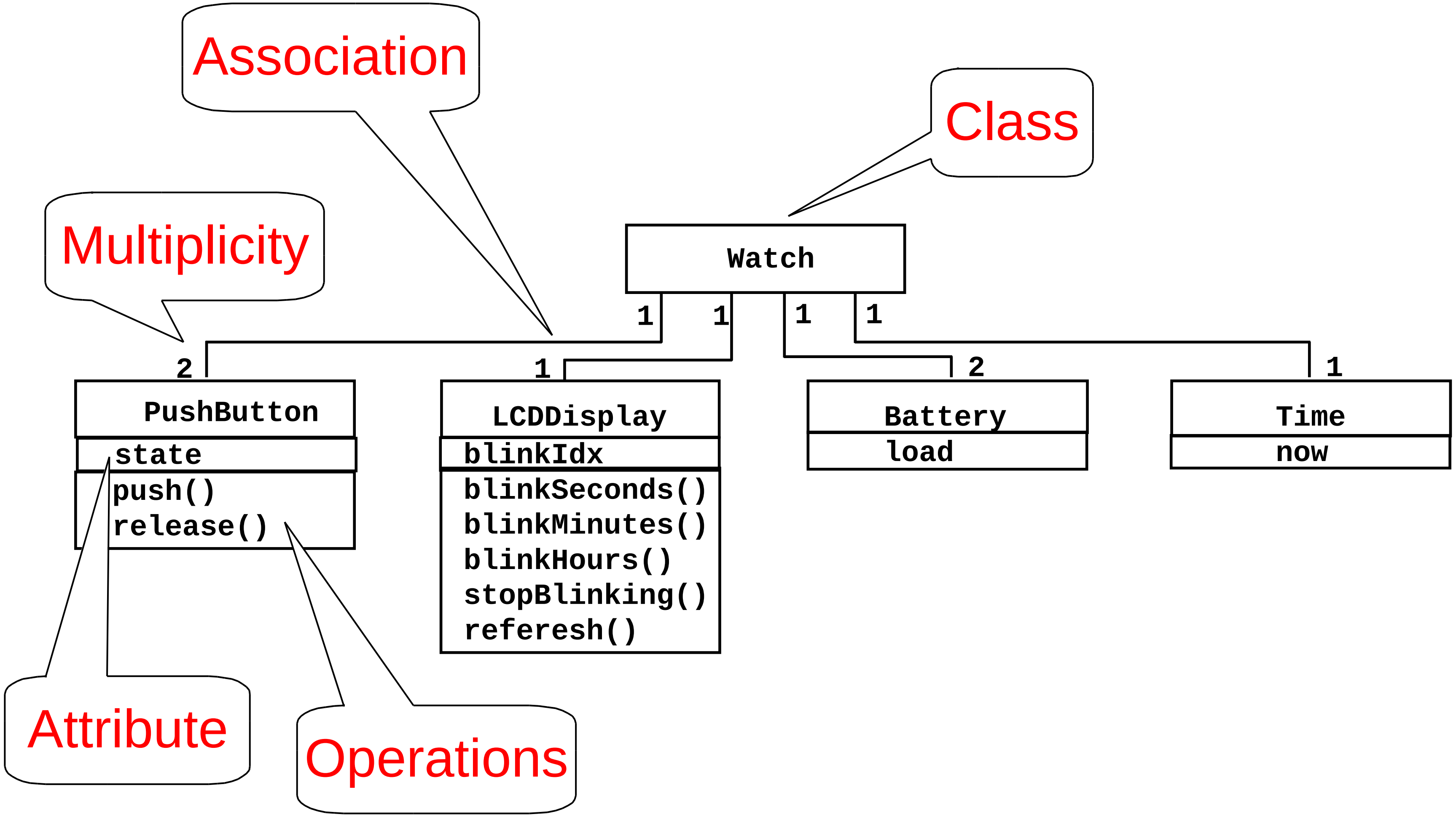
UML first pass: Use case diagrams



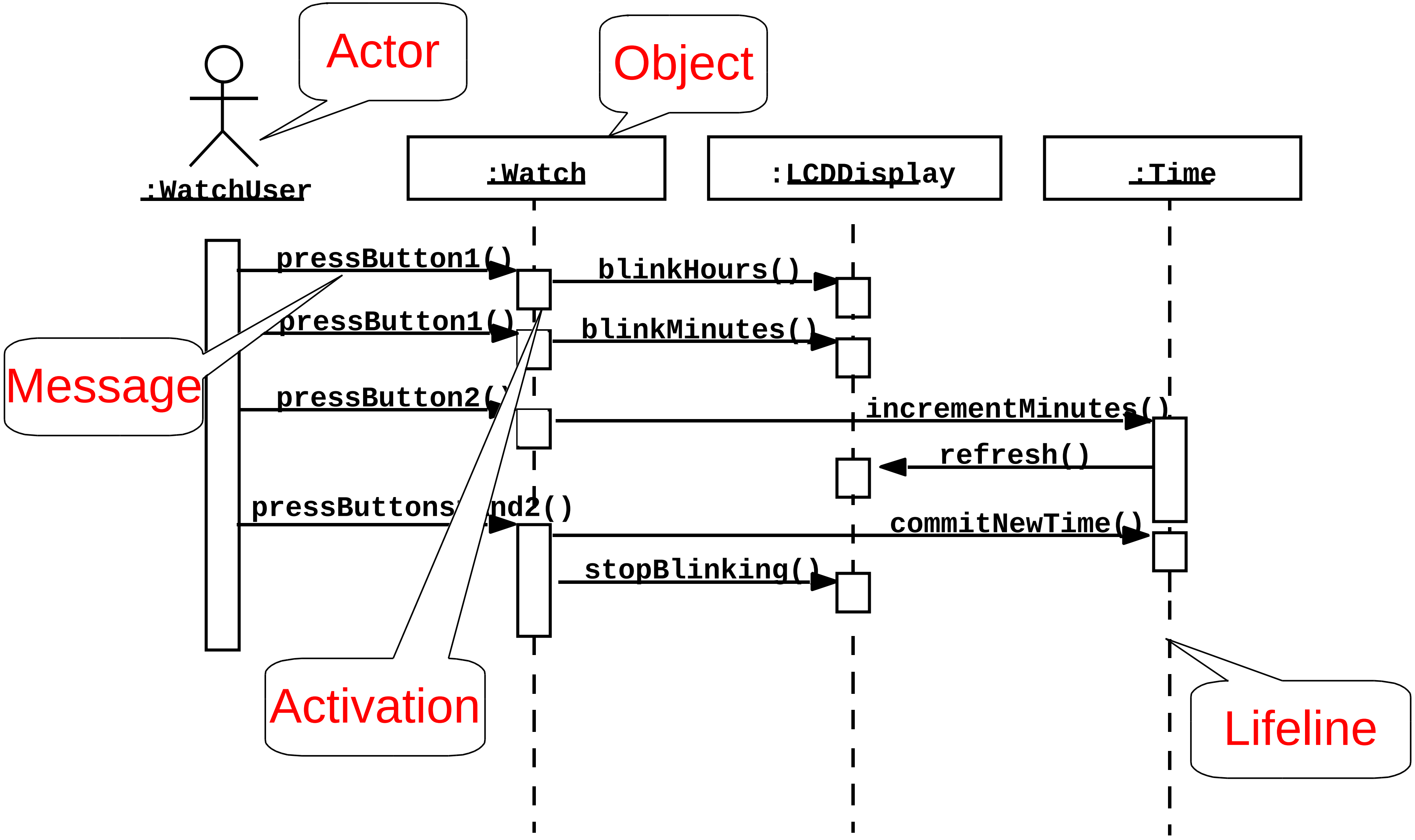
Use case diagrams represent the functionality of the system from user’s point of view

UML first pass: Class diagrams

Class diagrams represent the structure of the system

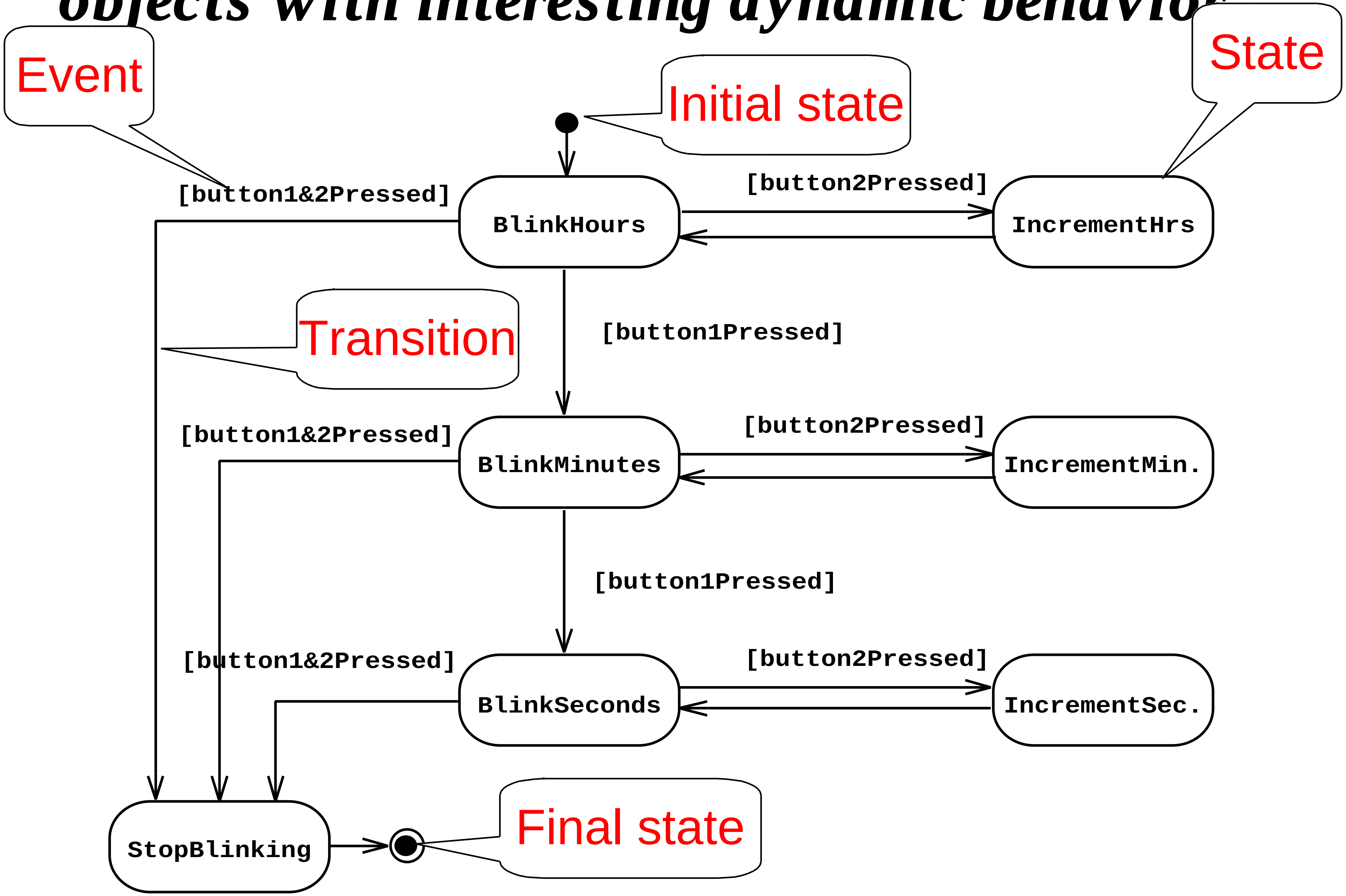


UML first pass: Sequence diagram



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

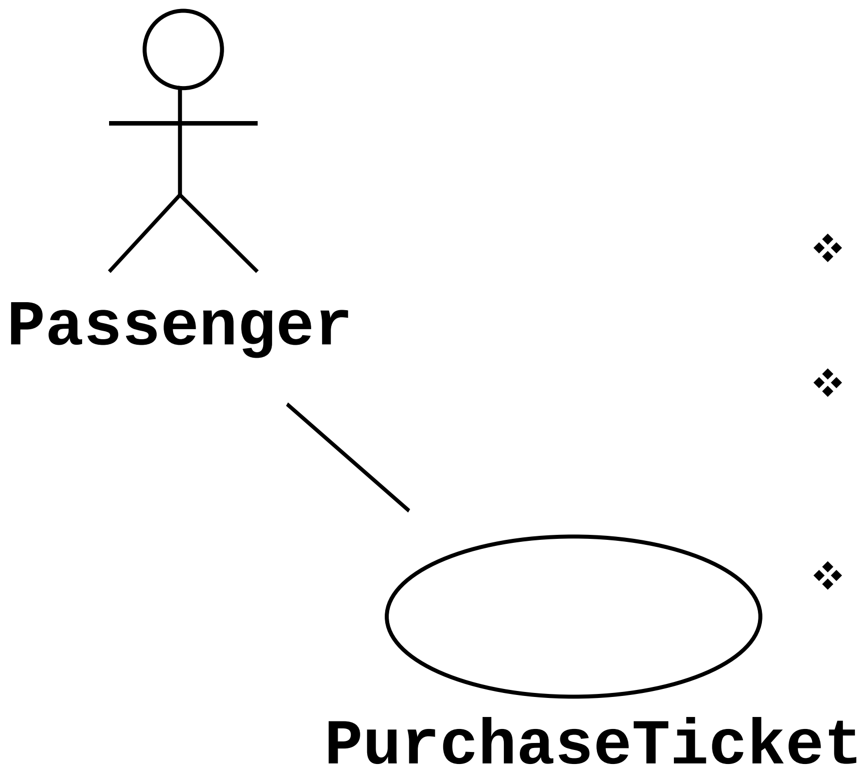
UML provide other notations that we will be introduced in subsequent lectures, as needed.

- Implementation diagrams
  - **Component diagrams**
  - **Deployment diagrams**
  - **Introduced in lecture on System Design (November 22)**
- Object constraint language
  - **Introduced in lecture on Object Design (December 21)**

UML Core Conventions

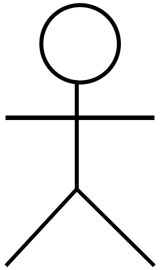
- Rectangles are classes or instances
- Ovals are functions or use cases
- Instances are denoted with an underlined names
  - <u>`myWatch:SimpleWatch`</u>
  - <u>`Joe:Firefighter`</u>
- Types are denoted with non underlined names
  - **`SimpleWatch`**
  - **`Firefighter`**
- Diagrams are graphs
  - **Nodes are entitites**
  - **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

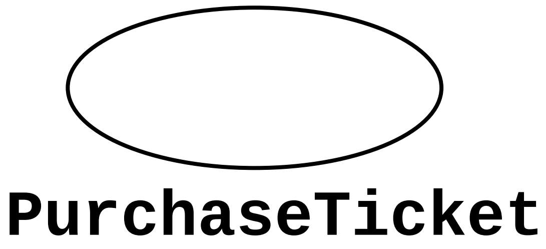


Passenger

- 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

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.

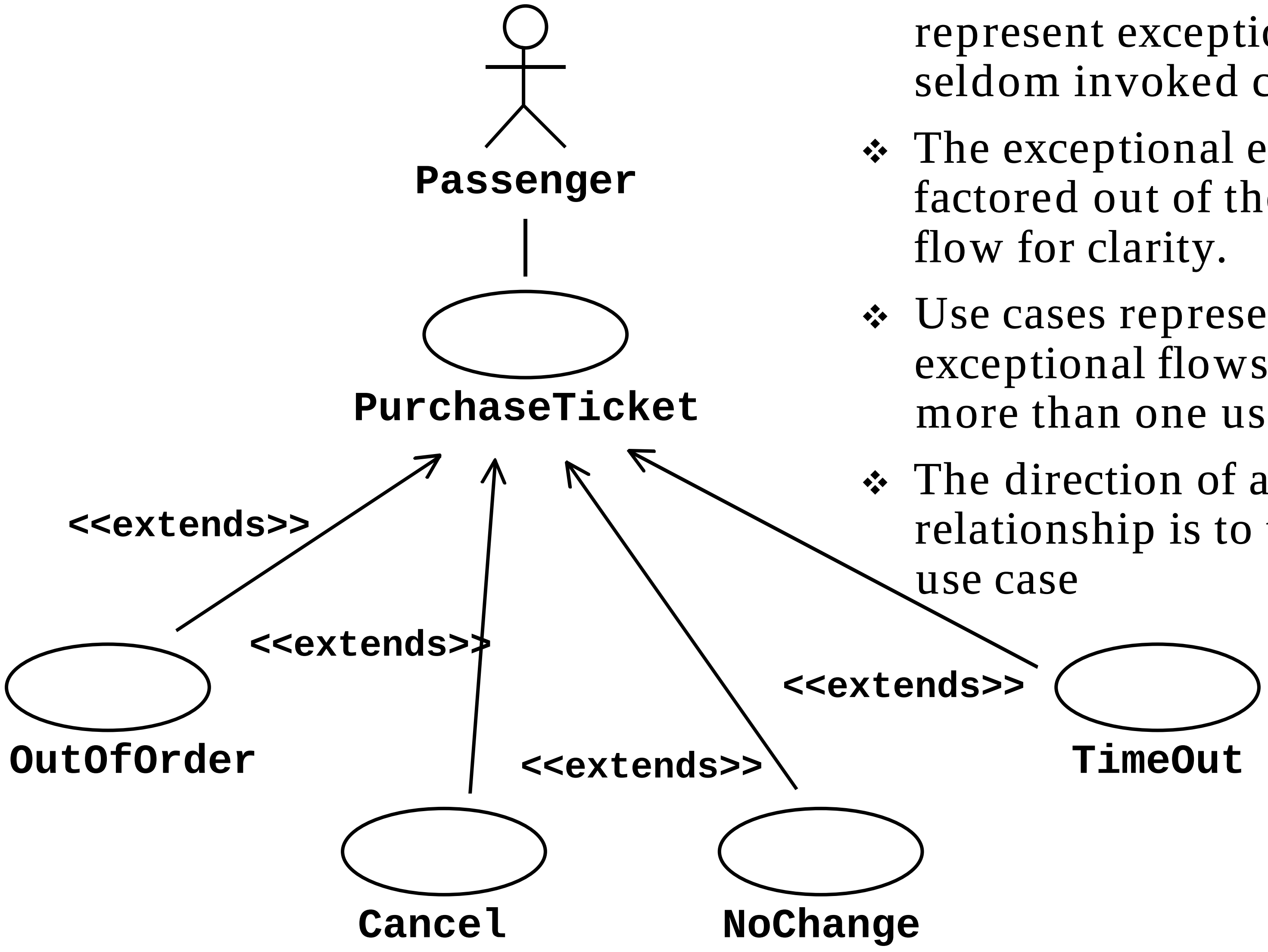
Event flow:

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?

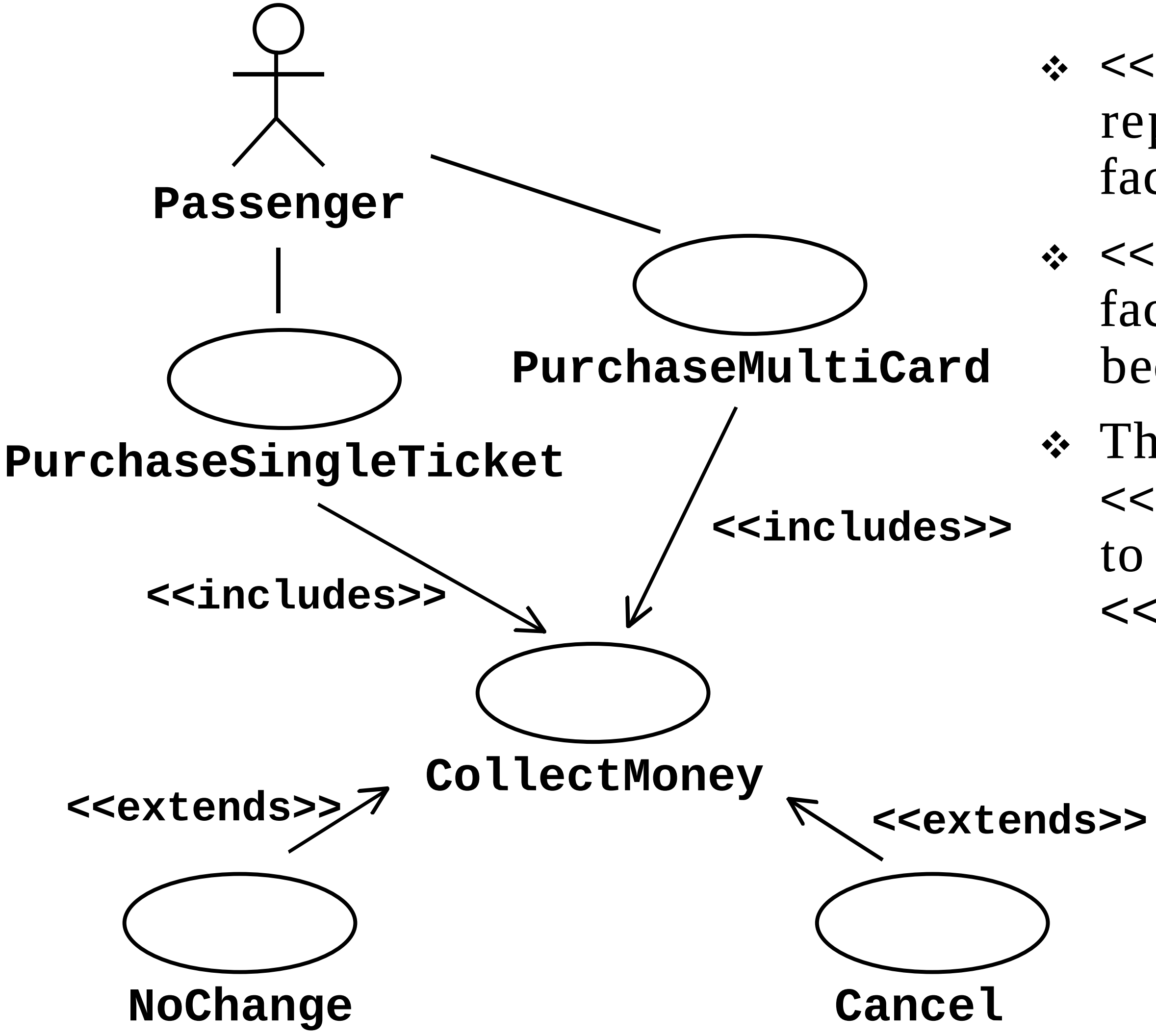
Exceptional cases!

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*

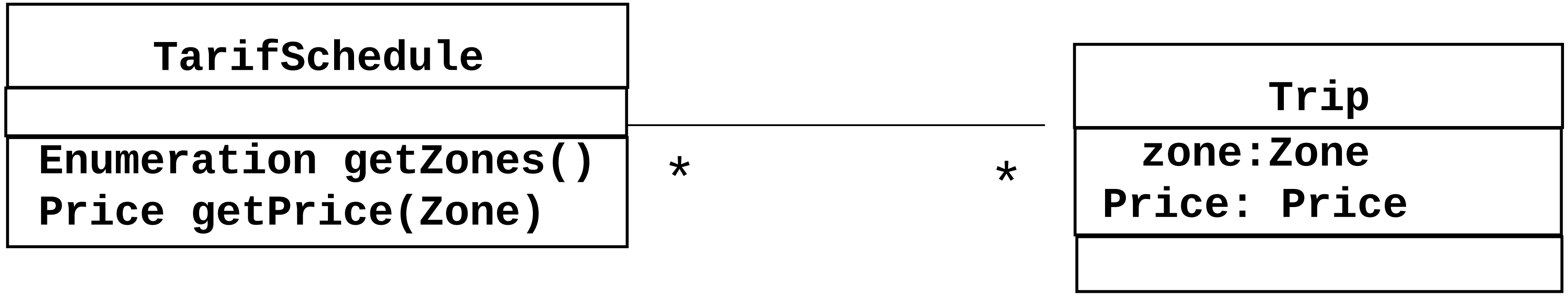


- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

Use Case Diagrams: Summary

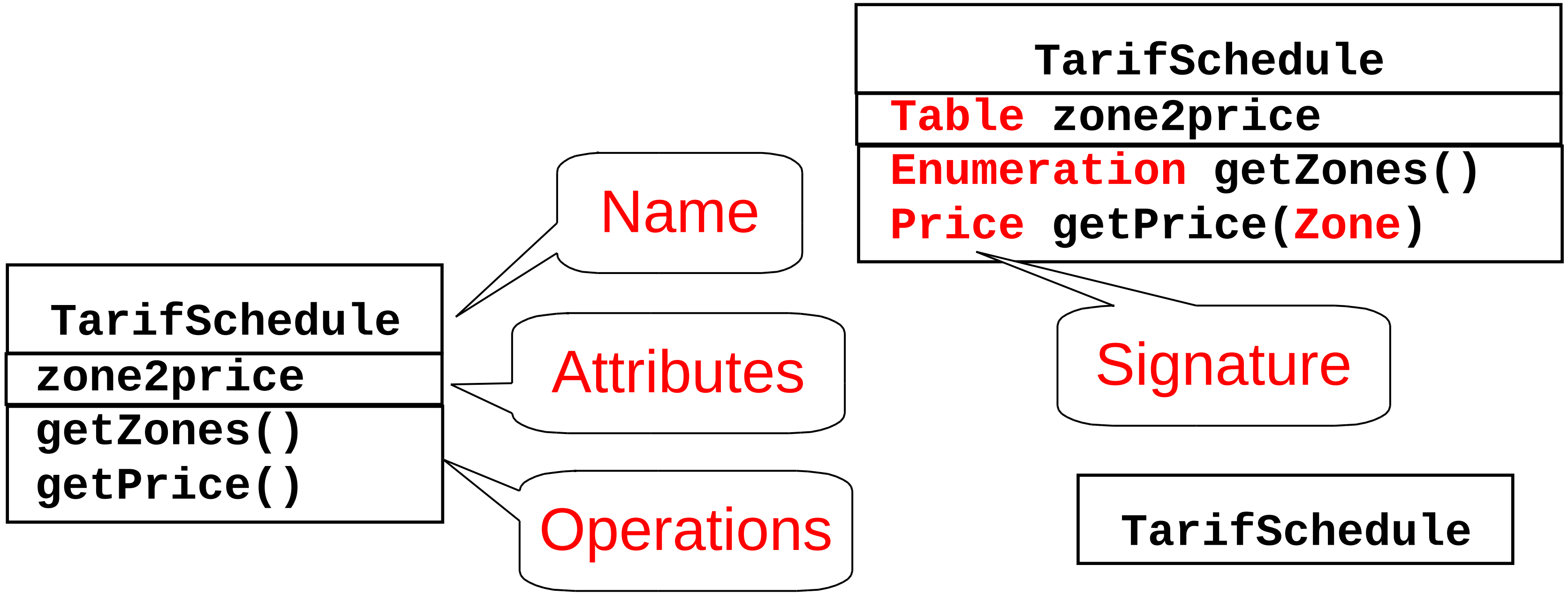
- Use case diagrams represent external behavior
- Use case diagrams are useful as an index into the use cases
- Use case descriptions provide meat of model, not the use case diagrams.
- All use cases need to be described for the model to be useful.

Class Diagrams



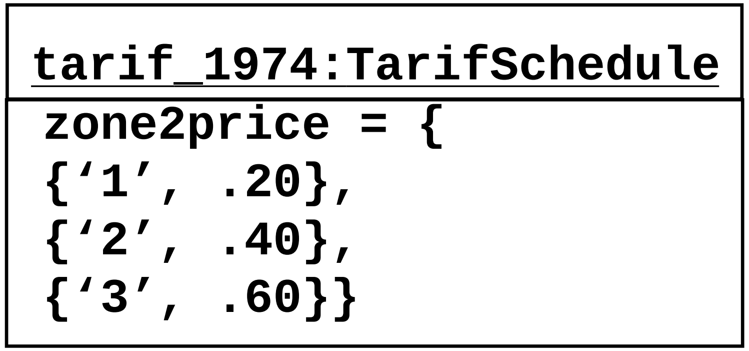
- Class diagrams represent the structure of the system.
- Used
  - **during requirements analysis to model problem domain concepts**
  - **during system design to model subsystems and interfaces**
  - **during object design to model classes.**

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



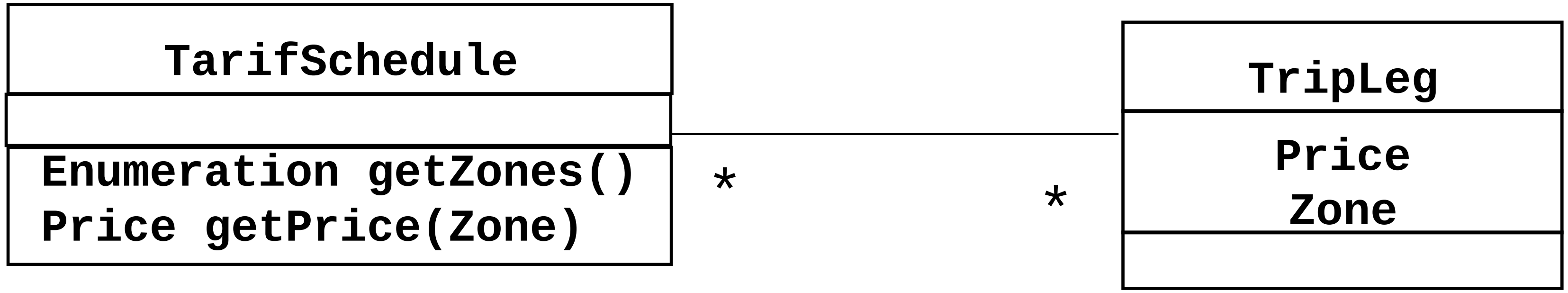
```
tarif_1974:TarifSchedule
zone2price = {
{‘1’, .20},
{‘2’, .40},
{‘3’, .60}}
```

- 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

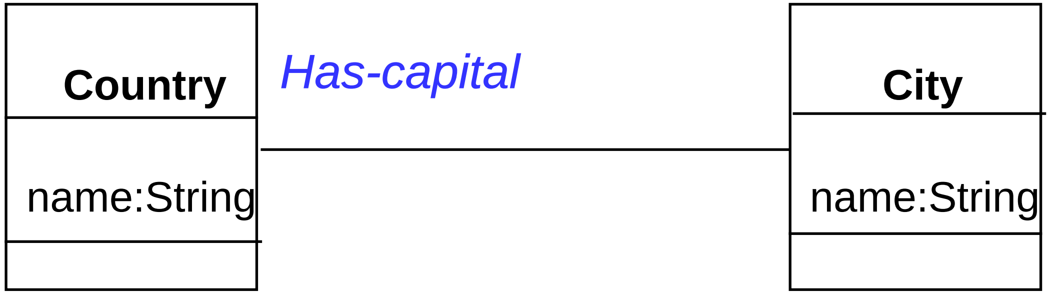
- What is the difference between an actor and a class and an instance?
- Actor:
    - **An entity outside the system to be modeled, interacting with the system (“Passenger”)**
- Class:
    - **An abstraction modeling an entity in the problem domain, inside the system to be modeled (“User”)**
- Object:
    - **A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).**

Associations

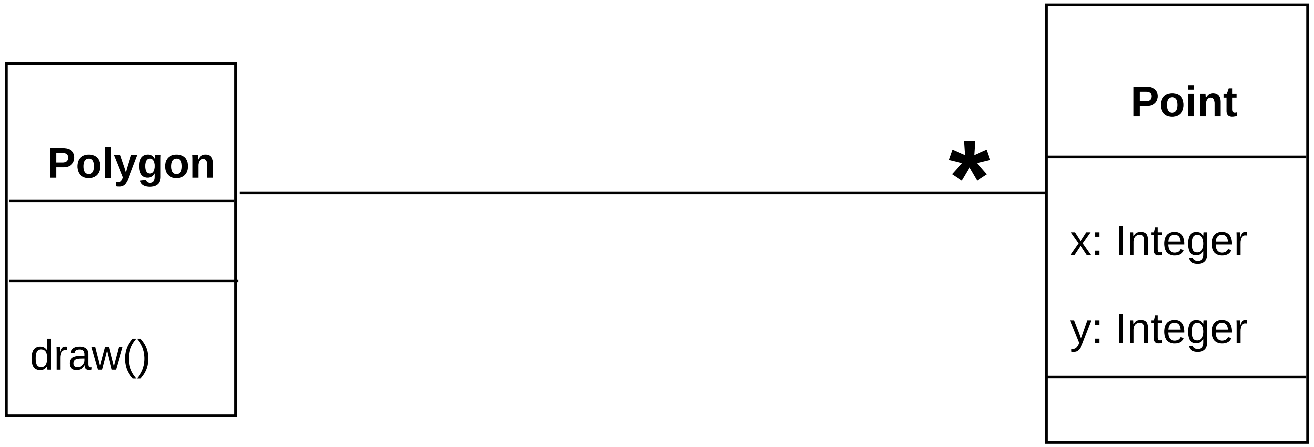


- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

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

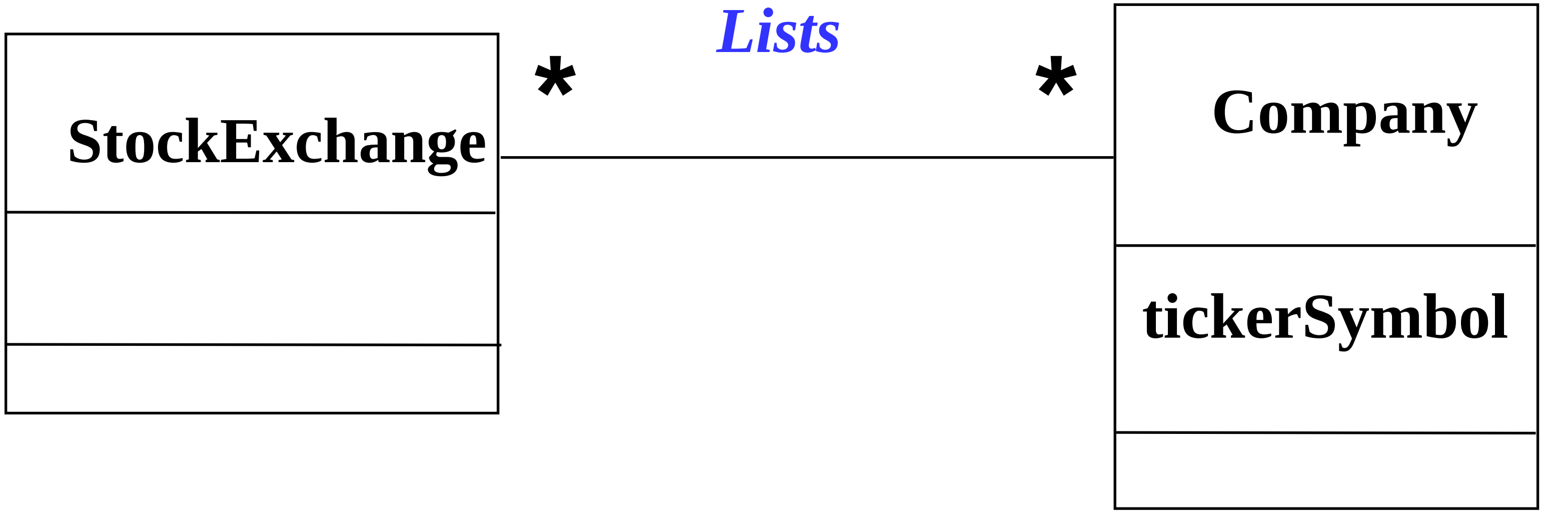


One-to-one association



One-to-many association

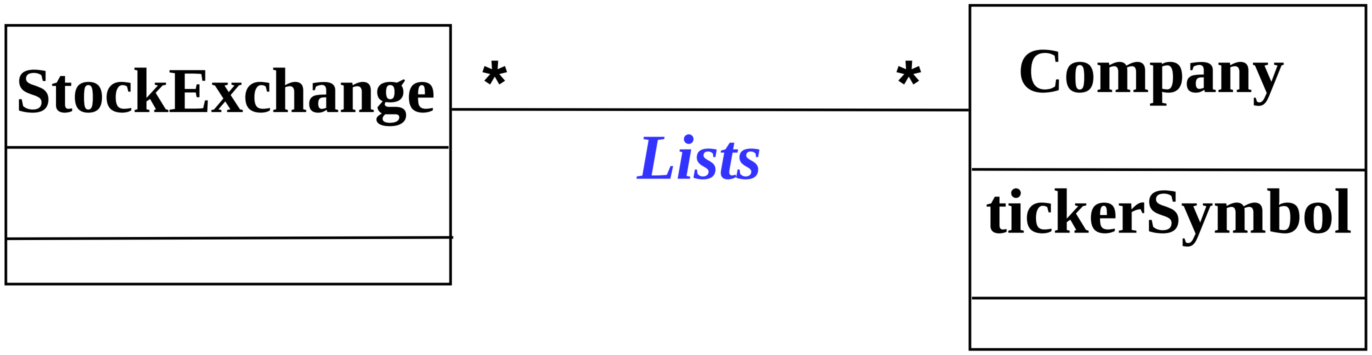
Many-to-Many Associations



From Problem Statement To Object Model

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

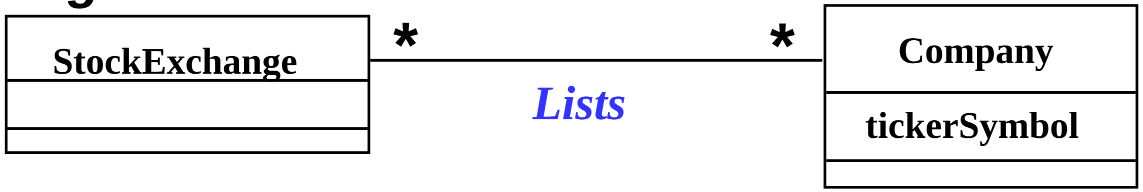
Class Diagram:



From Problem Statement to Code

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

Class Diagram:

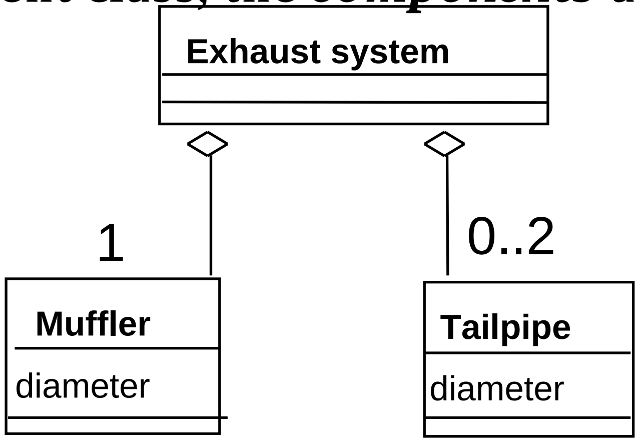


Java Code

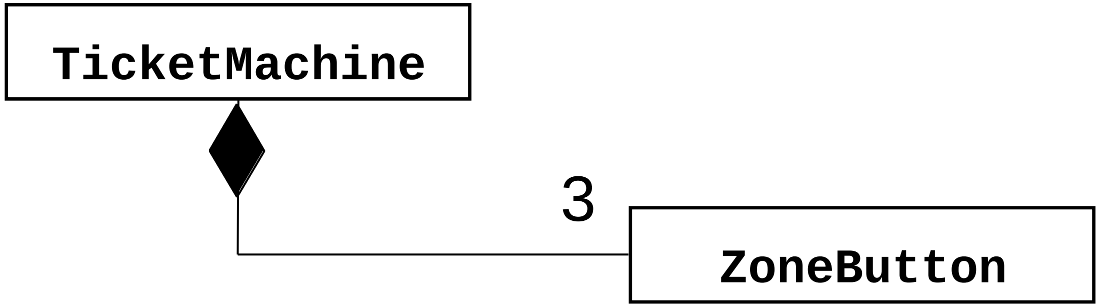
```
public class StockExchange
{
 public Vector m_Company = new Vector();
};

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

Aggregation

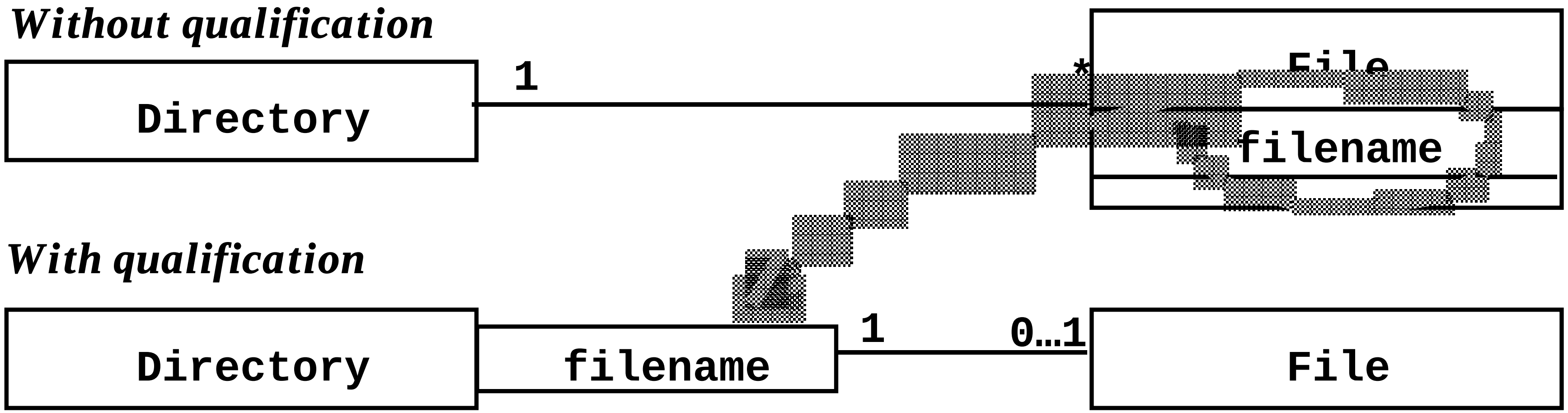
- An ***aggregation*** is a special case of association denoting a “consists of” hierarchy.
- The ***aggregate*** is the parent class, the ***components*** are the children class.



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

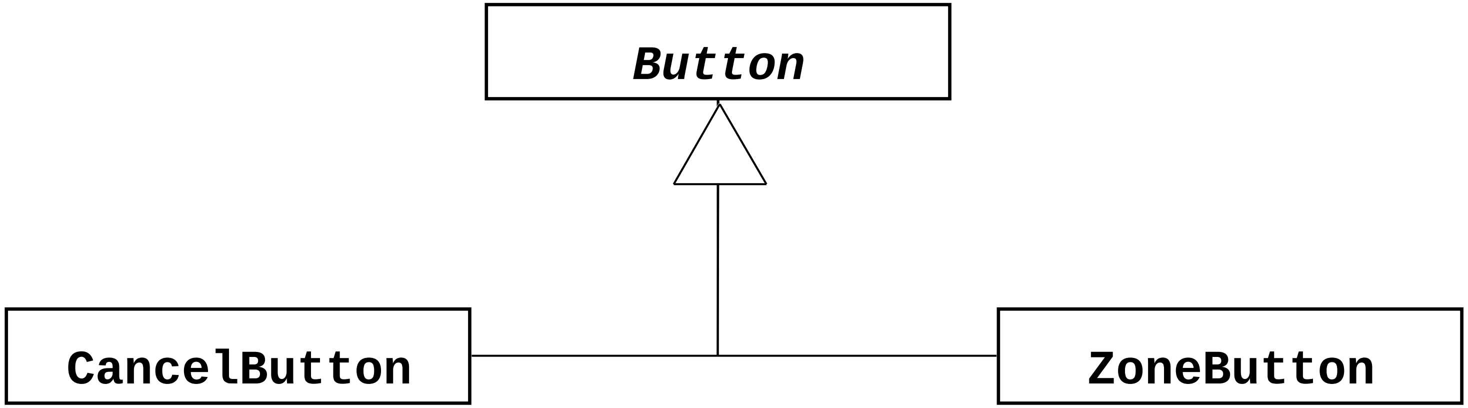


Qualifiers



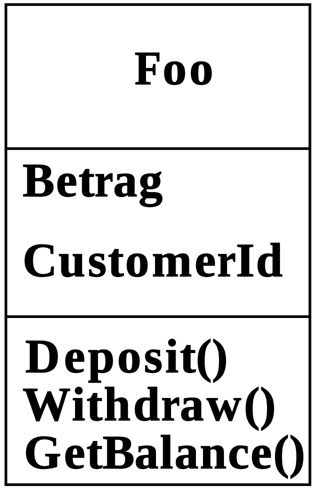
- Qualifiers can be used to reduce the multiplicity of an association.

Generalization



- Generalization relationships denote inheritance between classes.
- The children classes inherit the attributes and operations of the parent class.
- Generalization simplifies the model by eliminating redundancy.

Object Modeling in Practice: Class Identification



Foo

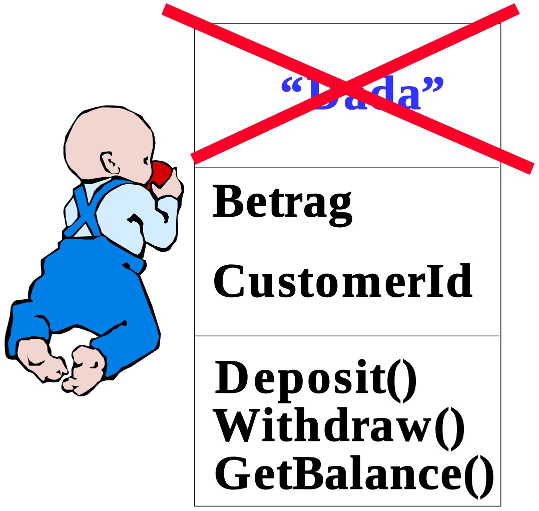
Betrag

CustomerId

Deposit()
Withdraw()
GetBalance()

Class Identification: Name of Class, Attributes and Methods

Object Modeling in Practice: Encourage Brainstorming

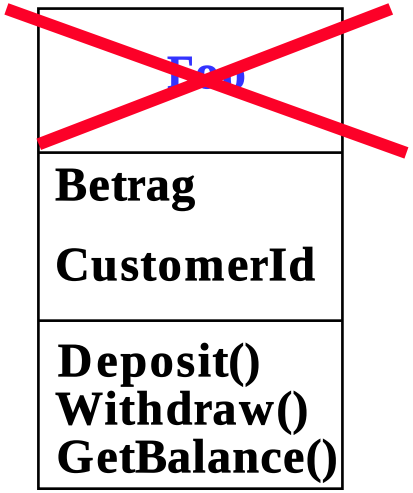


~~“Dada”~~

Betrag

CustomerId

Deposit()
Withdraw()
GetBalance()

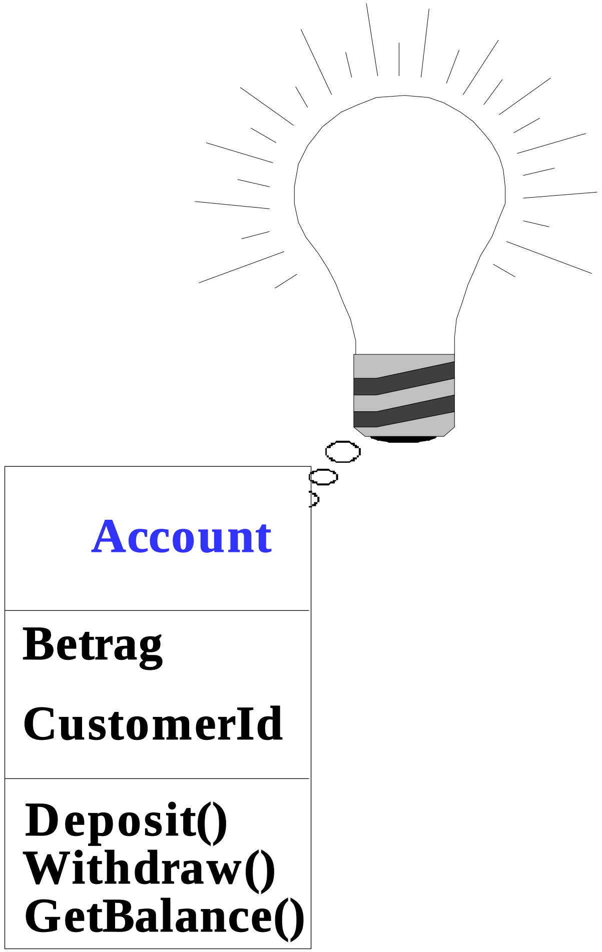


~~Foo~~

Betrag

CustomerId

Deposit()
Withdraw()
GetBalance()



Account

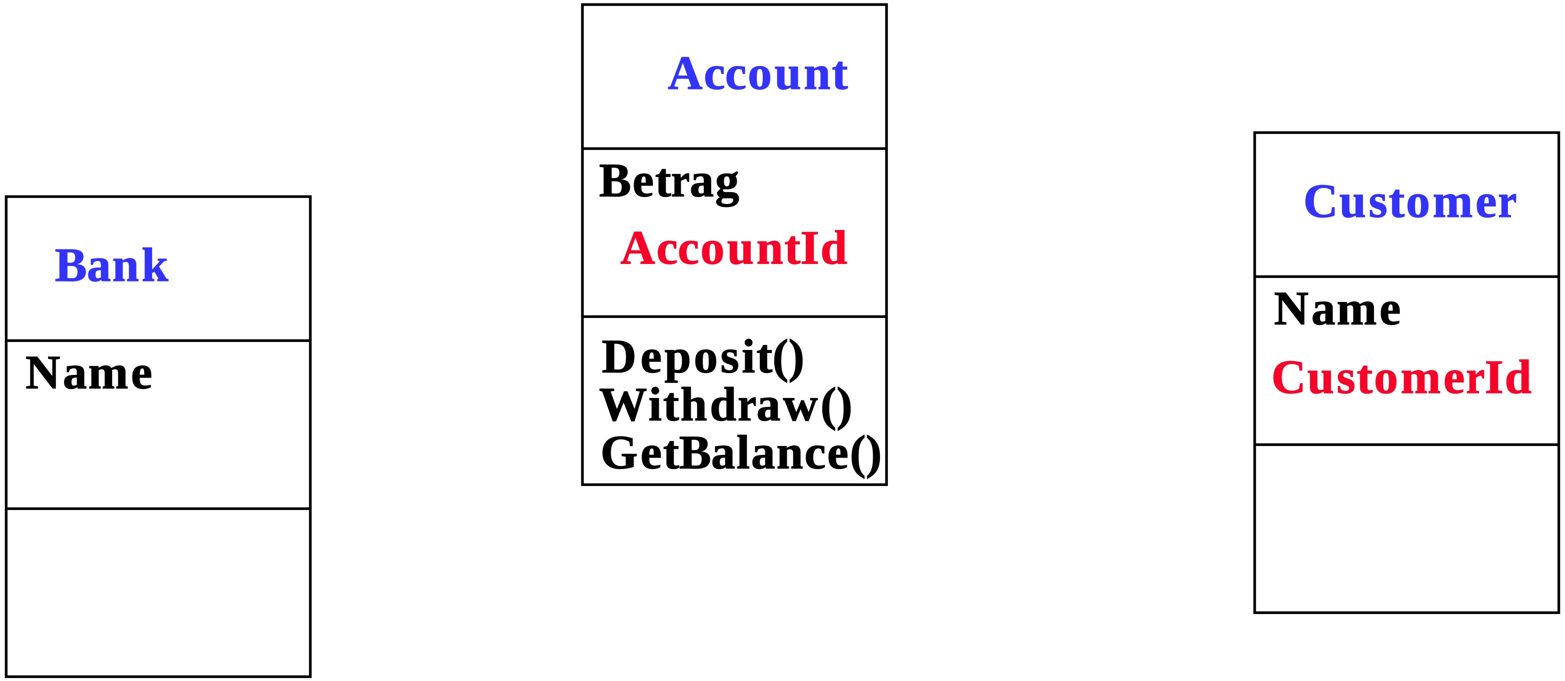
Betrag

CustomerId

Deposit()
Withdraw()
GetBalance()

Naming is important!
Is Foo the right name?

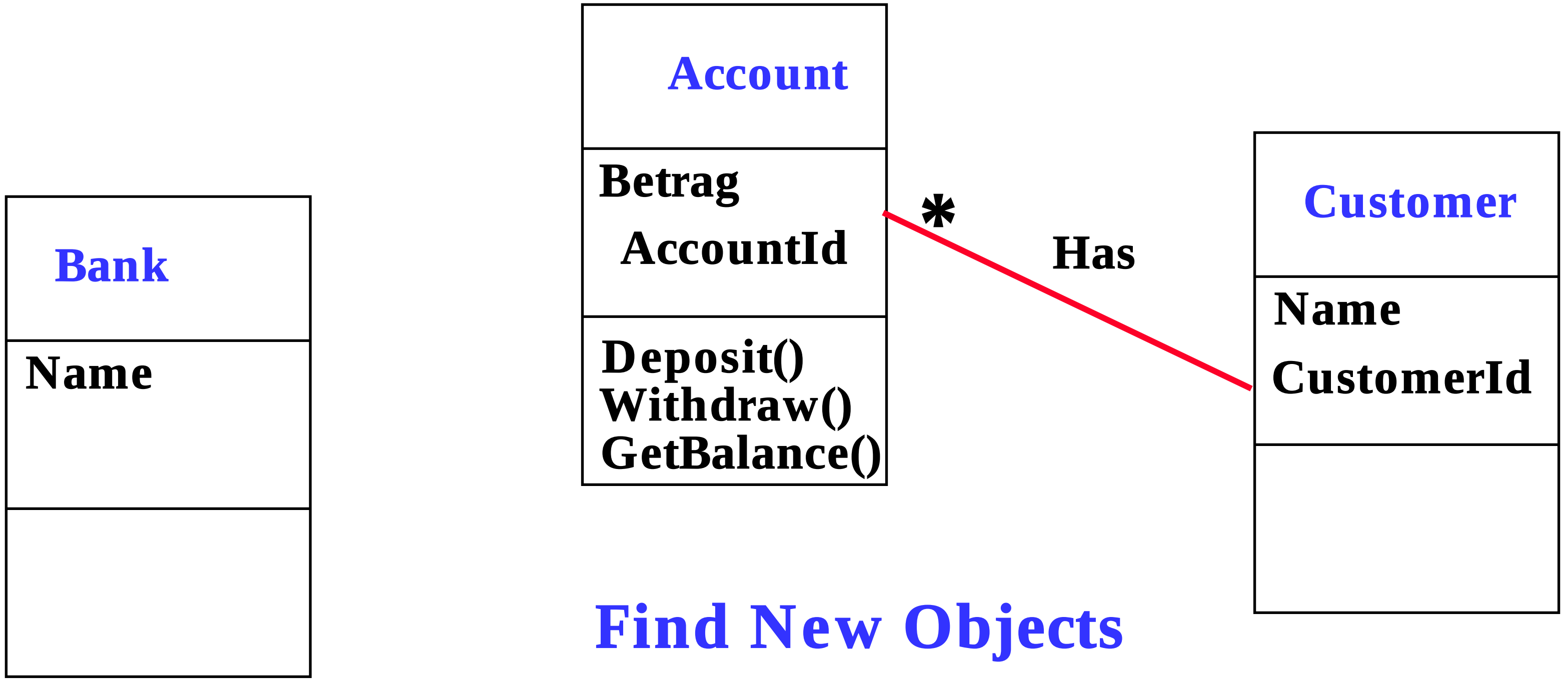
Object Modeling in Practice ctd



Find New Objects

Iterate on Names, Attributes and Methods

Object Modeling in Practice: A Banking System



Find New Objects

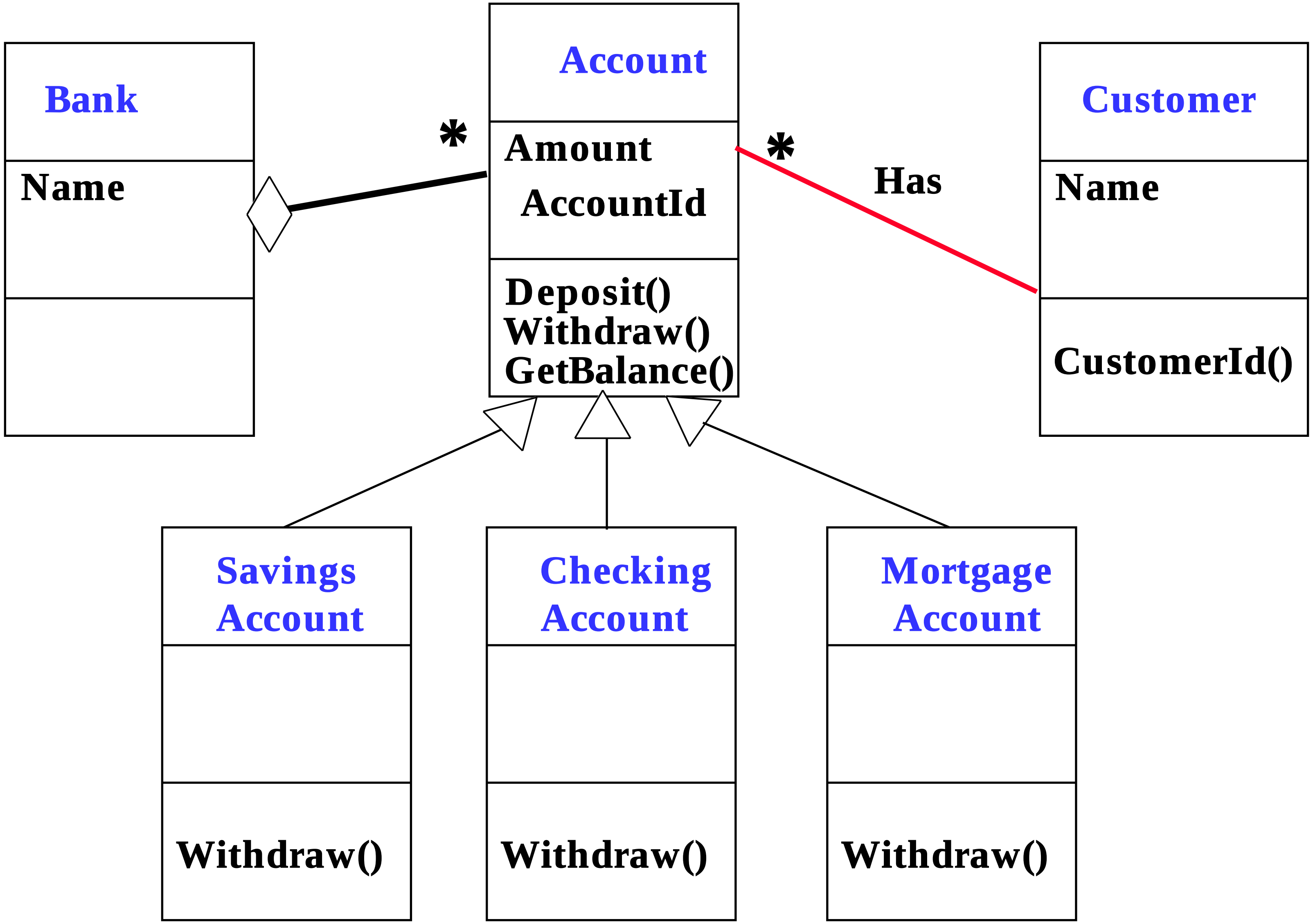
Iterate on Names, Attributes and Methods

Find Associations between Objects

Label the assocations

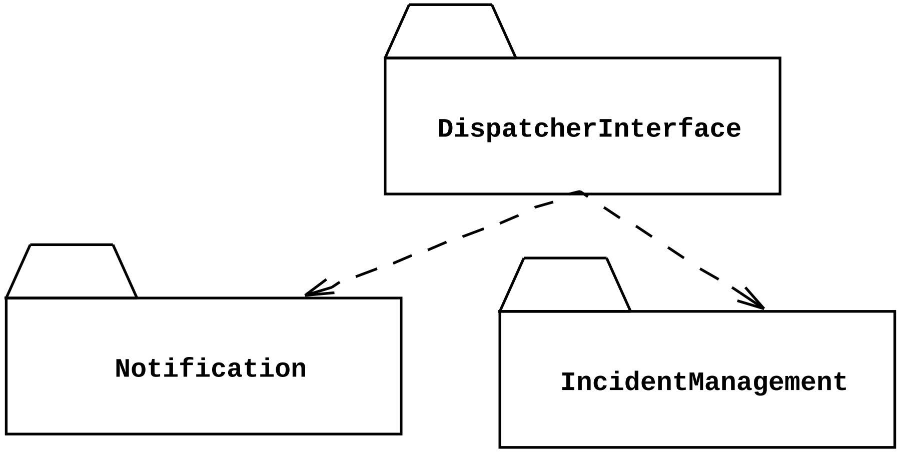
Determine the multiplicity of the assocations

Practice Object Modeling: Iterate, Categorize!



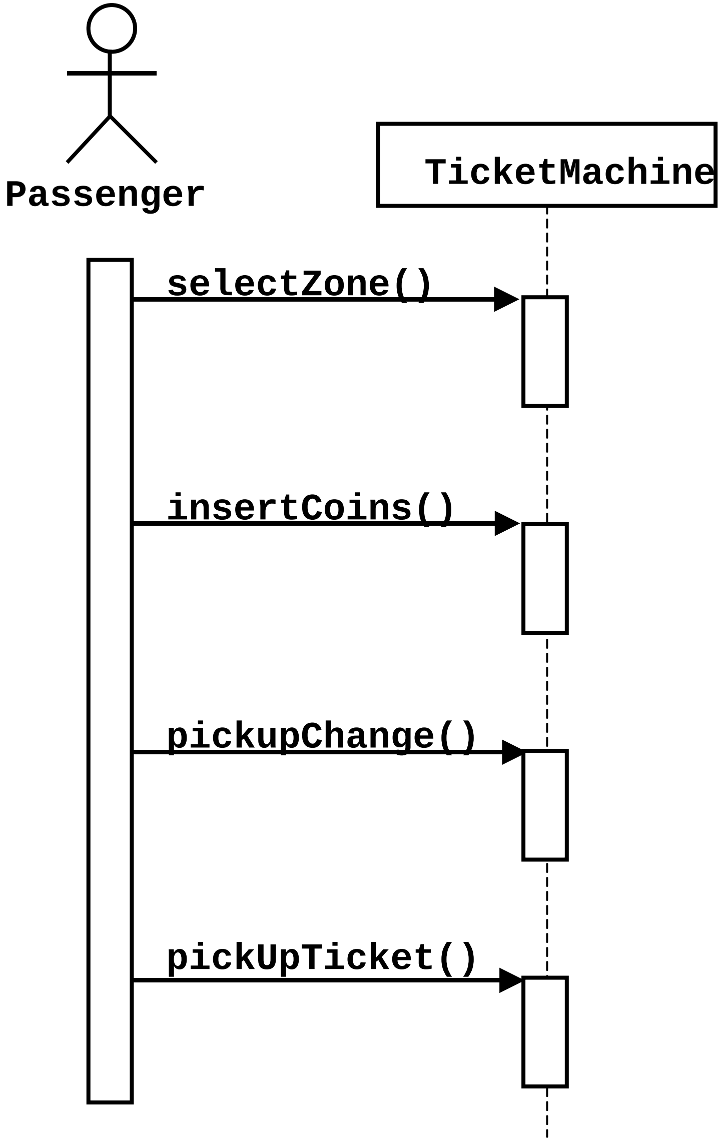
Packages

- 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.



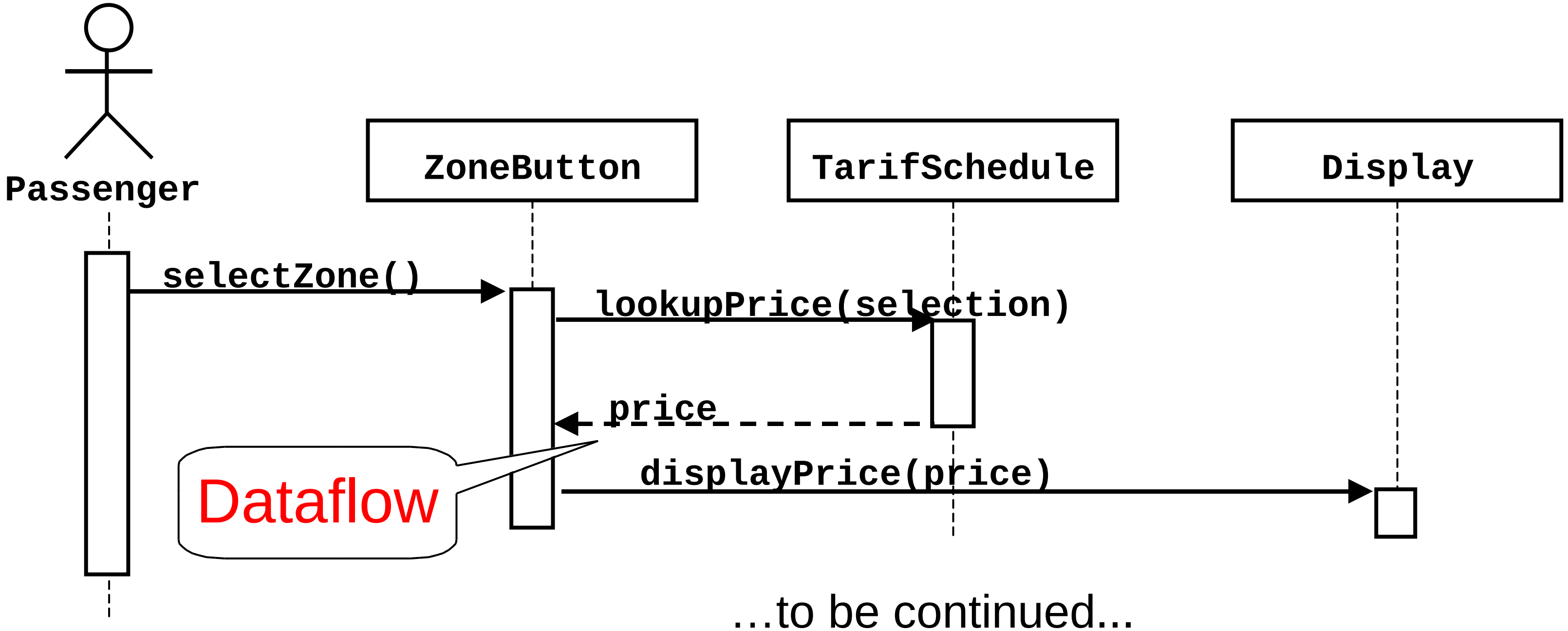
- A complex system can be decomposed into subsystems, where each subsystem is modeled as a package

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

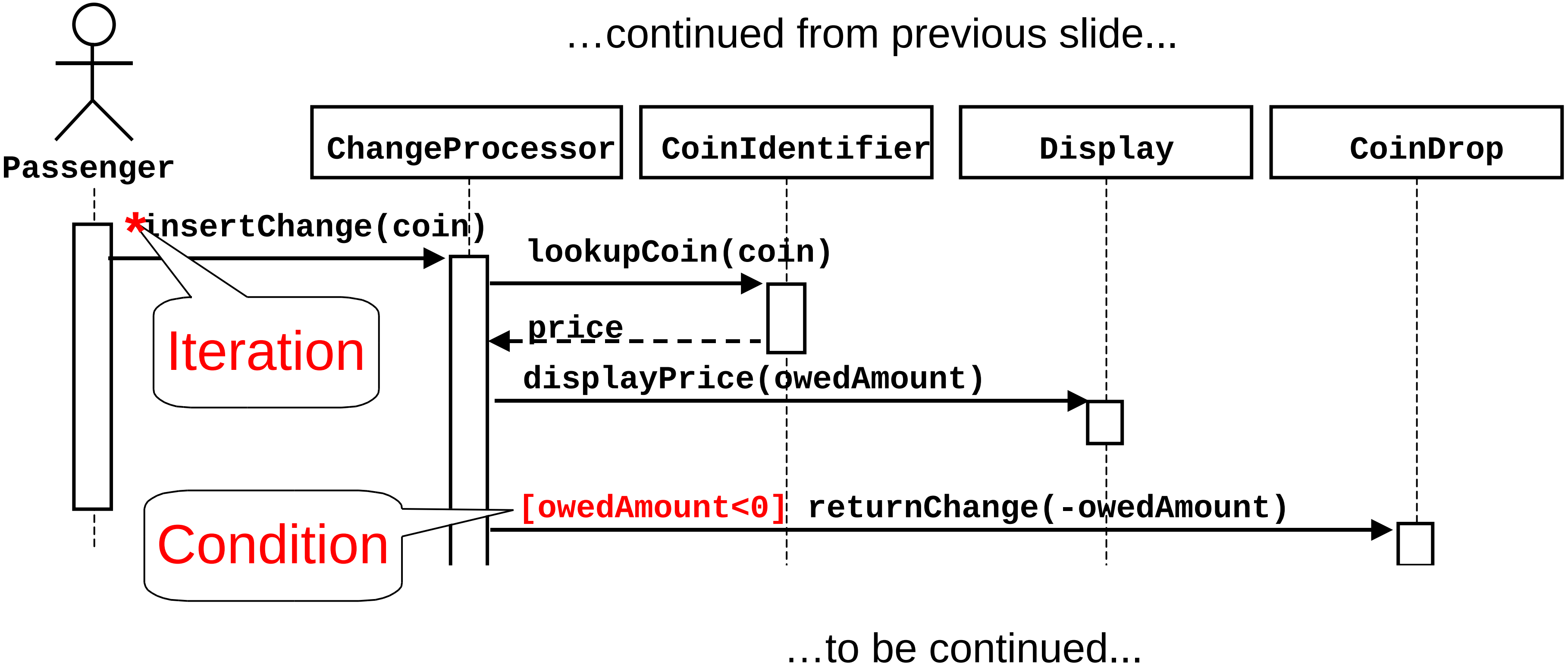
Nested messages



…to be continued...

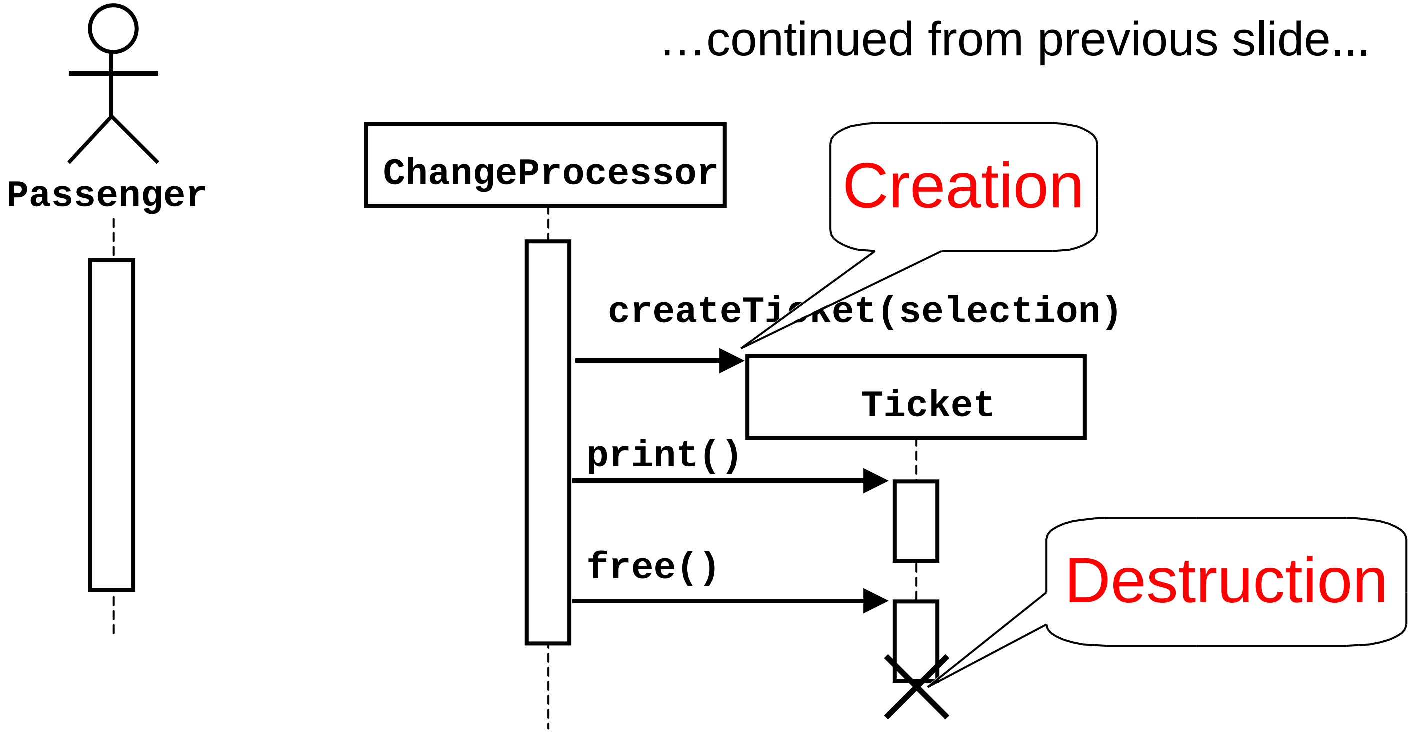
- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations
- Horizontal dashed arrows indicate data flow
- Vertical dashed lines indicate lifelines

Iteration & condition



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

Creation and destruction

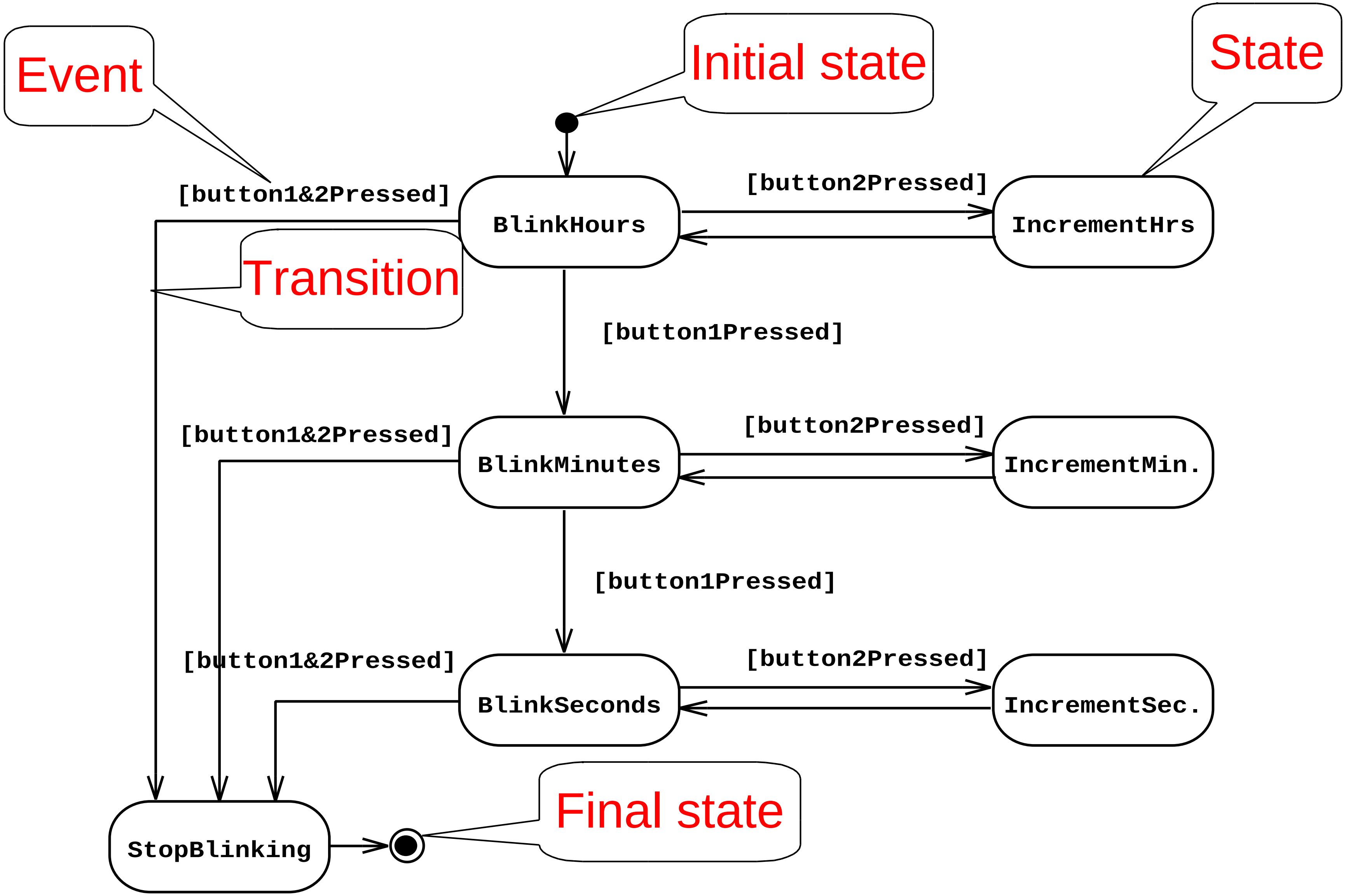


- Creation is denoted by a message arrow pointing to the object.
- Destruction is denoted by an X mark at the end of the destruction activation.
- In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

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.

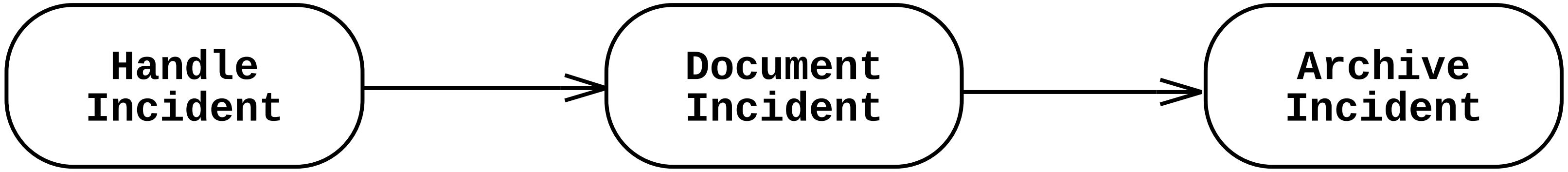
State Chart Diagrams



Represent behavior as states and transitions

Activity Diagrams

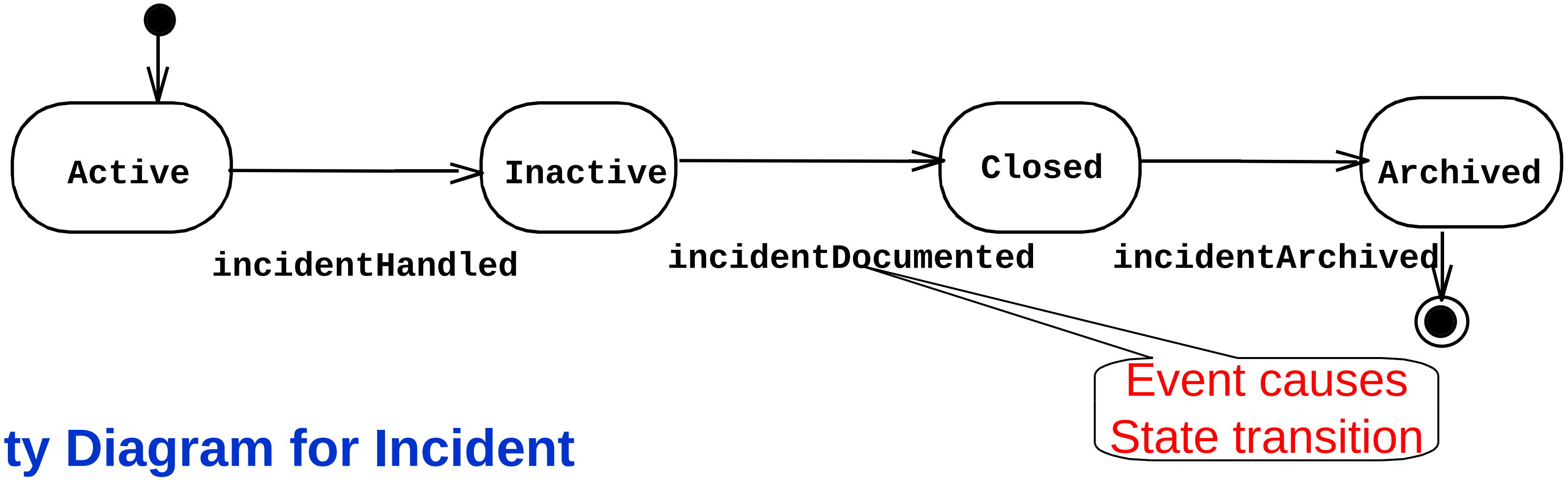
- An activity diagram shows flow control within a system



- 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**

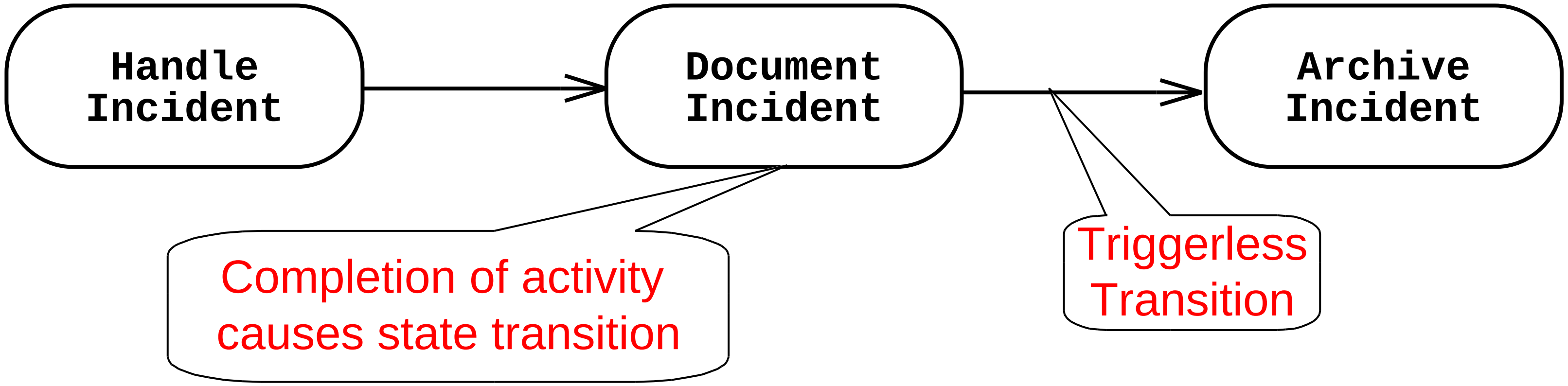
Statechart Diagram vs. Activity Diagram

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



Event causes State transition

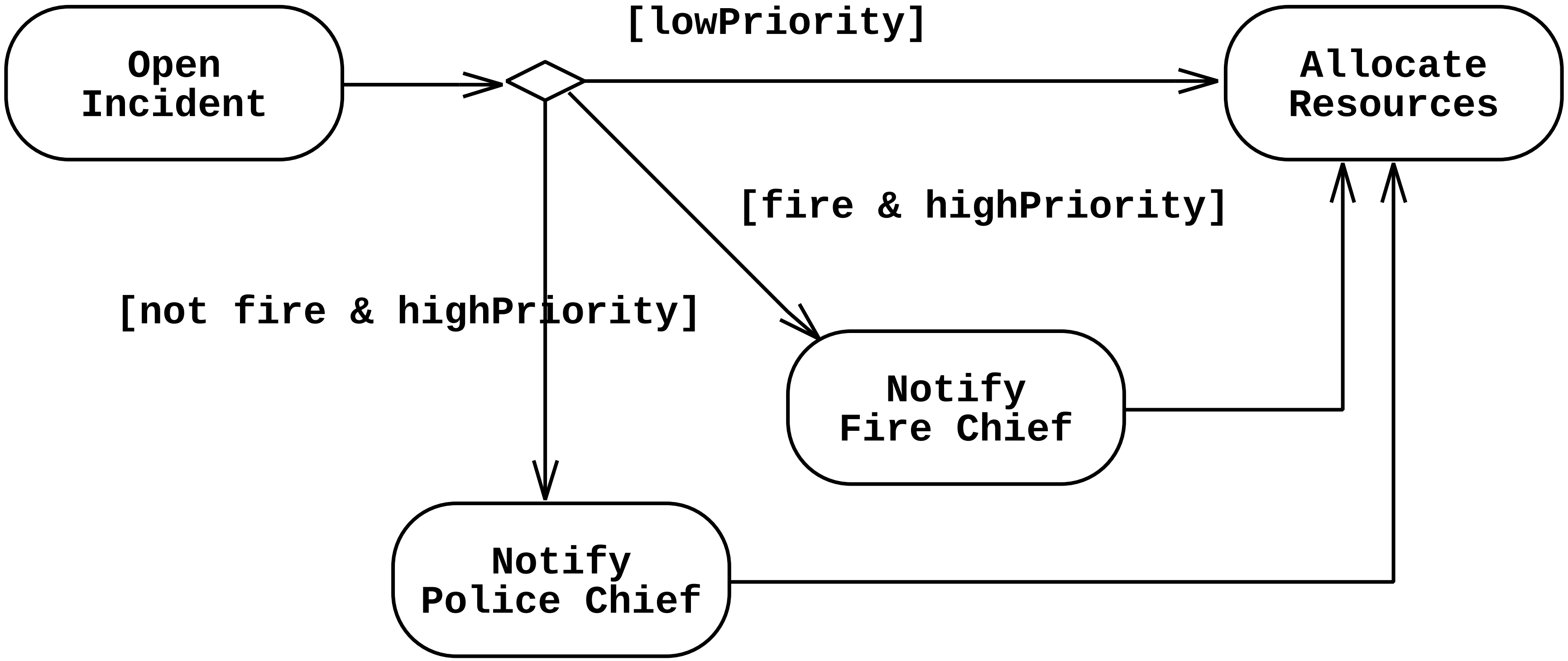
Activity Diagram for Incident
(State: Operation or Collection of Operations)



Completion of activity causes state transition

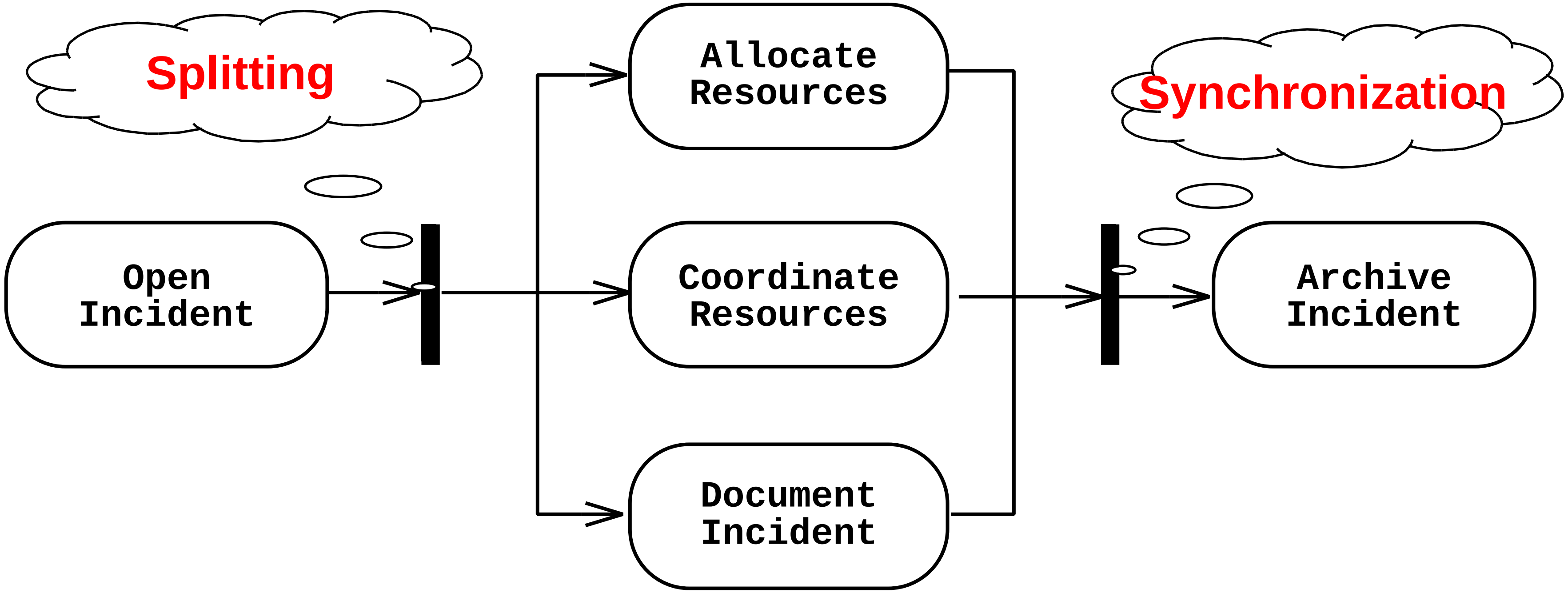
Triggerless Transition

Activity Diagram: Modeling Decisions



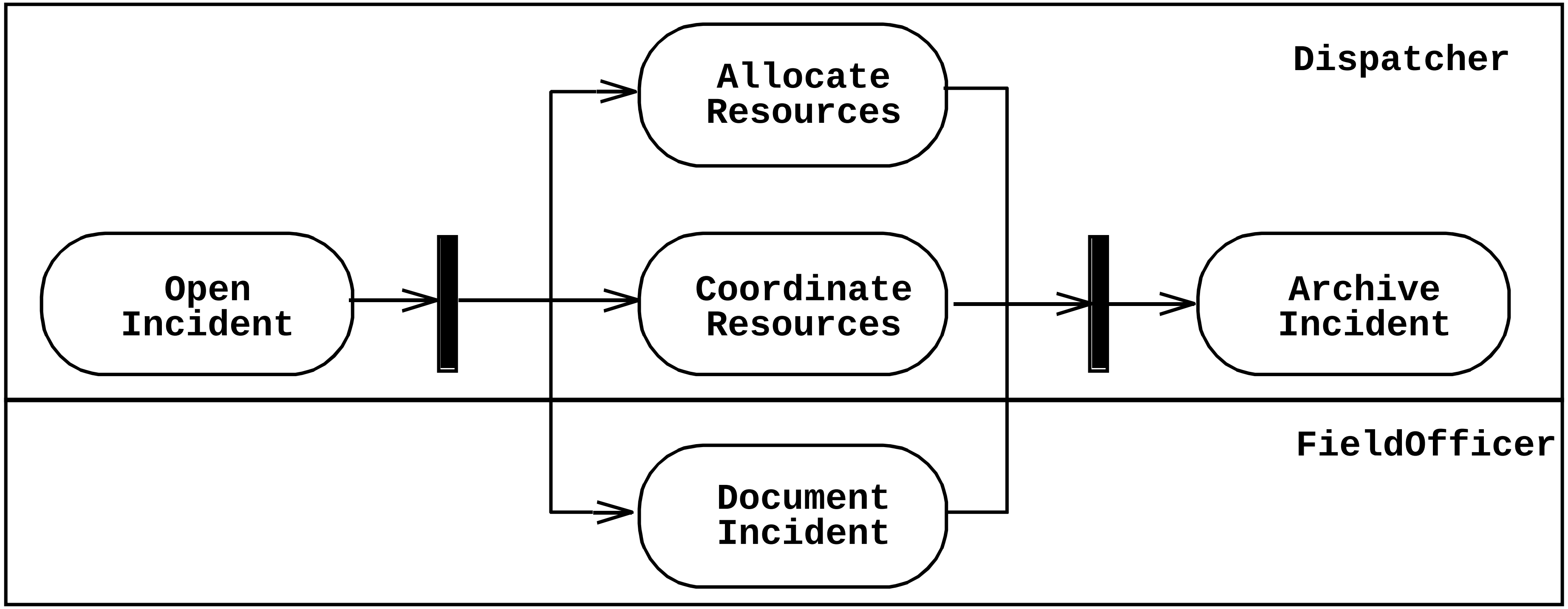
Activity Diagrams: Modeling Concurrency

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