Software Engineering

Professor Dr. Safa Amir Najim
Computer Information System Dept.
College of CS and IT
University of Basrah
2019-2020

Functional Modeling

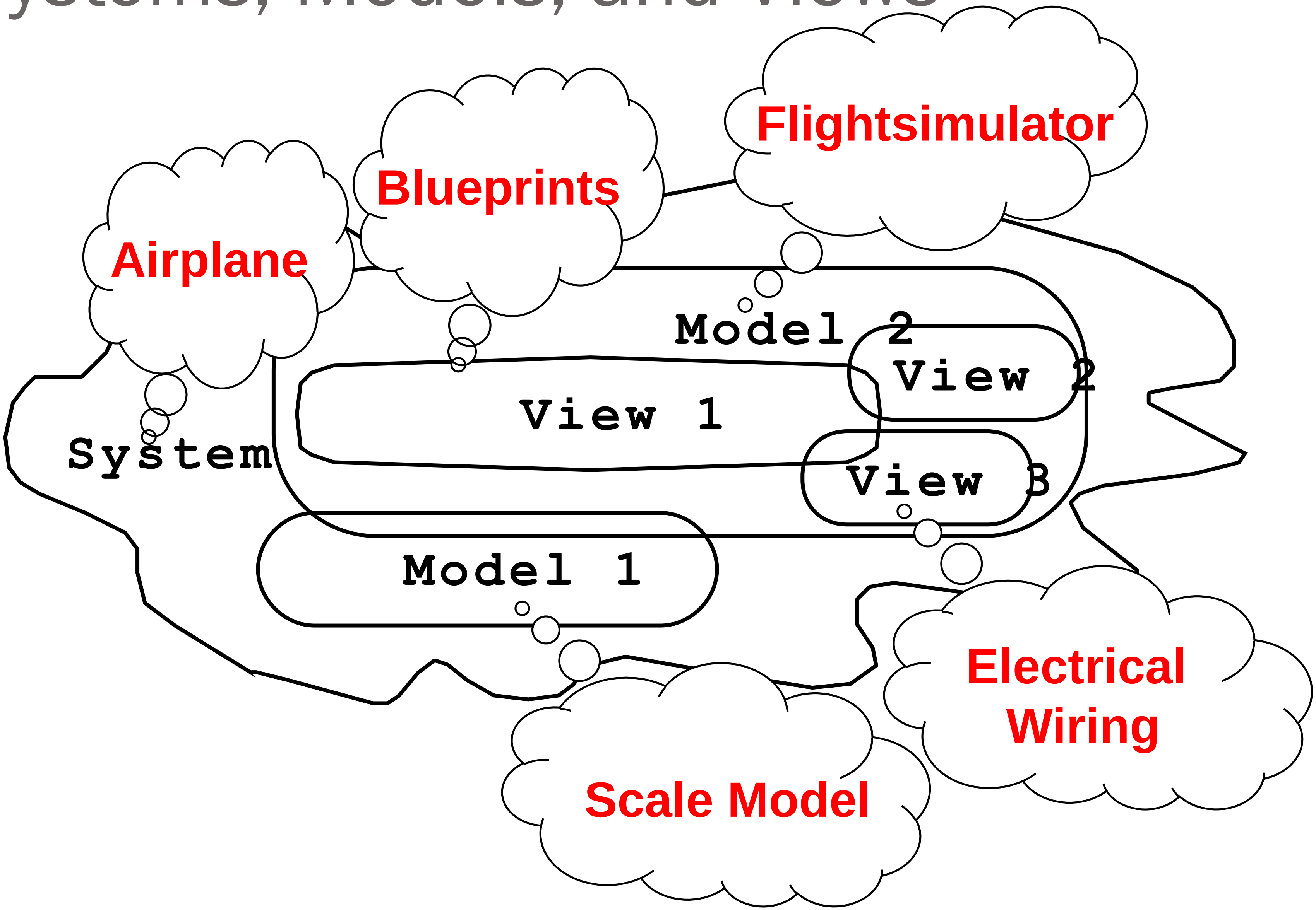
Chapter 4

Systems, Models, and Views

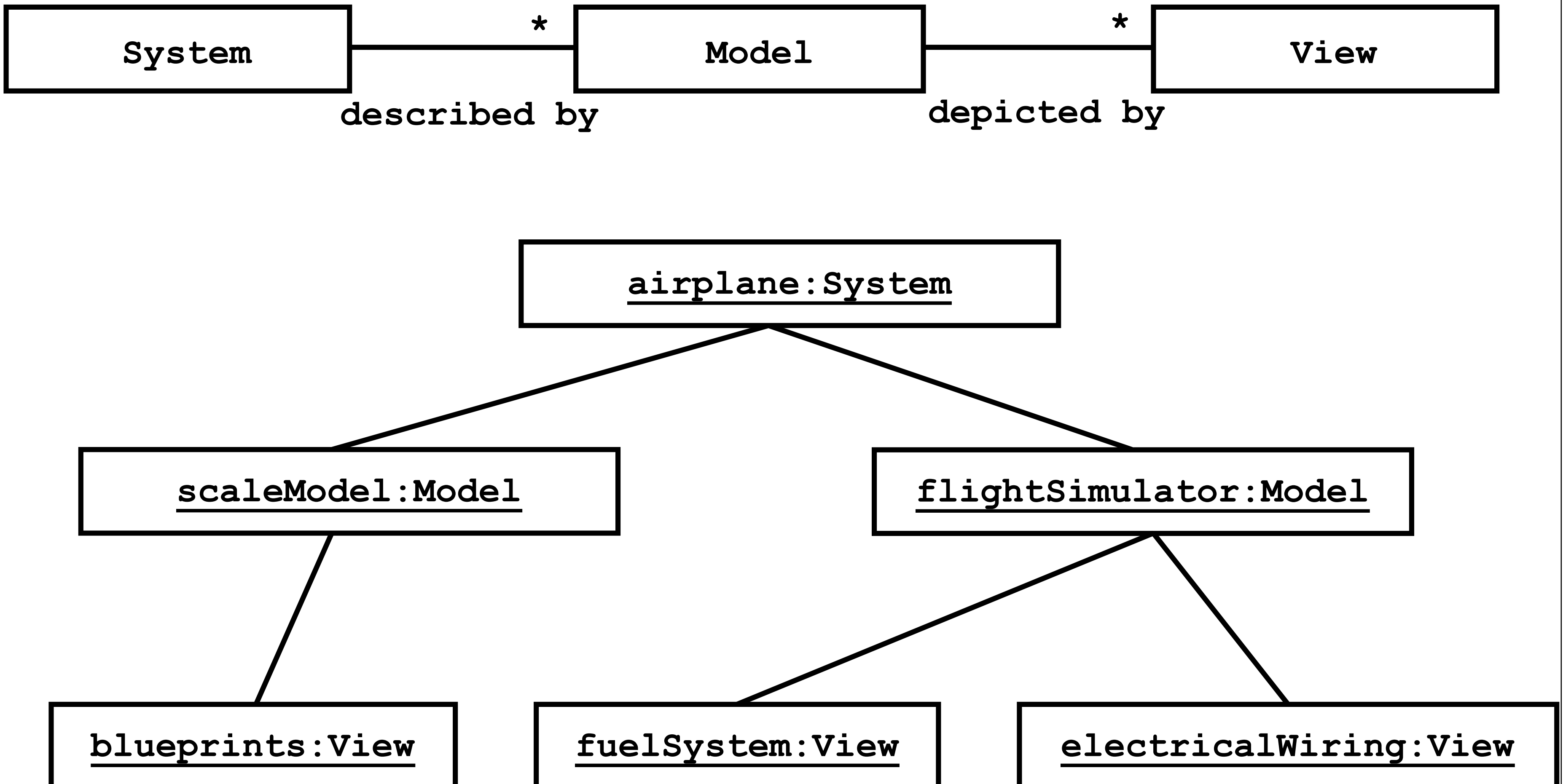
- A *model* is an abstraction describing system or a subset of a system
- A *view* depicts selected aspects of a model
- A *notation* is a set of graphical or textual rules for representing views

- Views and models of a single system may overlap each other

Systems, Models, and Views



Models, Views, and Systems



Why model software?

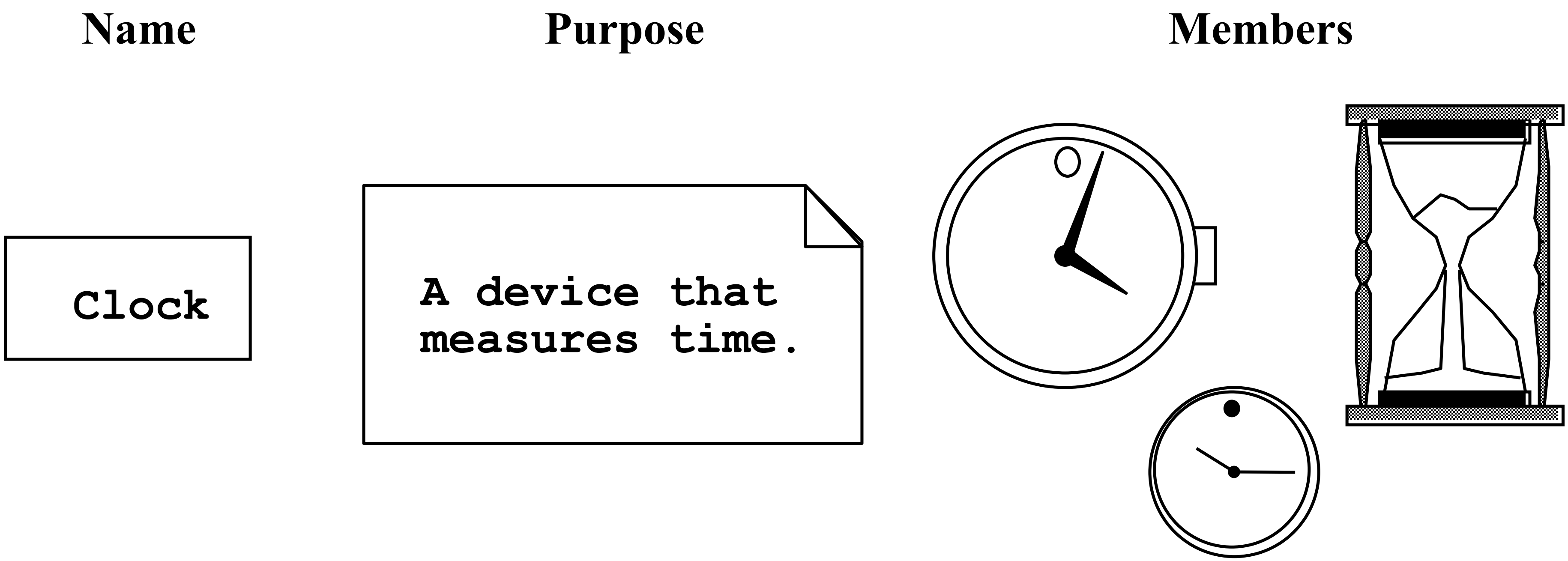
Software is already an abstraction: why model software?

- Software is getting larger, not smaller
  - NT 5.0 ~ 40 million lines of code
  - A single programmer cannot manage this amount of code in its entirety.
- Code is often not directly understandable by developers who did not participate in the development
- We need simpler representations for complex systems
  - Modeling is a mean for dealing with complexity

Concepts and Phenomena

- *Phenomenon*: An object in the world of a domain as you perceive it, for example:
  - The lecture you are attending
- *Concept*: Describes the properties of phenomena that are common, for example:
  - Lectures on software engineering
- A concept is a 3-tuple:
  - Its *Name* distinguishes it from other concepts.
  - Its *Purpose* are the properties that determine if a phenomenon is a member of a concept.
  - Its *Members* are the phenomena which are part of the concept.

Concepts and Phenomena



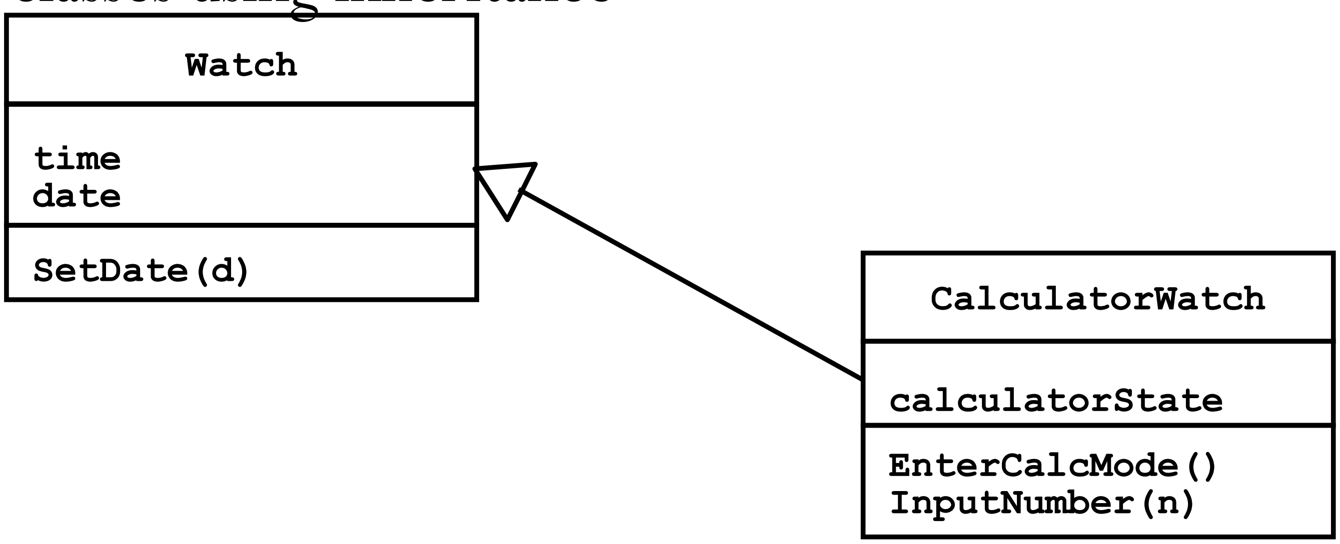
- Abstraction: Classification of phenomena into concepts
- Modeling: Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Concepts In Software: Type and Instance

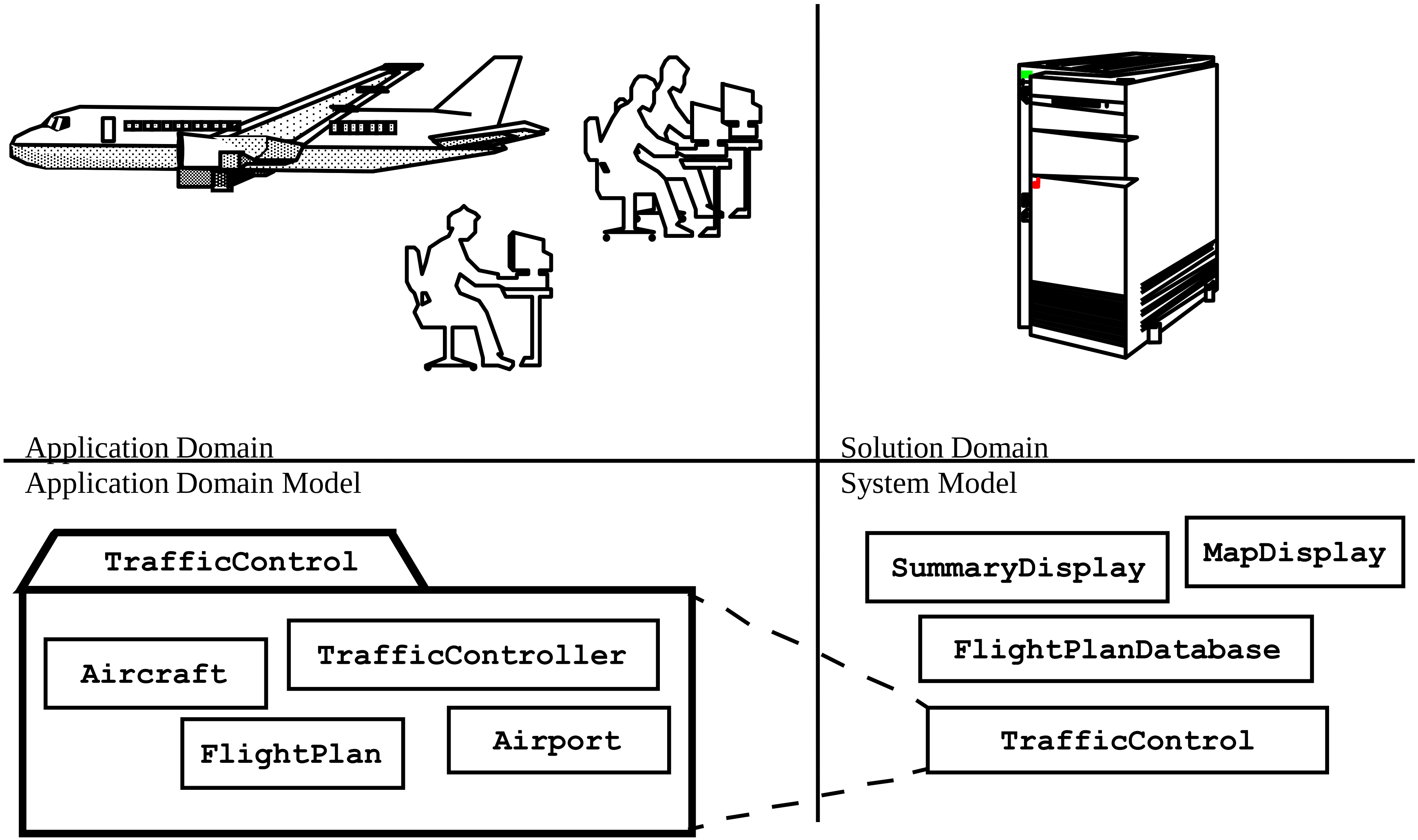
- Type:
  - An abstraction in the context of programming languages
  - Name: int, Purpose: integral number, Members: `0, -1, 1, 2, -2, . . .`
- Instance:
  - Member of a specific type
- The type of a variable represents all possible instances the variable can take.
- The relationship between "type" and "instance" is similar to that of "concept" and "phenomenon."
- Abstract data type:
  - Special type whose implementation is hidden from the rest of the system.

Class

- 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

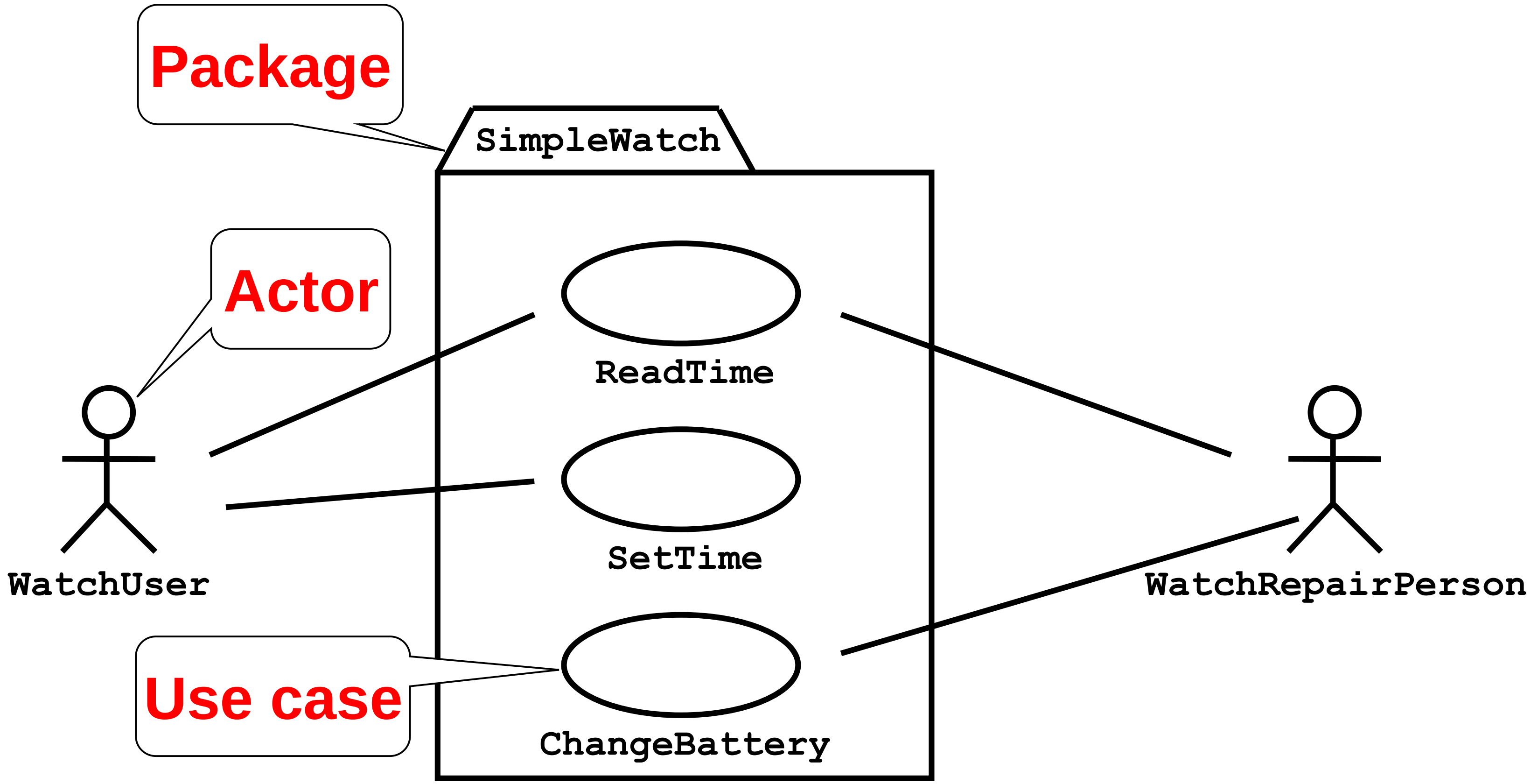


Object-Oriented Modeling



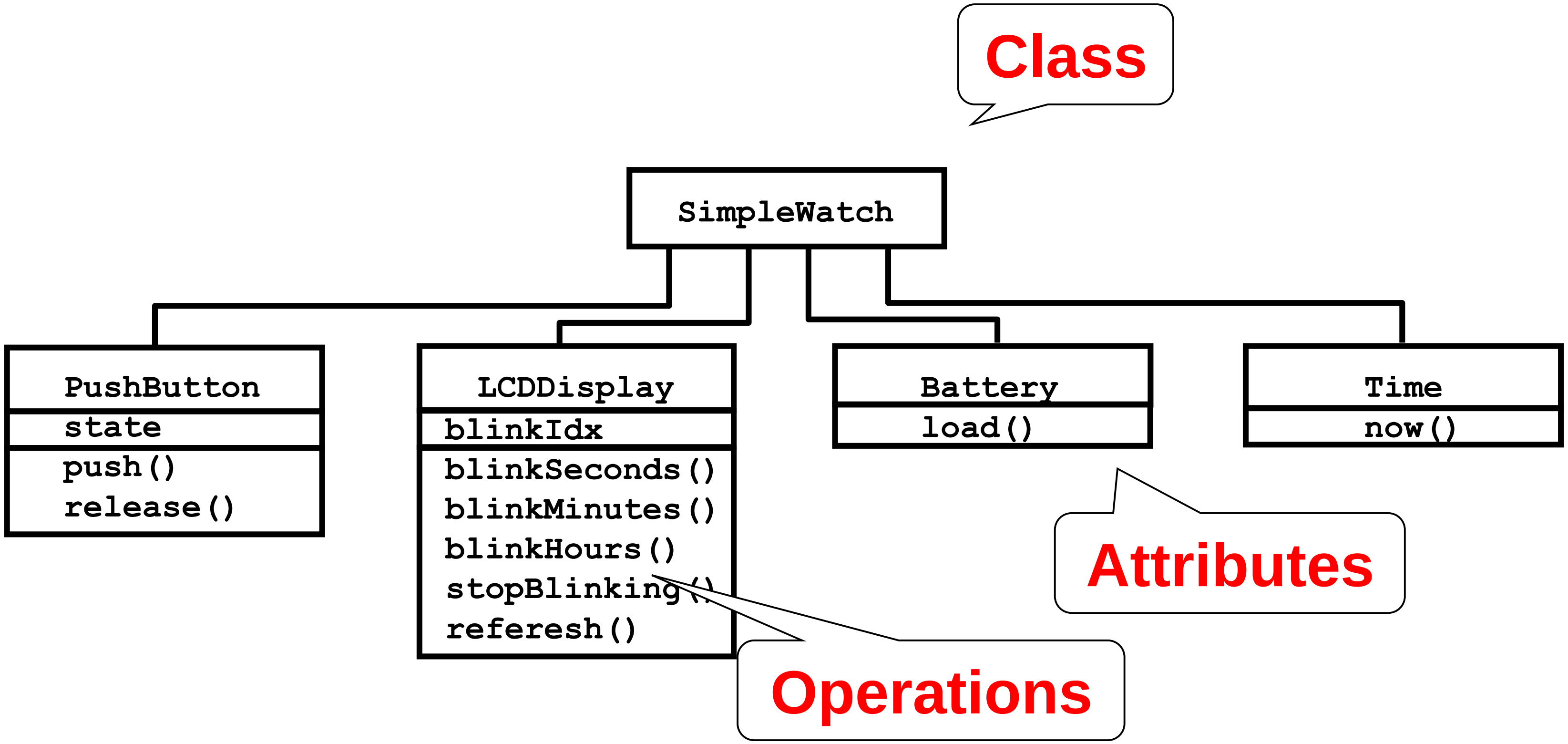
- Use case diagrams
  - Describe the functional behavior of the system as seen by the user.
- Class diagrams
  - Describe the static structure of the system: Objects, Attributes, and Associations.
- Sequence diagrams
  - Describe the dynamic behavior between actors and the system and between objects of the system.
- Statechart diagrams
  - Describe the dynamic behavior of an individual object as a finite state machine.
- Activity diagrams
  - Model the dynamic behavior of a system, in particular the workflow, i.e. a flowchart.

Use Case Diagrams



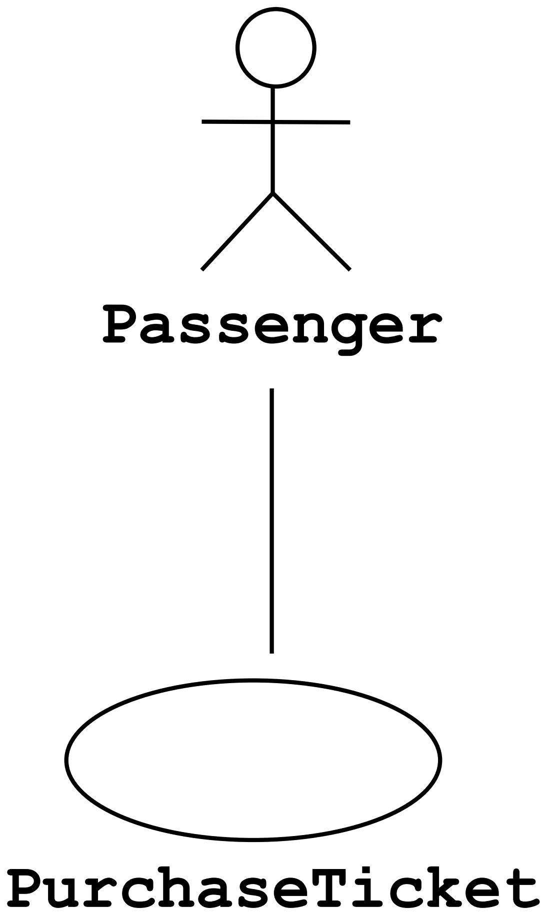
Use case diagrams represent the functionality of the system from user's point of view

Class Diagrams



Class diagrams represent the structure of the system

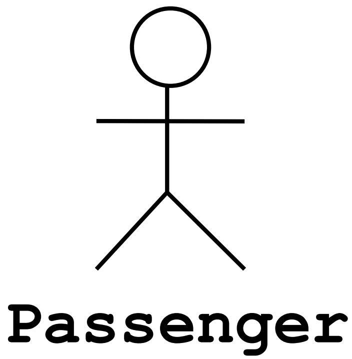
Use Case Diagrams



Used during requirements elicitation to represent external behavior

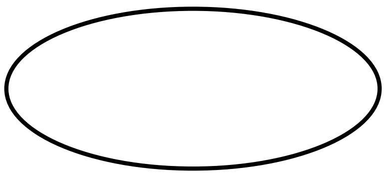
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

Actors



- 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 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.

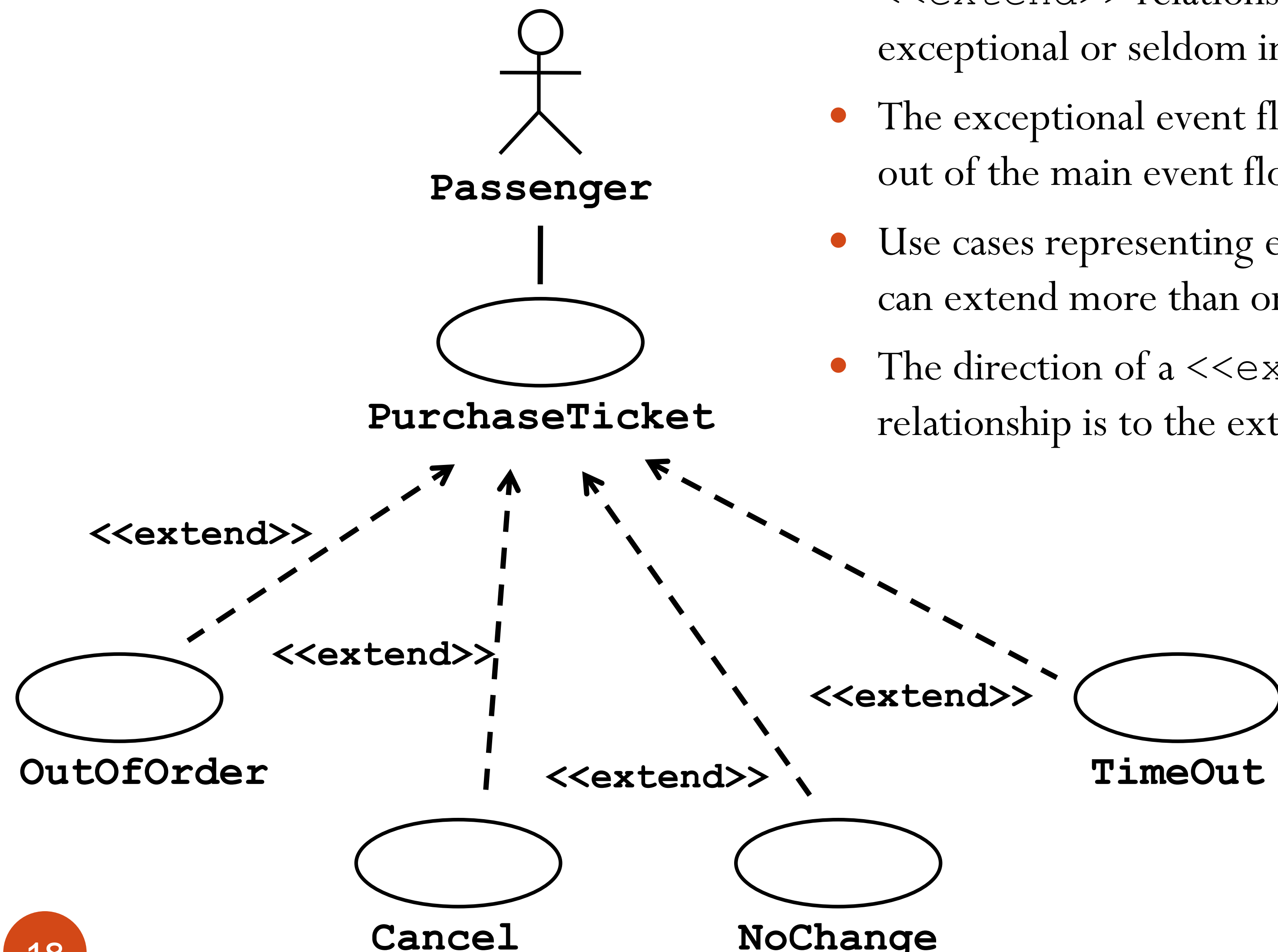
Event flow:

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?

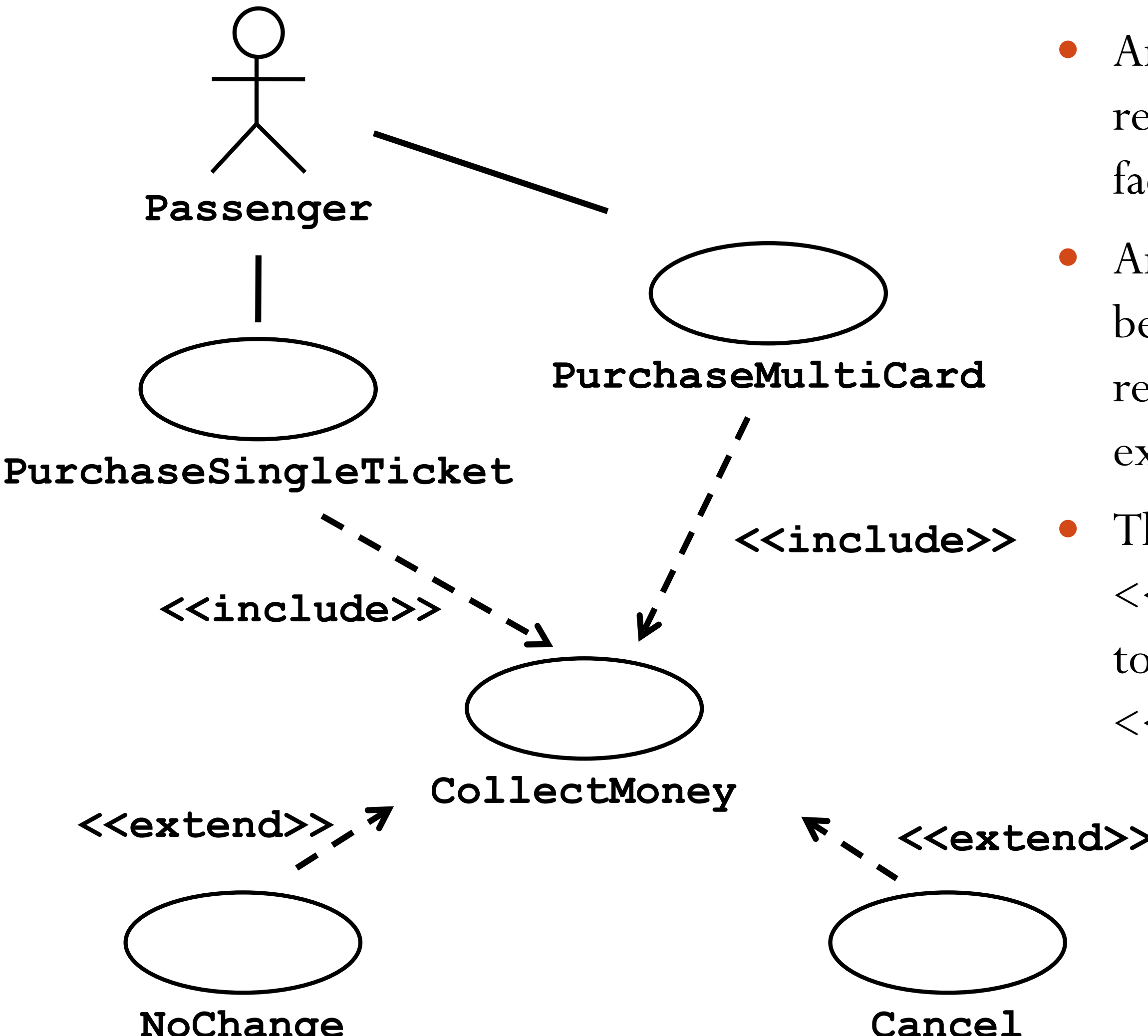
Exceptional cases!

The <<extend>> Relationship



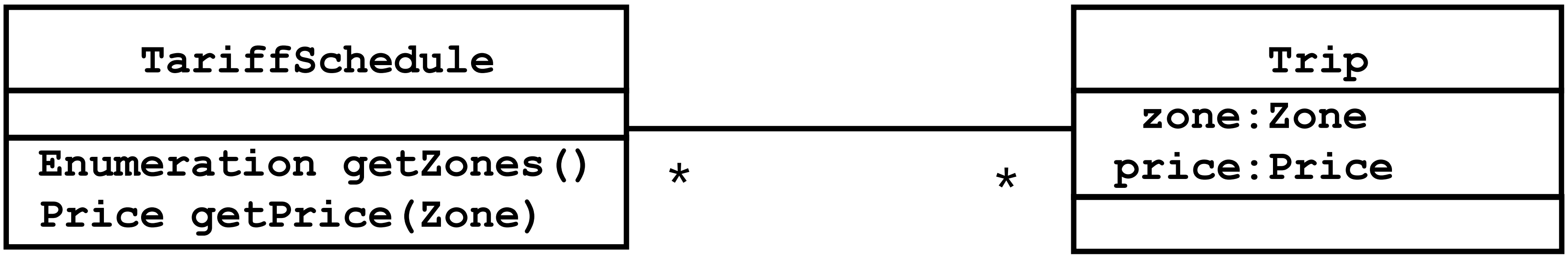
- <<extend>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extend>> relationship is to the extended use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

The <<include>> Relationship



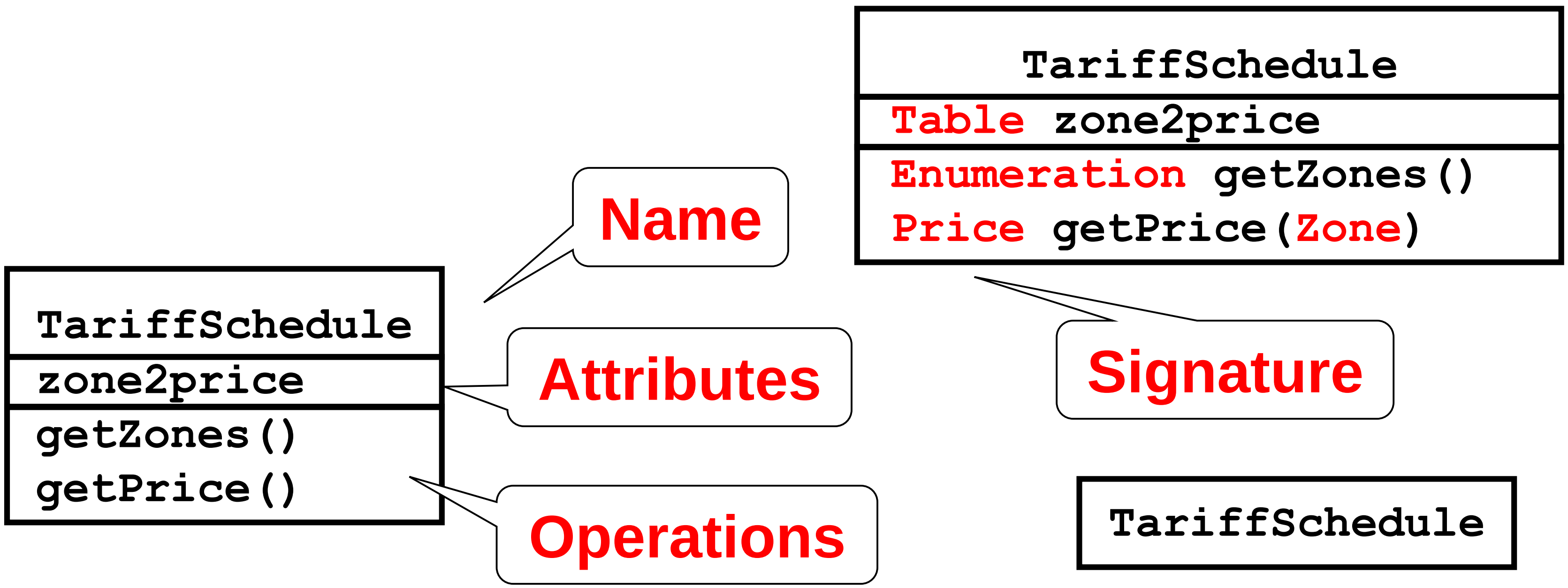
- An `<<include>>` relationship represents behavior that is factored out of the use case.
- An `<<include>>` represents behavior that is factored out for reuse, not because it is an exception.
- The direction of a `<<include>>` relationship is to the using use case (unlike `<<extend>>` relationships).

Class Diagrams



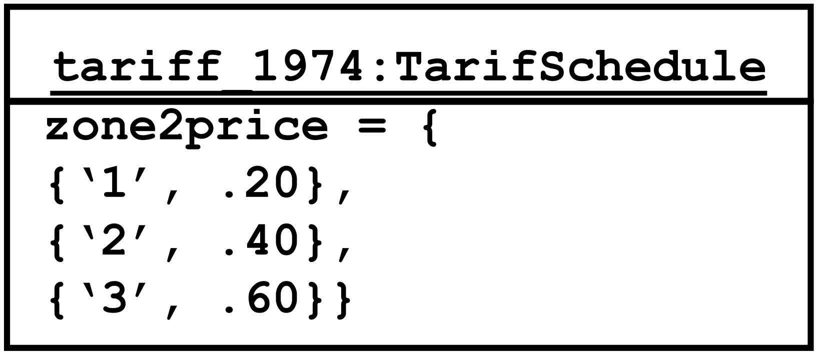
- Class diagrams represent the structure of the system.
- Class diagrams are used
  - during requirements analysis to model problem domain concepts
  - during system design to model subsystems and interfaces
  - during object design to model classes.

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

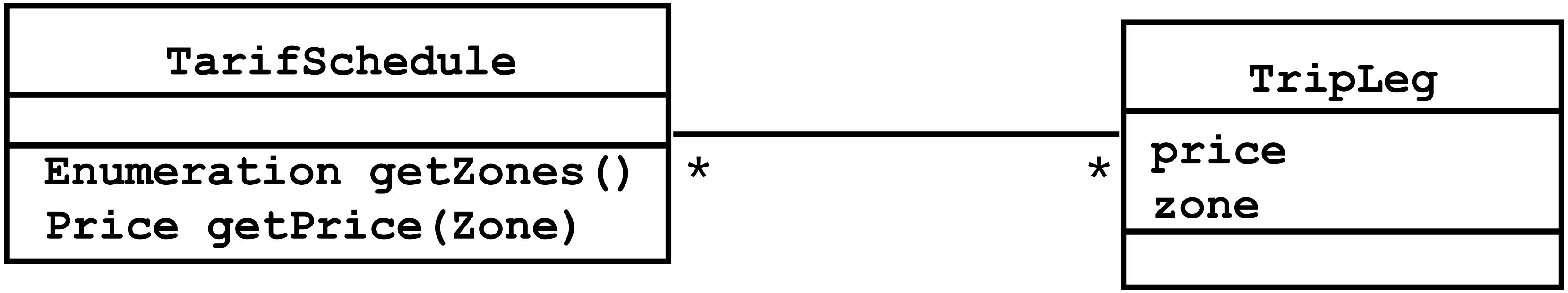


- An ***instance*** represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their ***values***.

Actor vs. Instances

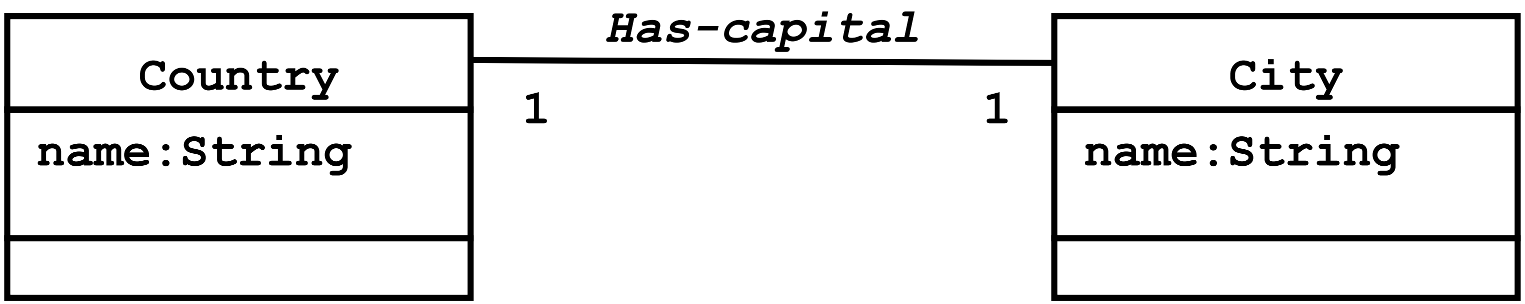
- What is the difference between an actor and a class and an instance?
- Actor:
  - An entity outside the system to be modeled, interacting with the system ("Pilot")
- Class:
  - An abstraction modeling an entity in the problem domain, inside the system to be modeled ("Cockpit")
- Object:
  - A specific instance of a class ("Joe, the inspector").

Associations

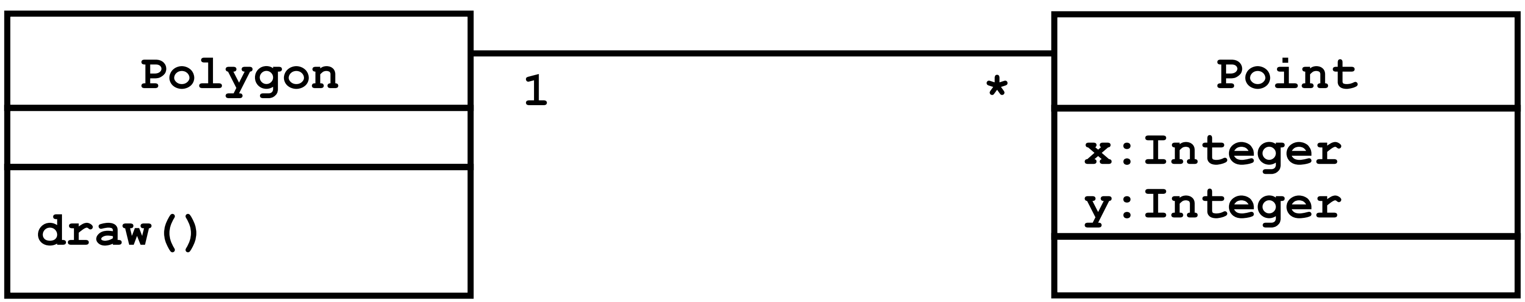


- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

1-to-1 and 1-to-Many Associations

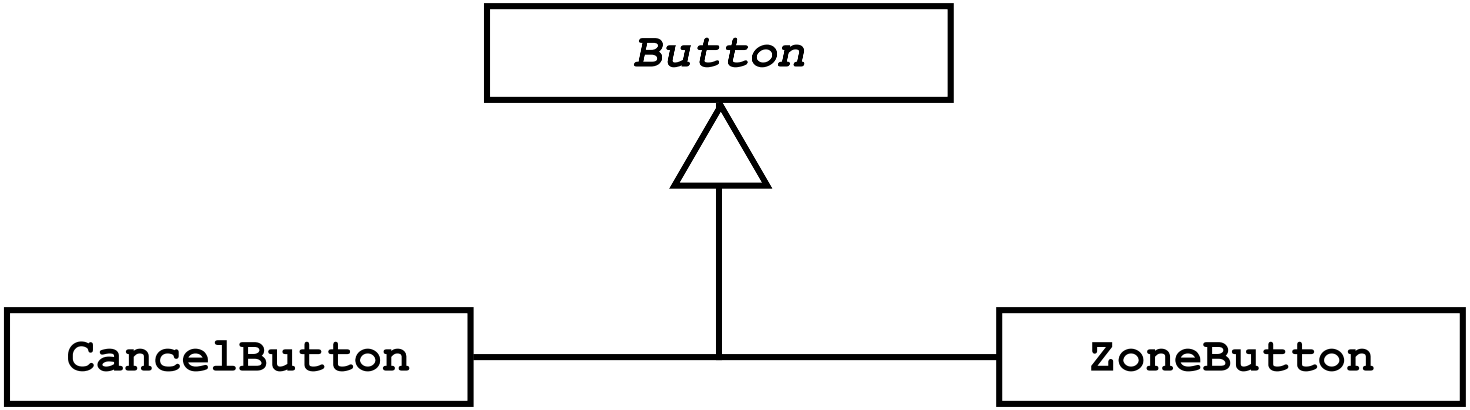


1-to-1 association



1-to-many 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.

From Problem Statement to Code

Problem Statement

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

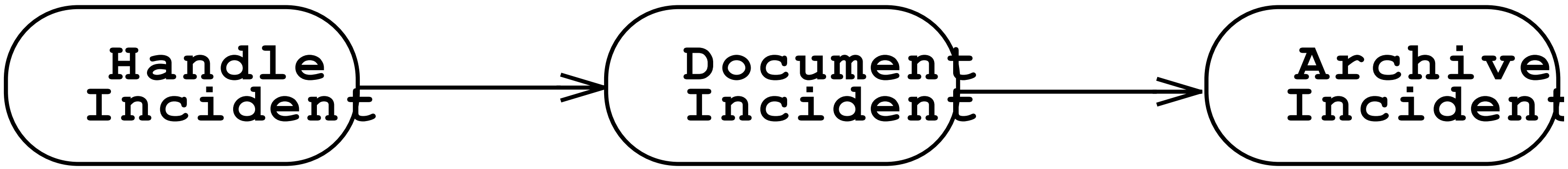
Class Diagram

Java Code

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

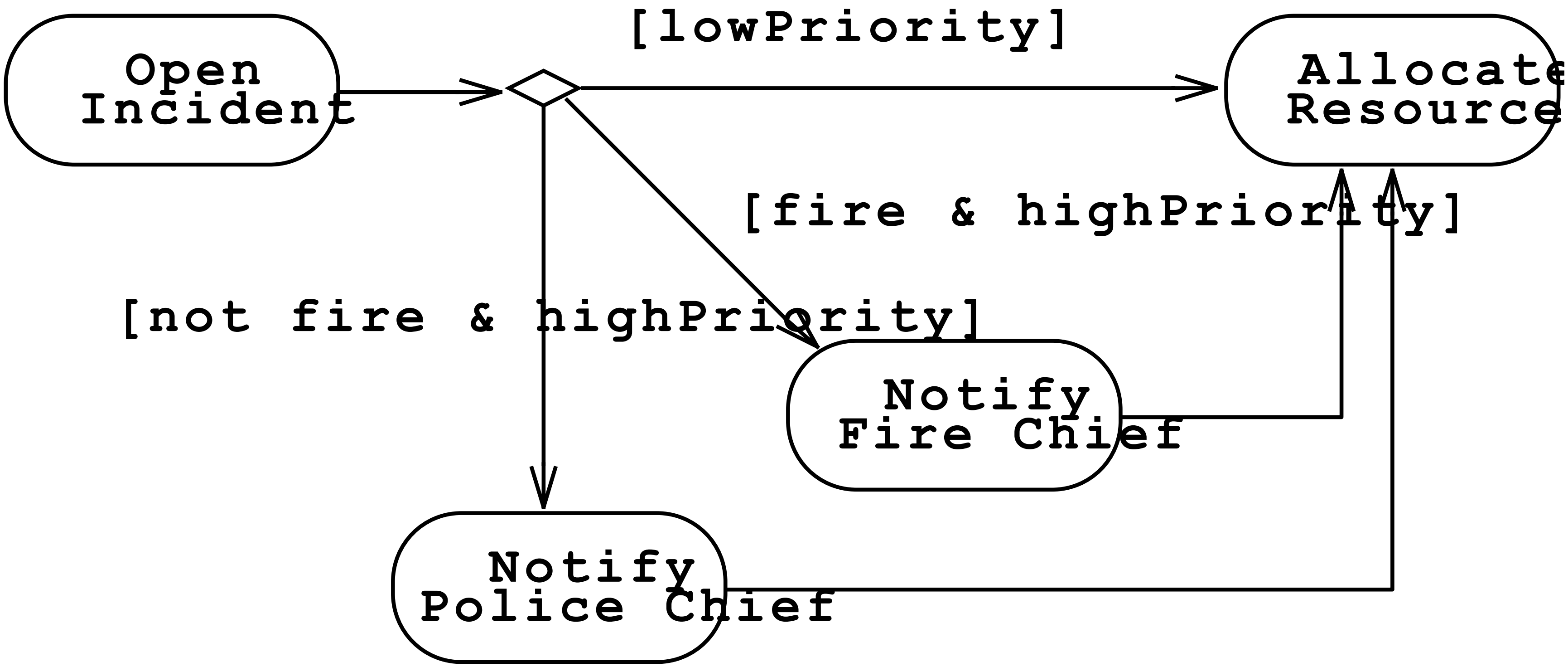
Activity Diagrams

- An activity diagram shows flow control within a system



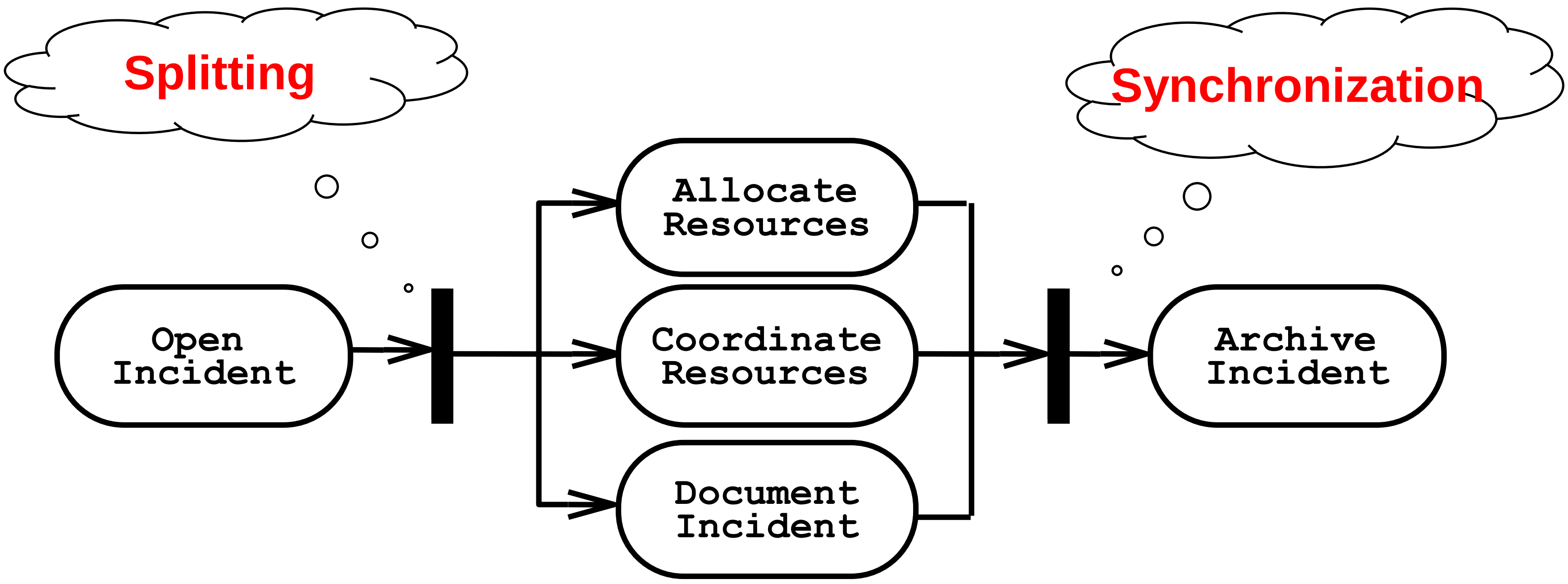
- In activity diagram, the states are activities ("functions")
- Two types of states:
  - *Action state:*
    - Cannot be decomposed any further
    - Happens "instantaneously" with respect to the level of abstraction used in the model
  - *Activity state:*
    - Can be decomposed further
    - The activity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

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.