UML IN SOFTWARE ENGG

DR TRIPTY SINGH

SHREEVIDYA

Overview

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

The Origin of UML

- The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor notations. UML has been designed for a broad range of applications. Hence, it provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design and deployment).

Why UML

- As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs. The primary goals in the design of the UML summarize by Page-Jones in Fundamental Object-Oriented Design in UML as follows:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
7. Integrate best practices.

What is UML and Why we use UML?

- Why we use UML?

➢ Use graphical notation: more clearly than natural language (imprecise) and code (too detailed).

➢ Help acquire an overall view of a system.

➢ UML is *not* dependent on any one language or technology.

➢ UML moves us from fragmentation to standardization.

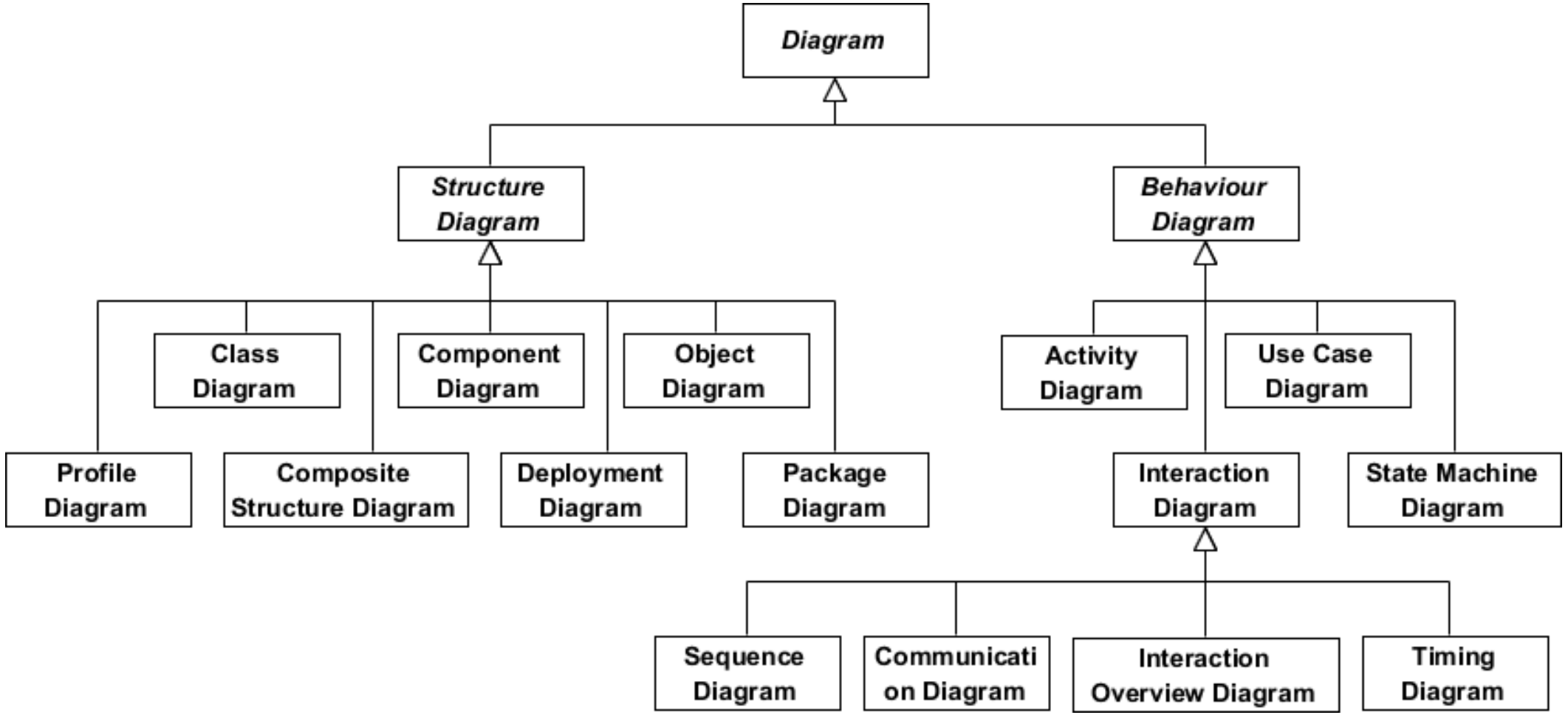
UML-An Overview

- UML - Before we begin to look at the theory of the UML, we are going to take a very brief run through some of the major concepts of the UML.
- The first thing to notice about the UML is that there are a lot of different diagrams (models) to get used to. The reason for this is that it is possible to look at a system from many different viewpoints. A software development will have many stakeholders playing a part. For Example:
- Analysts
- Designers
- Coders
- Testers
- QA
- The Customer
- Technical Authors
- All of these people are interested in different aspects of the system, and each of them require a different level of detail. For example, a coder needs to understand the design of the system and be able to convert the design to a low level code. By contrast, a technical writer is interested in the behavior of the system as a whole, and needs to understand how the product functions. The UML attempts to provide a language so expressive that all stakeholders can benefit from at least one UML diagram

UML-DIAGRAMS

- **Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts, there are seven types of structure diagram as follows:**
- Class Diagram
- Component Diagram
- Deployment Diagram
- Object Diagram
- Package Diagram
- Composite Structure Diagram
- Profile Diagram
- **Behavior diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time, there are seven types of behavior diagrams as follows:**
- Use Case Diagram
- Activity Diagram
- State Machine Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

UML-DIAGRAMS

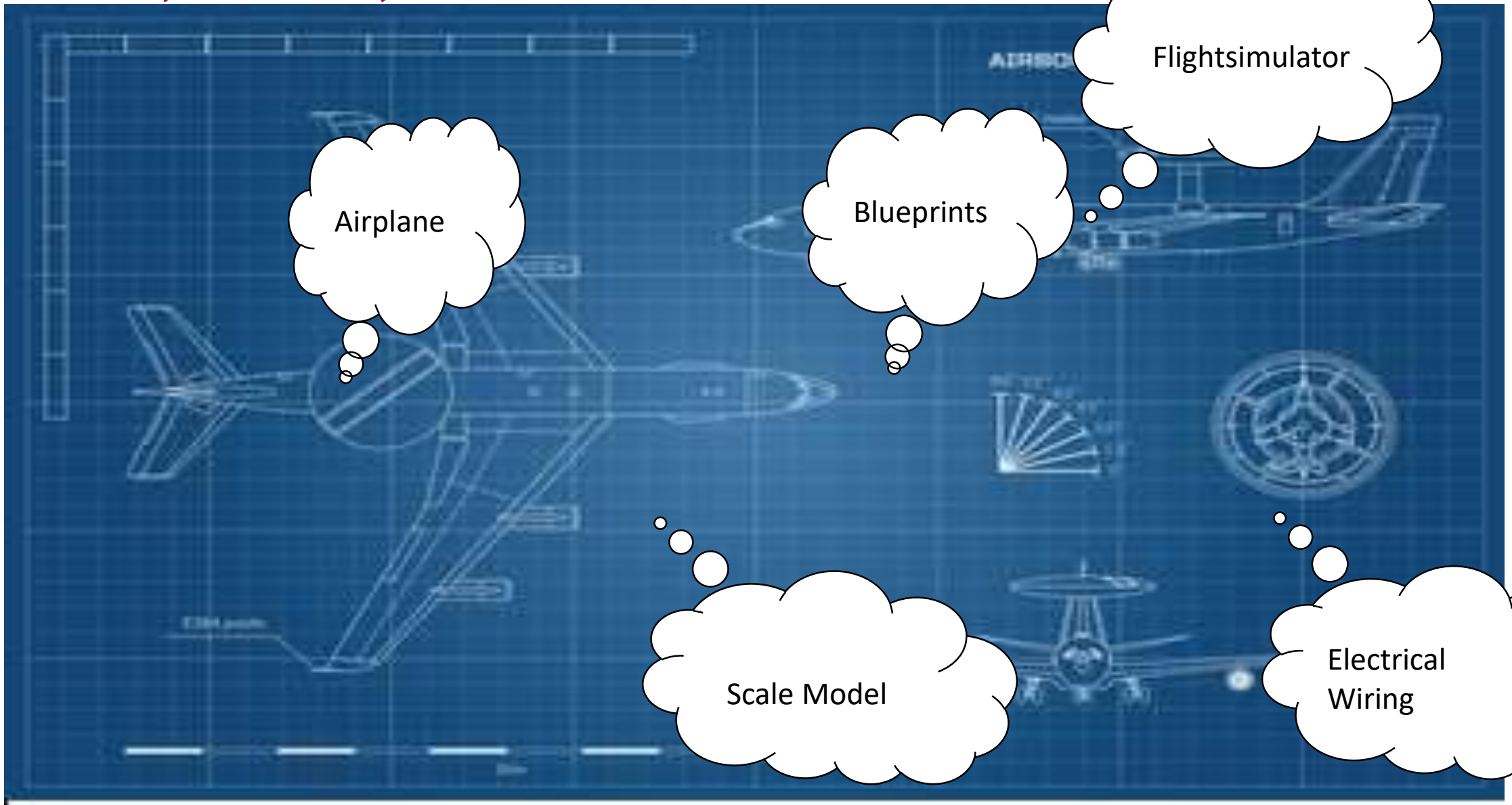


Systems, Models, and Views

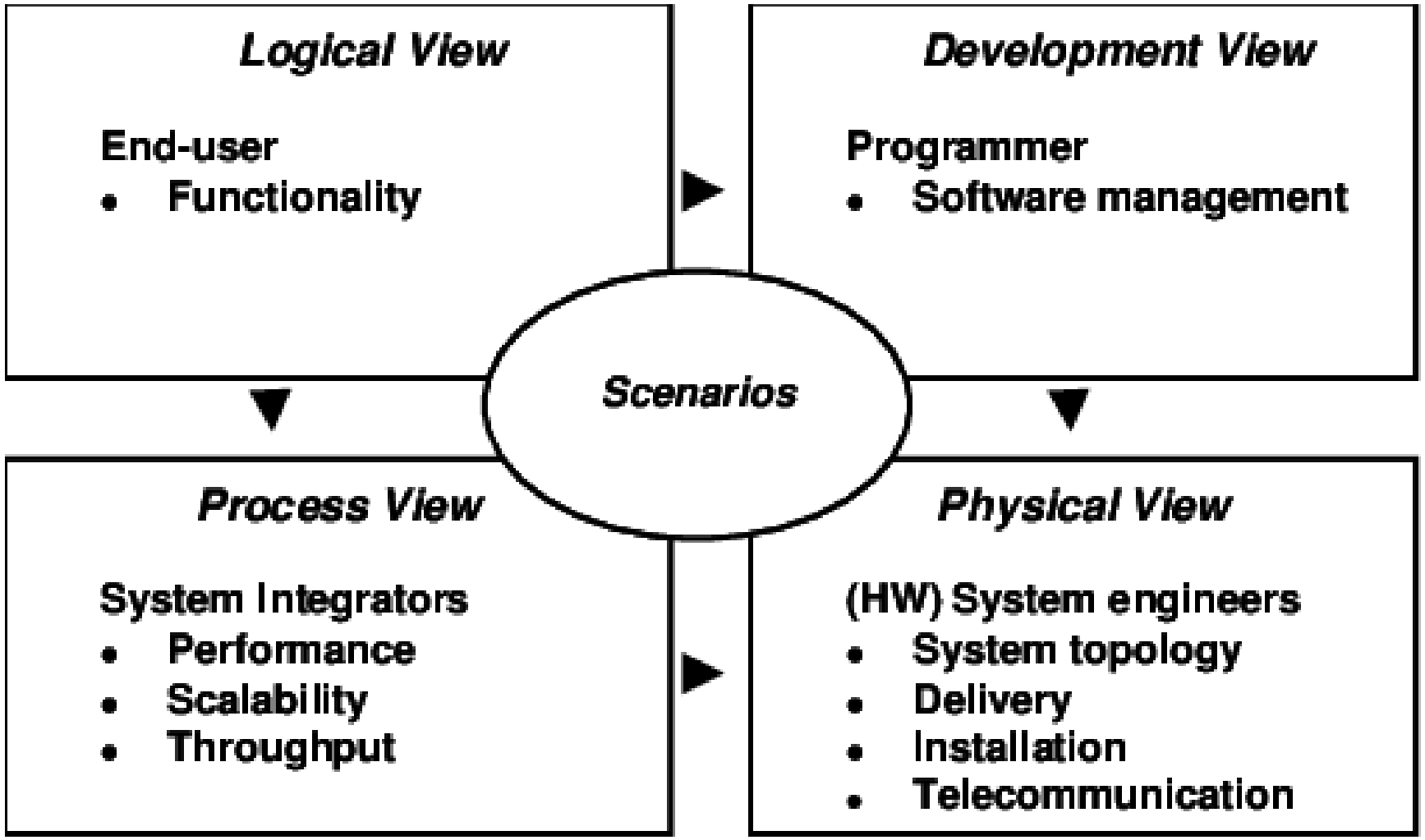
- A *model* is an abstraction describing system or a subset of a system
- A *view* depicts selected aspects of a model
- A *notation* is a set of graphical or textual rules for representing views

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

Systems, Models, and Views



Systems, Models, and Views



How to use UML diagrams to design software system?

- Types of UML Diagrams:

➢ Use Case Diagram

➢ Class Diagram

➢ Sequence Diagram

➢ Collaboration Diagram

➢ State Diagram

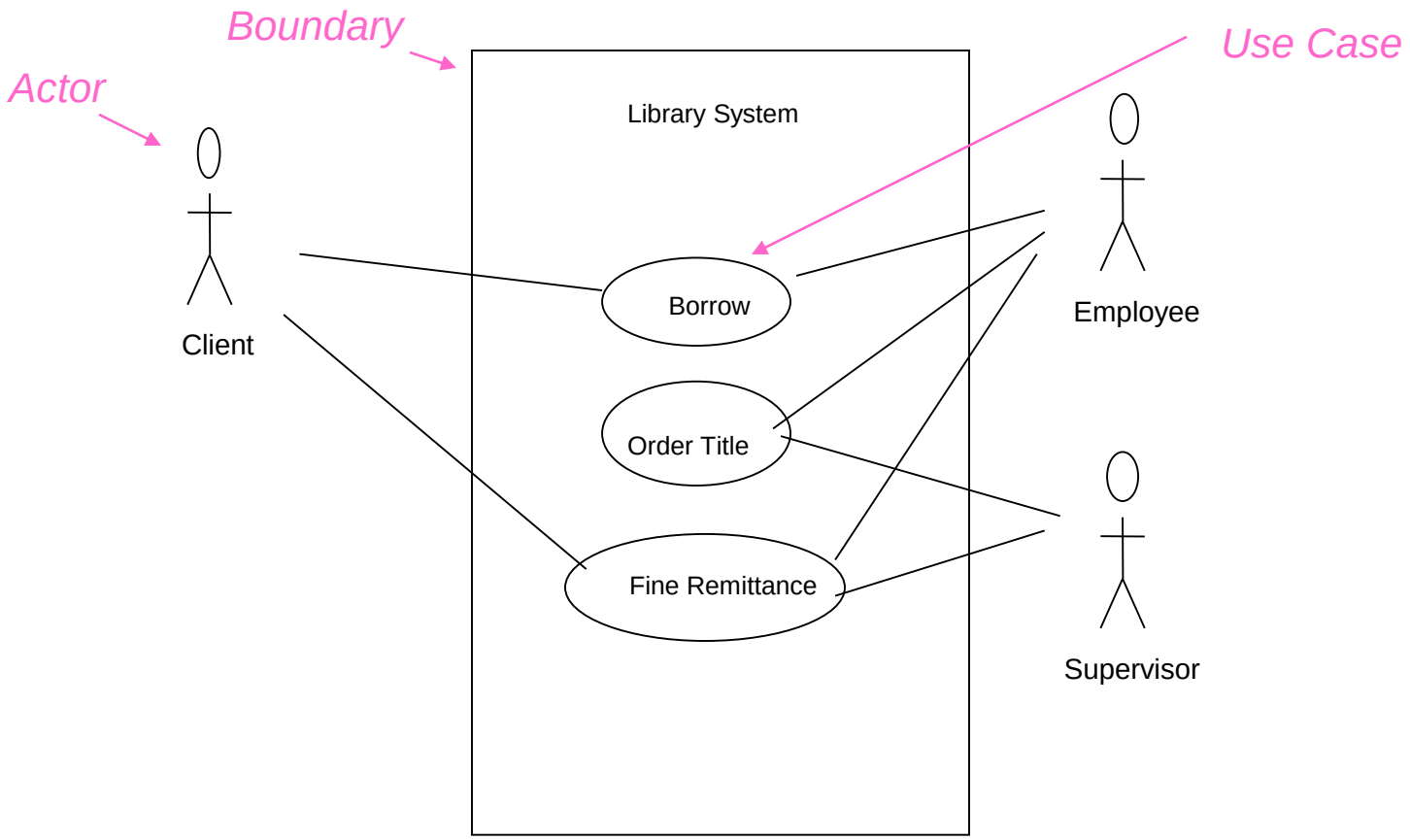
This is only a subset of diagrams … but are most widely used

Use-Case Diagrams

- A use-case diagram is a set of use cases
- A use case is a model of the interaction between
  - External users of a software product (actors) and
  - The software product itself
  - More precisely, an actor is a user playing a specific role

- describing a set of user **scenarios**
- capturing user requirements
- **contract** between end user and software developers

Use-Case Diagrams

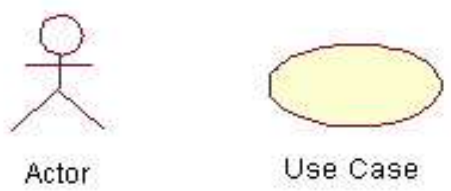


Use-Case Diagrams

- **Actors:** A role that a user plays with respect to the system, including human users and other systems. e.g., inanimate physical objects (e.g. robot); an external system that needs some information from the current system.

- **Use case:** A set of scenarios that describing an interaction between a user and a system, including alternatives.

- **System boundary**: rectangle diagram representing the boundary between the actors and the system.

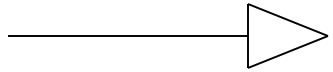


Use-Case Diagrams

- Association:

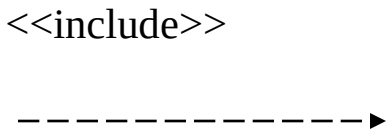
communication between an actor and a use case; Represented by a solid line.

- Generalization: relationship between one general use case and a special use case (used for defining special alternatives) Represented by a line with a triangular arrow head toward the parent use case.

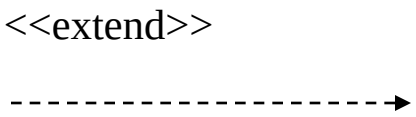


Use-Case Diagrams

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use "include" in stead of copying the description of that behavior.



Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares "extension points".



Use-Case Diagrams

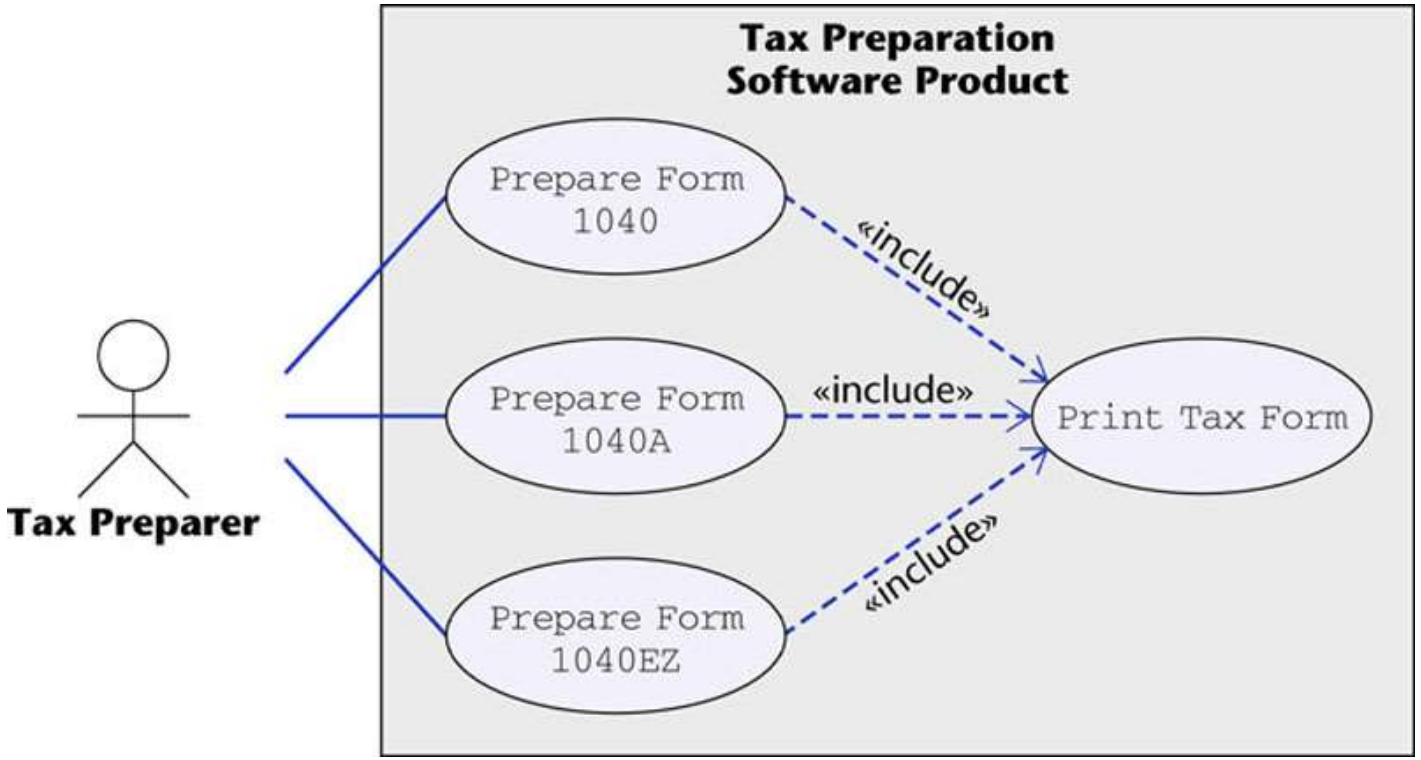
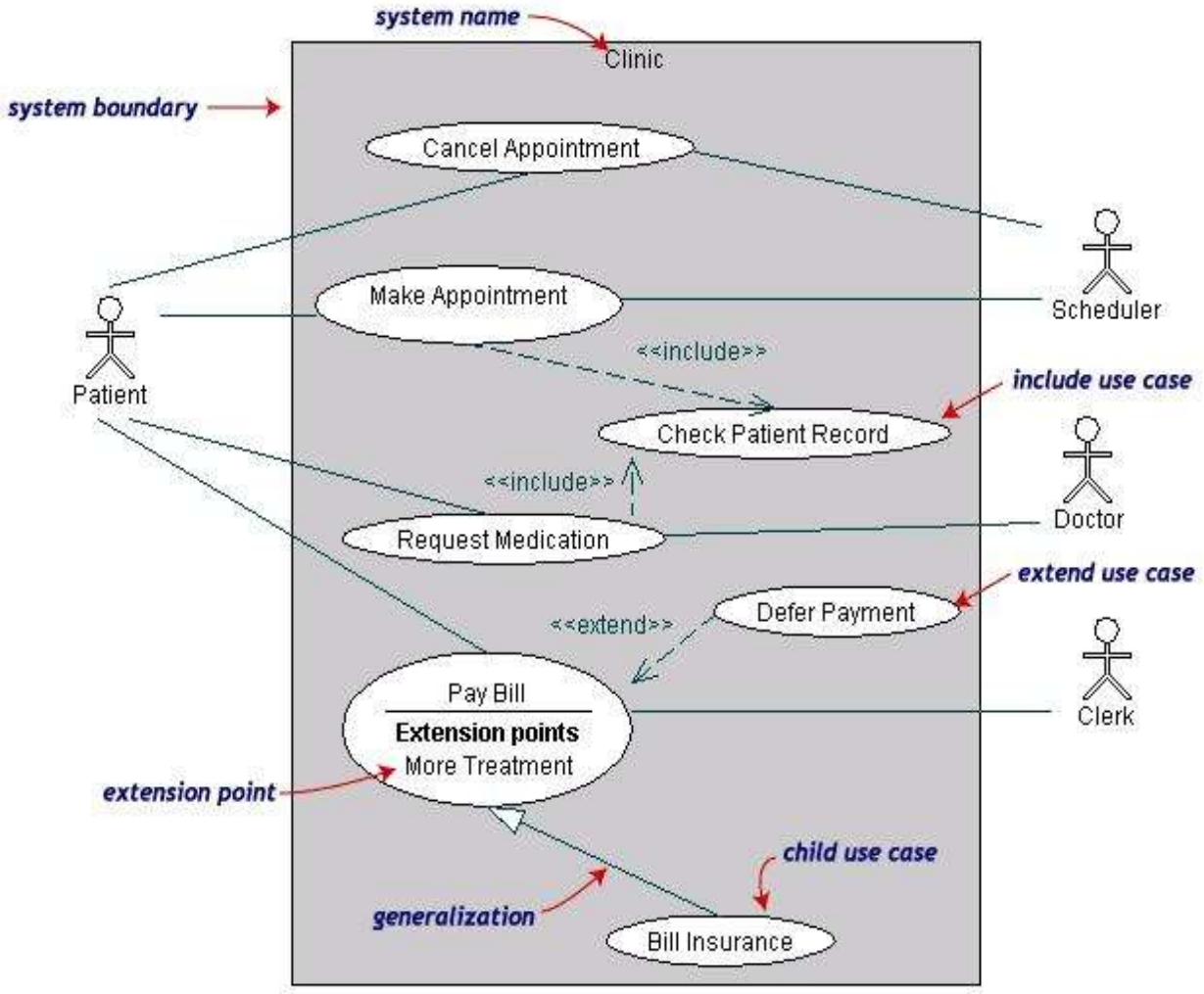


Figure 16.12

Use-Case Diagrams

- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask (include)
- The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)
- **Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)



(TogetherSoft, Inc)

Class diagram

- A class diagram depicts classes and their interrelationships
- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

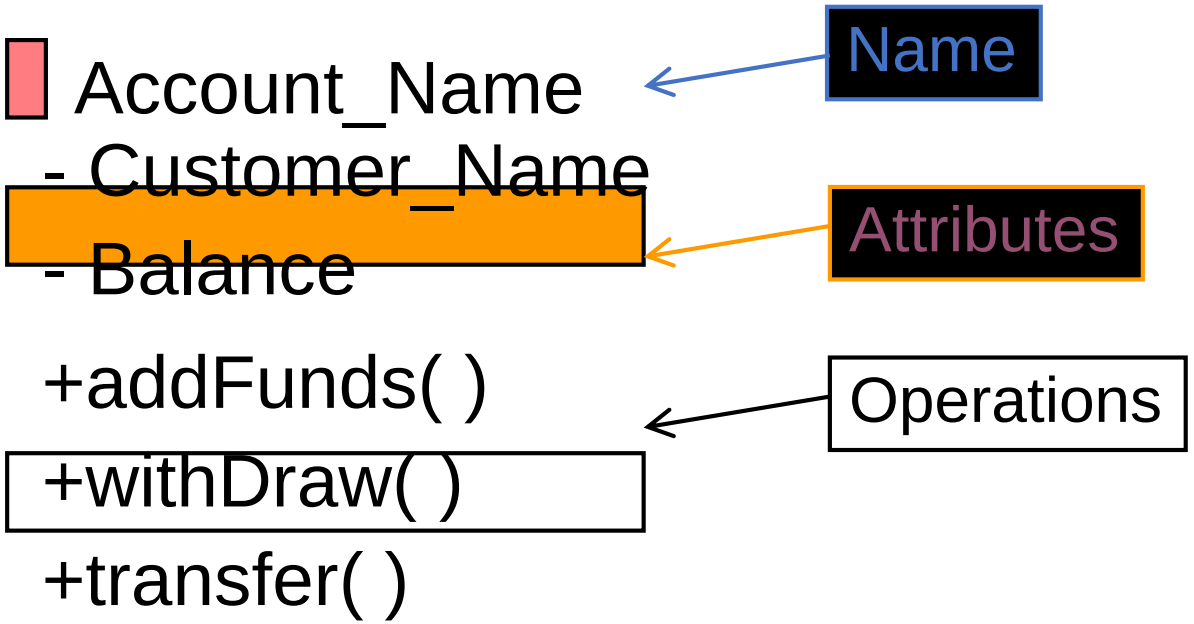
Class diagram

- Each class is represented by a rectangle subdivided into three compartments
  - Name
  - Attributes
  - Operations

- Modifiers are used to indicate visibility of attributes and operations.
  - '+' is used to denote *Public* visibility (everyone)
  - '#' is used to denote *Protected* visibility (friends and derived)
  - '-' is used to denote *Private* visibility (no one)

- By default, attributes are hidden and operations are visible.

Class diagram

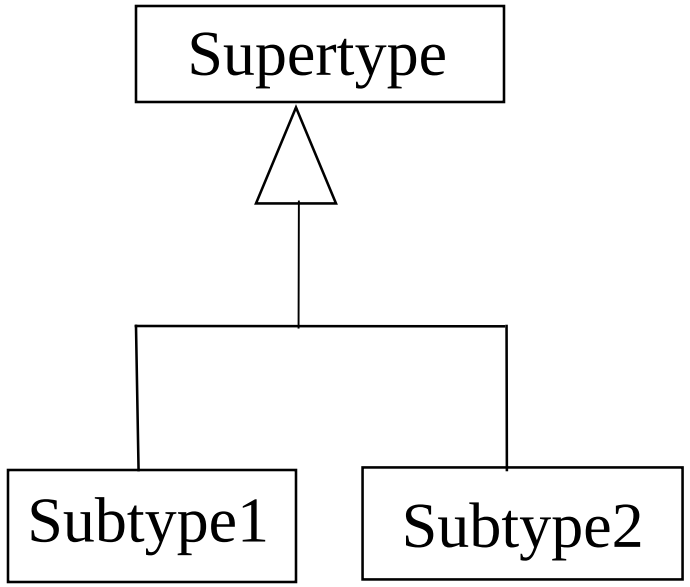


OO Relationships

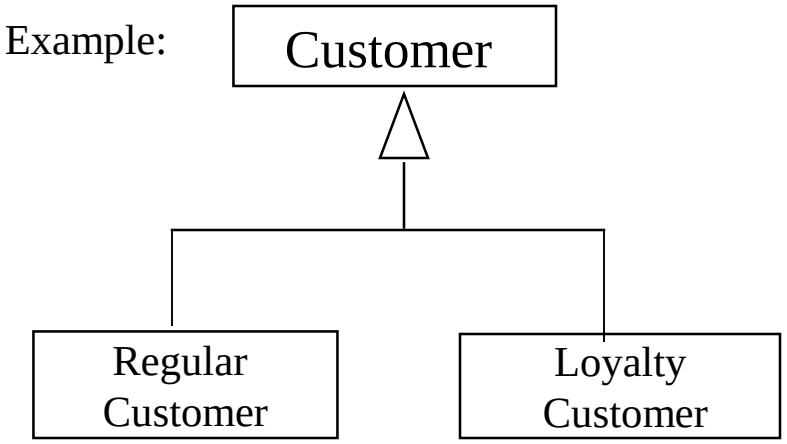
- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (student enrolls in course)

- Associations can be further classified as
  - Aggregation
  - Composition

OO Relationships: **Generalization**



Example:



-Inheritance is a required feature of object orientation

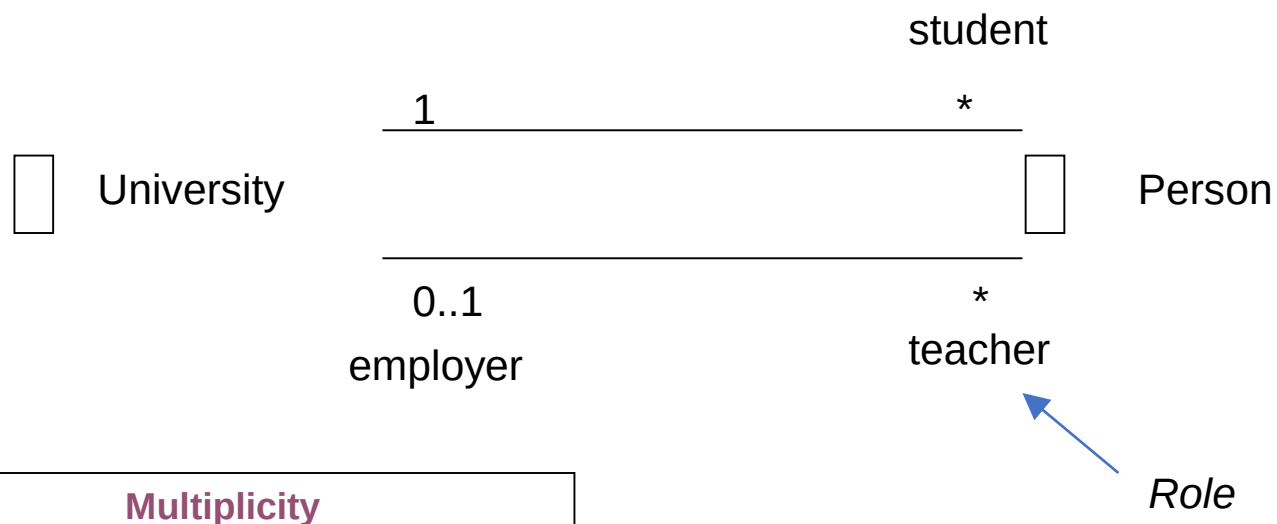
-Generalization expresses a parent/child relationship among related classes.

-Used for abstracting details in several layers

OO Relationships: **Association**

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.
- Association has two ends
  - Role names (e.g. enrolls)
  - Multiplicity (e.g. One course can have many students)
  - Navigability (unidirectional, bidirectional)

Association: Multiplicity and Roles



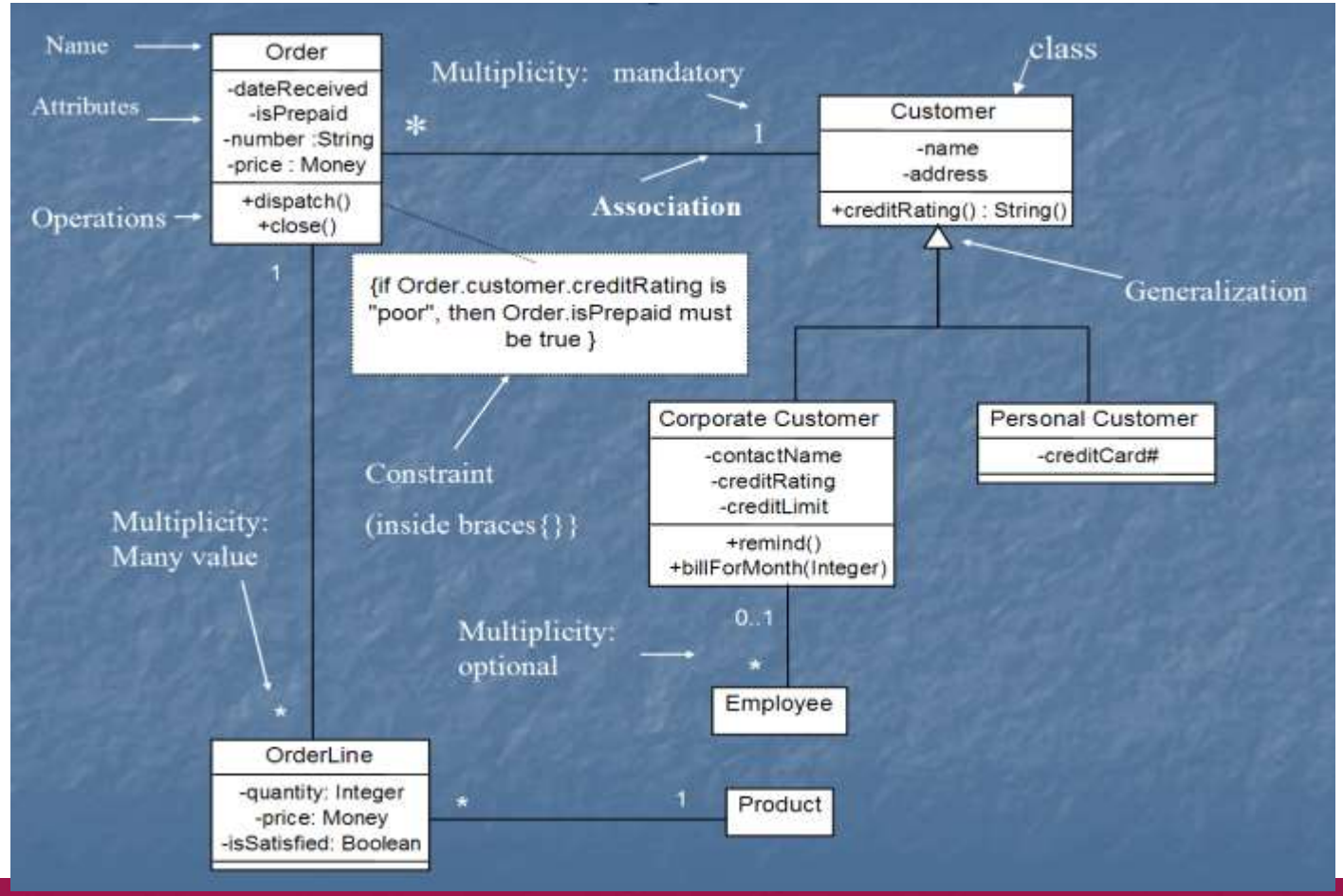
Multiplicity

Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

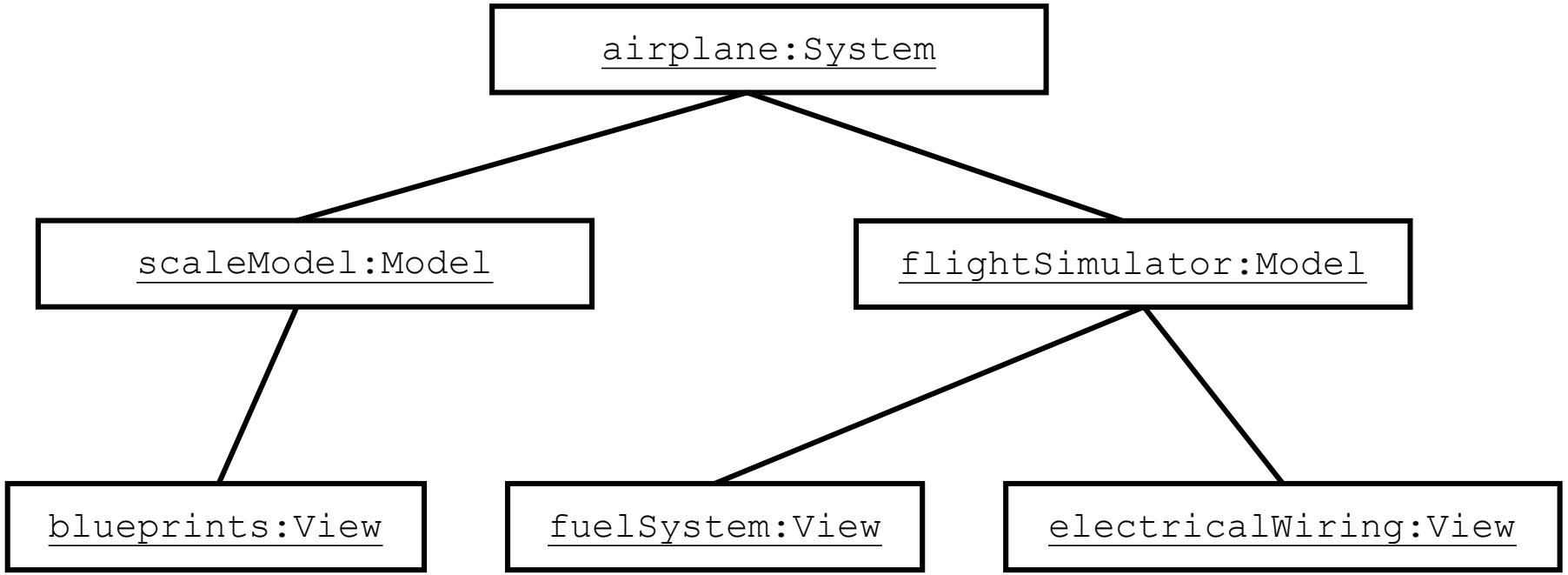
"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

Class diagram



[from *UML Distilled Third Edition*]

Models, Views, and Systems (UML)

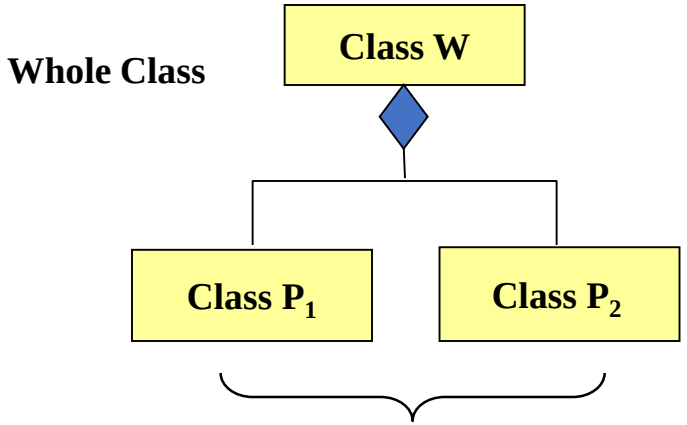


Association: Model to Implementation


```
Class Student {
   Course enrolls[4];
}

Class Course {
  Student have[];
}
```

OO Relationships: **Composition**



[From Dr.David A. Workman]

Example

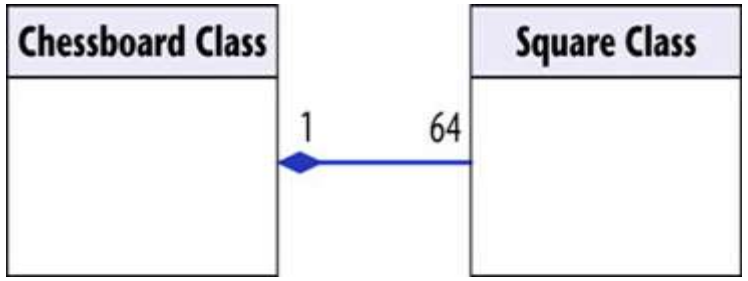


Figure 16.7

Association

Models the part–whole relationship

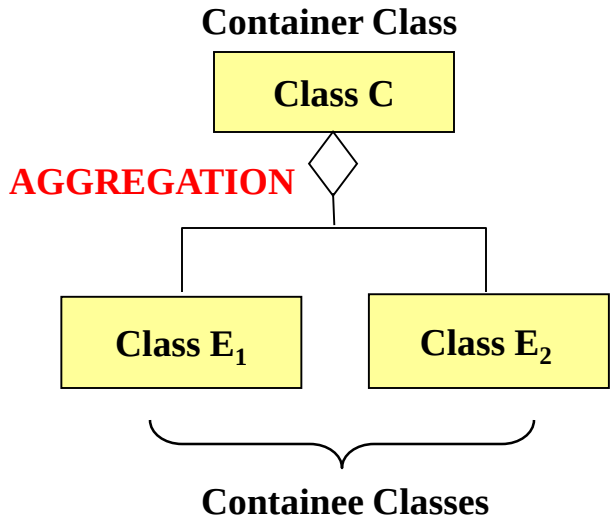
Composition

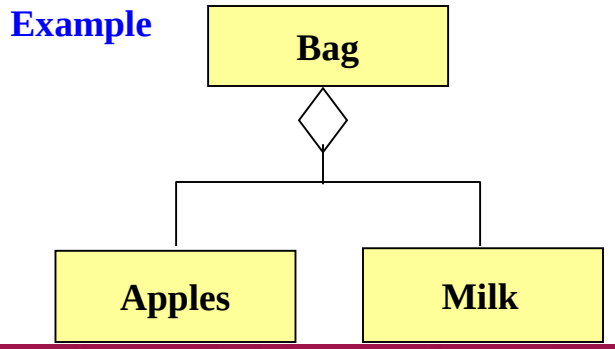
Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

Example:

A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

OO Relationships: **Aggregation**





Aggregation:
expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

express a more informal relationship than composition expresses.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

[From Dr.David A. Workman]

Aggregation vs. Composition

- **Composition** is really a strong form of **association**
  - components have only one owner
  - components cannot exist independent of their owner
  - components live or die with their owner
  - e.g. Each car has an engine that can not be shared with other cars.

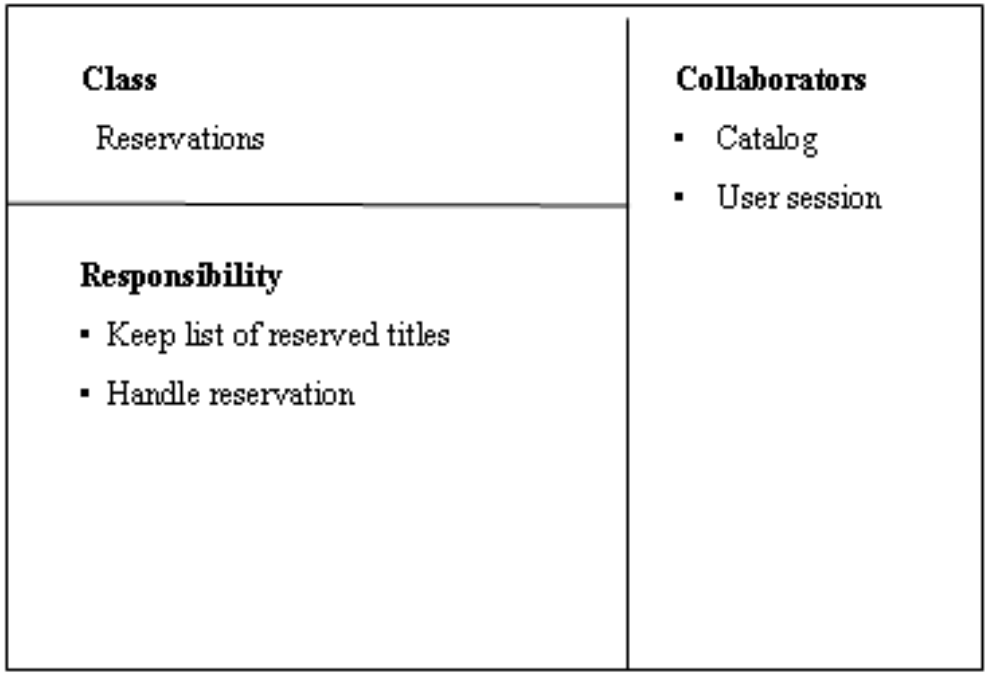
- **Aggregations**

may form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate. e.g. Apples may exist independent of the bag.

Good Practice: CRC Card

Class Responsibility Collaborator

- easy to describe how classes work by moving cards around; allows to quickly consider alternatives.

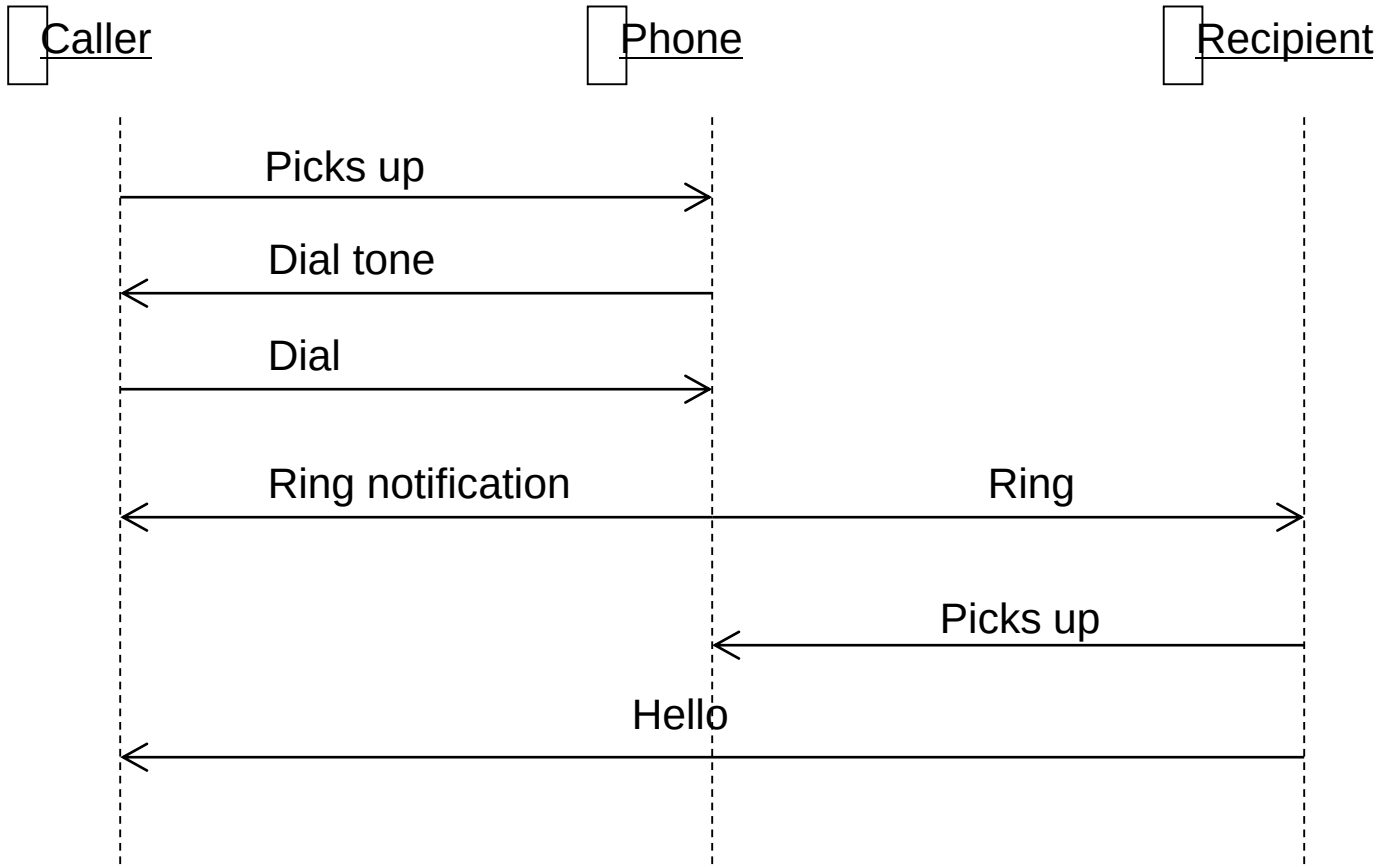


Interaction Diagrams

- show how objects interact with one another

- UML supports two types of interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams

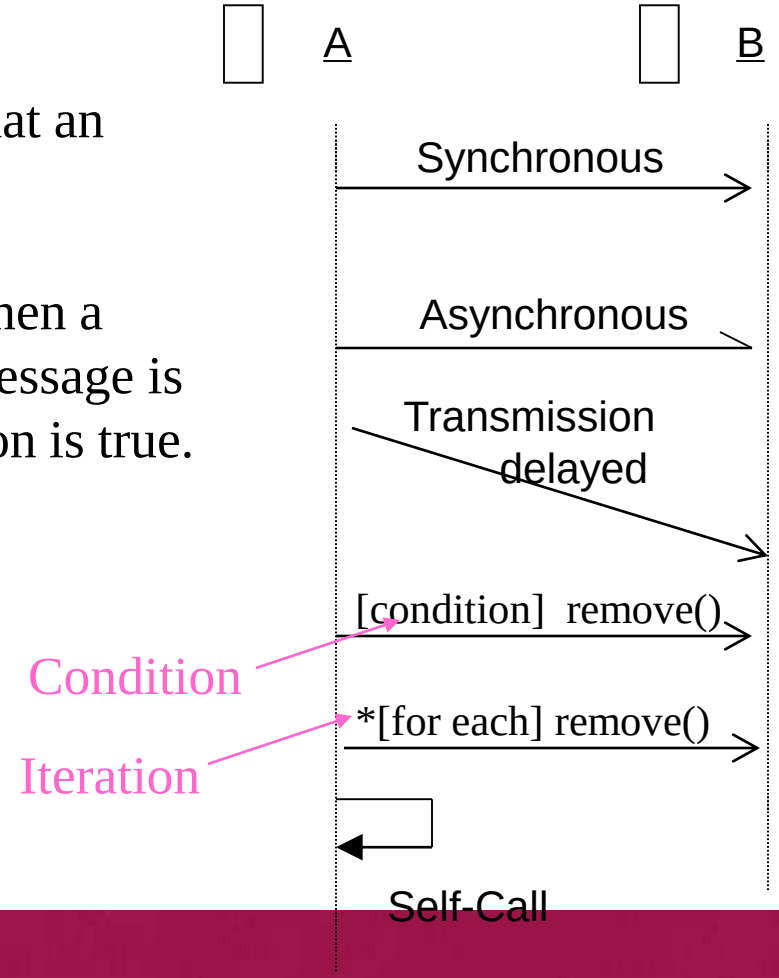
Sequence Diagram(make a phone call)



Sequence Diagram:Object interaction

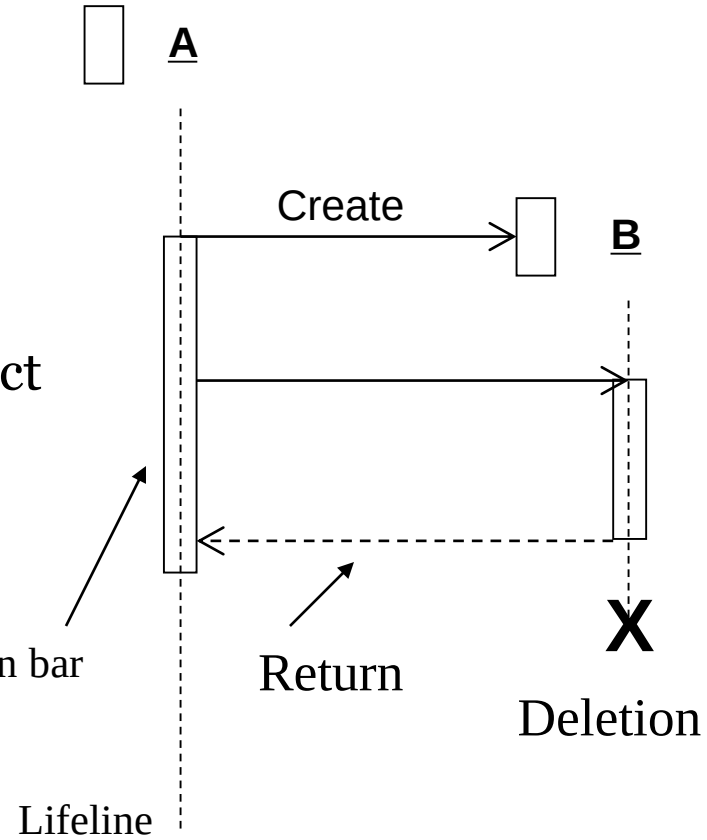
Self-Call: A message that an Object sends to itself.

Condition: indicates when a message is sent. The message is sent only if the condition is true.



Sequence Diagrams – Object Life Spans

- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an 'X' on lifeline
  - Object's life ends at that point



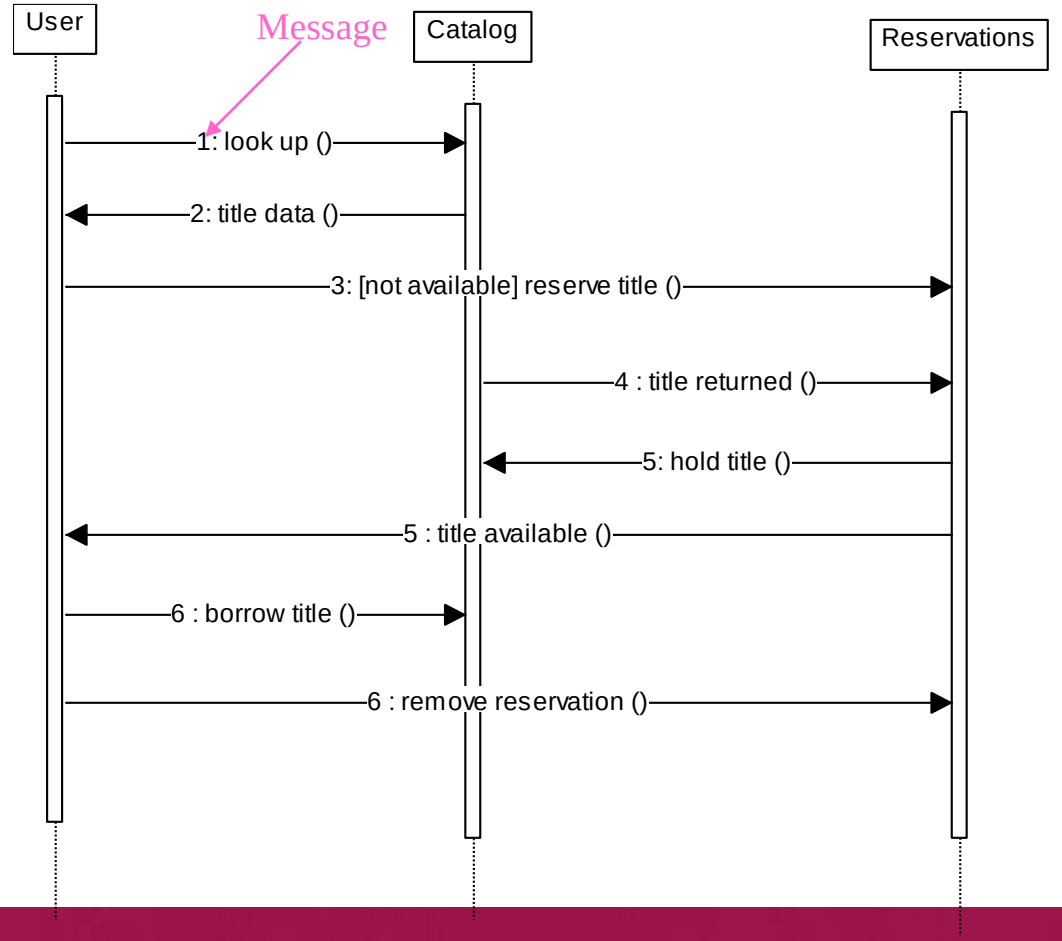
Sequence Diagram

•Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.

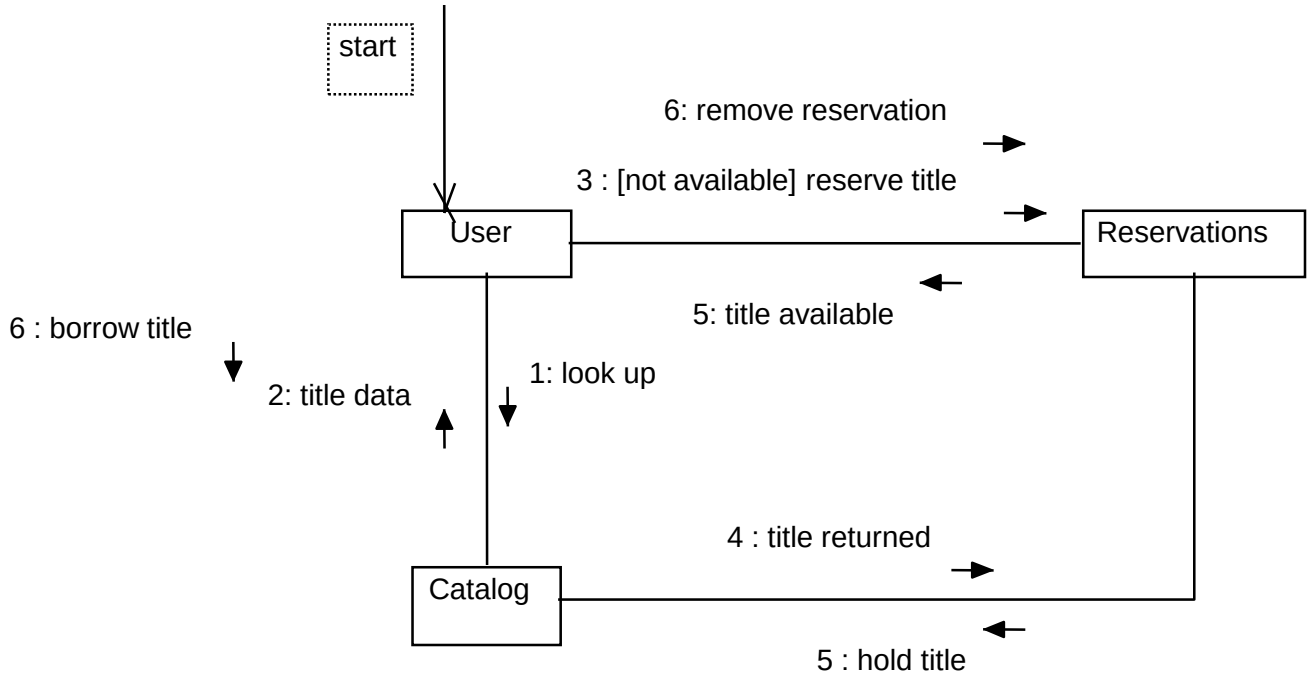
•The horizontal dimension shows the objects participating in the interaction.

•The vertical arrangement of messages indicates their order.

•The labels may contain the seq. # to indicate concurrency.



Interaction Diagrams: Collaboration diagrams



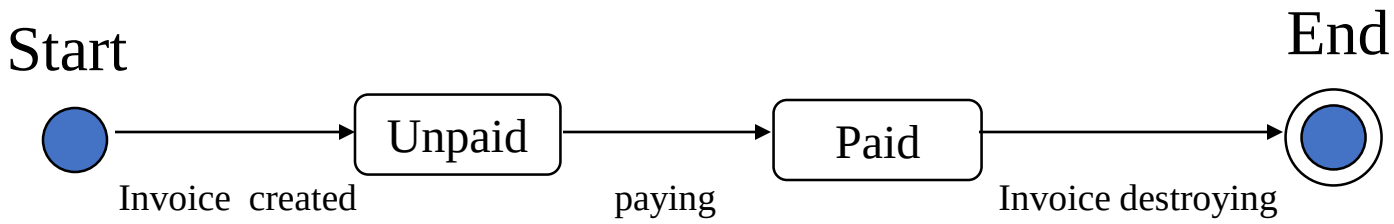
- Collaboration diagrams are equivalent to sequence diagrams. All the features of sequence diagrams are equally applicable to collaboration diagrams

- Use a sequence diagram when the transfer of information is the focus of attention

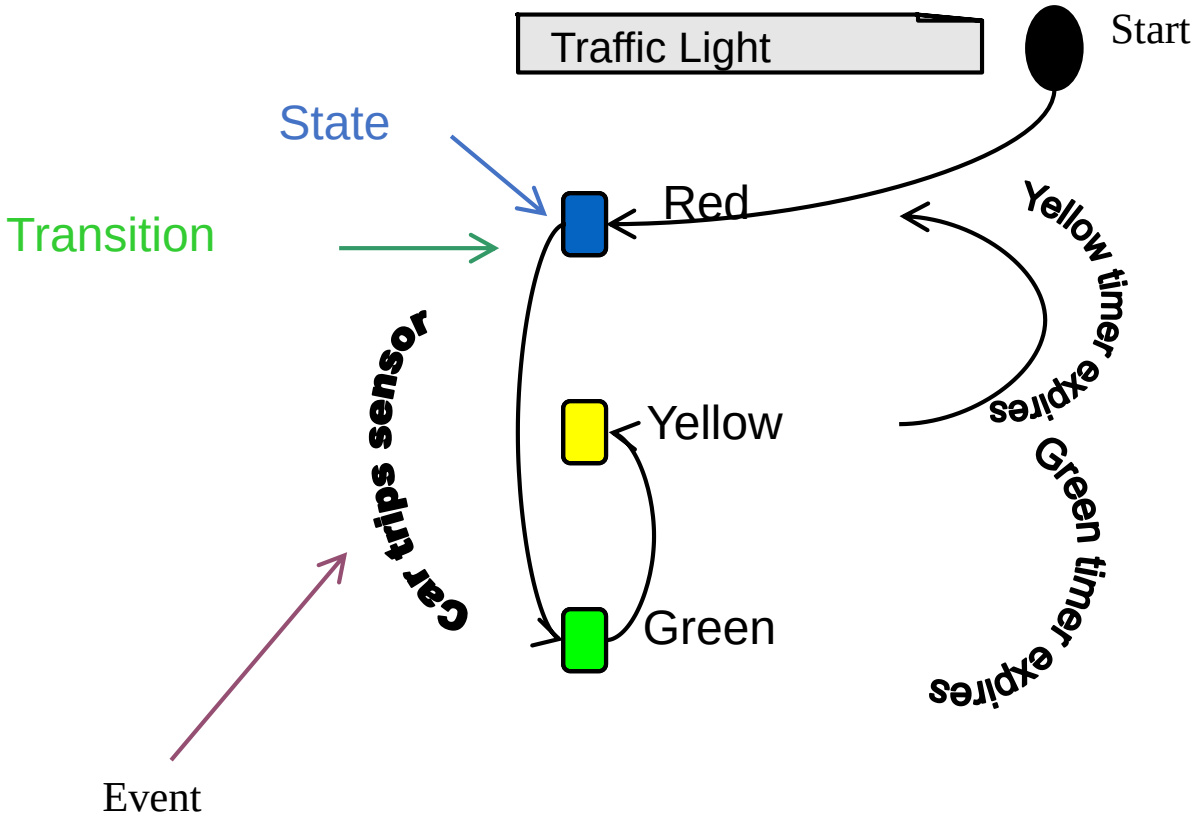
- Use a collaboration diagram when concentrating on the classes

State Diagrams (Billing Example)

State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.



State Diagrams (Traffic light example)



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

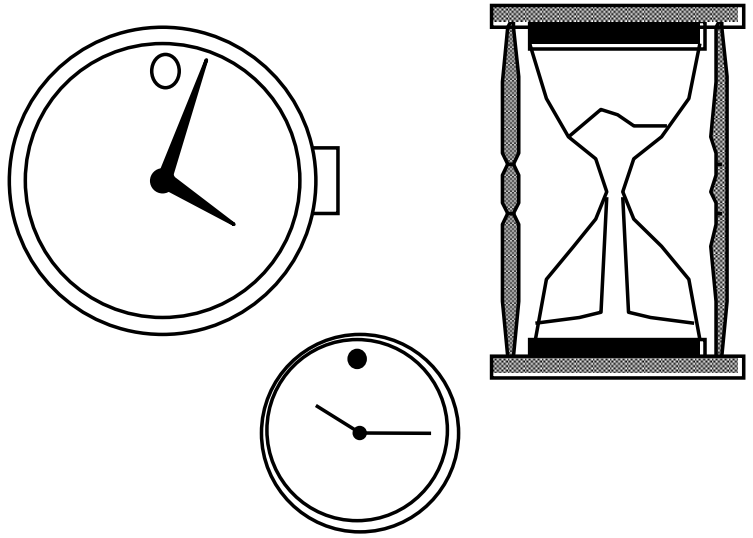
Name

Clock

Purpose

A device that measures time.

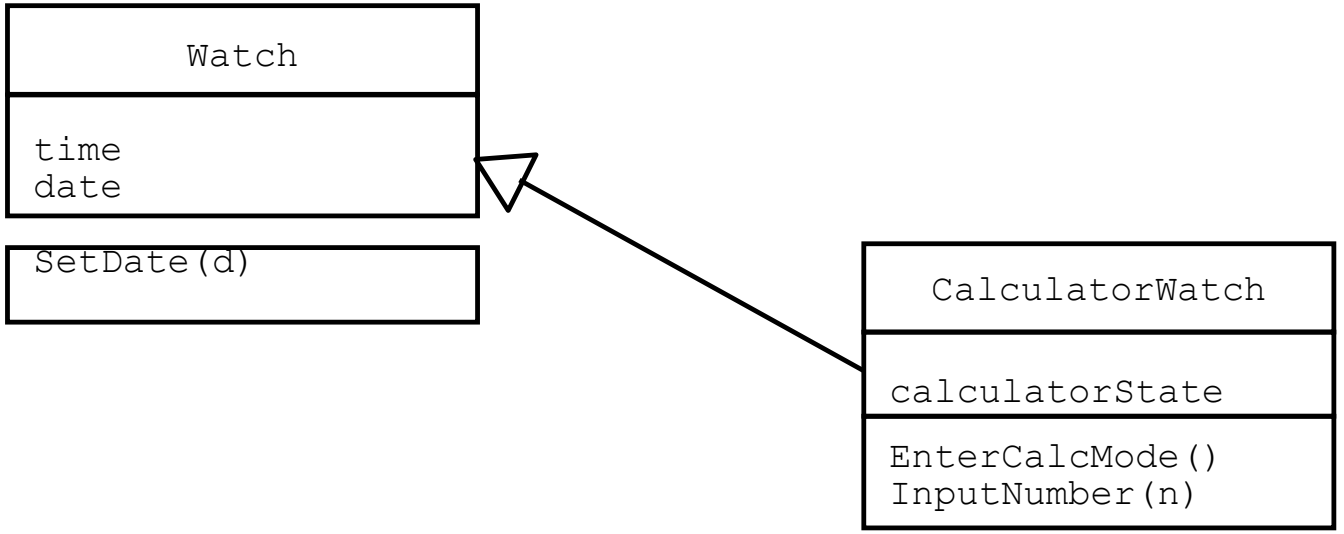
Members



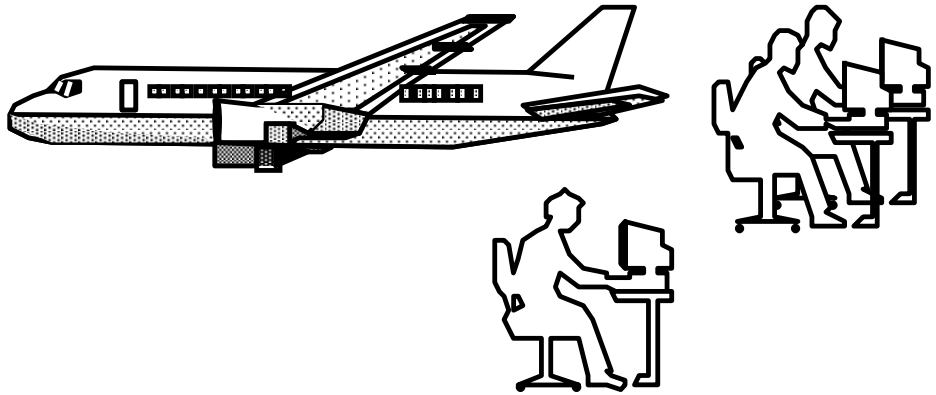
- Abstraction: Classification of phenomena into concepts
- Modeling: Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

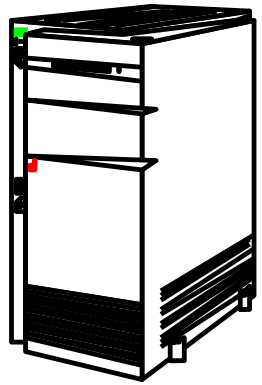
Class

- 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



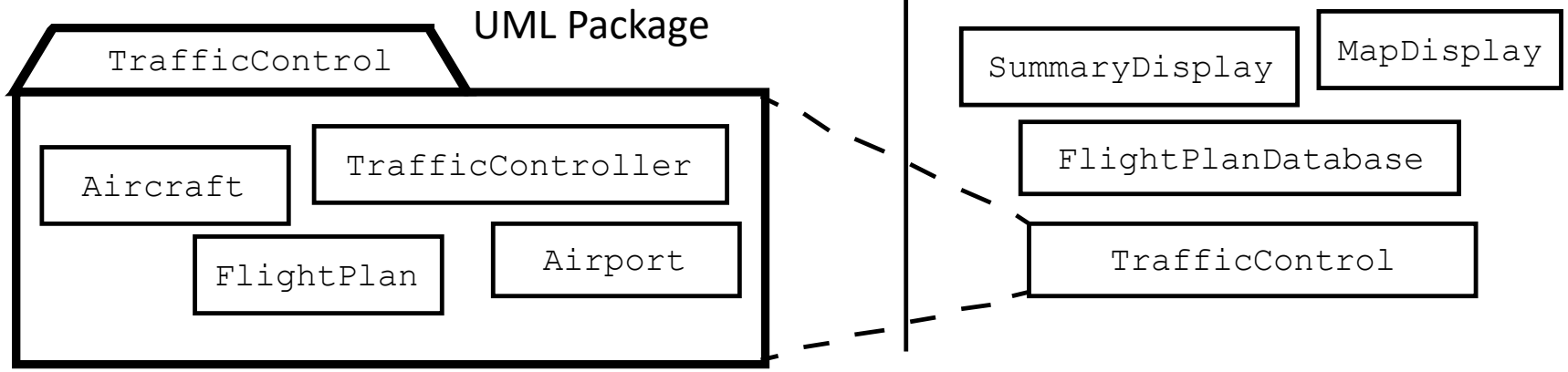


Application Domain

Application Domain Model

Solution Domain

System Model



Application and Solution Domain

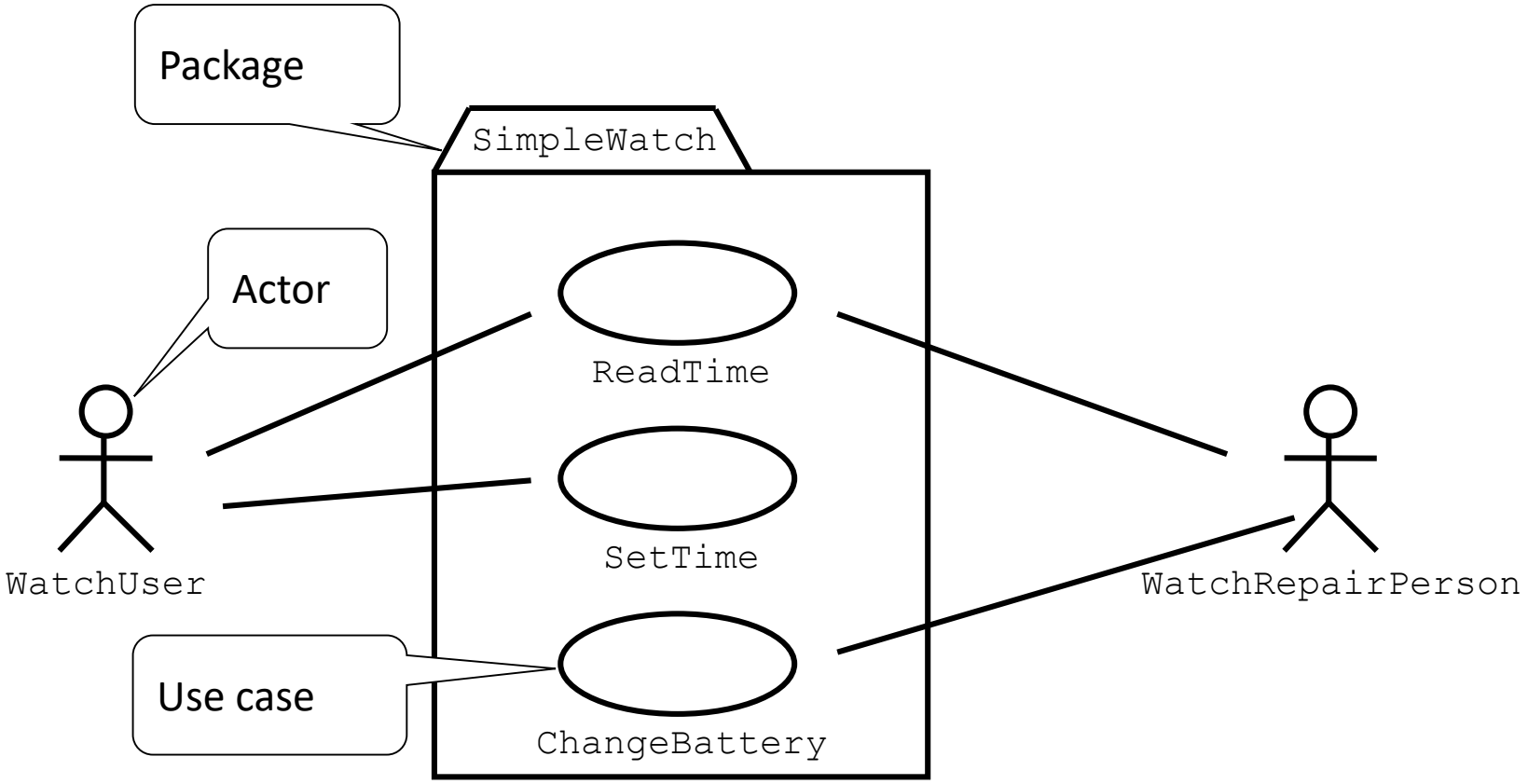
- Application Domain (Requirements Analysis):
  - The environment in which the system is operating

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

UML First Pass

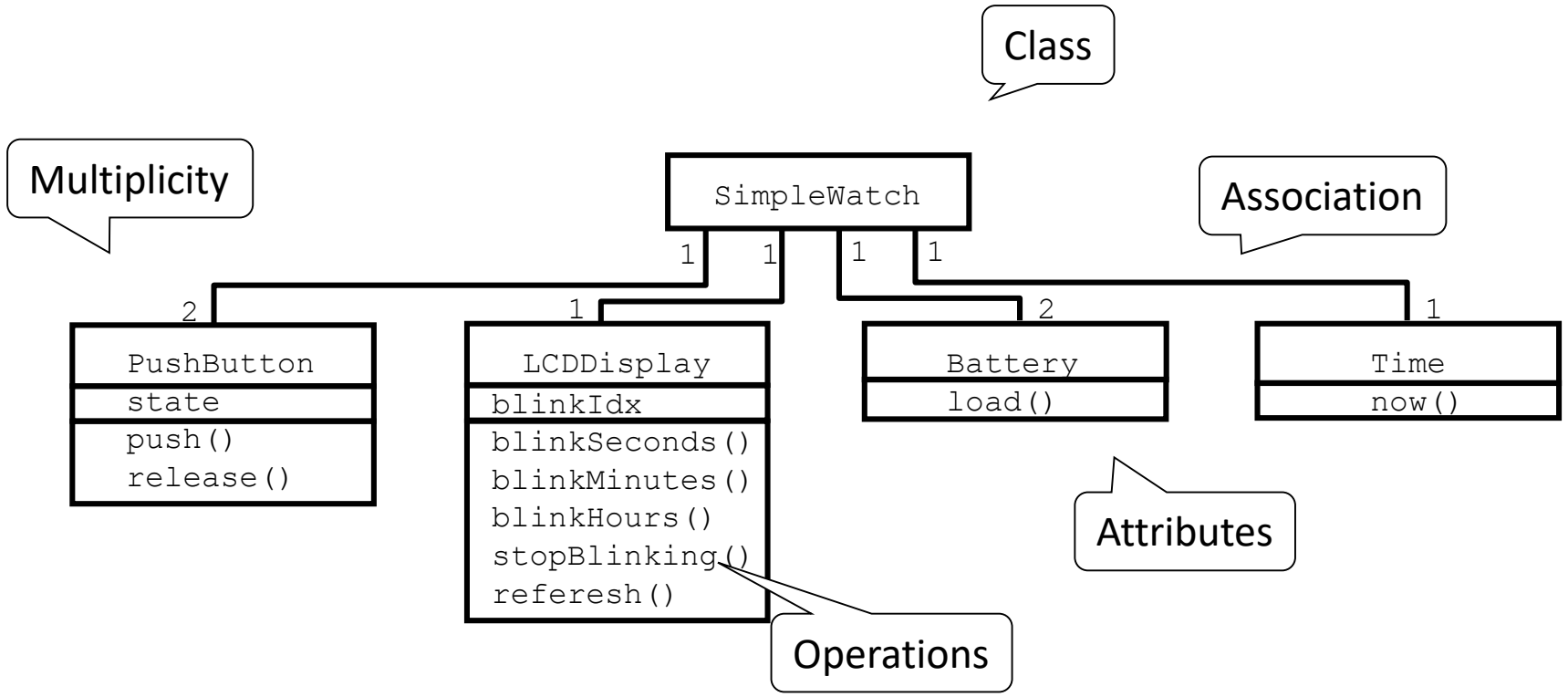
- Use case diagrams
  - Describe the functional behavior of the system as seen by the user.
- Class diagrams
  - Describe the static structure of the system: Objects, Attributes, and Associations.
- Sequence diagrams
  - Describe the dynamic behavior between actors and the system and between objects 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.

UML First Pass: Use Case Diagrams



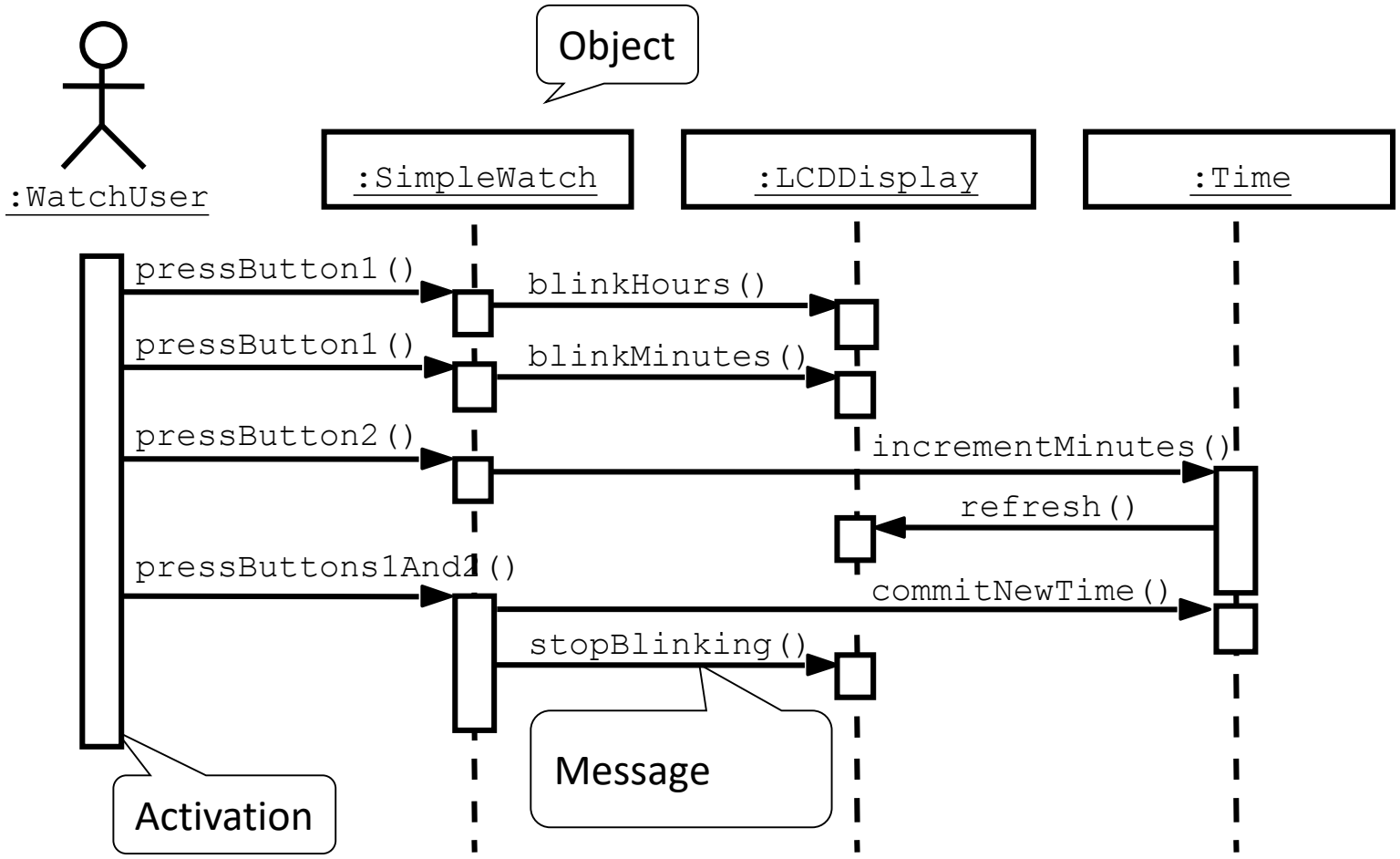
Use case diagrams represent the functionality of the system from user's point of view

UML First Pass: Class Diagrams



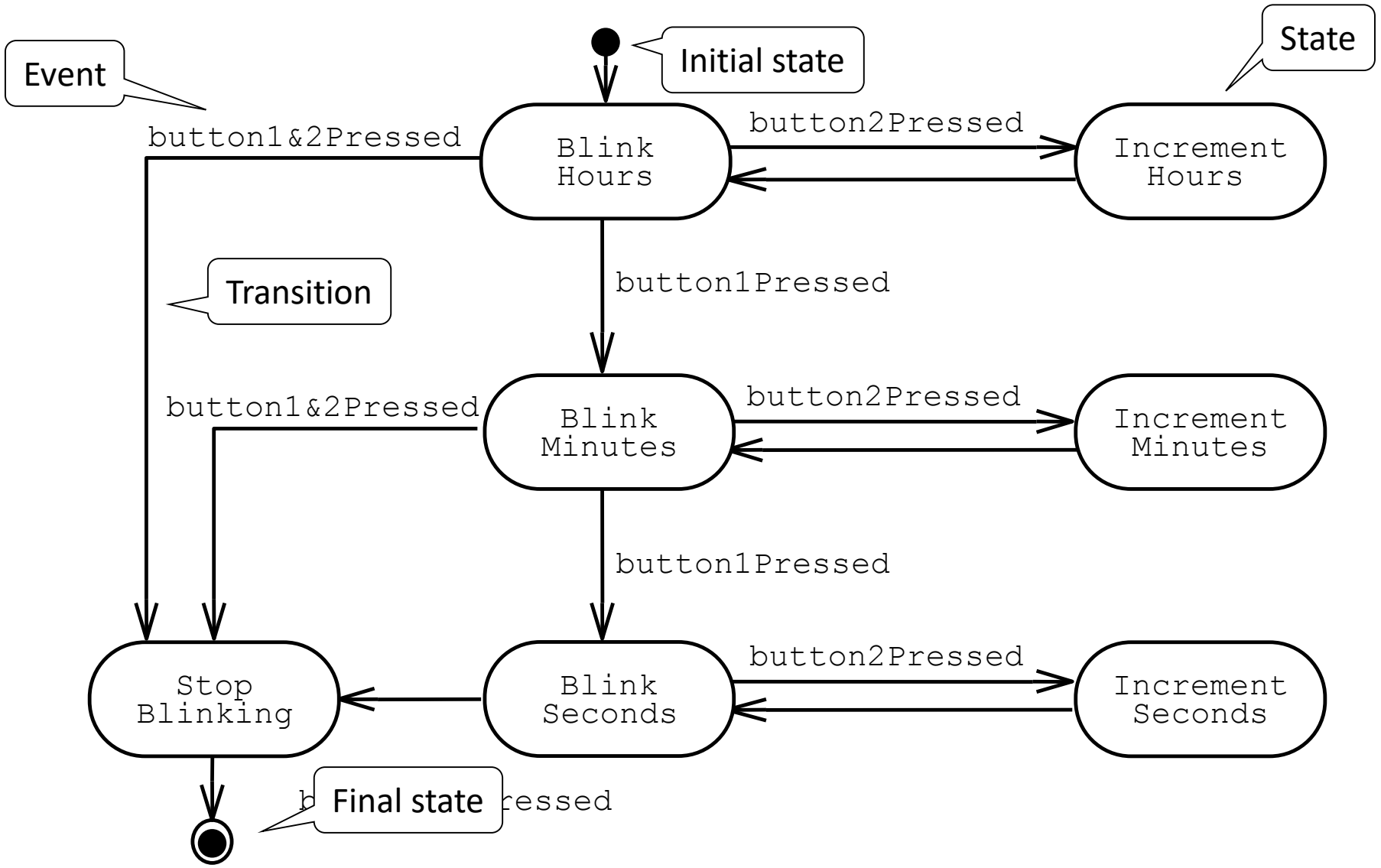
Class diagrams represent the structure of the system

UML First Pass: Sequence Diagram



Sequence diagrams represent the behavior as interactions

UML First Pass: Statechart Diagrams



Other UML Notations

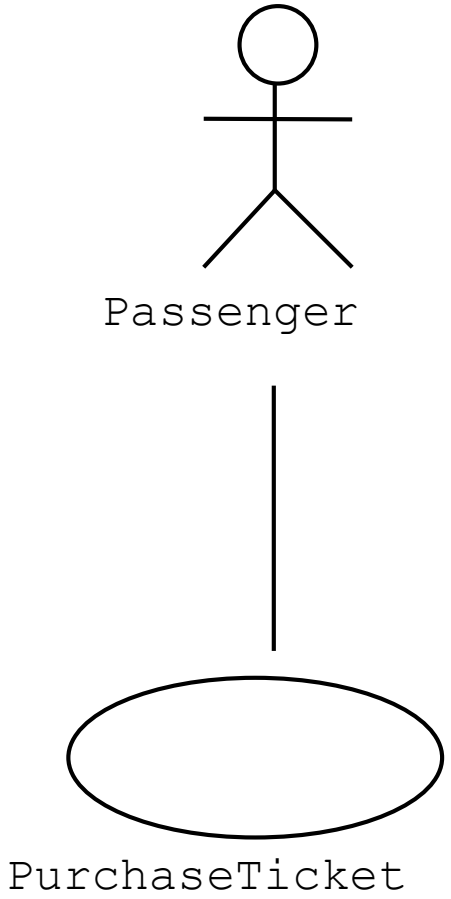
UML provide other notations that we will be introduced in subsequent lectures, as needed.

- Implementation diagrams
  - Component diagrams
  - Deployment diagrams
  - Introduced in lecture on System Design
- Object Constraint Language (OCL)
  - 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 nonunderlined names
  - `SimpleWatch`
  - `Firefighter`
- Diagrams are graphs
  - Nodes are entities
  - Arcs are relationships between entities

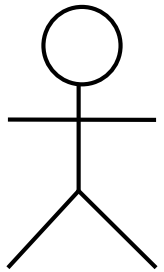
UML Second Pass: Use Case Diagrams



Used during requirements elicitation to represent external behavior

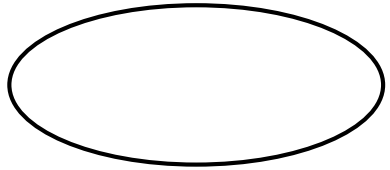
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality
- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

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



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.

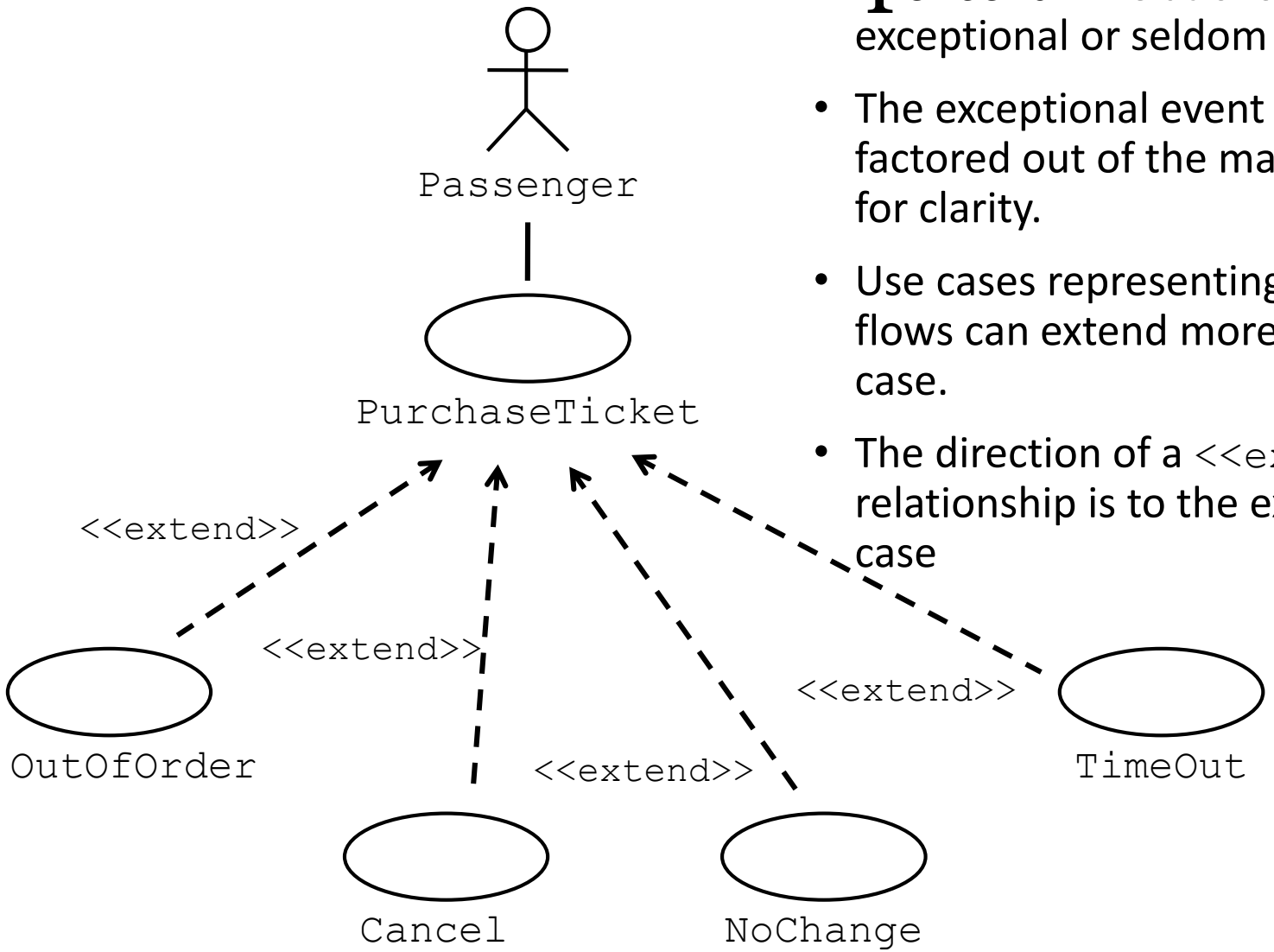
Event flow:

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?

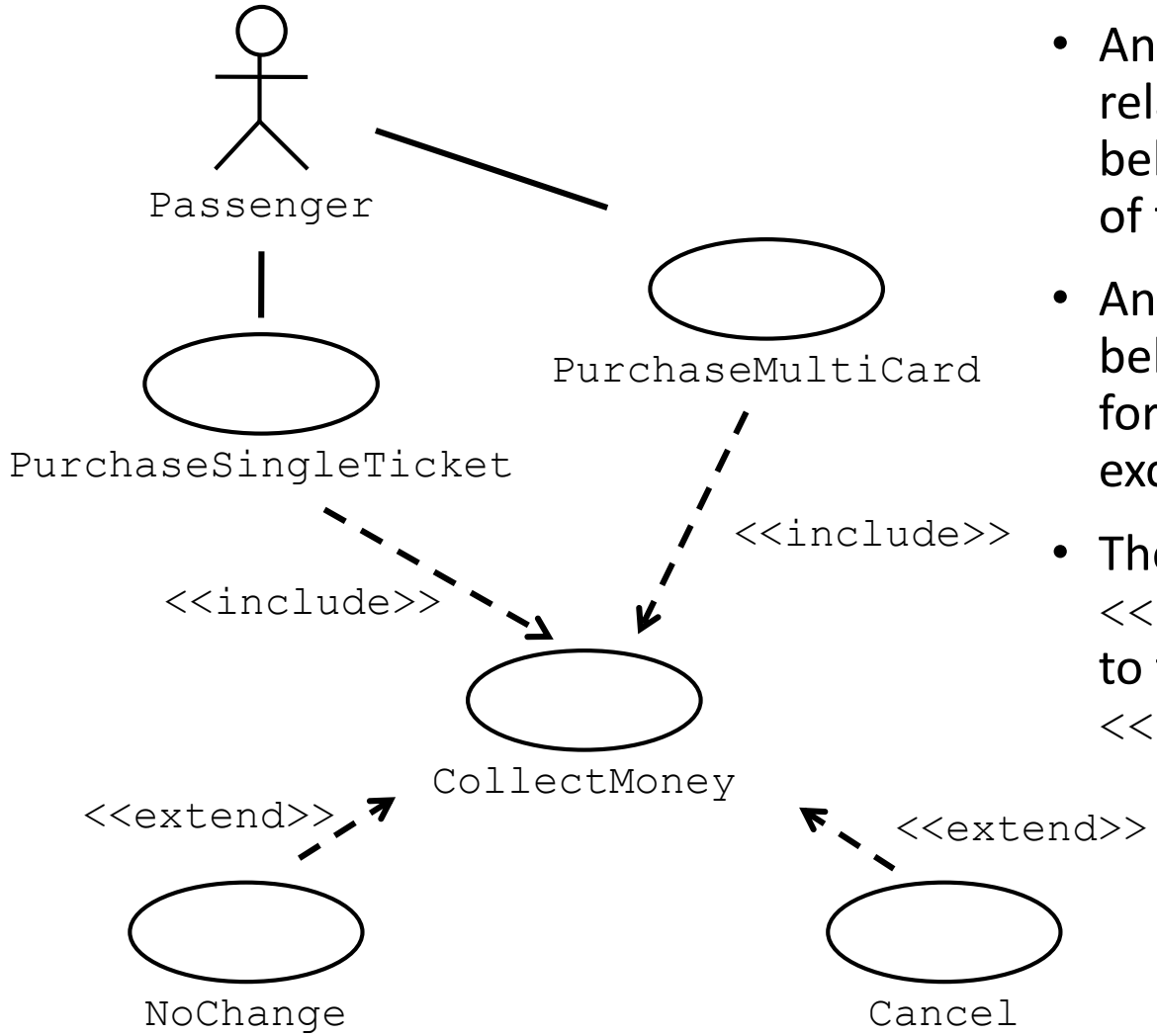
Exceptional cases!

The <<extend>> Relationship



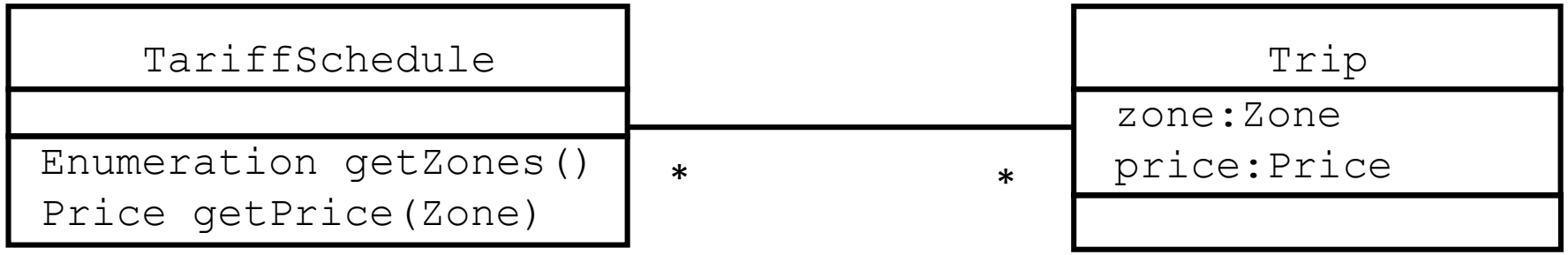
- <<extend>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extend>> relationship is to the extended use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

The <<include>> Relationship



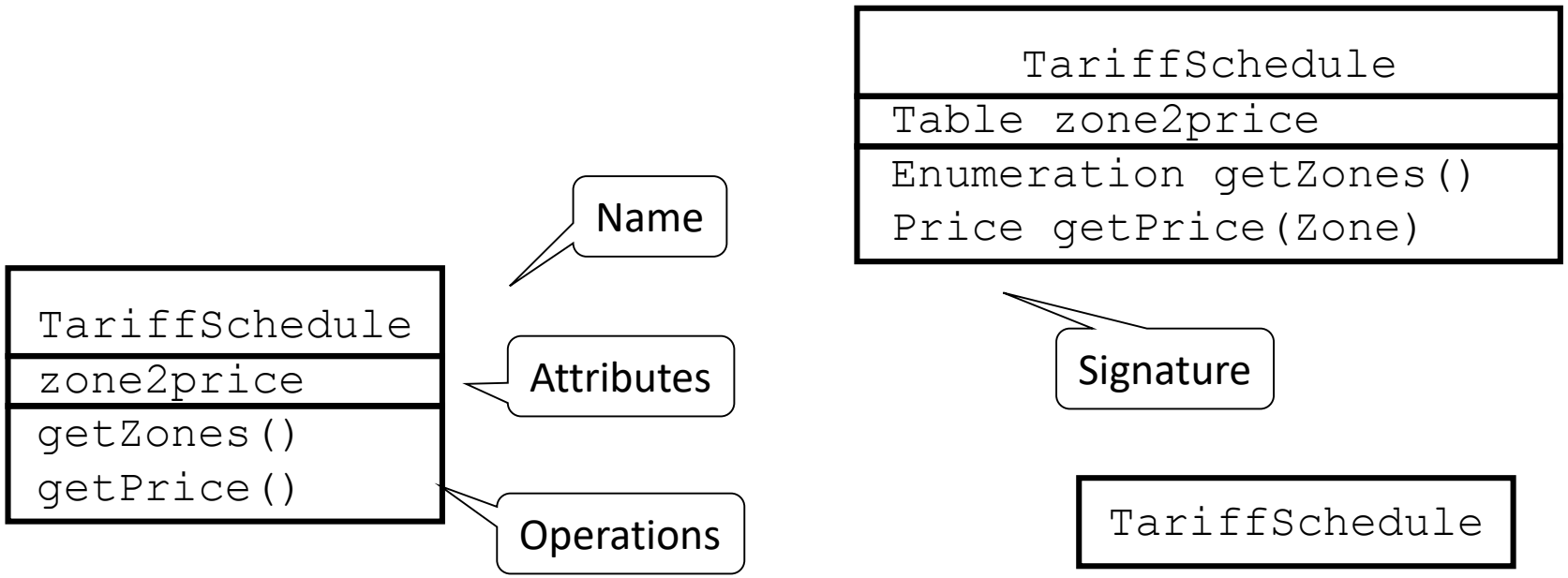
- An <<include>> relationship represents behavior that is factored out of the use case.
- An <<include>> represents behavior that is factored out for reuse, not because it is an exception.
- The direction of a <<include>> relationship is to the using use case (unlike <<extend>> relationships).

Class Diagrams



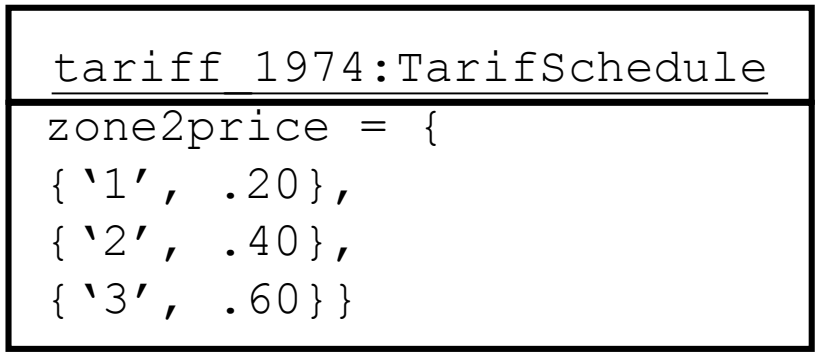
- Class diagrams represent the structure of the system.
- Class diagrams are used
  - during requirements analysis to model problem domain concepts
  - during system design to model subsystems and interfaces
  - during object design to model classes.

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

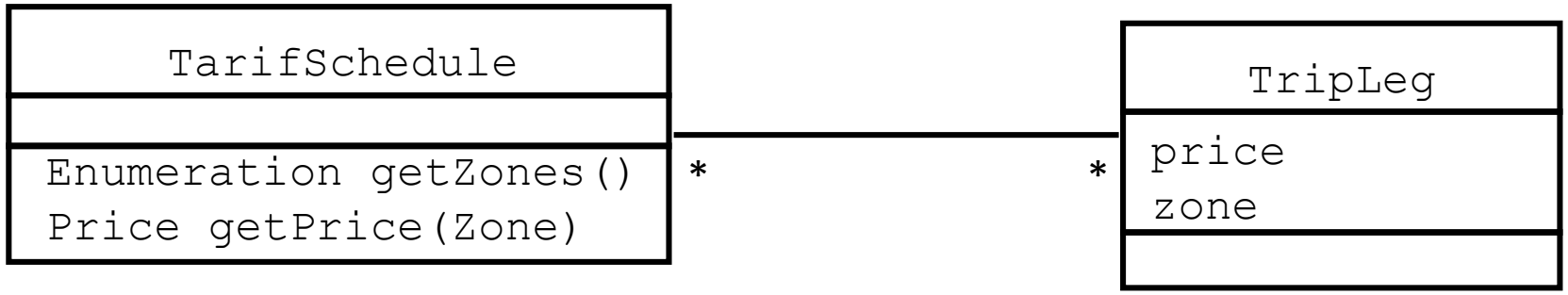


- An ***instance*** represents a phenomenon.
- The name of an instance is underlined and can contain the class of the instance.
- The attributes are represented with their ***values***.

Actor vs. Instances

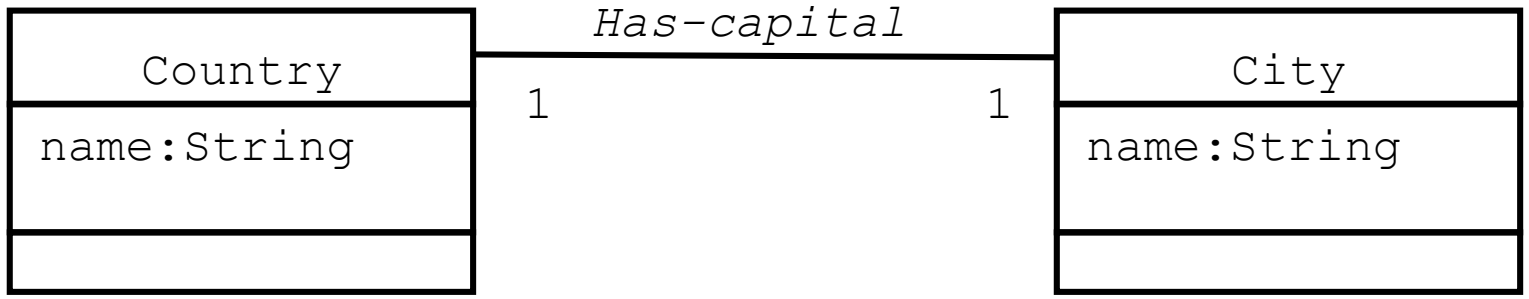
- What is the difference between an actor and a class and an instance?
- Actor:
  - An entity outside the system to be modeled, interacting with the system ("Pilot")
- Class:
  - An abstraction modeling an entity in the problem domain, inside the system to be modeled ("Cockpit")
- Object:
  - A specific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

Associations

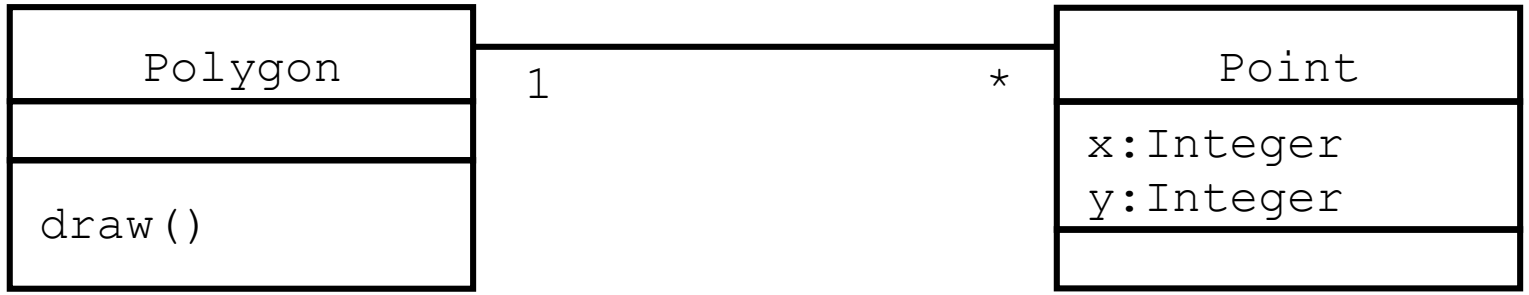


- Associations denote relationships between classes.
- The multiplicity of an association end denotes how many objects the source object can legitimately reference.

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



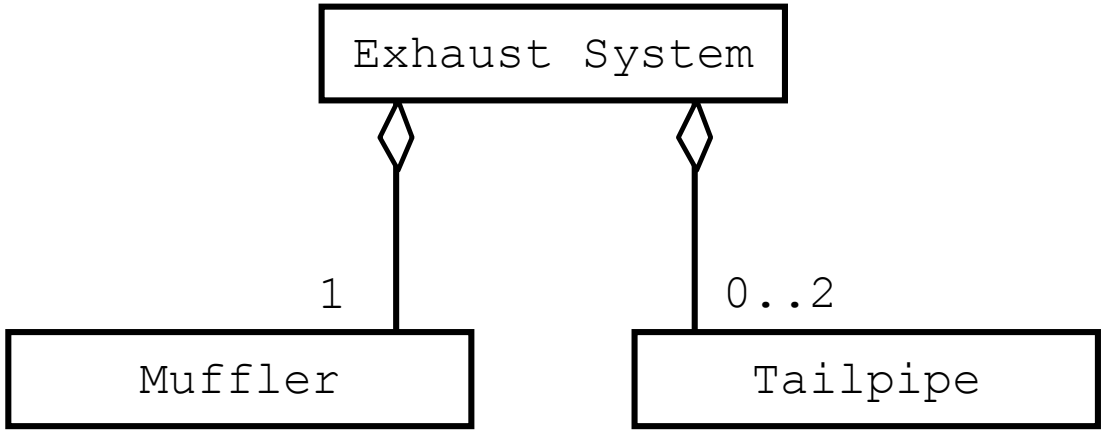
1-to-1 association



1-to-many association

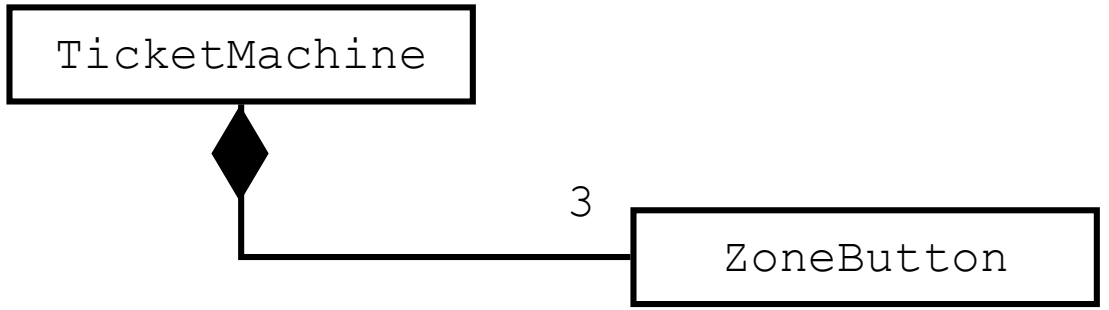
Aggregation

- 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.

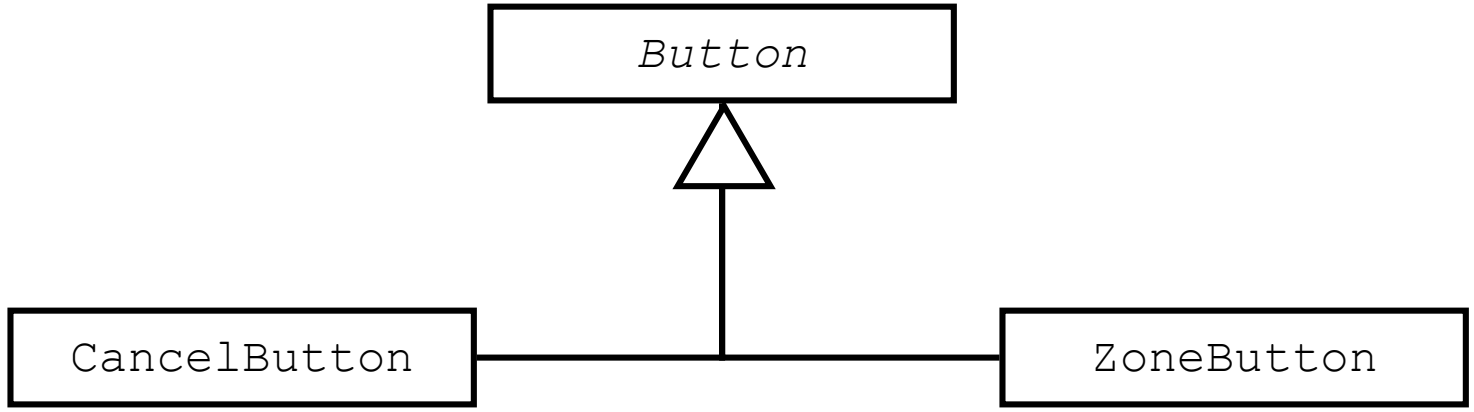


Composition

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



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

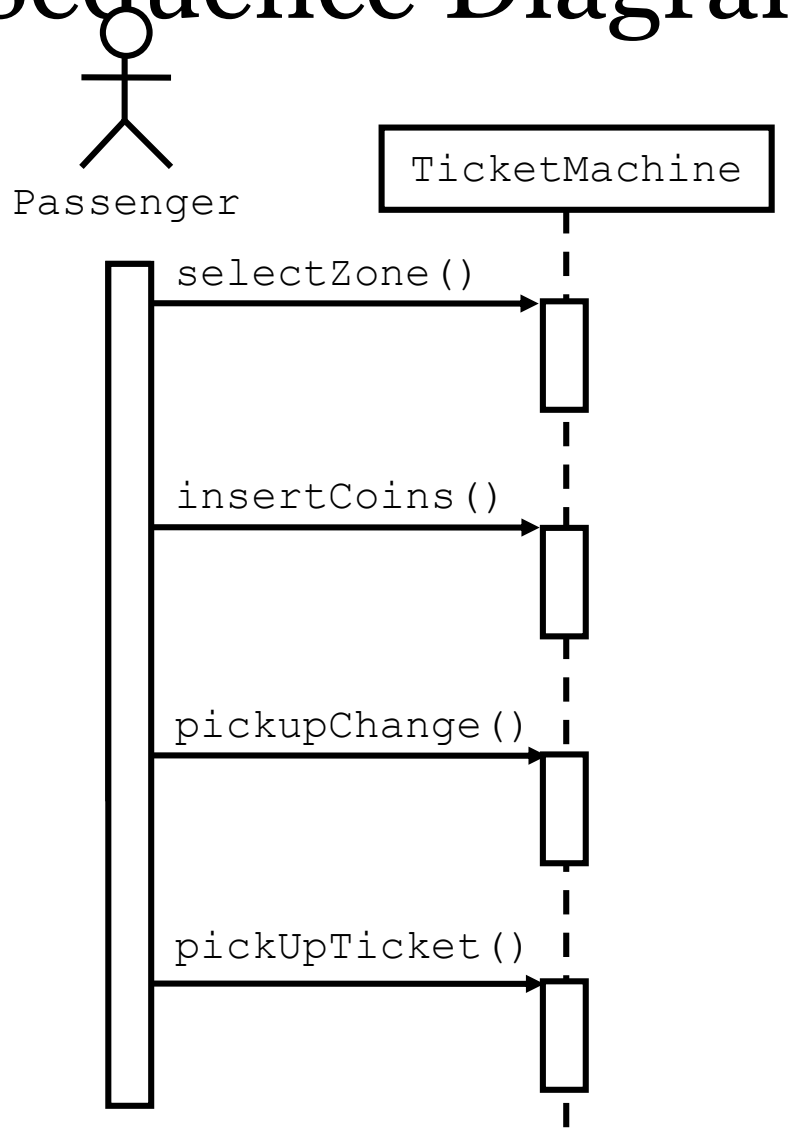
A stock exchange lists many companies. Each company is identified by a ticker symbol

Class Diagram

Java Code

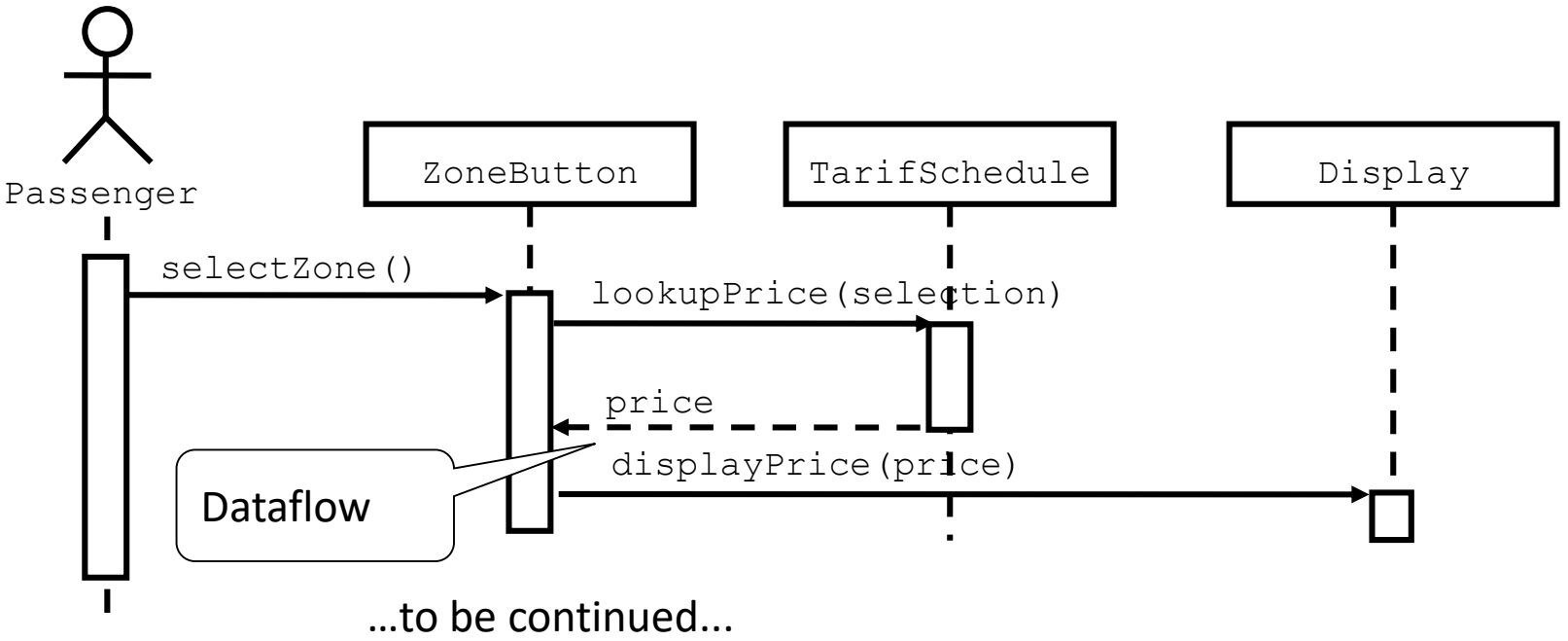
```
public class StockExchange {
    public Vector m_Company = new Vector();
};
public class Company {
    public int m_tickerSymbol;
    public Vector m_StockExchange = new Vector();
};
```

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

UML Sequence Diagrams: Nested Messages

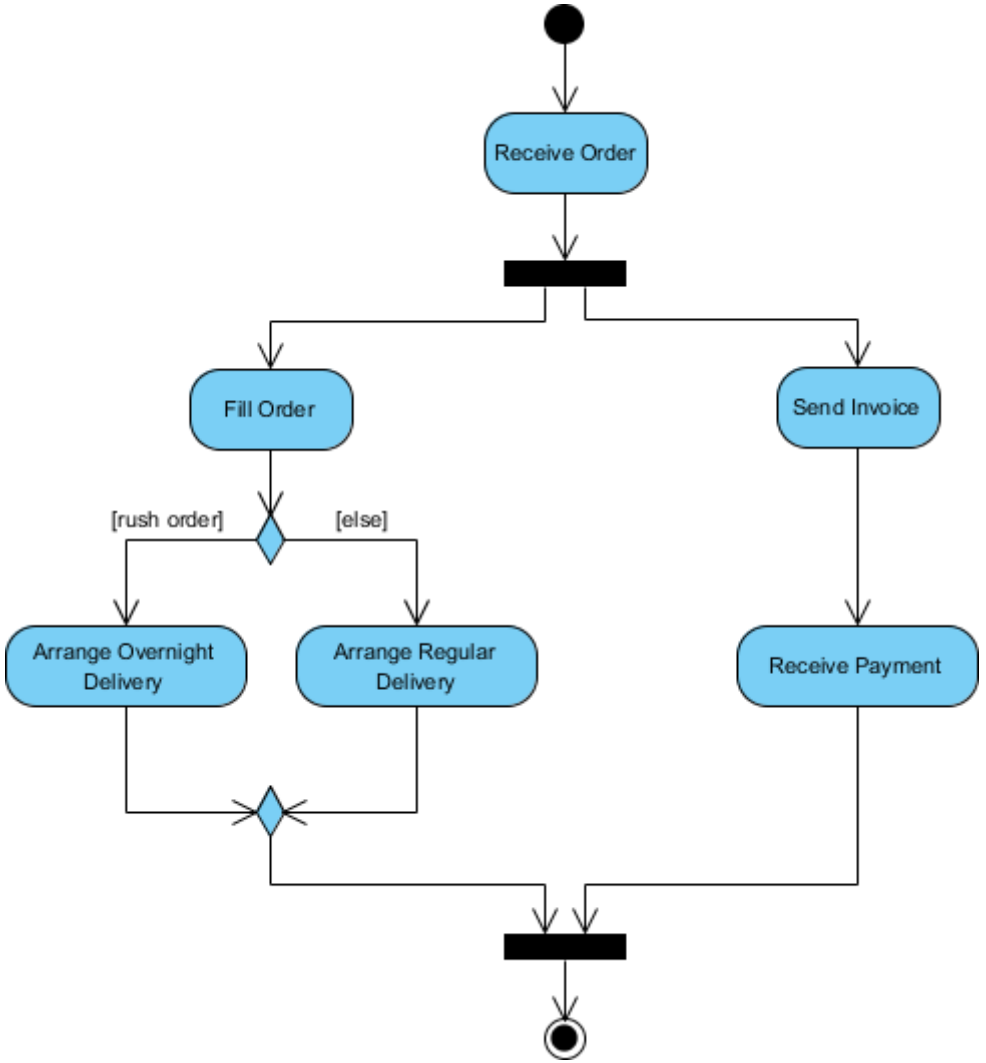


- The source of an arrow indicates the activation which sent the message
- An activation is as long as all nested activations

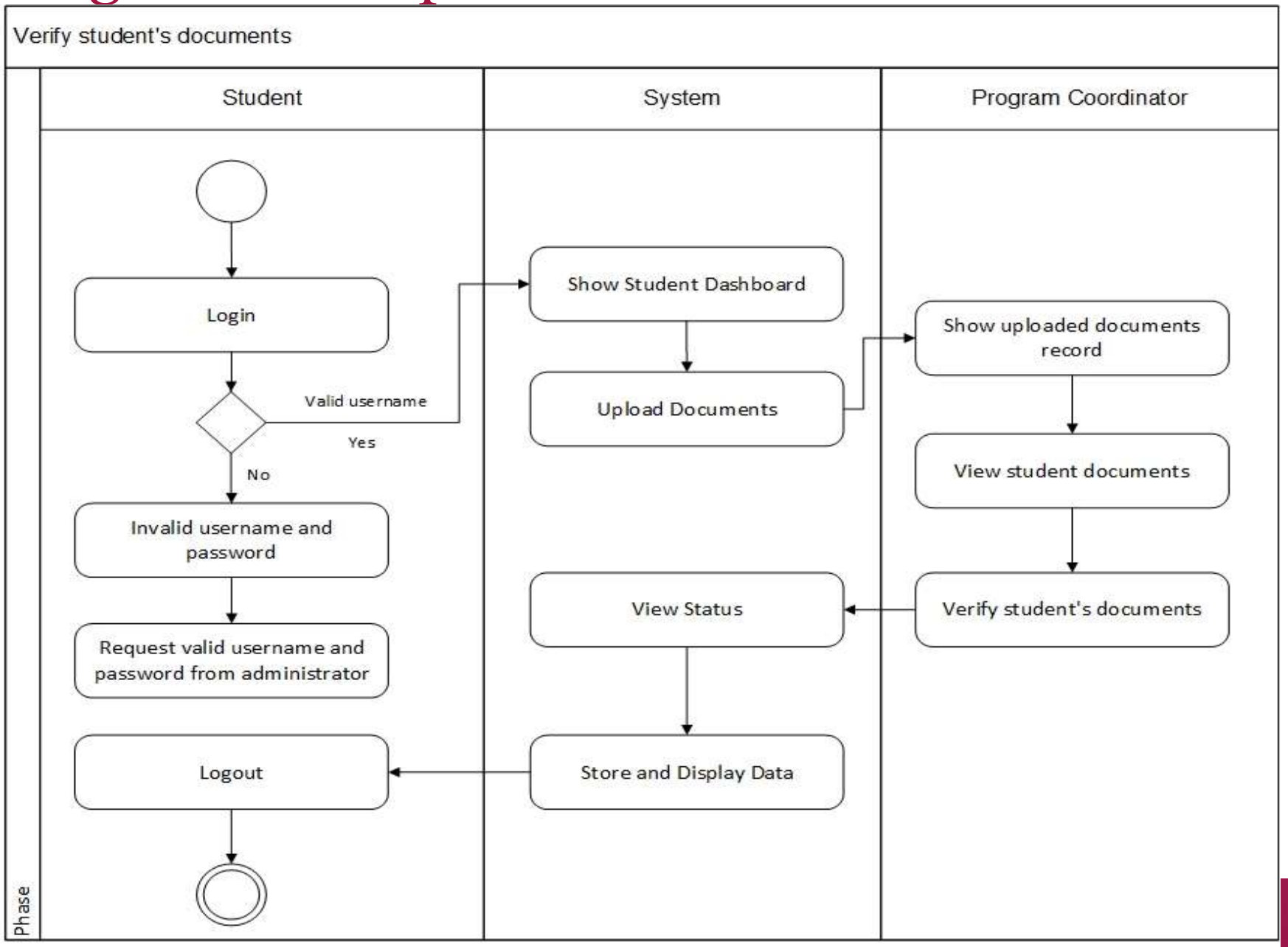
Sequence Diagram Observations

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

Activity Diagrams- simple



Activity Diagrams- complex



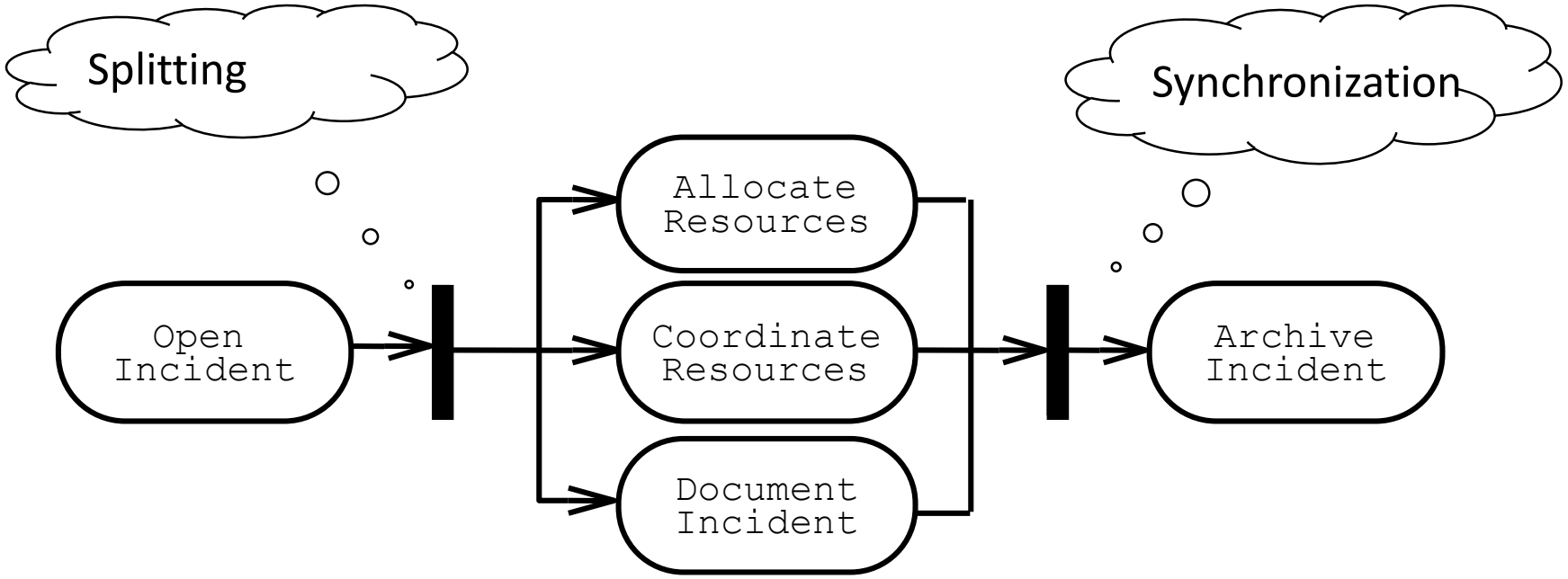
Activity Diagrams

- 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

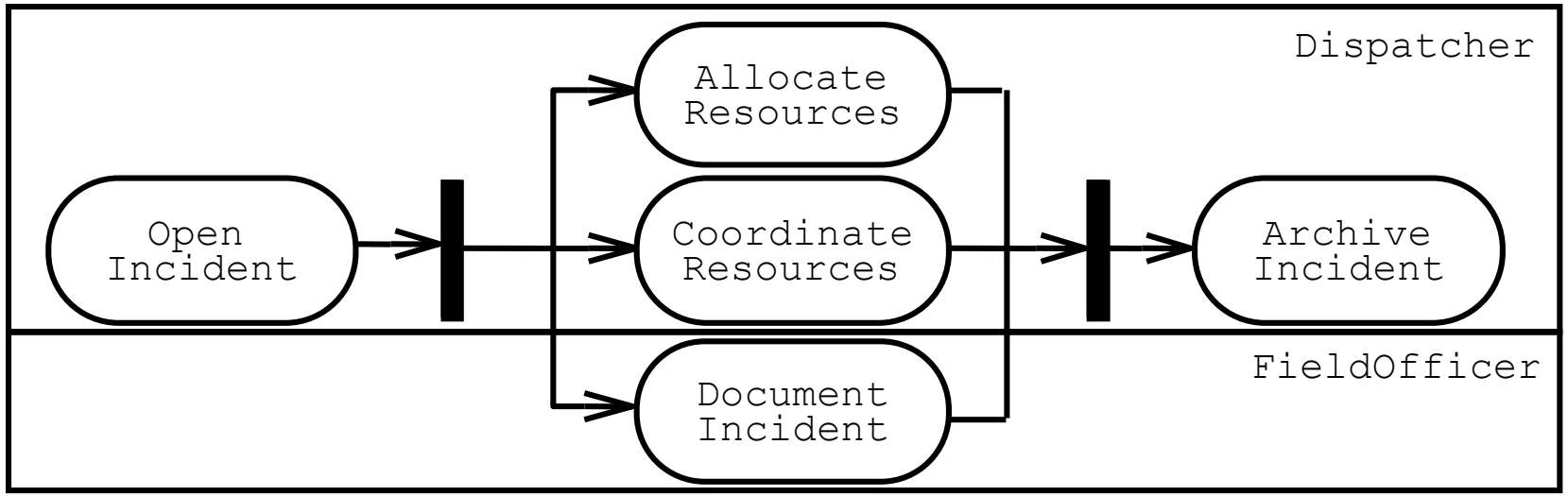
Activity Diagrams: Modeling Concurrency

- 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.



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

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