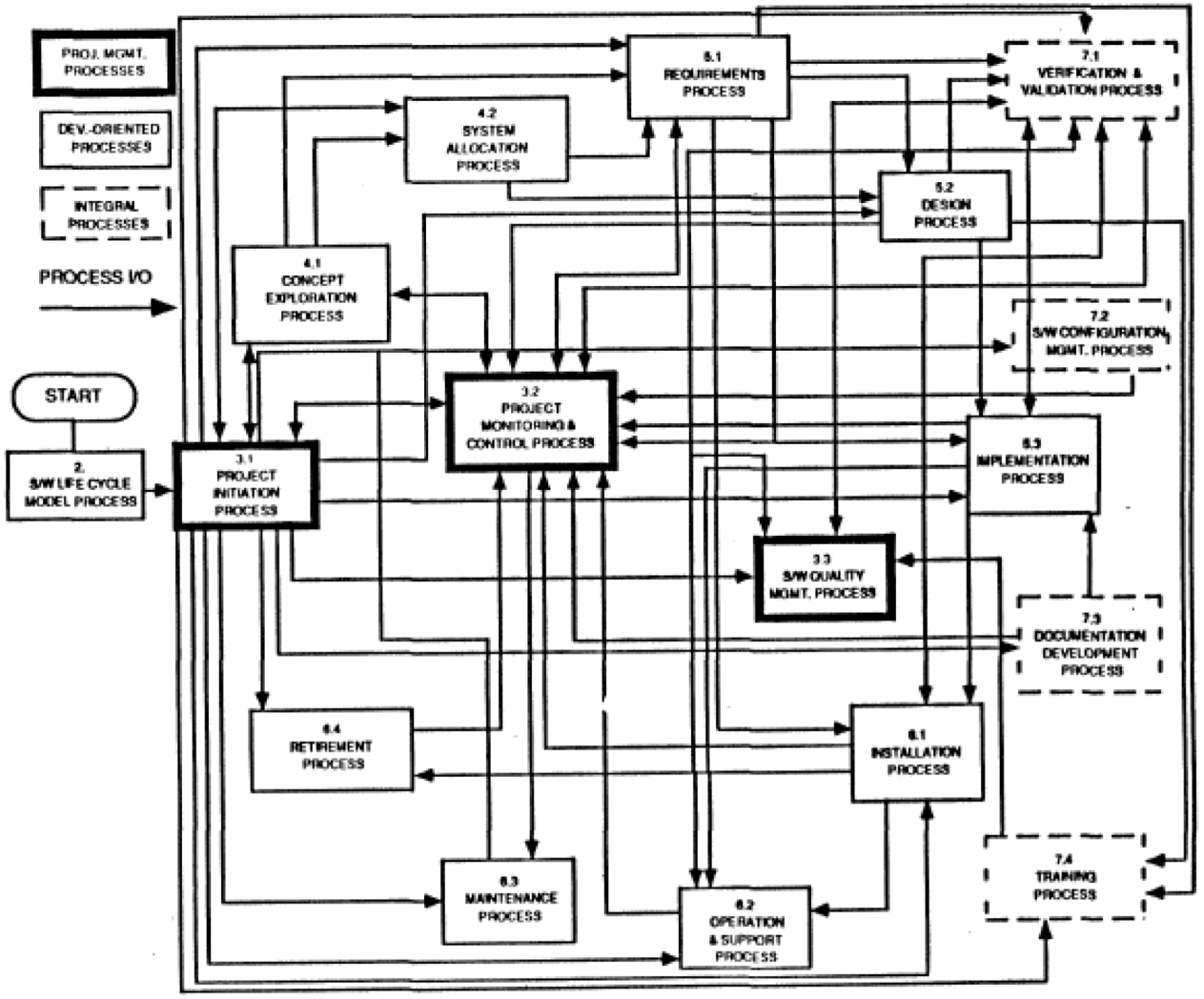
Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 2, Modeling with UML

Dealing with Complexity

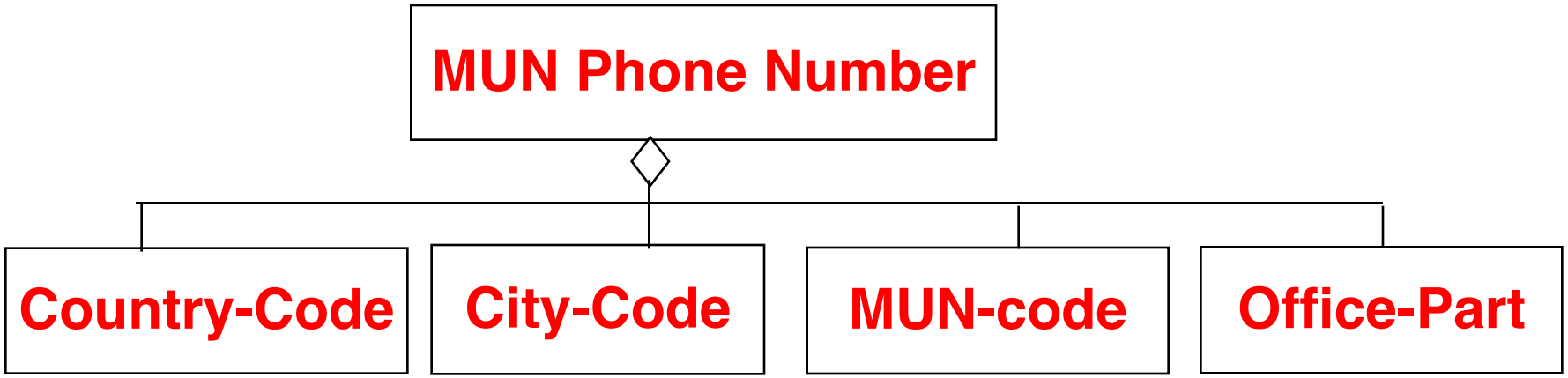
- Three ways to deal with complexity
  - Abstraction and Modeling
  - Decomposition
  - Hierarchy
- Introduction into the UML notation
- First pass on:
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams

What is the problem with this Drawing?



Abstraction

- Complex systems are hard to understand
  - The 7 +- 2 phenomena
    - Our short term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain
    - My Phone Number: +17098648632
- Chunking:
    - Group collection of objects to reduce complexity
    - State-code, city-code, MUN-code, Office-Part



Model

- A model is an abstraction of a system (abstraction allows us to ignore unessential details
  - A system that no longer exists
  - An existing system
  - A future system to be built.

We use Models to describe Software Systems

- Object model: What is the structure of the system?
- Functional model: What are the functions of the system?
- Dynamic model: How does the system react to external events?

- System Model: Object model + functional model + dynamic model

Decomposition

- A technique used to master complexity ("divide and conquer")
- Two major types of decomposition
  - Functional decomposition
  - Object-oriented decomposition
- Functional decomposition
  - The system is decomposed into modules
  - Each module is a major function in the application domain
  - Modules can be decomposed into smaller modules.

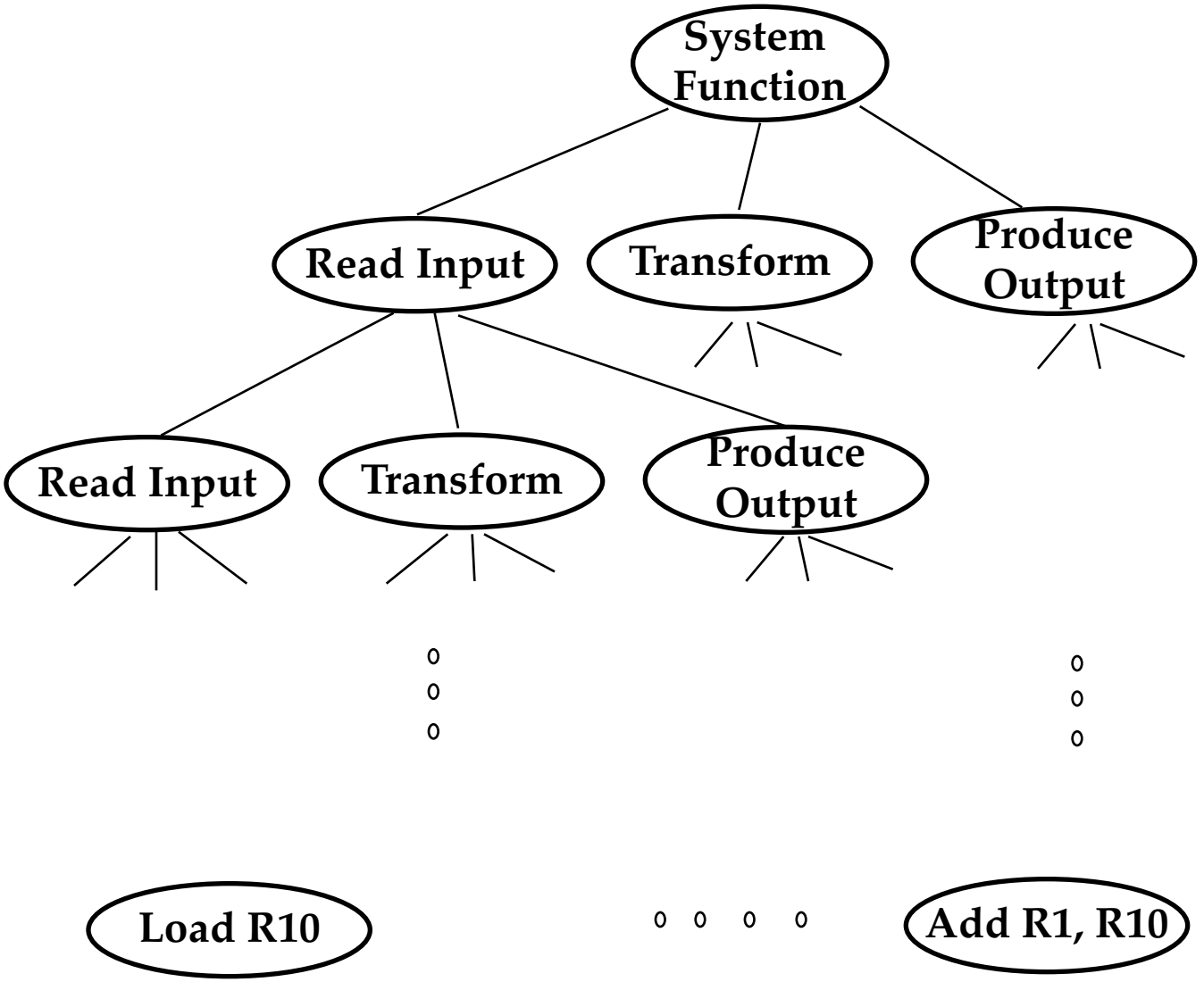
Decomposition (cont'd)

- Object-oriented decomposition
  - The system is decomposed into classes ("objects")
  - Each class is a major entity in the application domain
  - Classes can be decomposed into smaller classes

- Object-oriented vs. functional decomposition

Which decomposition is the right one?

Functional Decomposition



Top Level functions

Level 1 functions

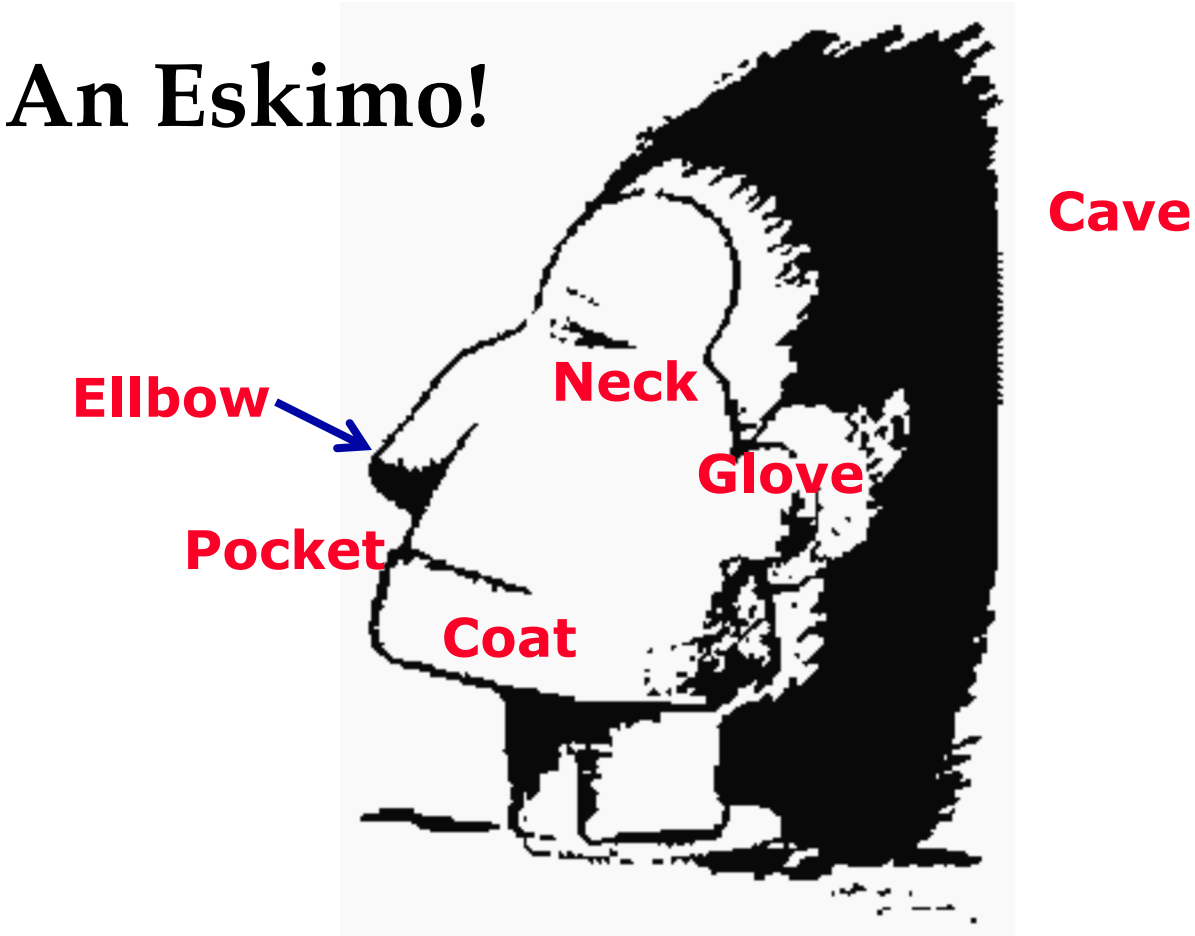
Level 2 functions

Machine instructions

Functional Decomposition

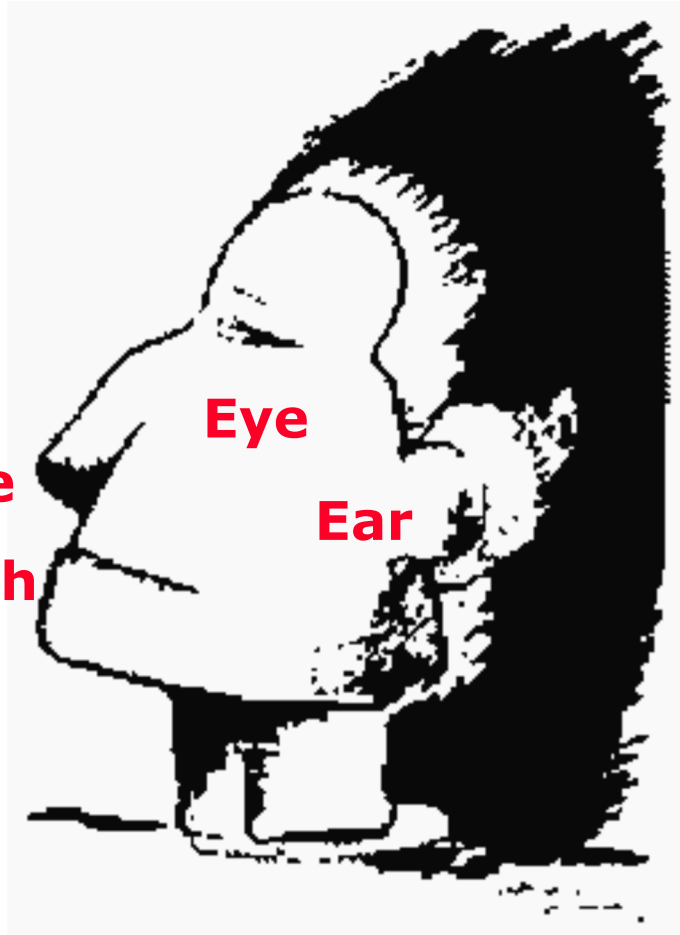
- The functionality is spread all over the system
- Maintainer must understand the whole system to make a single change to the system
- Consequence:
  - Source code is hard to understand
  - Source code is complex and impossible to maintain
  - User interface is often awkward and non-intuitive.

Object-oriented decomposition – What is This?



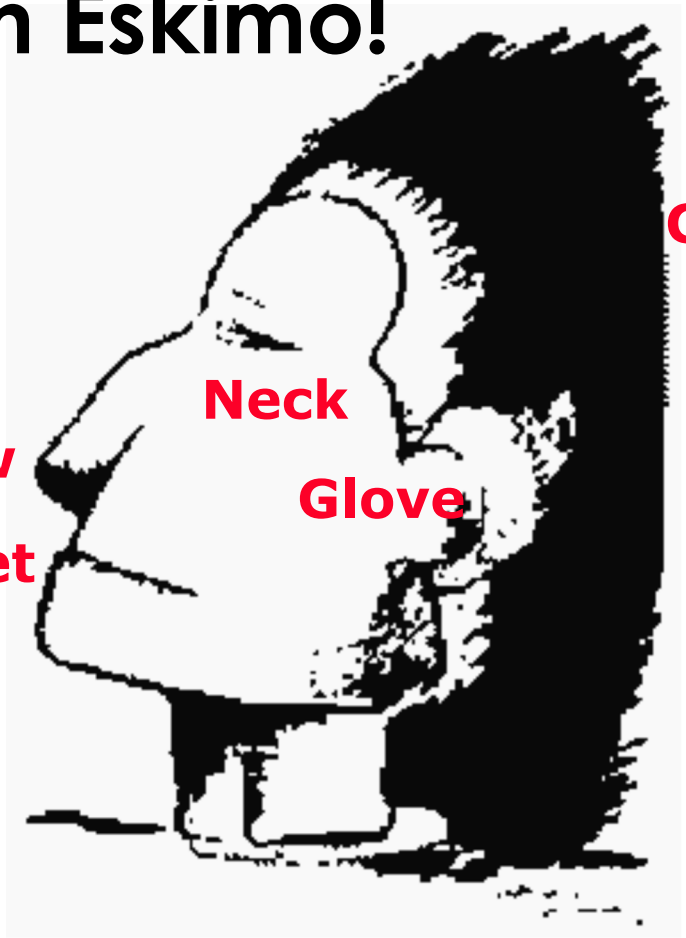
Object-oriented decomposition – What is This?

A Face!

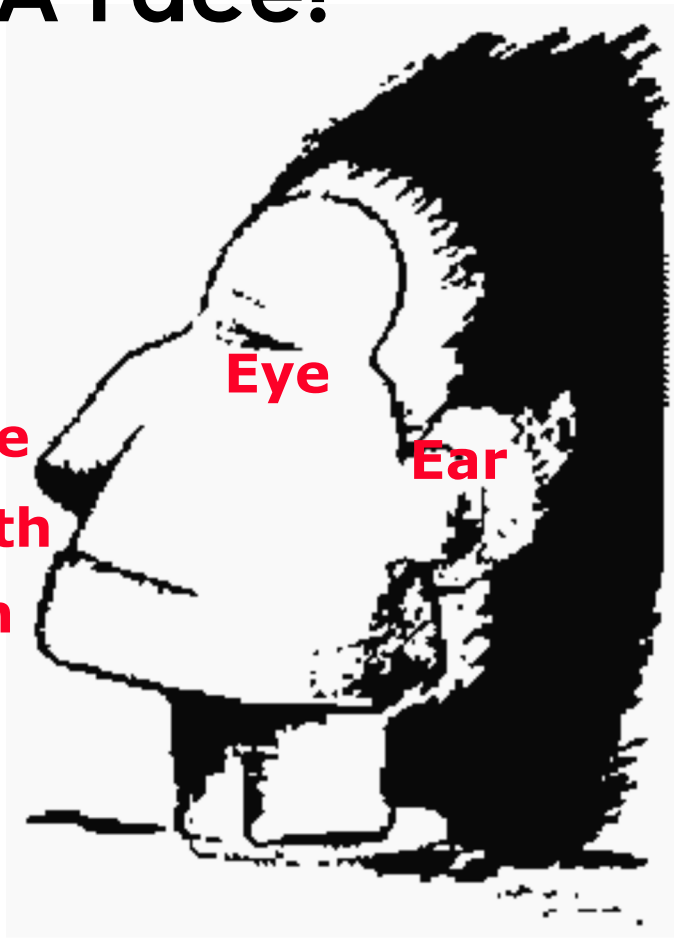


Object-oriented decomposition – What is This?

An Eskimo!



A Face!



Class Identification

- **Basic assumptions:**
  - We can find the *classes for a new software system:* Greenfield Engineering
  - We can identify the *classes in an existing system*: Reengineering
  - We can create a *class-based interface to an existing system:* Interface Engineering

Class Identification (cont'd)

- **Why can we do this?**
  - Philosophy, science, experimental evidence
- **What are the limitations?**
  - Depending on the purpose of the system, different objects might be found
- **Crucial**

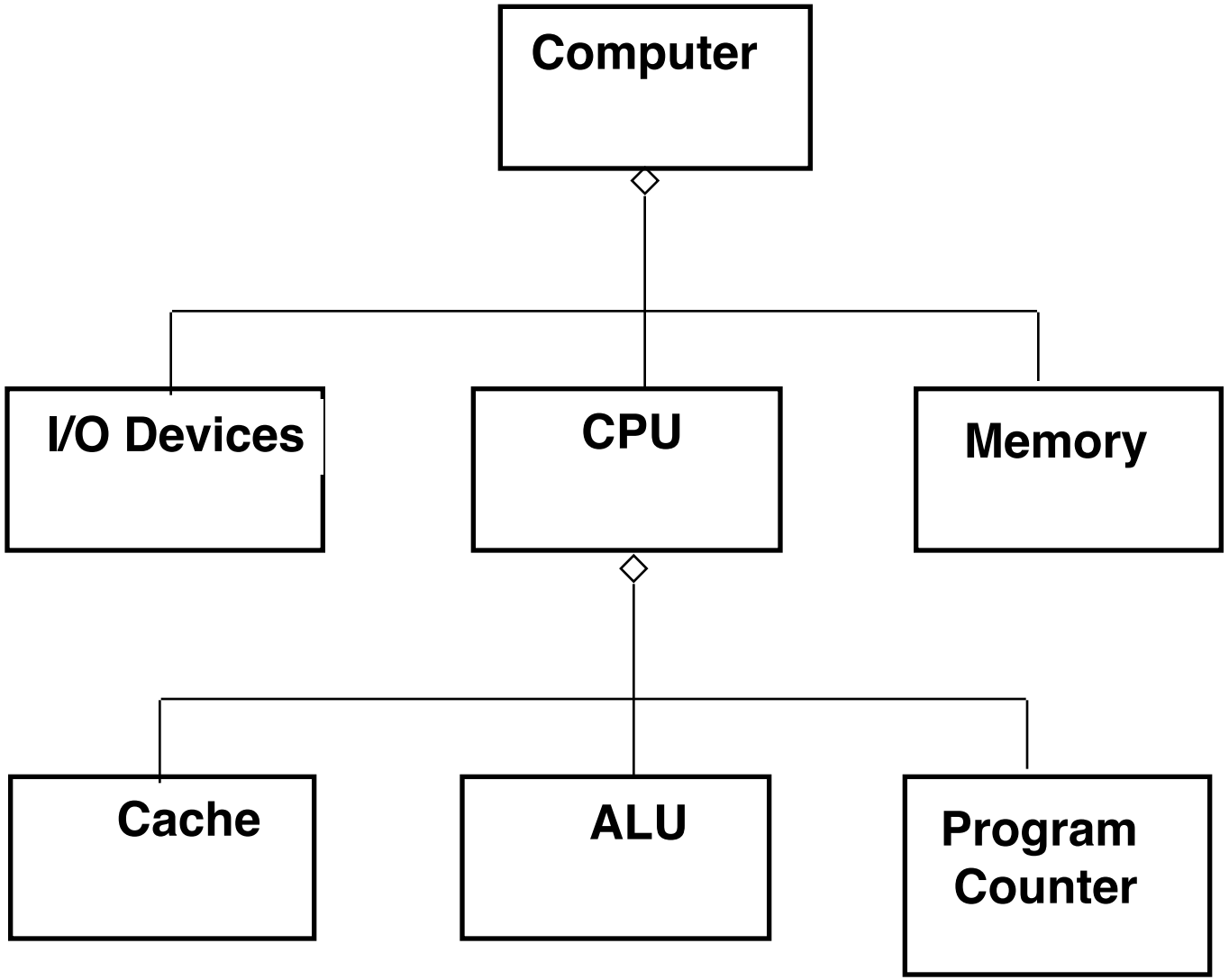
  Identify the purpose of a system

Hierarchy

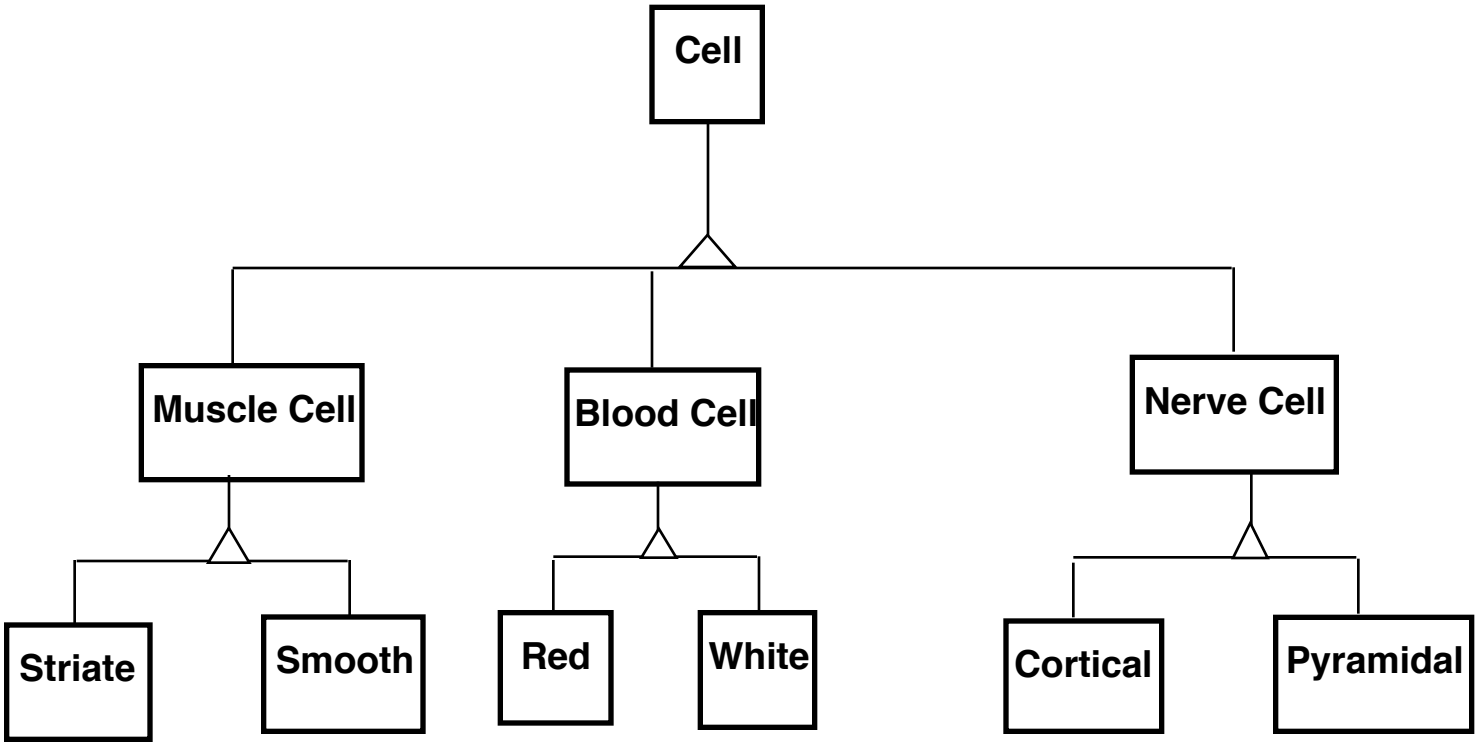
- So far we got abstractions
  - This leads us to classes and objects
  - “Chunks”

- Another way to deal with complexity is to provide relationships between these chunks
- One of the most important relationships is hierarchy
- 2 special hierarchies
  - "Part-of" hierarchy
  - "Is-kind-of" hierarchy

Part-of Hierarchy (Aggregation)



Is-Kind-of Hierarchy (Taxonomy)



Where are we now?

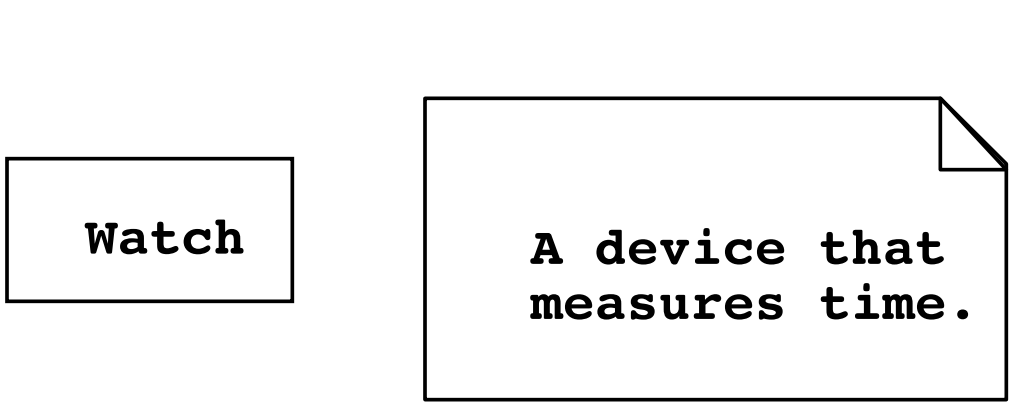
- Three ways to deal with complexity:
  - Abstraction, Decomposition, Hierarchy
- Object-oriented decomposition is good
  - Unfortunately, depending on the purpose of the system, different objects can be found
- How can we do it right?
  - Start with a description of the functionality of a system
  - Then proceed to a description of its structure
- Ordering of development activities
  - Software lifecycle

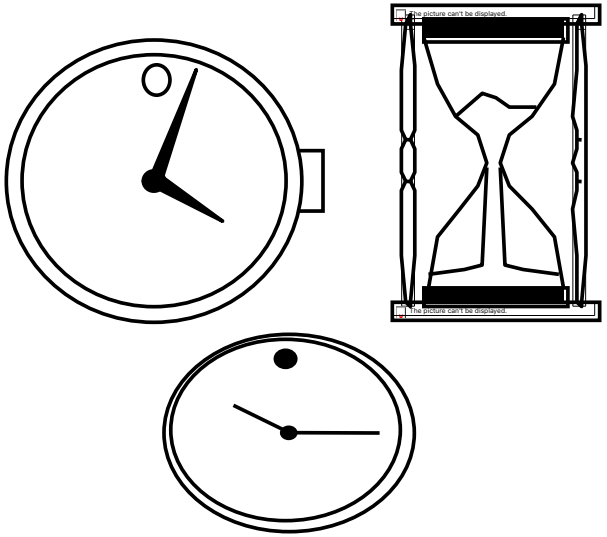
Concepts and Phenomena

- Phenomenon
  - An object in the world of a domain as you perceive it
    - Examples: This lecture at 9:35, my black watch
- Concept
  - Describes the common properties of phenomena
    - Example: All lectures on software engineering
    - Example: All black watches
- A Concept is a 3-tuple:
  - **Name:** The name distinguishes the concept 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, Phenomena, Abstraction and Modeling

Name **Purpose** **Members**





Definition Abstraction:

- Classification of phenomena into concepts

Definition Modeling:

- Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Systems

- A *system* is an organized set of communicating parts
  - Natural system: A system whose ultimate purpose is not known
  - Engineered system: A system which is designed and built by engineers for a specific purpose
- The parts of the system can be considered as systems again
  - In this case we call them *subsystems*

Examples of natural systems:

- Universe, earth, ocean

Examples of engineered systems:

- Airplane, watch, GPS

Examples of subsystems:

- Jet engine, battery, satellite.

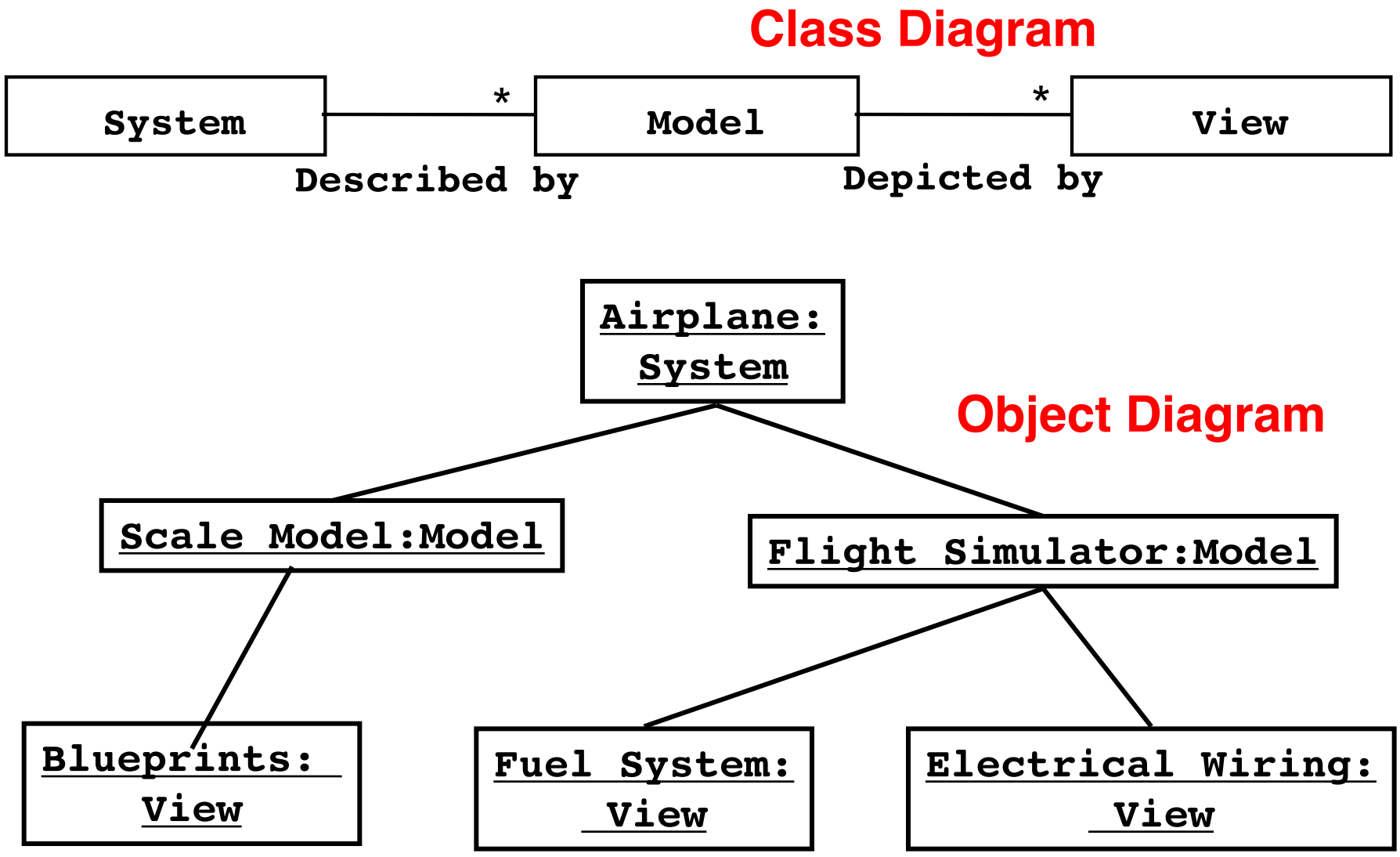
Systems, Models and Views

- A ***model*** is an abstraction describing a system or a subsystem
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting models and views:
  - formal notations, "napkin designs"

System: Airplane

Models:	**Views:**
Flight simulator	Electrical wiring, Fuel system
Scale model	Blueprint of the airplane components,

Systems, Models and Views (UML Notation)

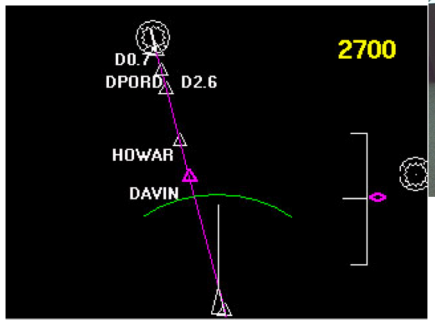


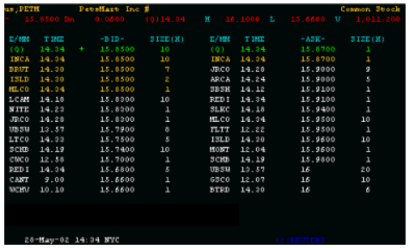
Application vs Solution Domain

- Application Domain (Analysis):
  - The environment in which the system is operating
- Solution Domain (Design, Implementation):
  - The technologies used to build the system
- Both domains contain abstractions that we can use for the construction of the system model.

Object-oriented Modeling

Application Domain
(Phenomena)





Solution Domain
(Phenomena)

System Model (Concepts) *(Analysis)*

UML Package

TrafficControl

Aircraft

TrafficController

Airport

FlightPlan

System Model (Concepts) *(Design)*

MapDisplay

Summary Display

FlightPlanDatabase

TrafficControl

What is UML?

- UML (Unified Modeling Language)
  - Nonproprietary standard for modeling software systems, OMG
  - Convergence of notations used in object-oriented methods
    - OMT (James Rumbaugh and collegues)
    - Booch (Grady Booch)
    - OOSE (Ivar Jacobson)
- Current Version: UML 2.5.1
  - Information at the OMG portal http://www.uml.org/
- Commercial tools: Rational (IBM),Together (Borland), Visual Architect (business processes, BCD)
- Open Source tools: ArgoUML, StarUML, Umbrello
- Commercial and Opensource: PoseidonUML (Gentleware)

UML: First Pass

- You can model 80% of most problems by using about 20 % UML
- We teach you those 20%

- 80-20 rule: Pareto principle
  - http://www.ephorie.de/hindle_pareto-prinzip.htm

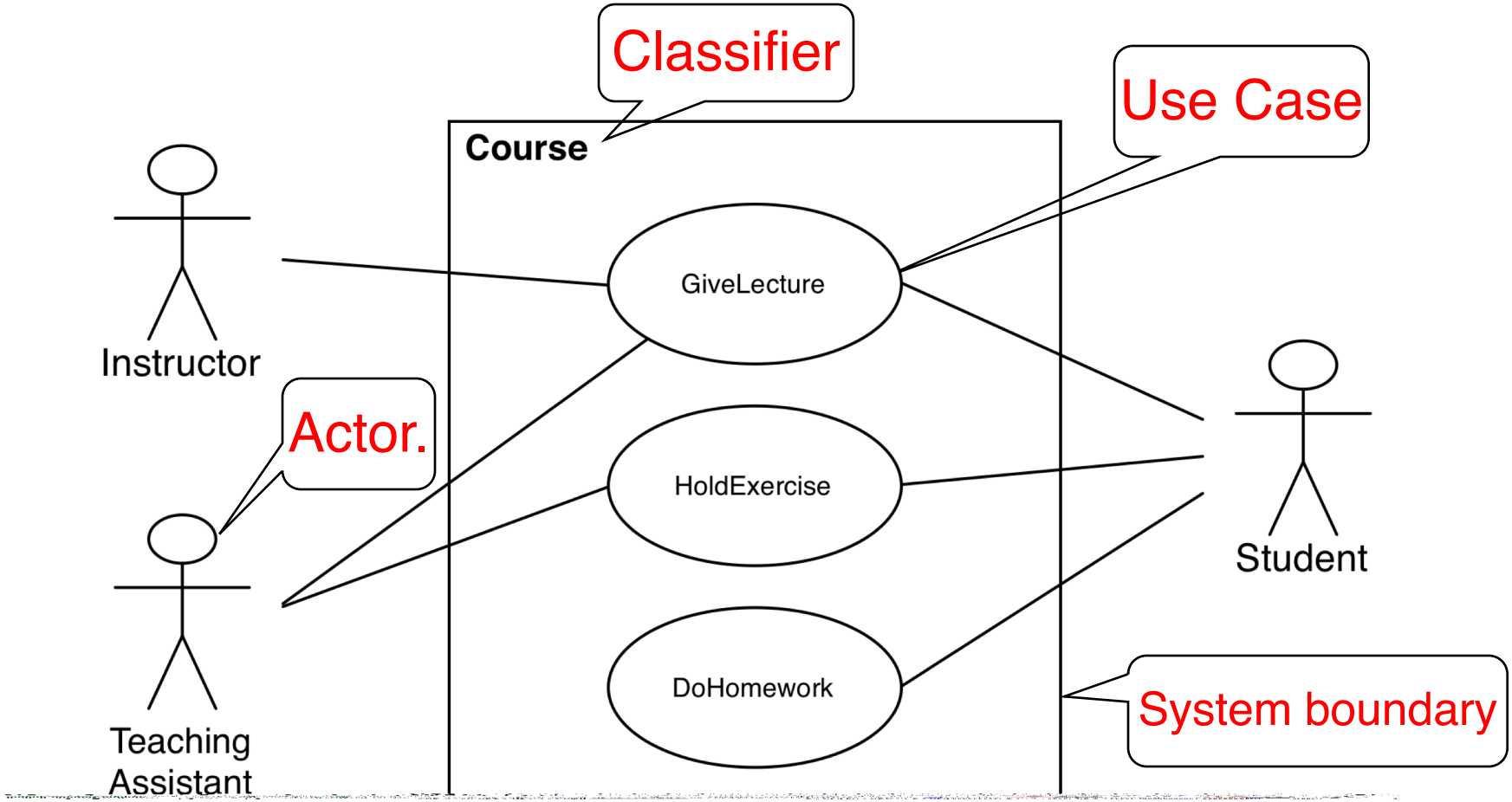
UML First Pass

- 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 objects of the system
- Statechart diagrams
  - Describe the dynamic behavior of an individual object
- Activity diagrams
  - Describe the dynamic behavior of a system, in particular the workflow.

UML Core Conventions

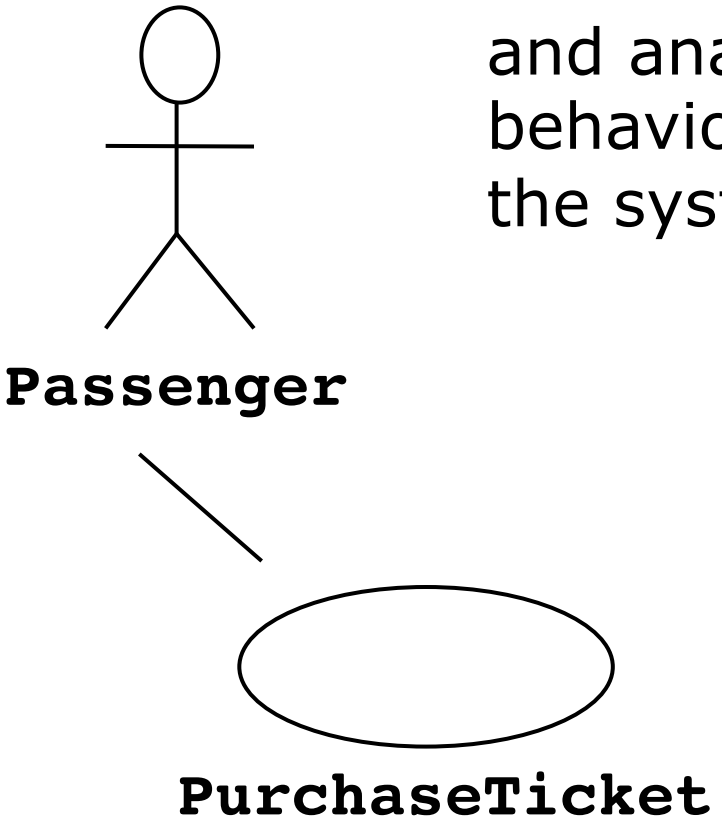
- All UML Diagrams denote graphs of nodes and edges
  - Nodes are entities and drawn as rectangles or ovals
  - Rectangles denote classes or instances
  - Ovals denote functions
- Names of Classes are not underlined
  - `SimpleWatch`
  - `Firefighter`
- Names of Instances are underlined
  - `myWatch:SimpleWatch`
  - `Joe:Firefighter`
- An edge between two nodes denotes a relationship between the corresponding entities

UML first pass: Use case diagrams



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

UML Use Case Diagrams



Used during requirements elicitation and analysis to represent external behavior ("visible from the outside of the system")

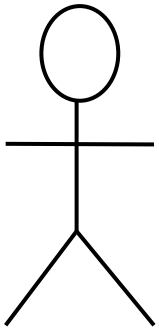
An ***Actor*** represents a role, that is, a type of user of the system

A ***use case*** represents a class of functionality provided by the system

Use case model:

The set of all use cases that completely describe the functionality of the system.

Actors



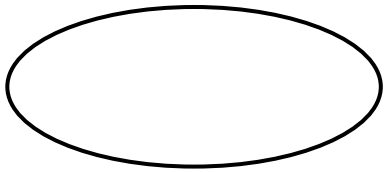
Passenger

- An actor is a model for an external entity which interacts (communicates) with the system:
  - User
  - External system (Another system)
  - Physical environment (e.g. Weather)
- An actor has a unique name and an optional description
- Examples:
  - **Passenger**: A person in the train
  - **GPS satellite**: An external system that provides the system with GPS coordinates.

Optional Description

Name

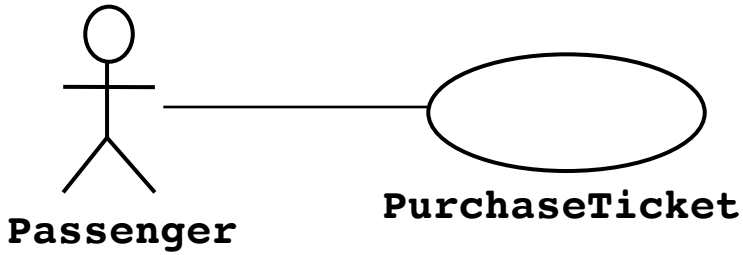
Use Case



PurchaseTicket

- A use case represents a class of functionality provided by the system
- Use cases can be described textually, with a focus on the event flow between actor and system
- The textual use case description consists of 6 parts:
  1. Unique name
  2. Participating actors
  3. Entry conditions
  4. Exit conditions
  5. Flow of events
  6. Special requirements.

Textual Use Case Description Example



1. *Name:* `Purchase ticket`

2. *Participating actor:*
   `Passenger`

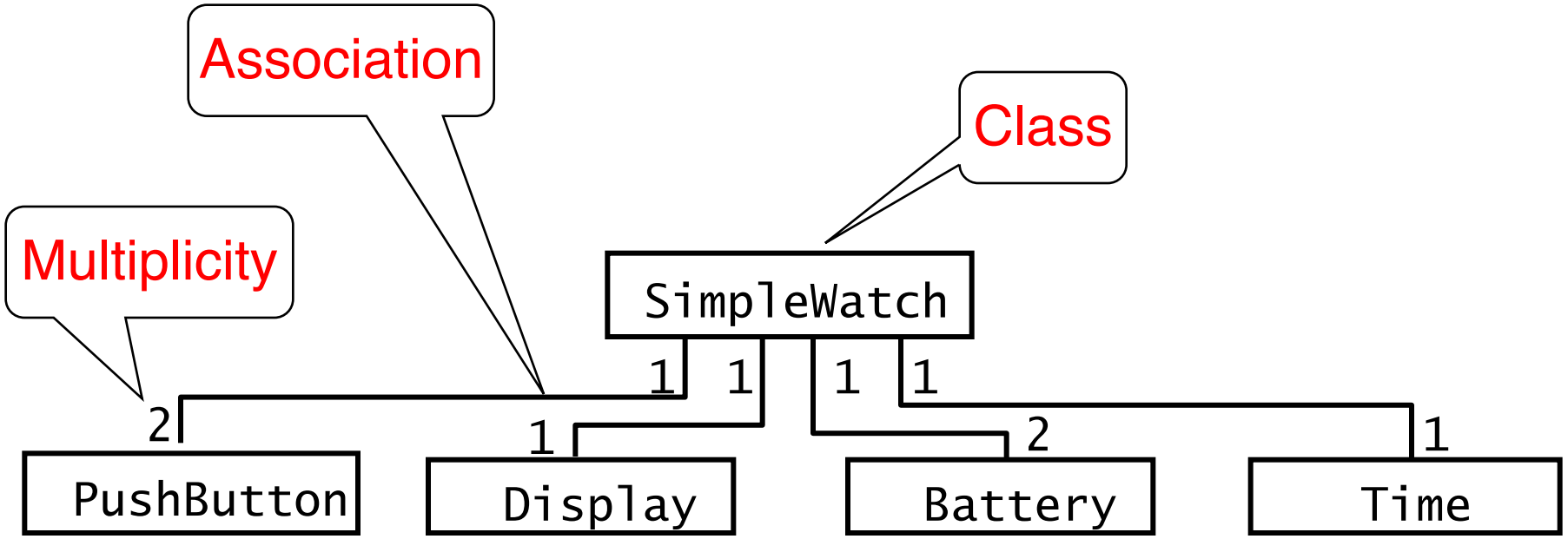
3. *Entry condition:*
   - `Passenger` stands in front of ticket distributor
   - `Passenger` has sufficient money to purchase ticket

4. *Exit condition:*
   - `Passenger` has ticket

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

6. *Special requirements:*
   None.

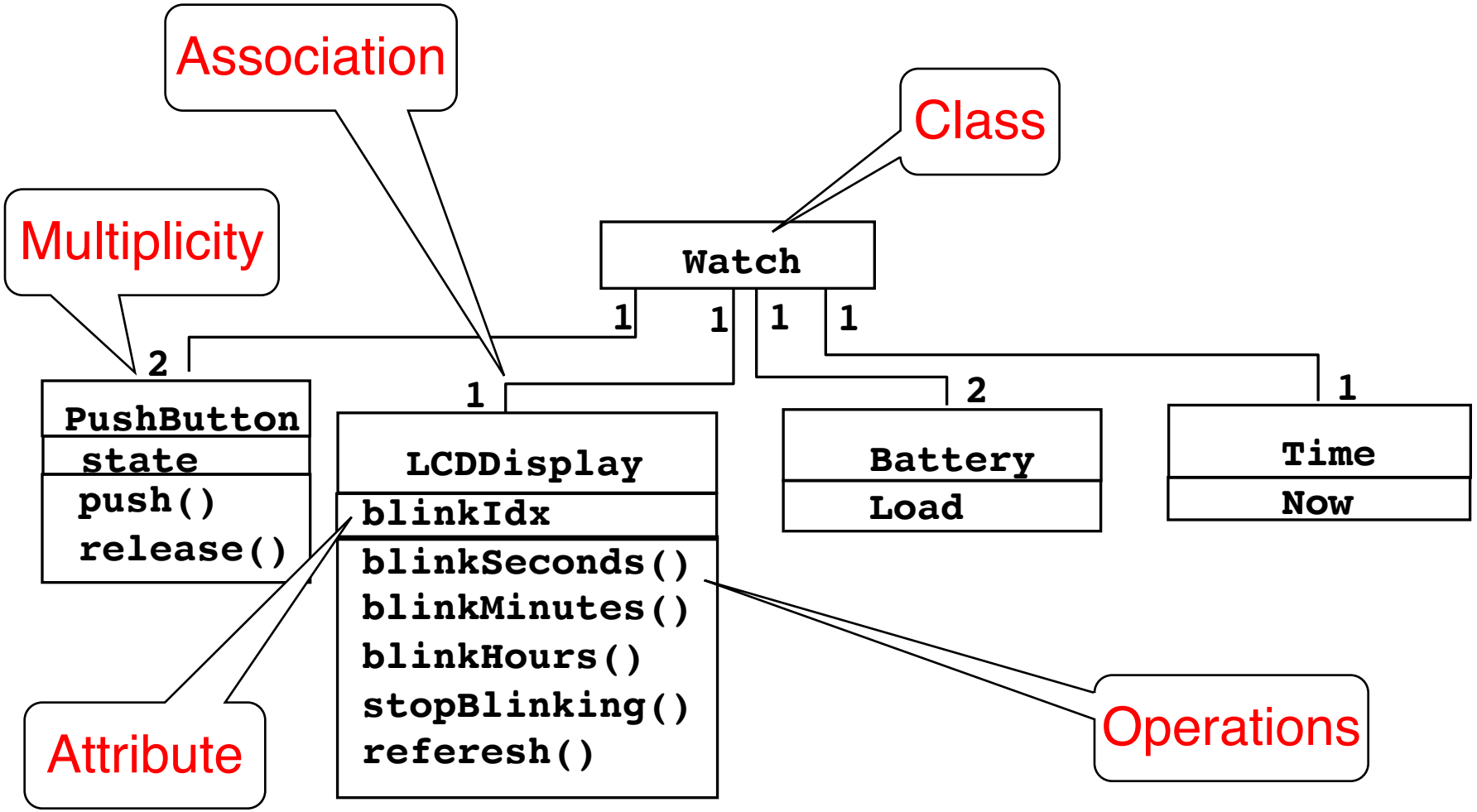
UML first pass: Class diagrams



Class diagrams represent the structure of the system

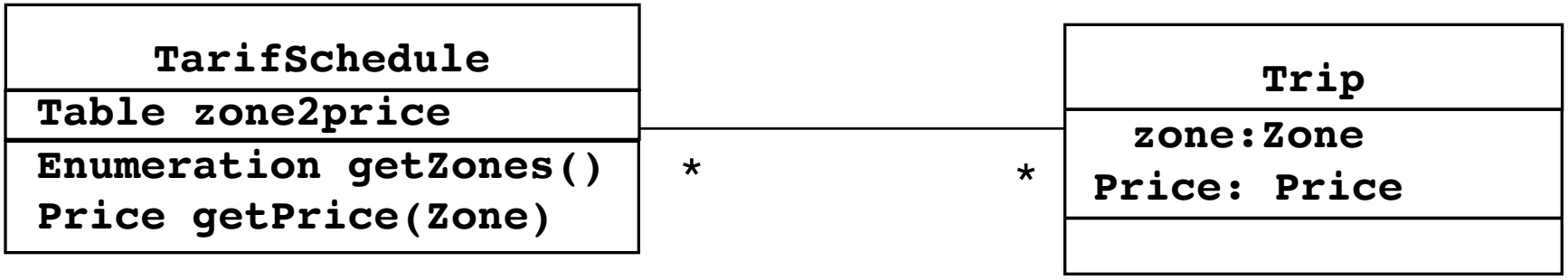
UML first pass: Class diagrams

Class diagrams represent the structure of the system

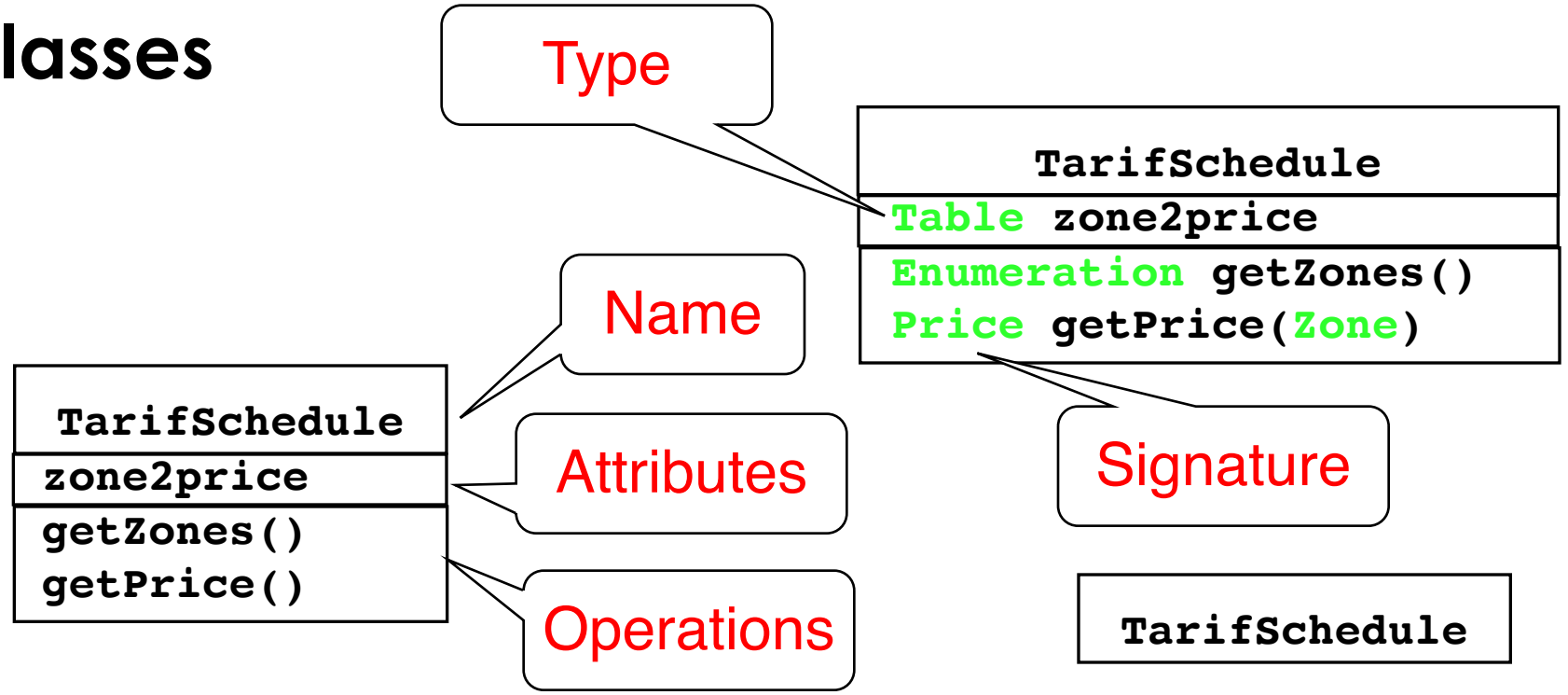


Class Diagrams

- Class diagrams represent the structure of the system
- Used
  - during requirements analysis to model application domain concepts
  - during system design to model subsystems
  - during object design to specify the detailed behavior and attributes of classes.



Classes

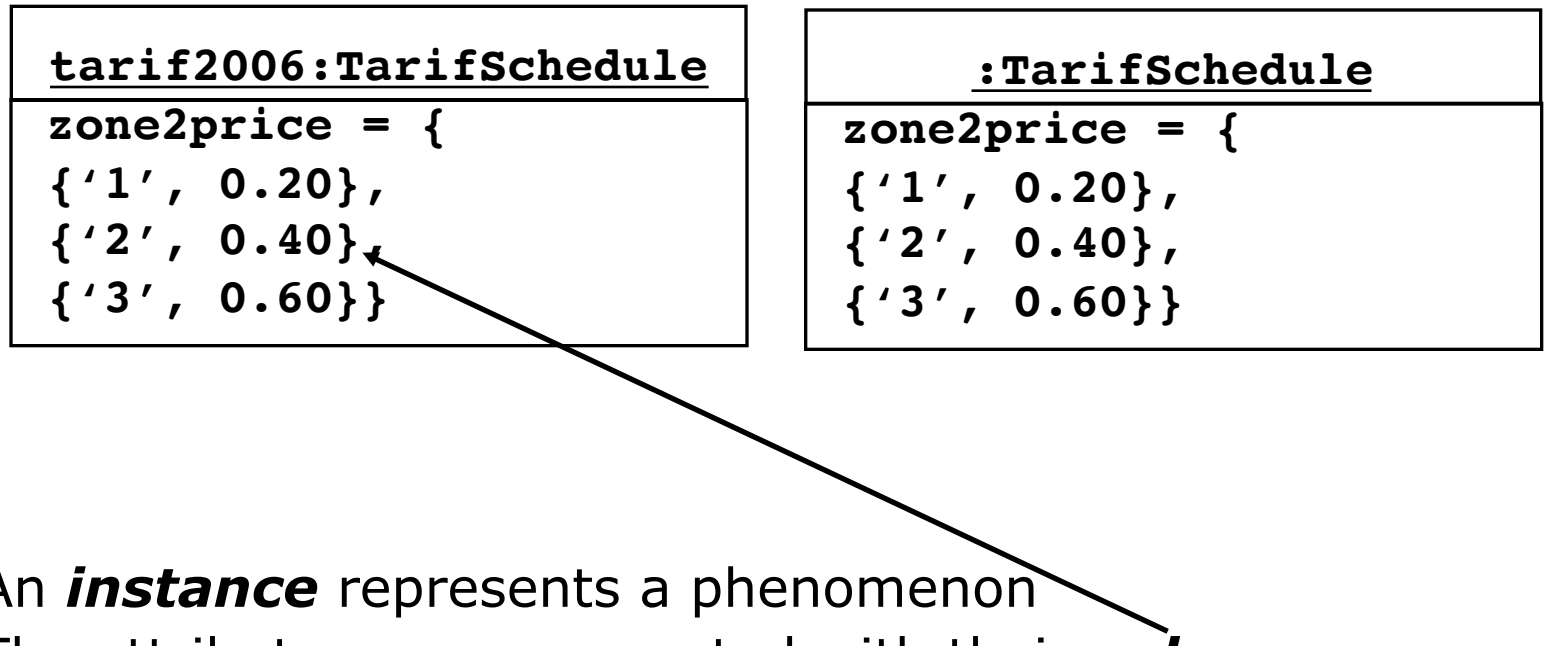


- A ***class*** represents a concept
- A class encapsulates state ***(attributes)*** and behavior ***(operations)***

  Each attribute has a ***type***
  Each operation has a ***signature***

The class name is the only mandatory information

Instances

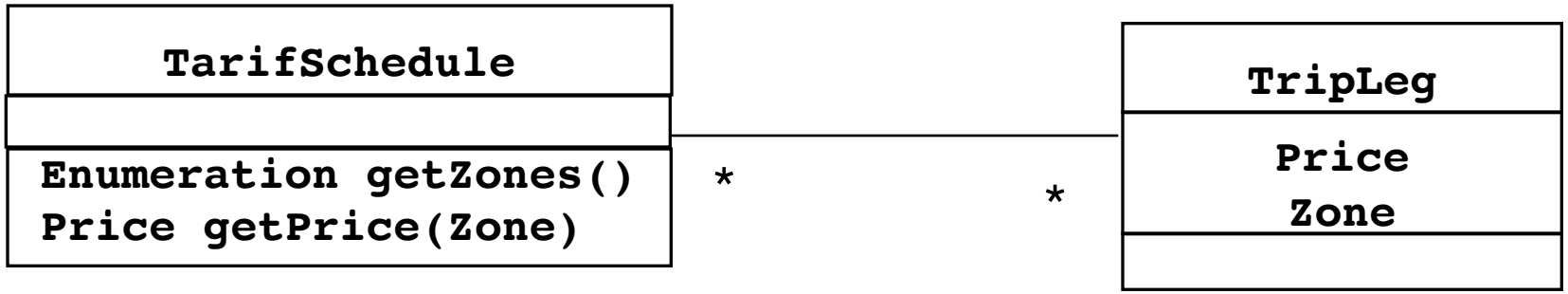


- An ***instance*** represents a phenomenon
- The attributes are represented with their ***values***
- The name of an instance is underlined
- The name can contain only the class name of the instance (anonymous instance)

Actor vs Class vs Object

- **Actor**
  - An entity outside the system to be modeled, interacting with the system (“Passenger”)
- **Class**
  - An abstraction modeling an entity in the application or solution domain
  - The class is part of the system model (“User”, “Ticket distributor”, “Server”)
- **Object**
  - A specific instance of a class (“Joe, the passenger who is purchasing a ticket from the ticket distributor”).

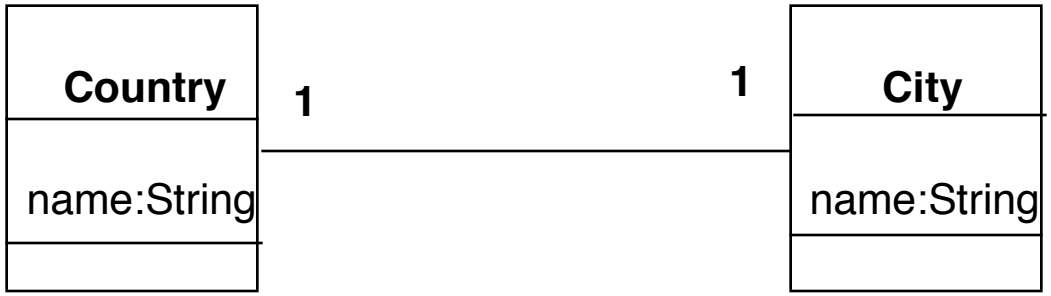
Associations



Associations denote relationships between classes

The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

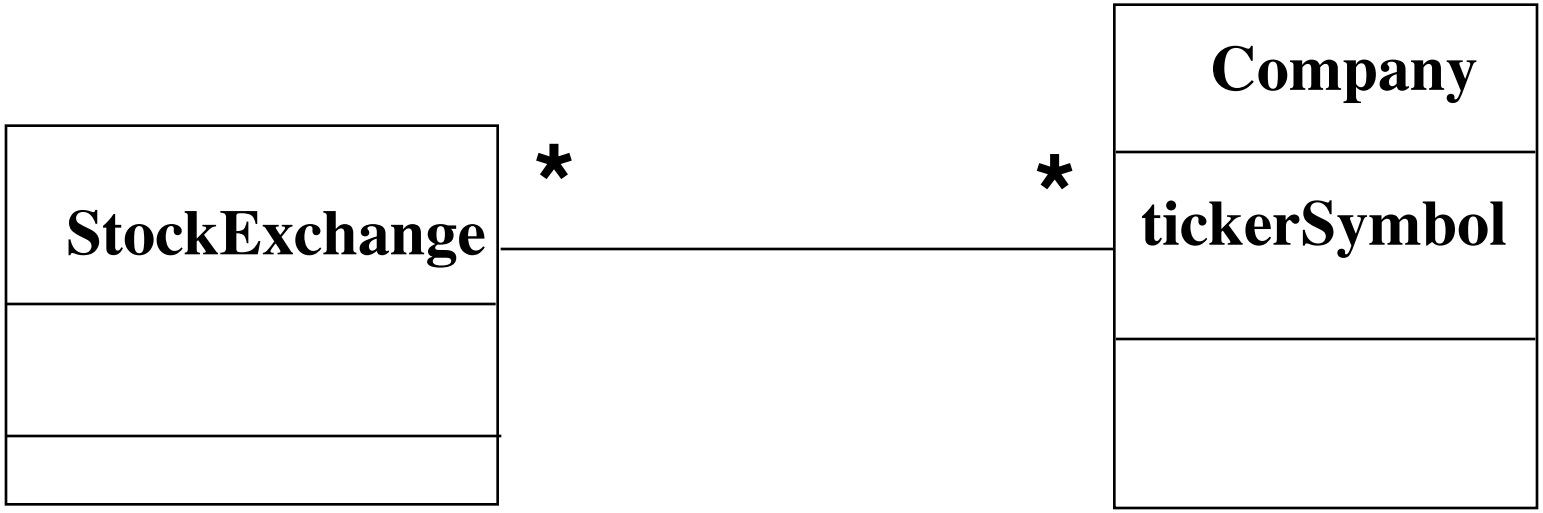
1-to-1 and 1-to-many Associations



1-to-1 association

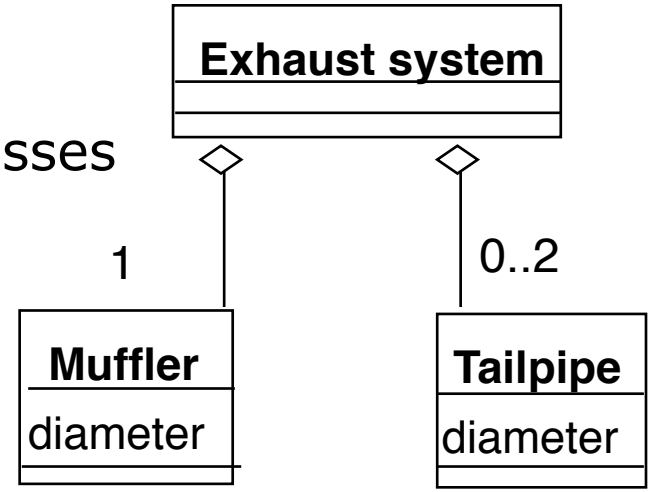
1-to-many association

Many-to-Many Associations

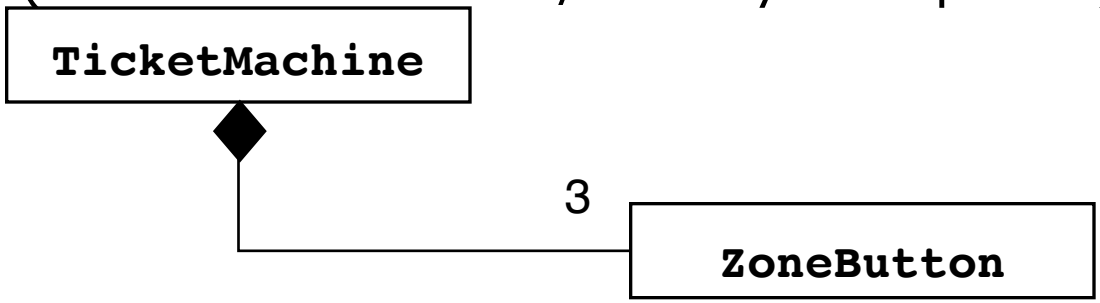


Aggregation

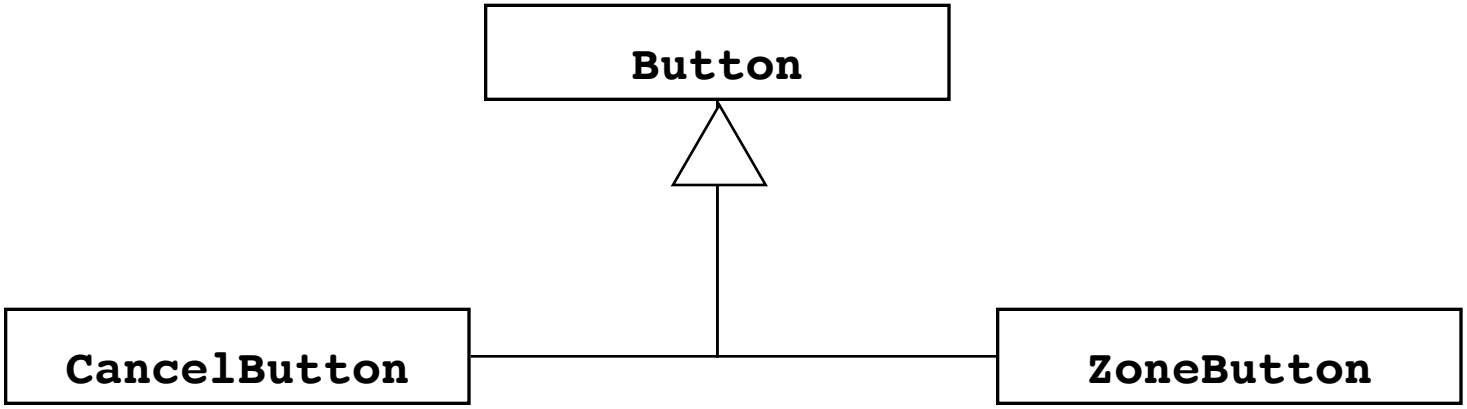
- An *aggregation* is a special case of association denoting a "consists-of" hierarchy
- The *aggregate* is the parent class, the components are the children classes



A solid diamond denotes *composition*: A strong form of aggregation where the *life time of the component instances* is controlled by the aggregate. That is, the parts don't exist on their won ("the whole controls/destroys the parts")



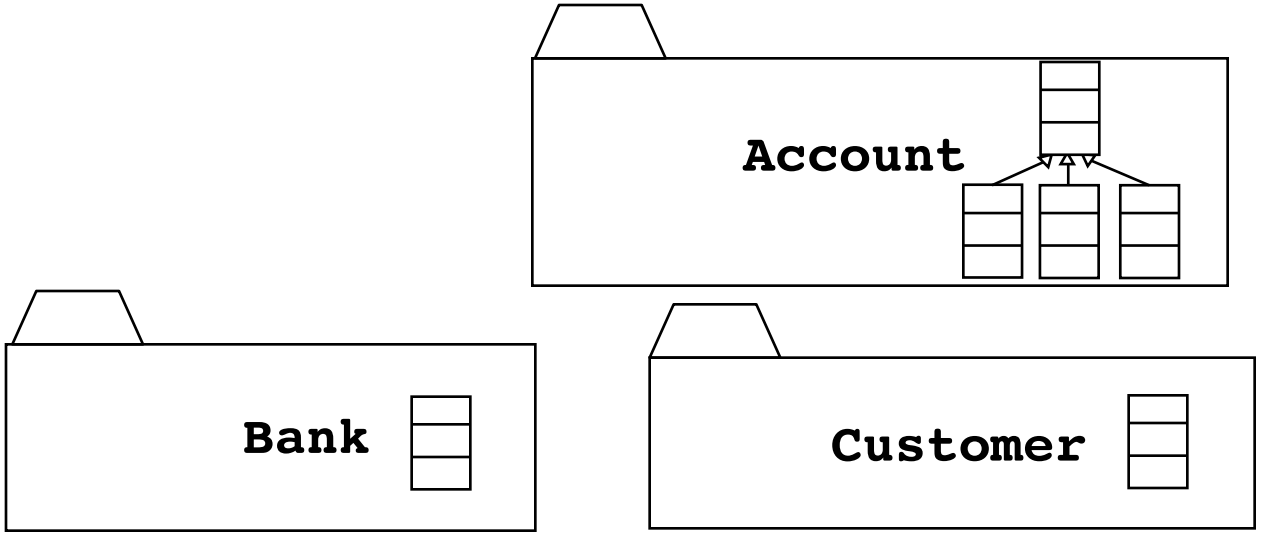
Inheritance



- *Inheritance* is another special case of an association denoting a "kind-of" hierarchy
- Inheritance simplifies the analysis model by introducing a taxonomy
- The **children classes** inherit the attributes and operations of the **parent class.**

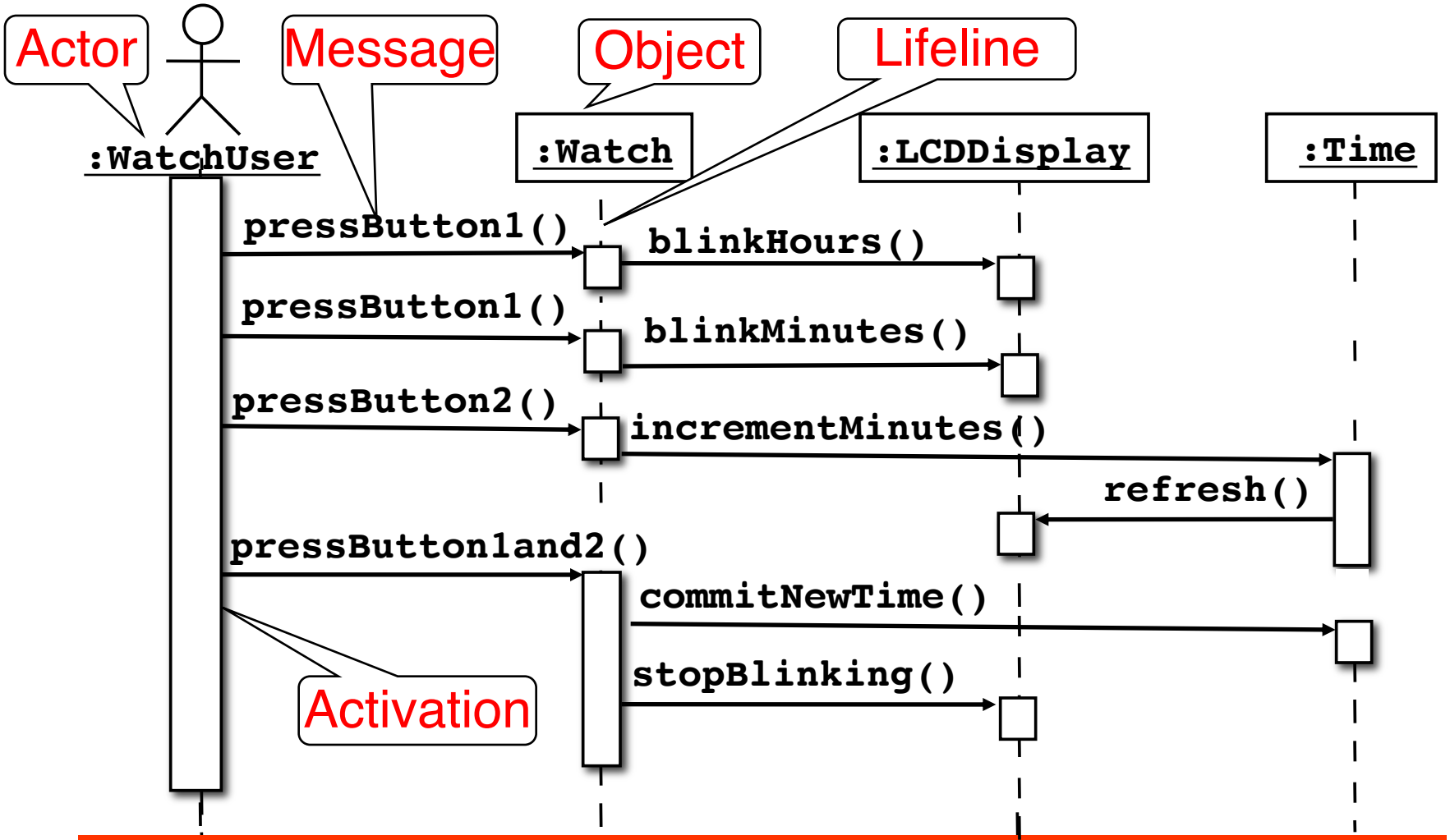
Packages

- Packages help you to organize UML models to increase their readability
- We can use the UML package mechanism to organize classes into subsystems



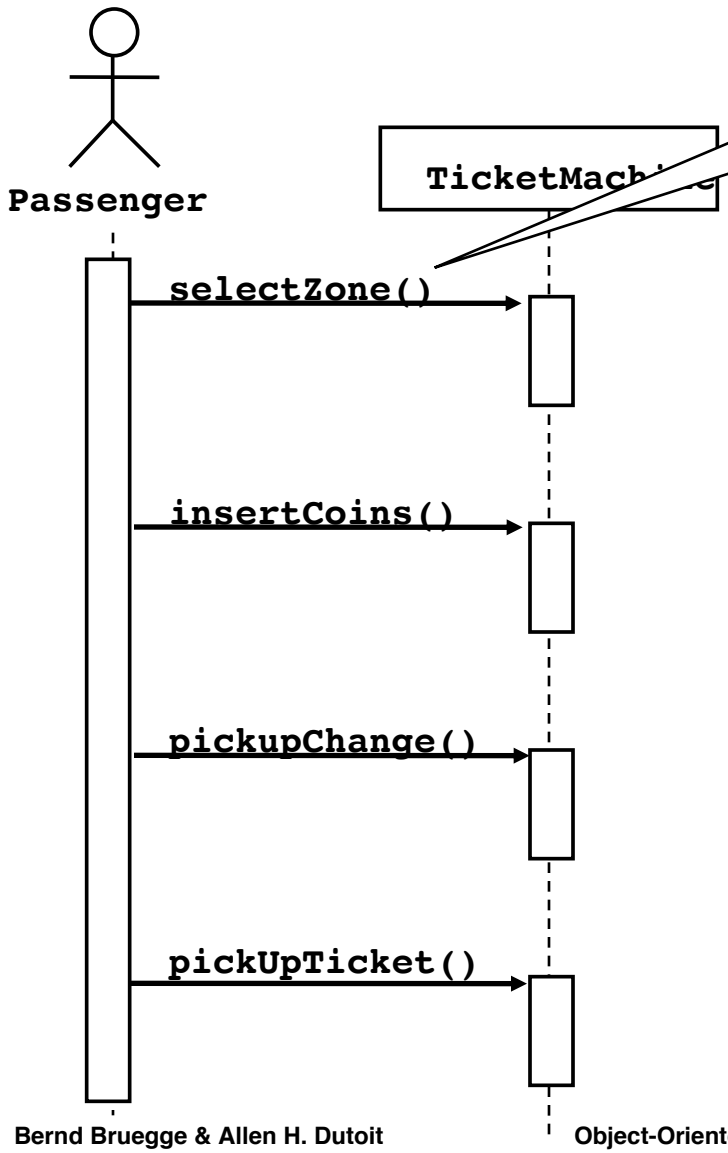
- Any complex system can be decomposed into subsystems, where each subsystem is modeled as a package.

UML first pass: Sequence diagram



Sequence diagrams represent the behavior of a system as messages ("interactions") between *different objects*

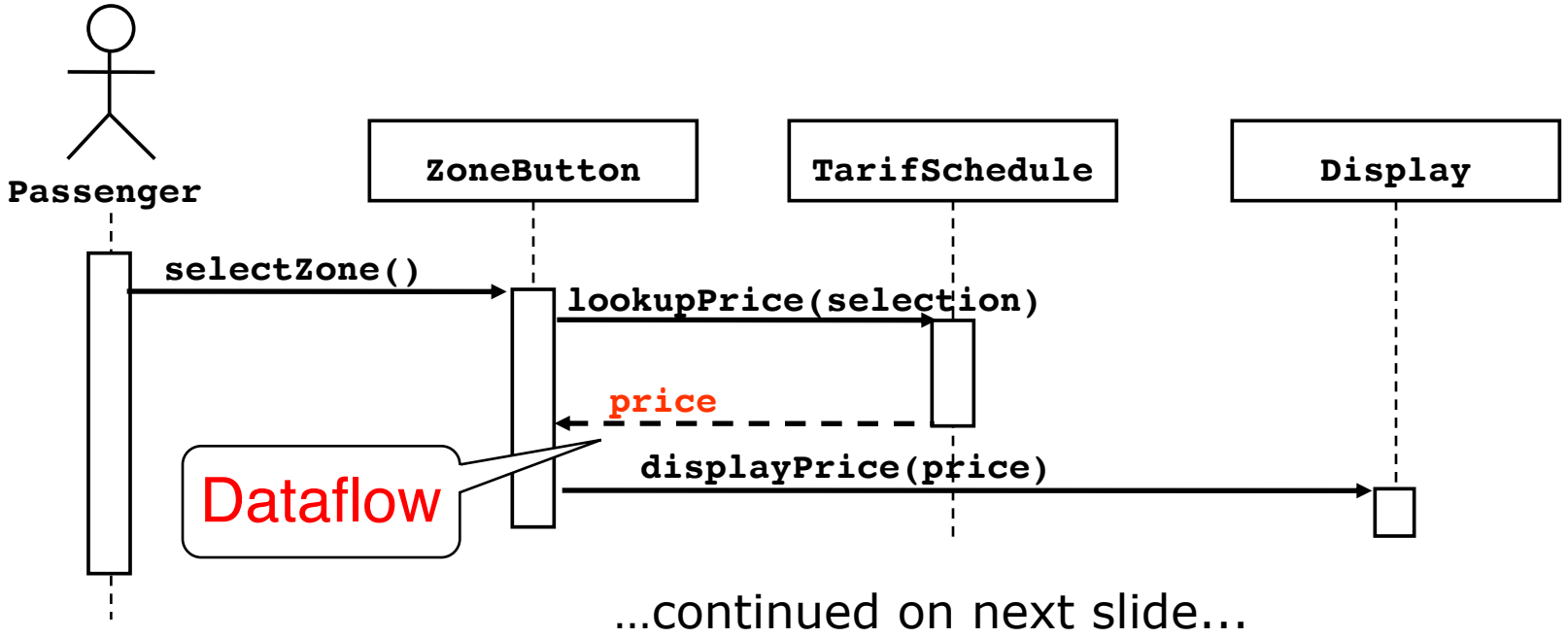
Sequence Diagrams



Focus on Controlflow

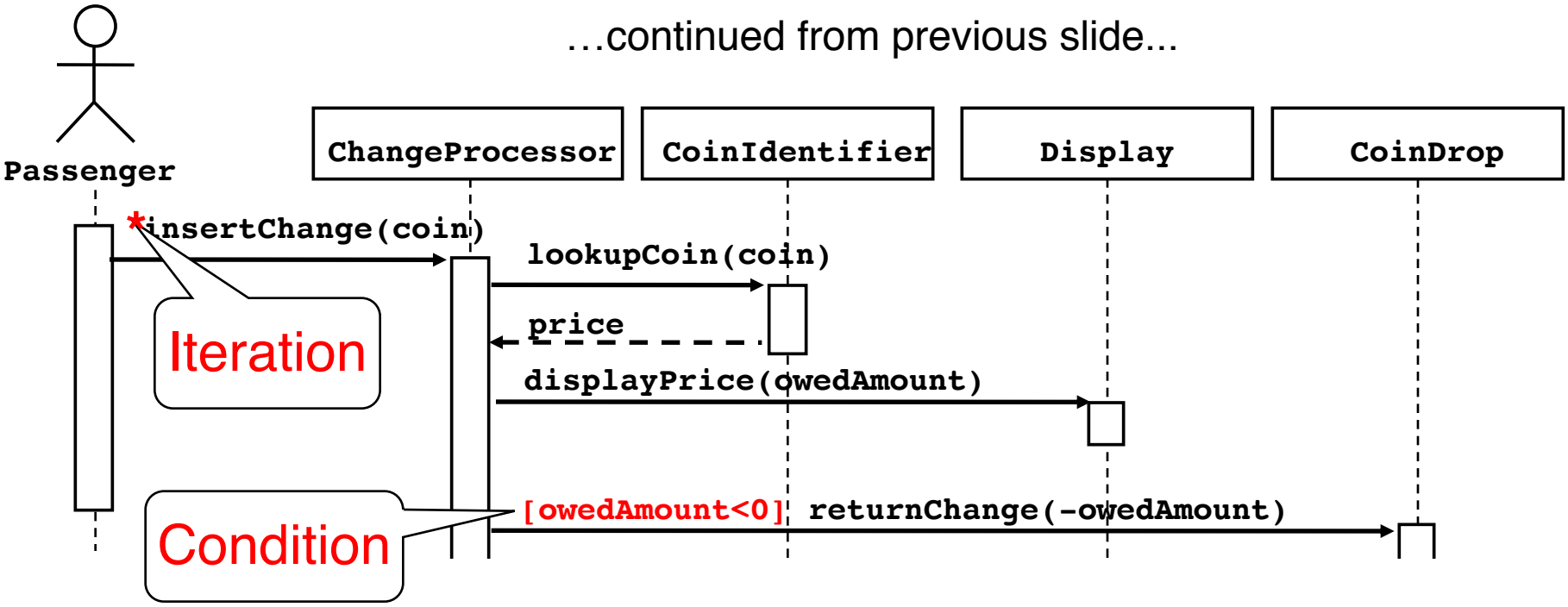
- Used during analysis
  - To refine use case descriptions
  - to find additional objects ("participating objects")
- Used during system design
  - to refine subsystem interfaces
- ***Instances*** are represented by rectangles. ***Actors*** by sticky figures
- ***Lifelines*** are represented by dashed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

Sequence Diagrams can also model the Flow of Data



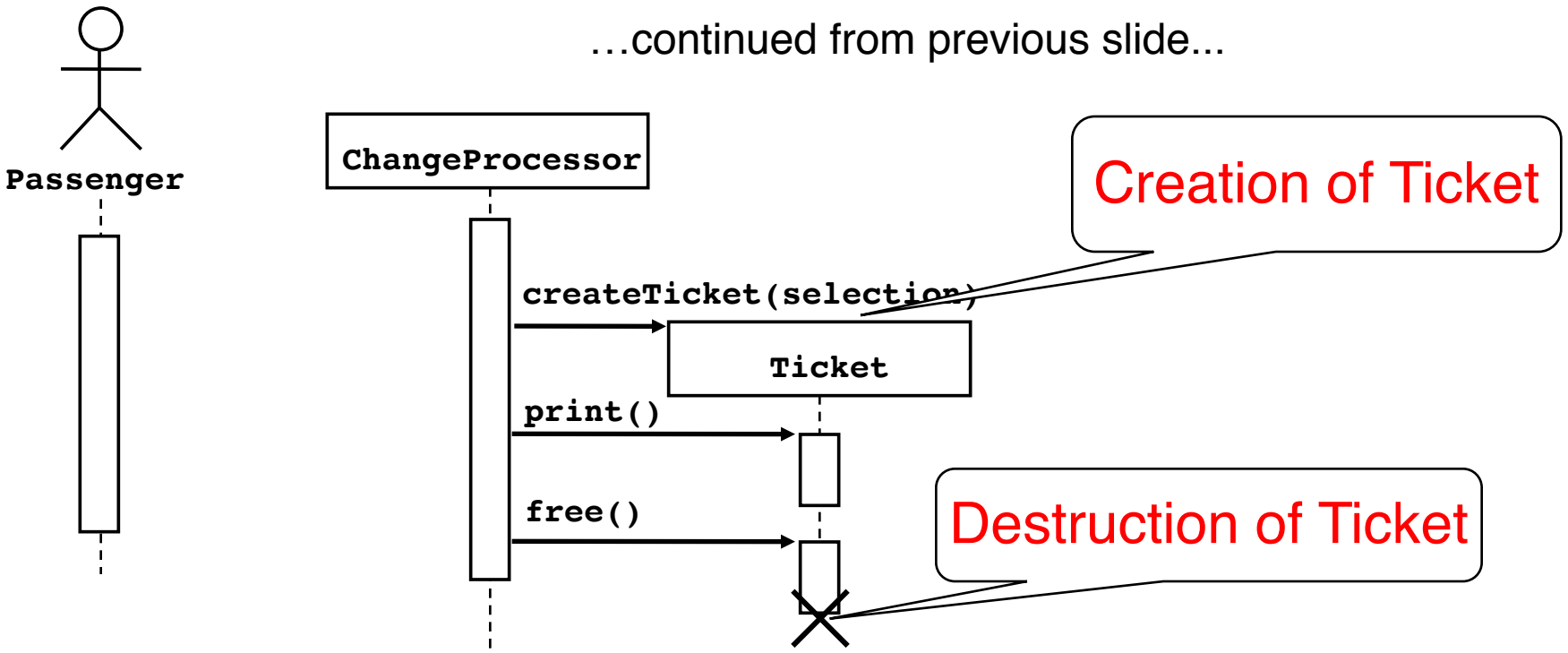
- The source of an arrow indicates the activation which sent the message
- Horizontal dashed arrows indicate data flow, for example return results from a message

Sequence Diagrams: 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

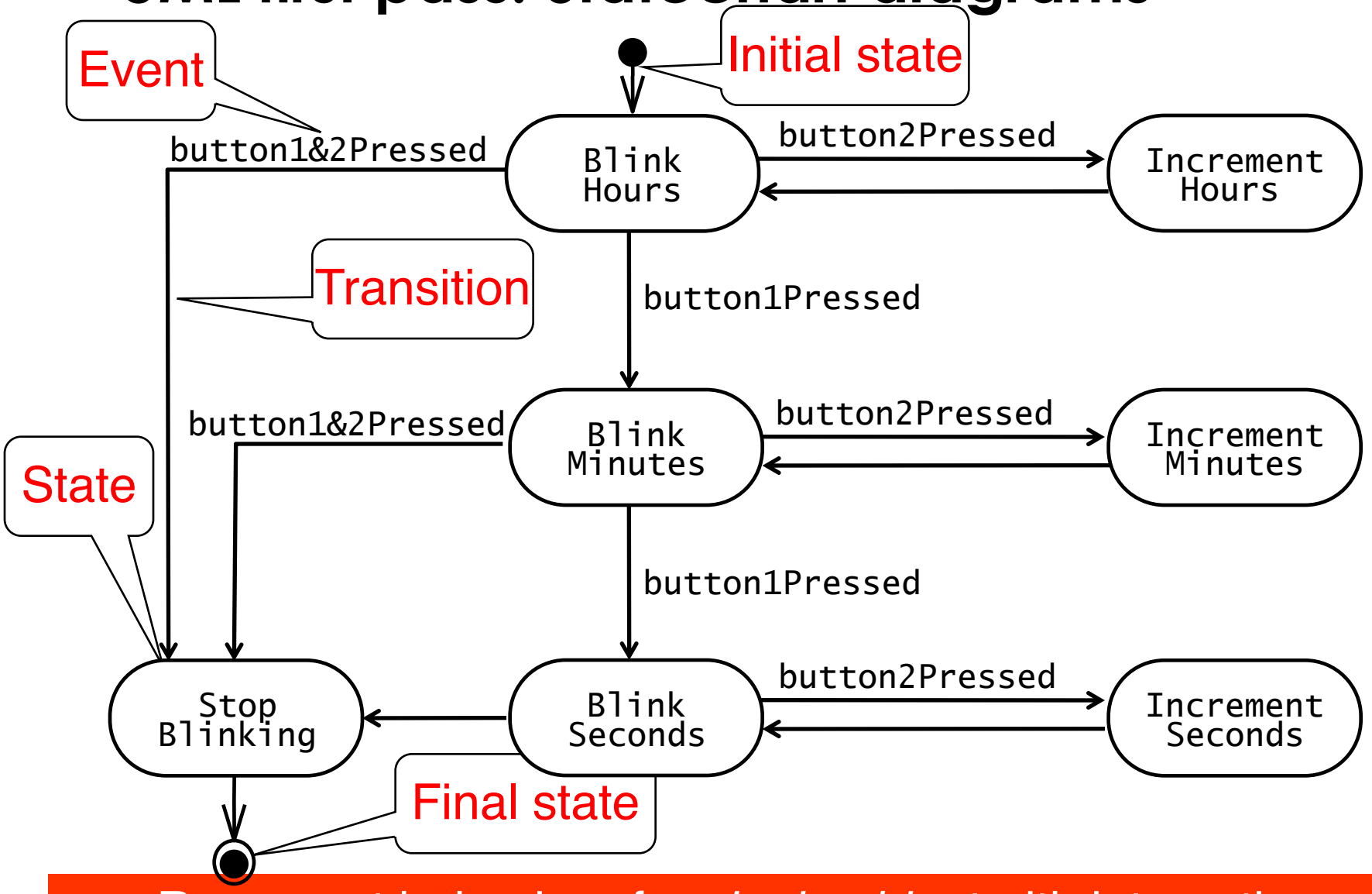


- Creation is denoted by a message arrow pointing to the object
- Destruction is denoted by an X mark at the end of the destruction activation
  - In garbage collection environments, destruction can be used to denote the end of the useful life of an object.

Sequence Diagram Properties

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

UML first pass: Statechart diagrams



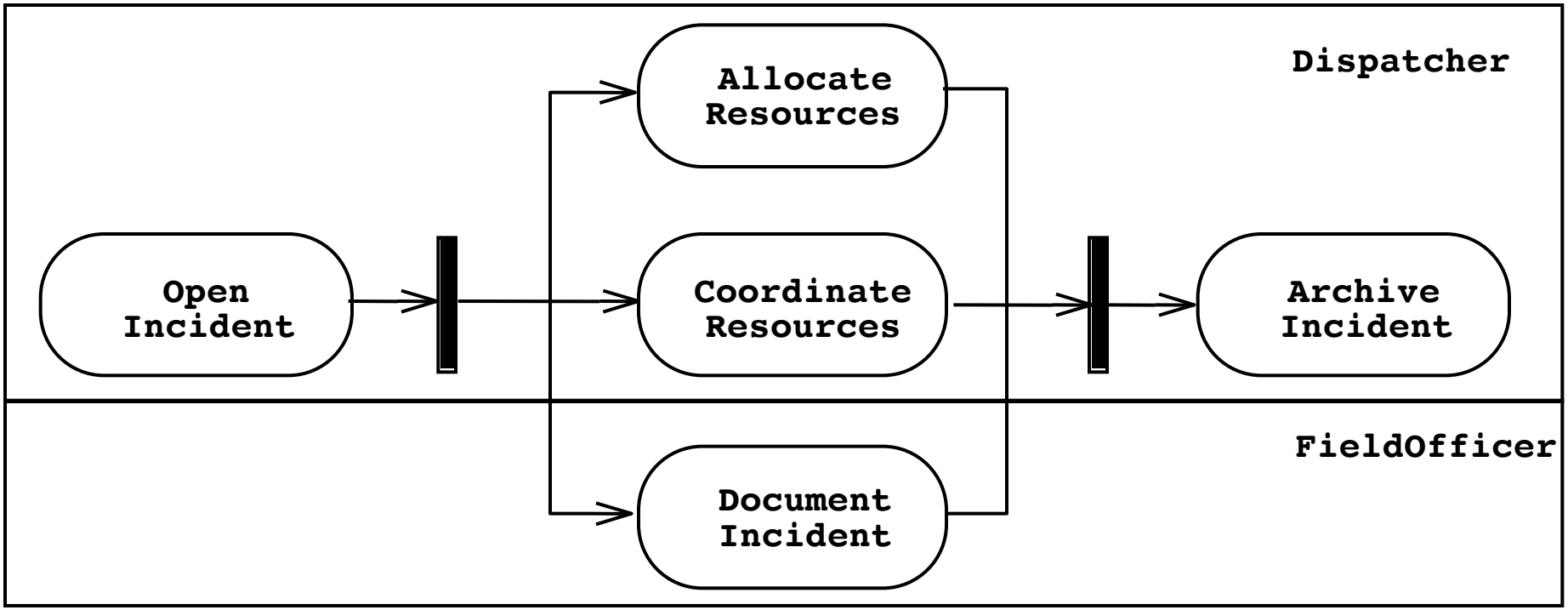
Represent behavior of *a single object* with interesting dynamic behavior.

Activity Diagrams

- An activity diagram is a special case of a state chart diagram
- The states are activities ("functions")
- An activity diagram is useful to depict the workflow in a system

Activity Diagrams: Grouping of Activities

- Activities may be grouped into swimlanes to denote the object or subsystem that implements the activities.



What should be done first? Coding or Modeling?

- It all depends….
- Forward Engineering
  - Creation of code from a model
  - Start with modeling
  - Greenfield projects
- Reverse Engineering
  - Creation of a model from existing code
  - Interface or reengineering projects
- Roundtrip Engineering
  - Move constantly between forward and reverse engineering
  - Reengineering projects
  - Useful when requirements, technology and schedule are changing frequently.