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

Chapter 2, Modeling with UML, Part 1

Software Engineering

Lecture 1.2: Introduction to UML
(Adapted from slides by Bruegge & Dutoit)
Spring 2019

Thomas Troels Hildebrandt, Professor
Software, Data, People & Society Section
Department of Computer Science

UNIVERSITY OF COPENHAGEN

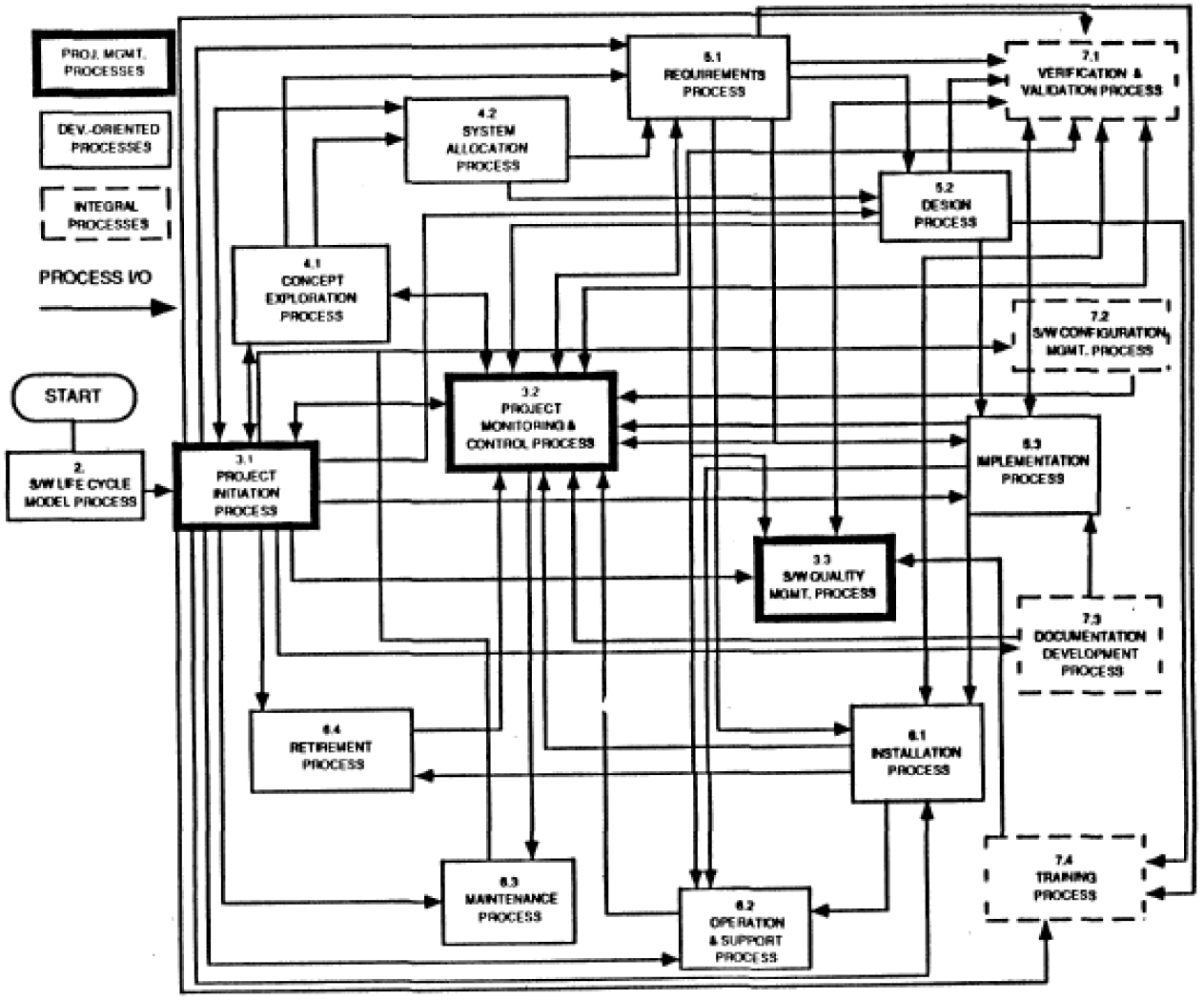
Odds and Ends

- Reading for this Week:
  - Chapter 1 and 2, Bruegge&Dutoit, Object-Oriented Software Engineering
- Access to the Lecture Portal
- Lectures Slides:
  - Will be posted after each lecture.

Overview for the Lecture

- Introduction to the UML notation
- First pass on:
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - Statechart diagrams
  - Activity diagrams
- A bit more details on Use case diagrams

What is the problem with this Drawing?



Abstraction

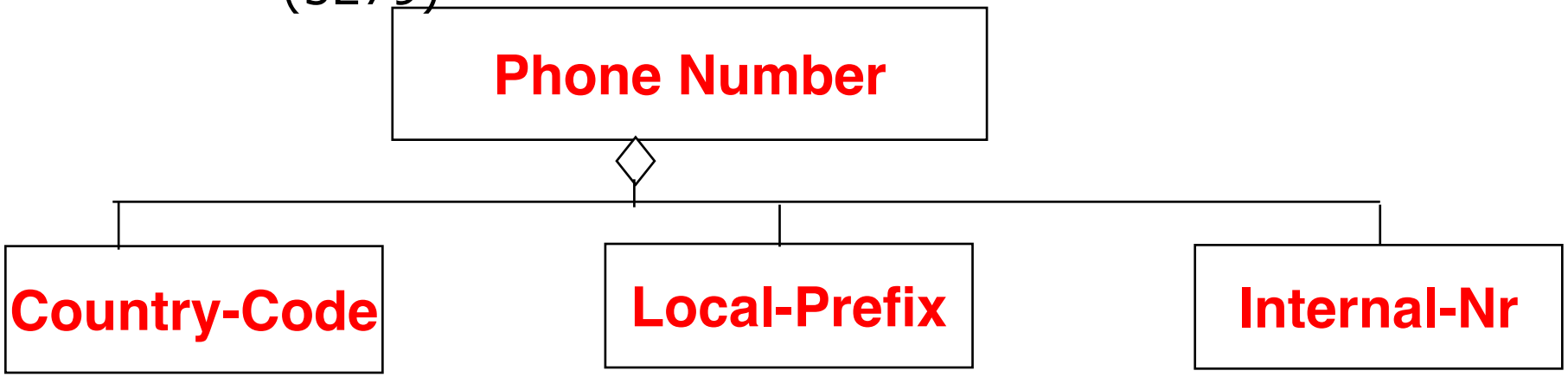
- Complex systems are hard to understand
  - The 7 +- 2 phenomena
    - Our short term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain
    - My Old ITU Phone Number: +4572185279

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 Old ITU Phone Number: +4572185279
- Chunking:
    - Group collection of objects to reduce complexity
    - 3 chunks:
      - Country-code, Local-Prefix, Internal-Nr

Abstraction

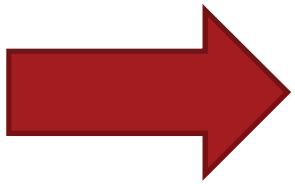
- Complex systems are hard to understand
  - The 7 +- 2 phenomena
    - Our short term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain
    - My Old ITU Phone Number: +4572185279
- Chunking:
    - Group collection of objects to reduce complexity
    - Country-code (45), Local Prefix (7218), Internal-Nr (5279)

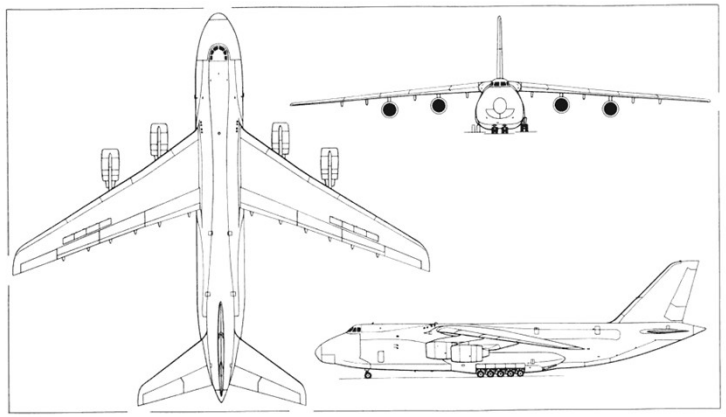


Abstraction and Models

We create **models** of a system to abstract away from its complexity to something manageable and understandable.

By using formal and standardized **notations** for our models we avoid ambiguity.





Models must be falsifiable

- Karl Popper ("Objective Knowledge):
  - There is no absolute truth when trying to understand reality
  - One can only build theories, that are "true" until somebody finds a counter example
- Falsification: The act of disproving a theory or hypothesis
- The truth of a theory is never certain. We must use phrases like:
  - "by our best judgement", "using state-of-the-art knowledge"
- In software engineering any model is a theory:
  - We build models and try to find counter examples by:
    - Requirements validation, user interface testing, review of the design, source code testing, system testing, etc.
- Testing: The act of disproving a model.

Abstraction

- Abstraction allows us to ignore unessential details
- Two definitions for abstraction:
  - Abstraction is a *thought process* where ideas are distanced from objects
    - **Abstraction as activity**
  - Abstraction is the *resulting idea* of a thought process where an idea has been distanced from an object
    - **Abstraction as entity**
- Ideas can be expressed by models

Models

- A model is an abstraction of a system
  - A system that no longer exists
  - An existing system
  - A future system to be built.

Another Example of a System to be Built

http://www.youtube.com/watch?v=S6Eu-Uh69BE

We use Models to describe Software Systems

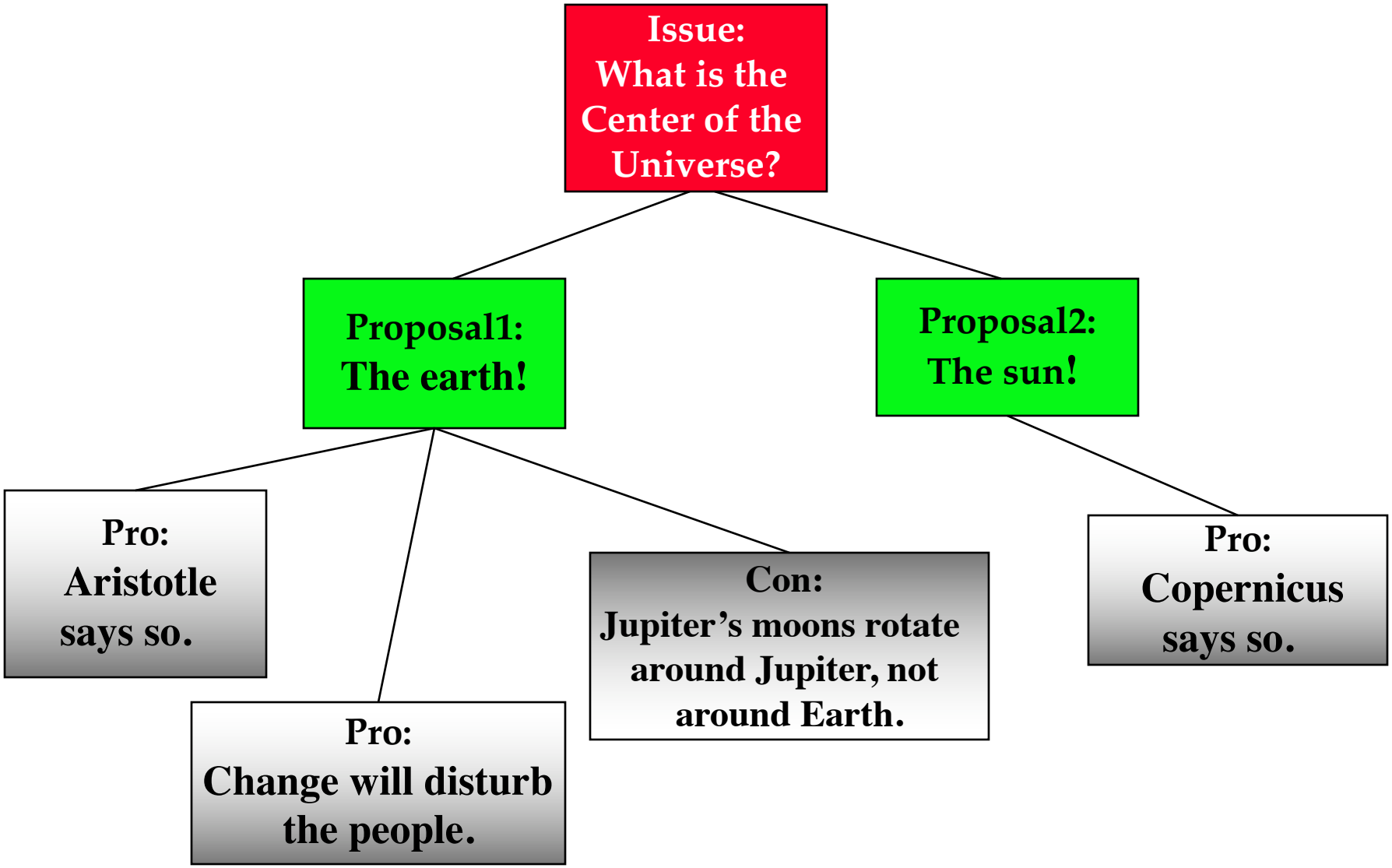
- Object model: What is the structure of the system?
- Functional model: What are the functions of the system?
- Dynamic model: How does the system react to external events?

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

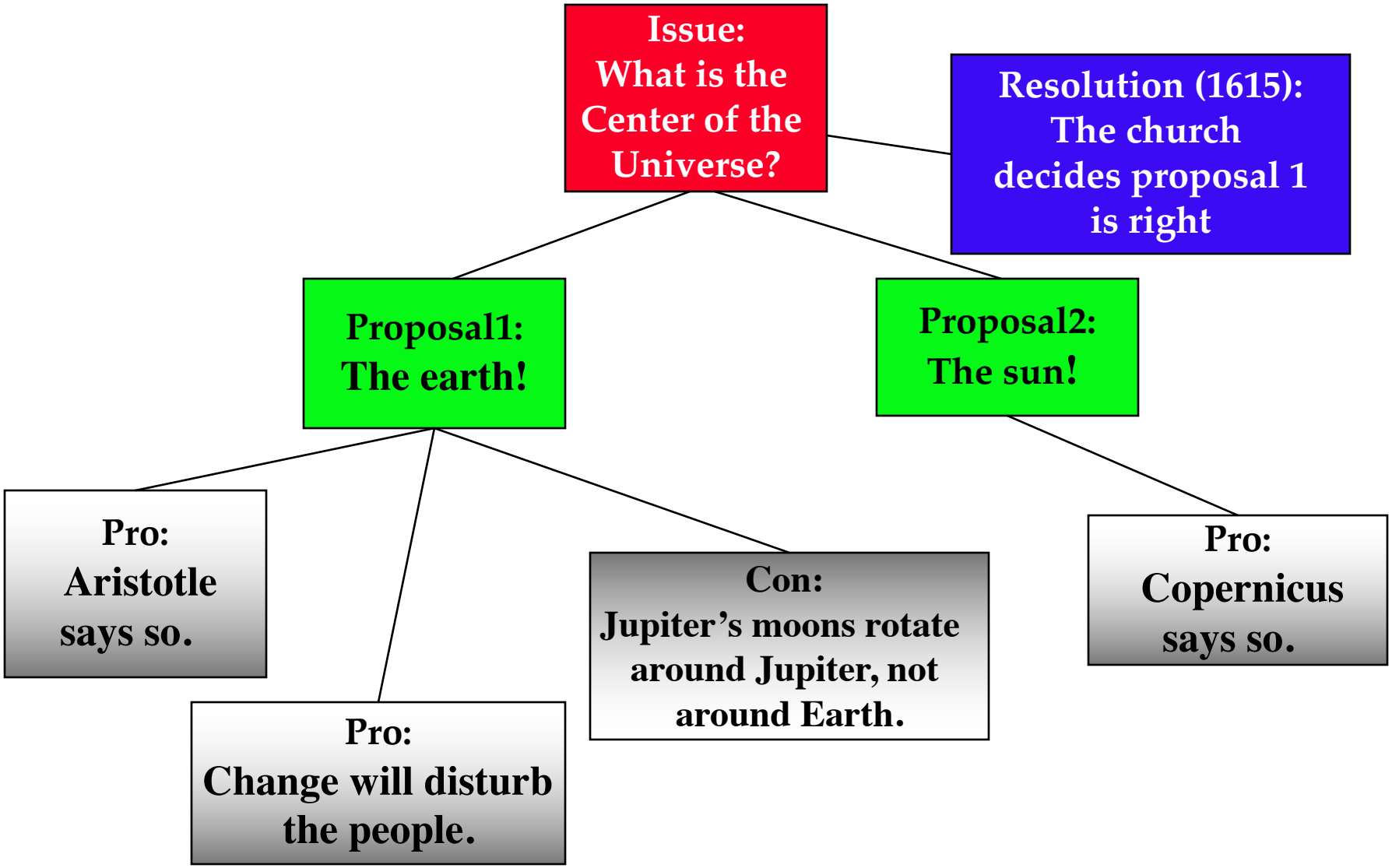
Other models used to describe Software System Development

- Task Model:
  - PERT Chart: What are the dependencies between tasks?
  - Schedule: How can this be done within the time limit?
  - Organization Chart: What are the roles in the project?
- Issues Model:
  - What are the open and closed issues?
    - What blocks me from continuing?
  - What constraints were imposed by the client?
  - What resolutions were made?
    - These lead to action items

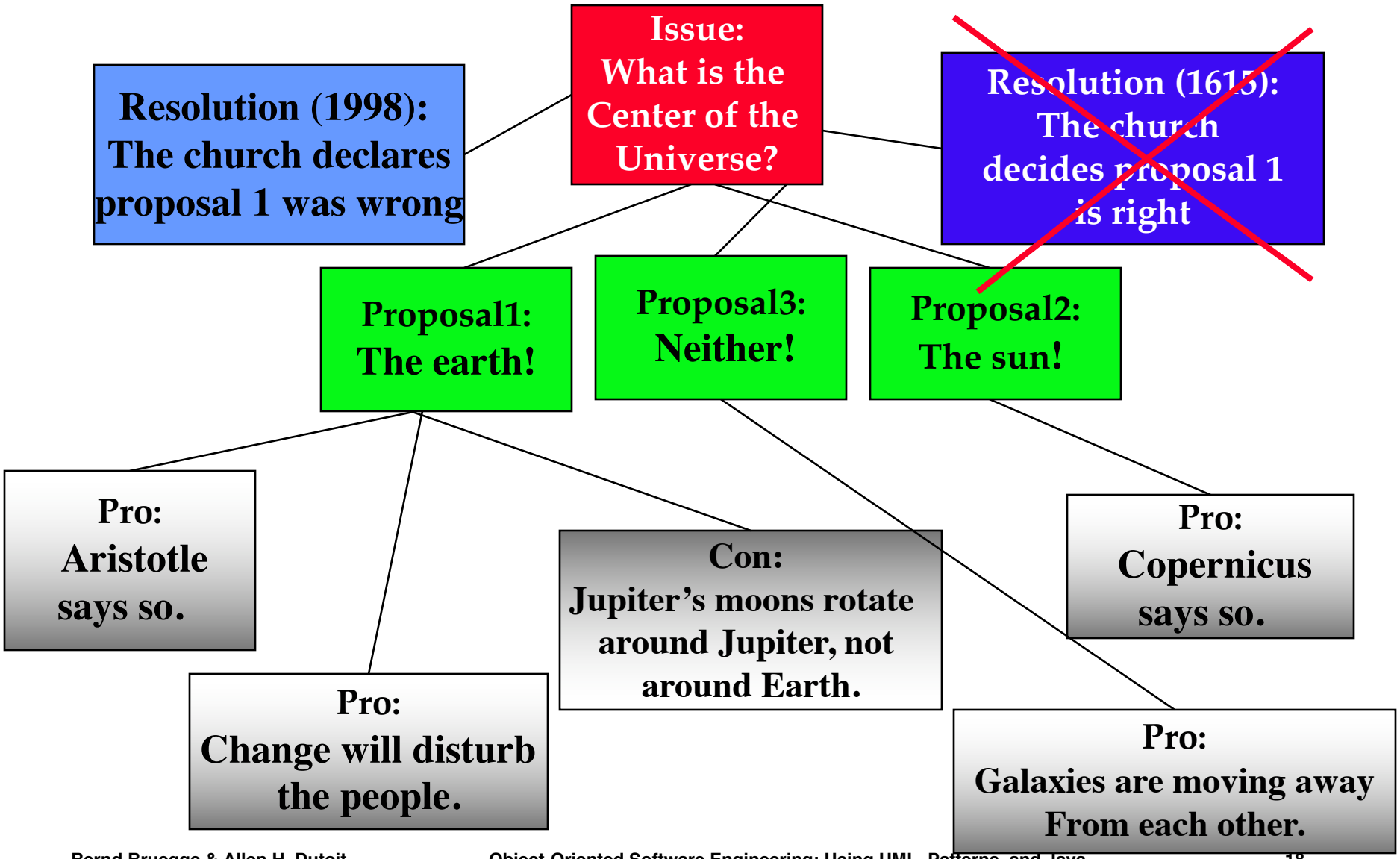
Issue-Modeling



Issue-Modeling



Issue-Modeling



2. Technique to deal with Complexity: 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.

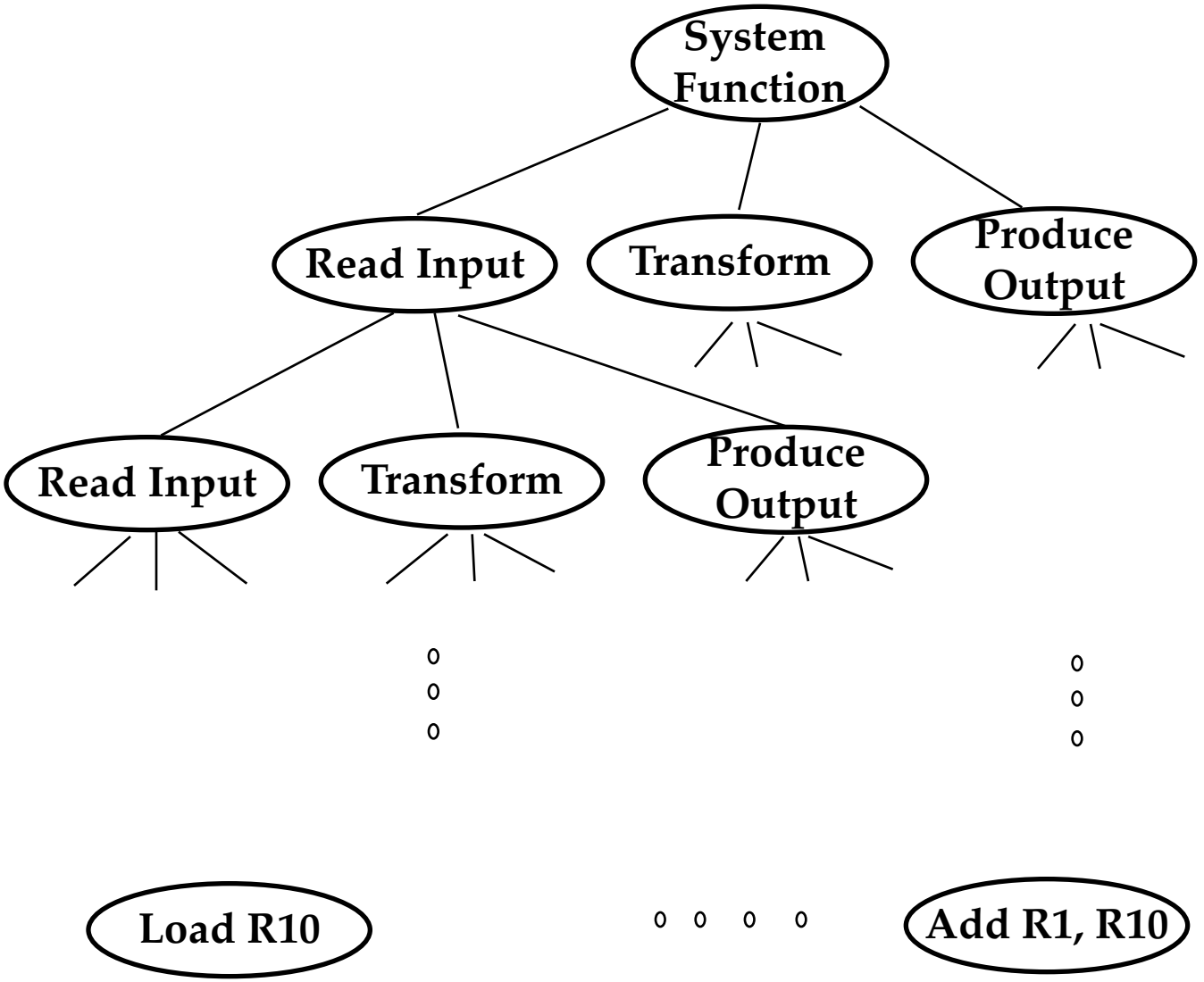
Decomposition (cont'd)

- 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

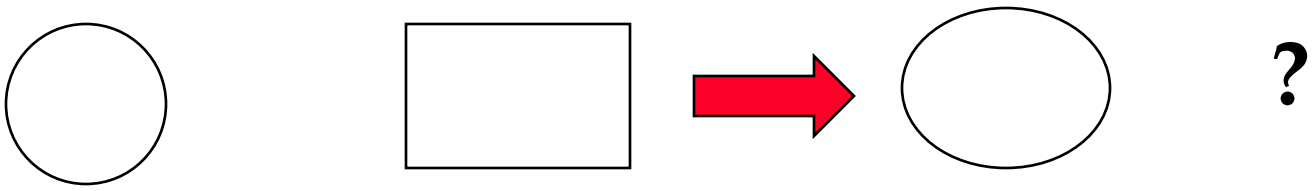
Functional Decomposition

- 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.

Functional Decomposition

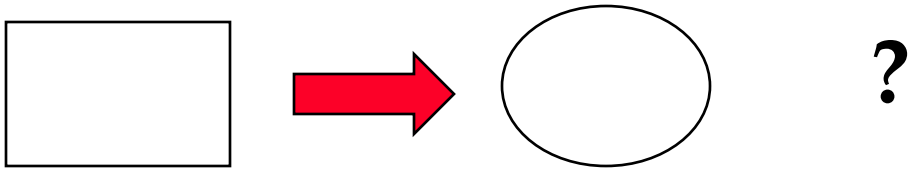
- The functionality is spread all over the system
- Maintainer must understand the whole system to make a single change to the system
- Consequence:
  - Source code is hard to understand
  - Source code is complex and impossible to maintain
  - User interface is often awkward and non-intuitive

- Example: Microsoft Powerpoint's Autoshapes
  - How do I change a square into a circle?

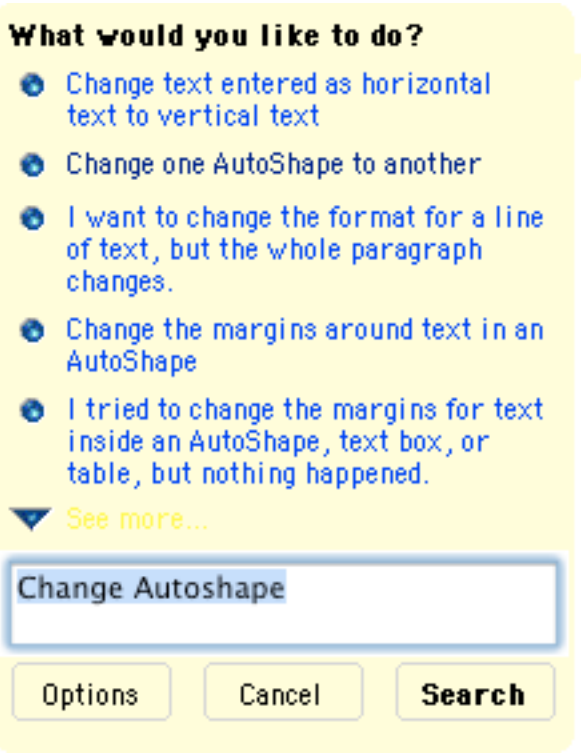


Changing a Square into a Circle

First Attempt:
Check the Format Menu: Autoshape



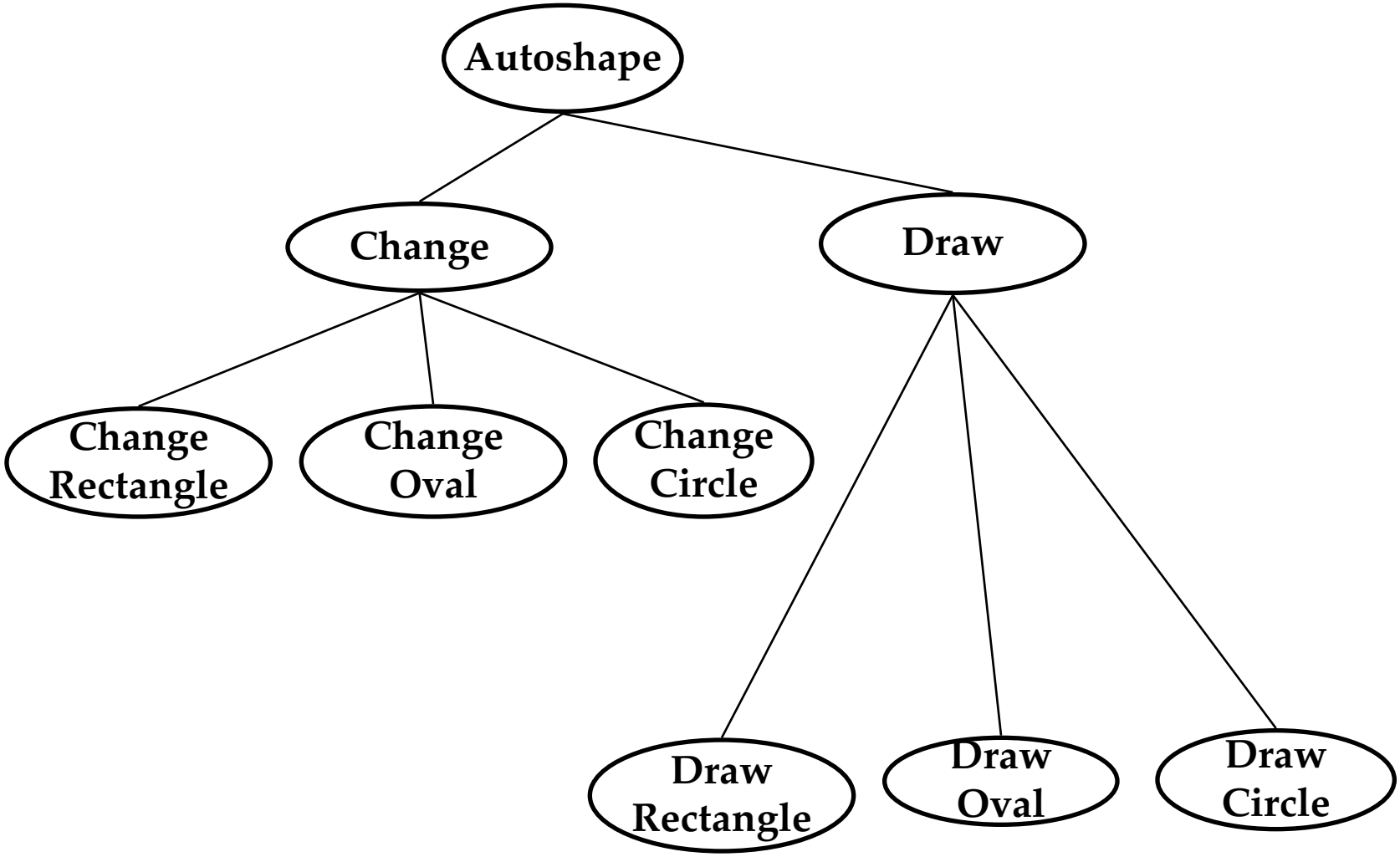
Second Attempt: Ask the Help Assistant



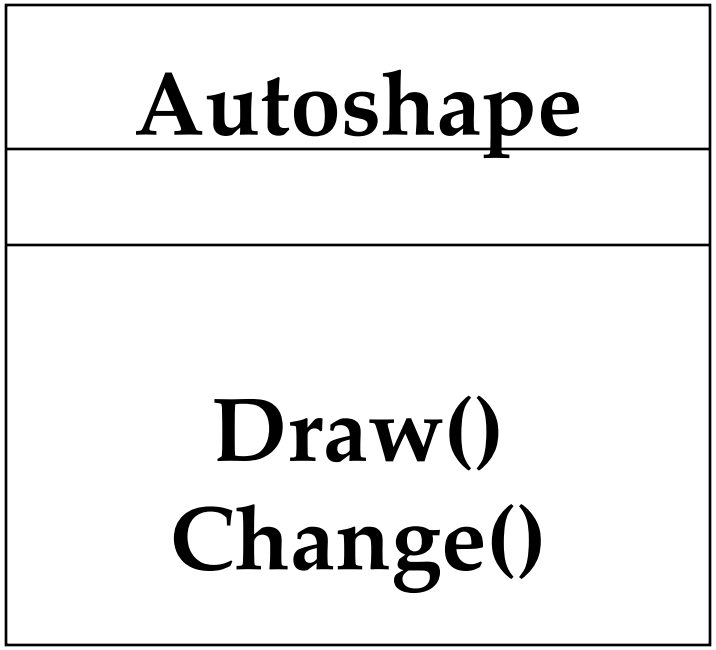
Change one AutoShape to another:

1. Select the AutoShape you want to change.
2. On the Drawing toolbar, click Draw , click Change AutoShape, point to a category, and then click the shape you want.

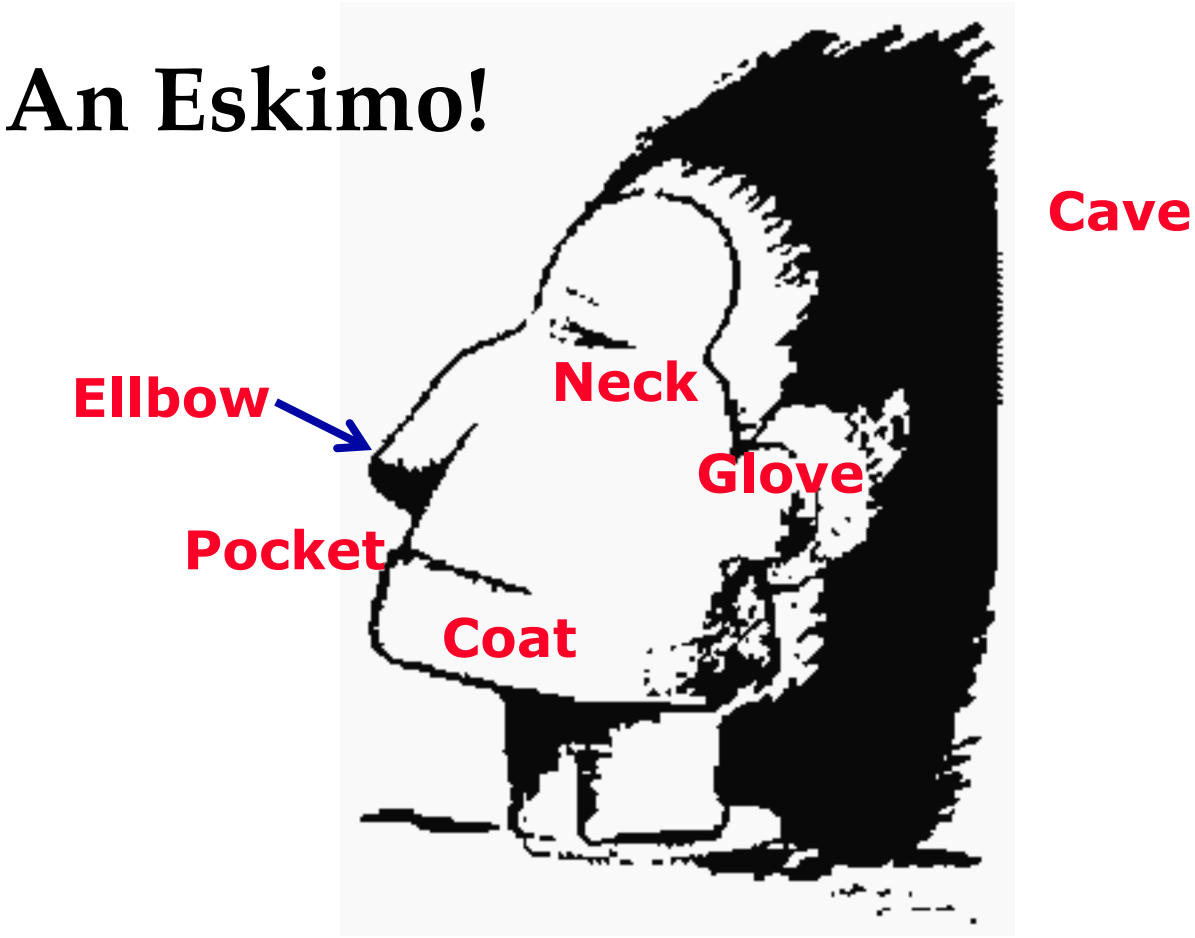
Functional Decomposition: Autoshape



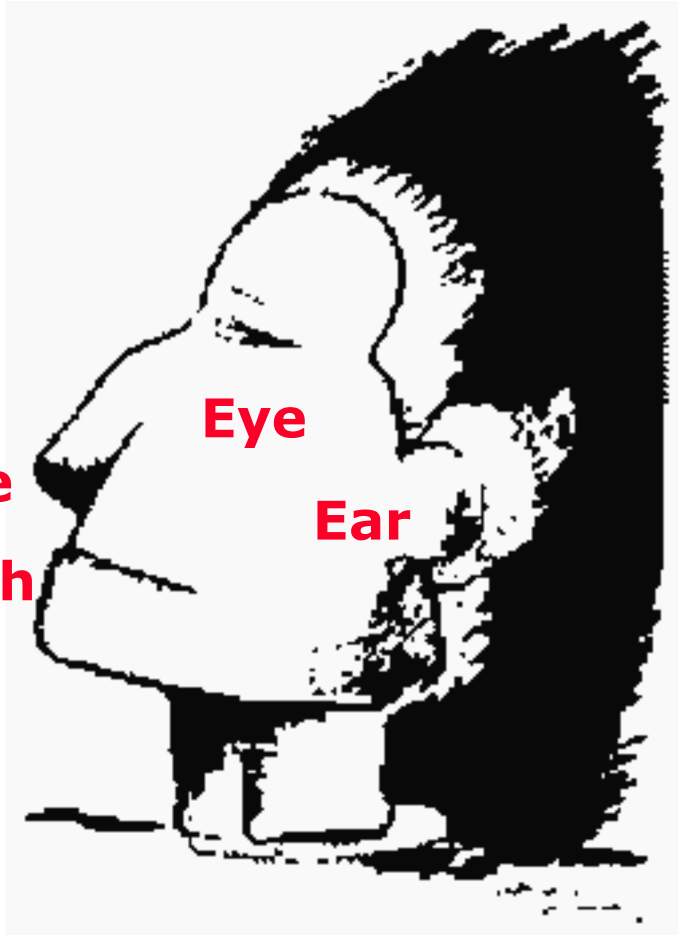
Object-Oriented View



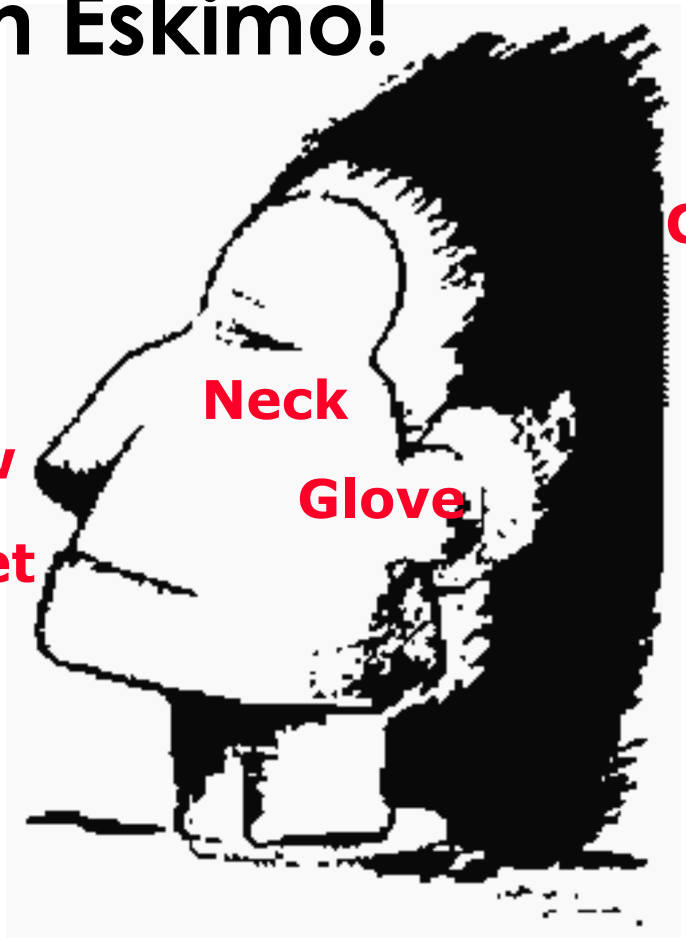
What is This?



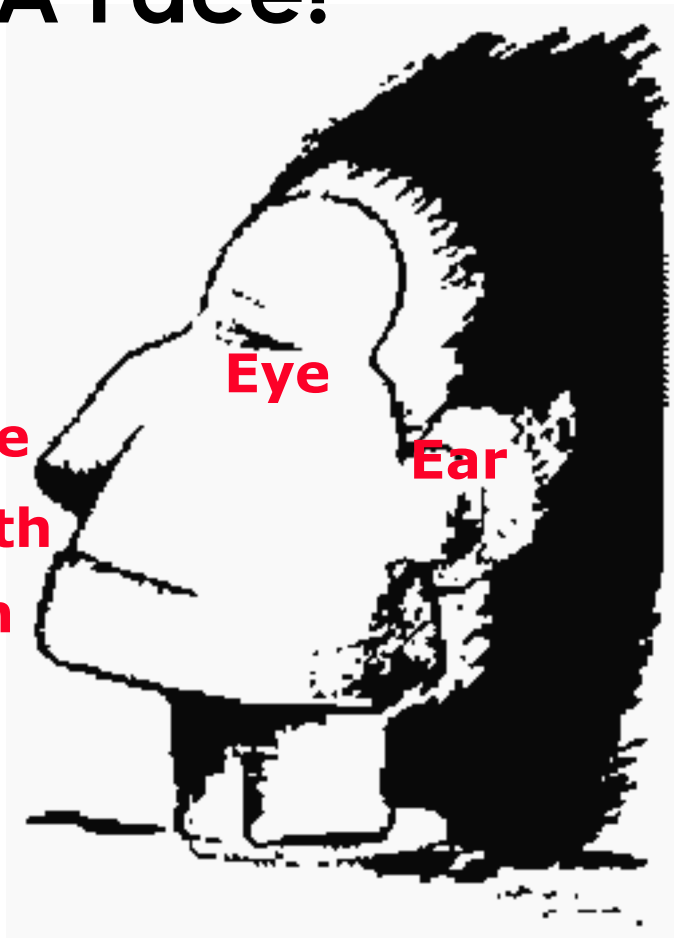
A Face!



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.

Where are we?

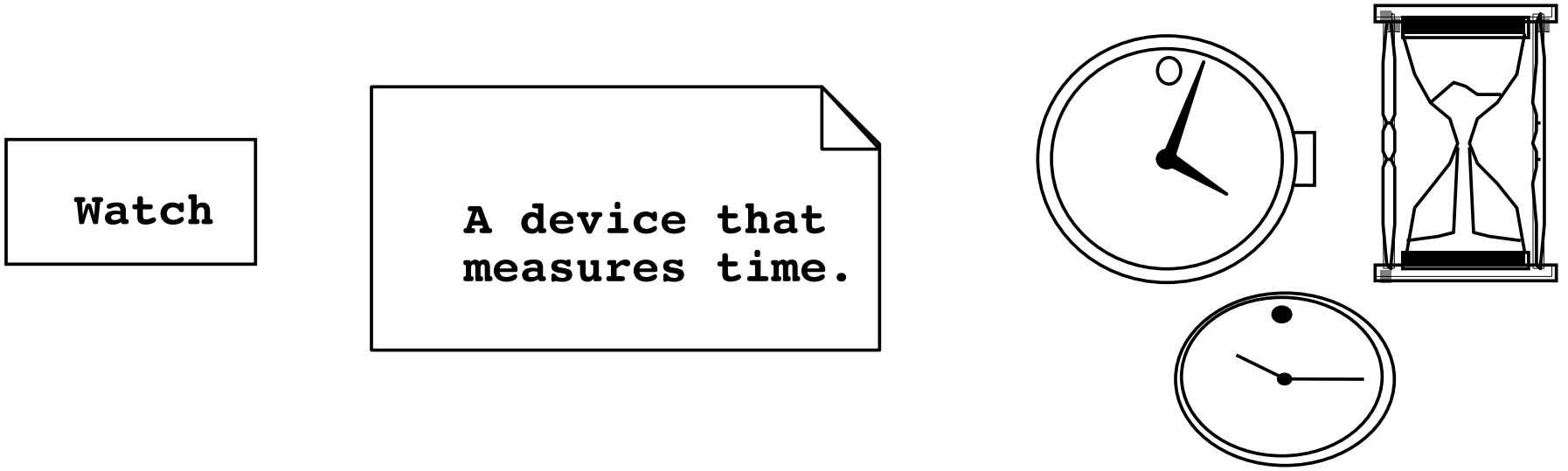
- Three ways to deal with complexity:
  - Abstraction, Decomposition, Hierarchy
- Object-oriented decomposition is fairly good
  - Unfortunately, depending on the purpose of the system, different objects can be found
- How can we do it right?
  - Start with a description of the functionality of a system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

Concepts and Phenomena

- Phenomenon
  - An object in the world of a domain as you perceive it
    - Examples: This lecture, 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:

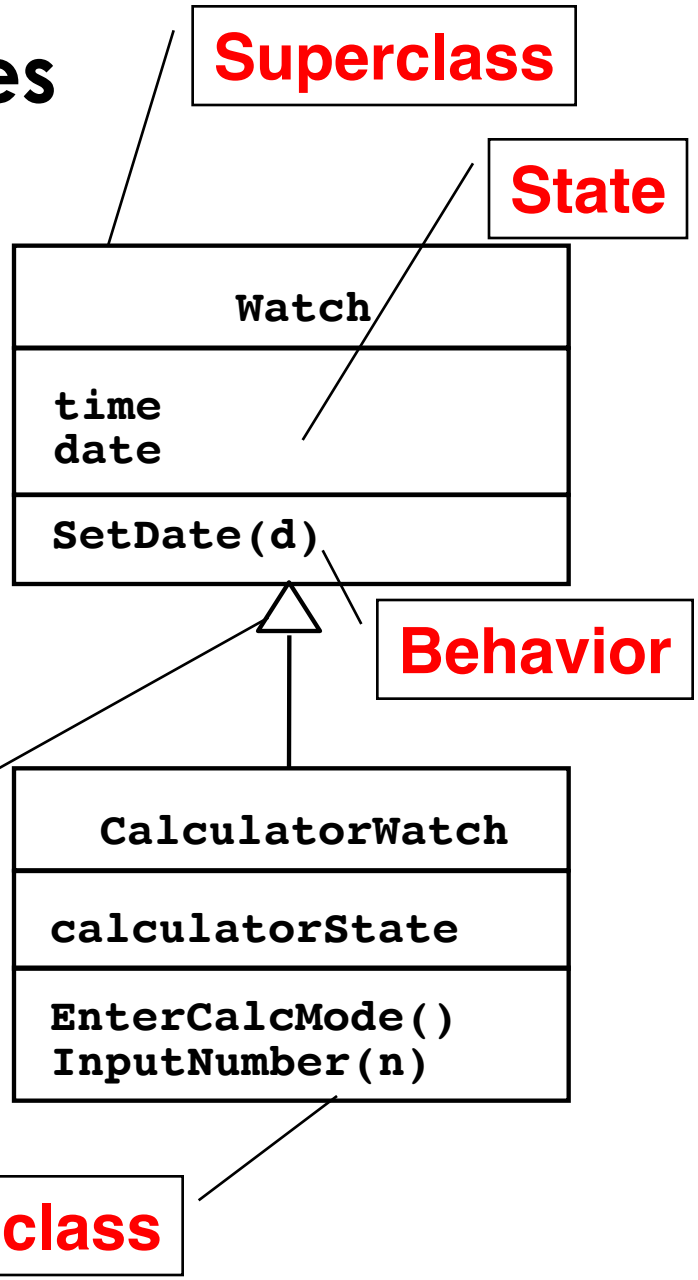
- Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Abstract Data Types & Classes

- **Abstract data type**
  - A type whose implementation is hidden from the rest of the system
- **Class:**
  - An abstraction in the context of object-oriented languages
  - A class encapsulates state and behavior
    - Example: Watch

Subclasses can be defined in terms of other classes using inheritance

- Example: CalculatorWatch



Type and Instance

- Type:
  - An concept in the context of programming languages
  - Name: int
  - Purpose: integral number
  - Members: `0, -1, 1, 2, -2,…`
- Instance:
  - Value/Member of a specific type

- The type of a variable represents all possible instances, i.e. values, the variable can hold

The following relationships are similar:

Type <-> Value
Concept <-> Phenomenon
Class <-> Object

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:

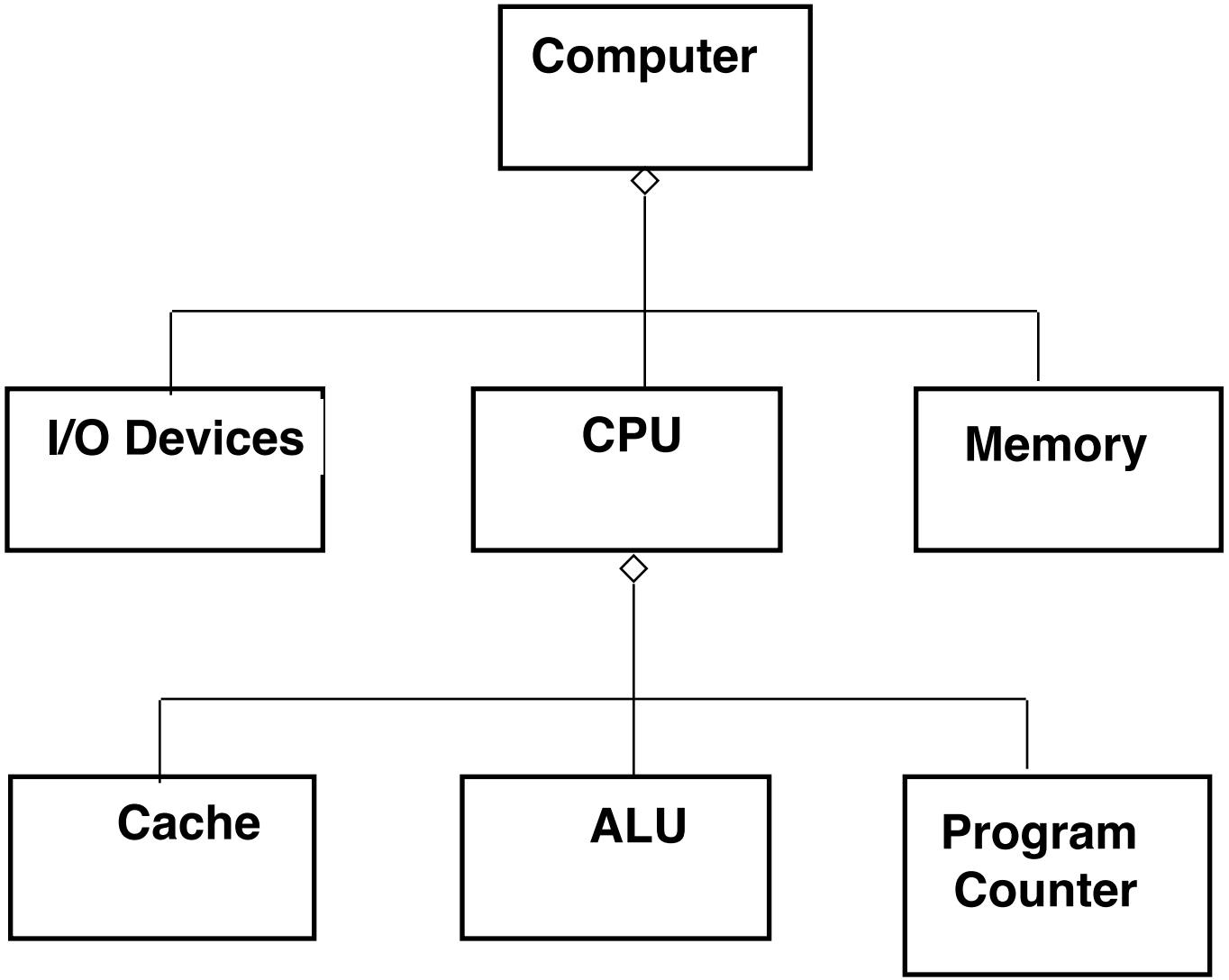
- Jet engine, battery, satellite.

3. Hierarchy

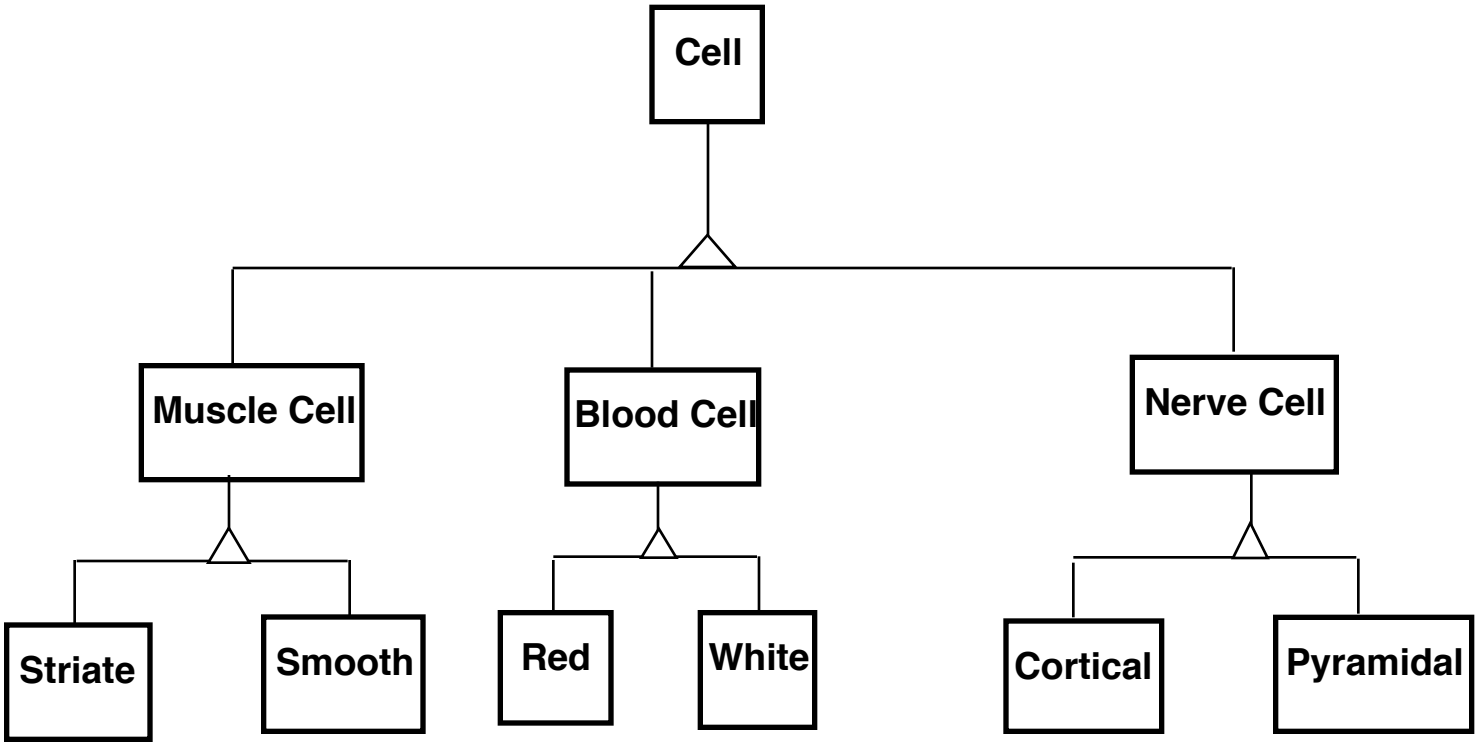
- So far we got abstractions
  - This leads us to classes and objects
  - “Chunks”

- 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)



Systems, Models and Views

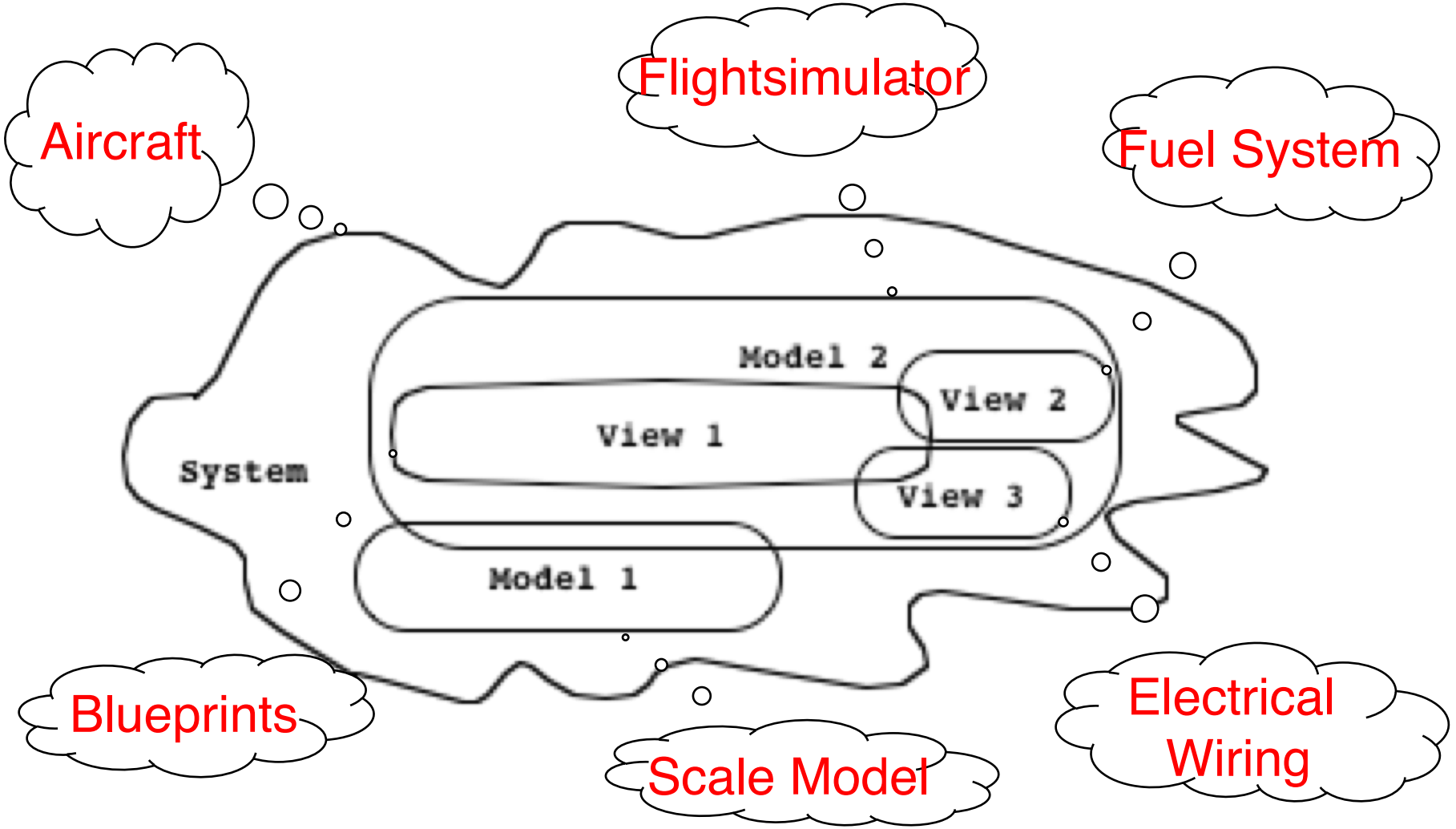
- A ***model*** is an abstraction describing a system or a subsystem
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting models and vi
  - formal notations, "napkin de

System: Airplane

Models:
Flight simulator
Scale model

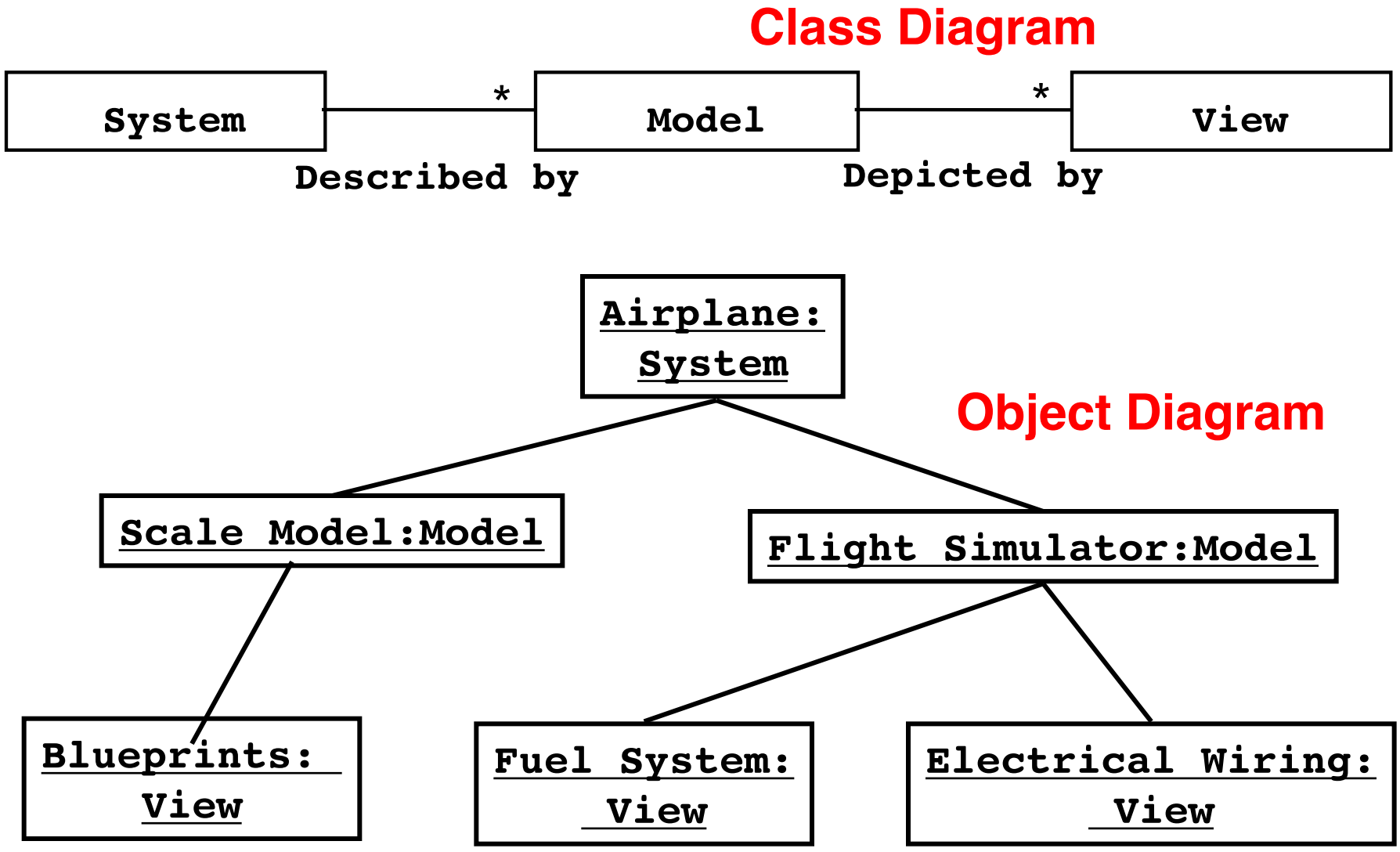
Views:
Blueprint of the airplane components
Electrical wiring diagram, Fuel system
Sound wave created by airplane

Systems, Models and Views ("Napkin" Notation)



Views and models of a complex system usually overlap

Systems, Models and Views (UML Notation)



Model-Driven Development

1. Build a platform-independent model of an applications functionality and behavior
   a) Describe model in modeling notation (UML)
   b) Convert model into platform-specific model
2. Generate executable from platform-specific model

Advantages:

- Code is generated from model ("mostly")
- Portability and interoperability

- Model Driven Architecture effort:
  - http://www.omg.org/mda/
- OMG: Object Management Group

Model-driven Software Development

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

Analysis results in analysis object model (UML Class Diagram):

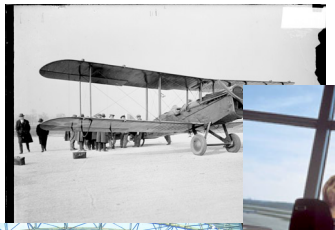
Implementation results in source code (Java):

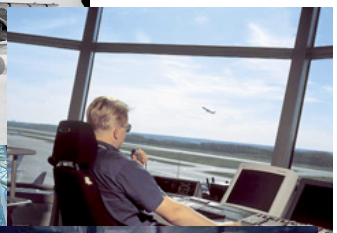
```
public class StockExchange {
    public m_Company = new Vector();
 };
public class Company  {
 public int m_tickerSymbol;
 public Vector m_StockExchange = new Vector();
};
```

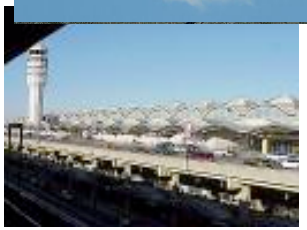
Application vs Solution Domain

- 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.

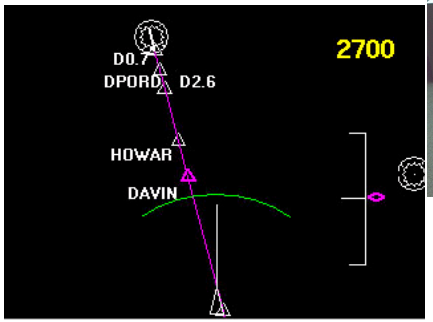
Object-oriented Modeling

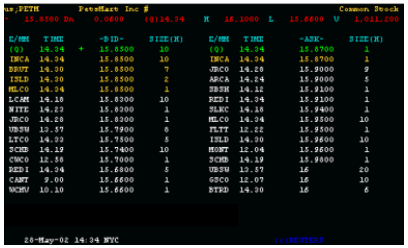






Application Domain
(Phenomena)





Solution Domain
(Phenomena)

System Model (Concepts) *(Analysis)*

UML Package

TrafficControl

Aircraft

TrafficController

Airport

FlightPlan

System Model (Concepts) *(Design)*

MapDisplay

Summary Display

FlightPlanDatabase

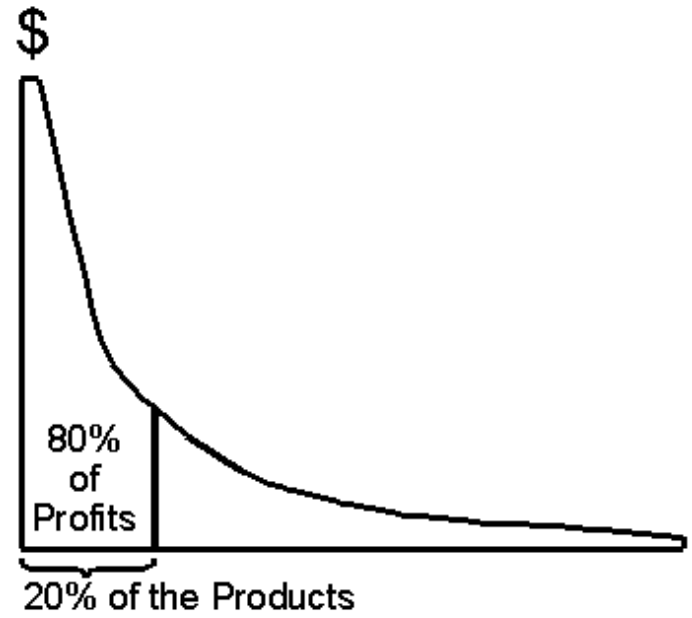
TrafficControl

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 (December 2017)
  - 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 solve 80% of the modeling problems by using 20 % UML
- We teach you those 20%
- 80-20 rule: Pareto principle



Vilfredo Pareto, 1848-1923
Introduced the concept of Pareto Efficiency,
Founder of the field of microeconomics.

What is UML? Unified Modeling Language

- Convergence of different notations used in object-oriented methods, mainly
  - OMT (James Rumbaugh and collegues), OOSE (Ivar Jacobson), Booch (Grady Booch)
- They also developed the Rational Unified Process, which became the Unified Process in 1999

25 year at GE Research, where he developed OMT, joined (IBM) Rational in 1994, CASE tool OMTool

At Ericsson until 1994, developed use cases and the CASE tool Objectory, at IBM Rational since 1995, http://www.ivarjacobson.com

Developed the Booch method ("clouds"), ACM Fellow 1995, and IBM Fellow 2003 http://www.booch.com/

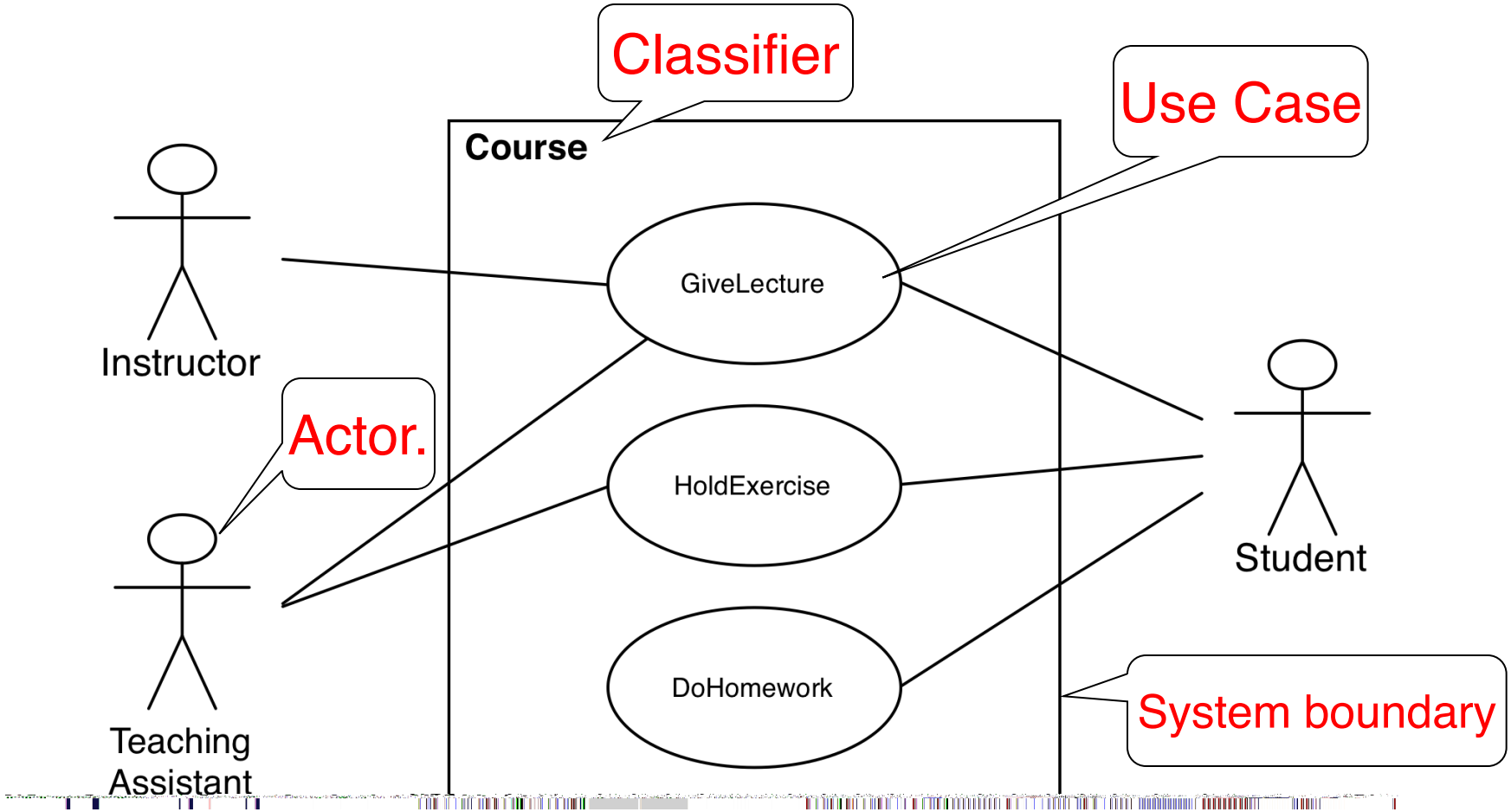
UML First Pass

- 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

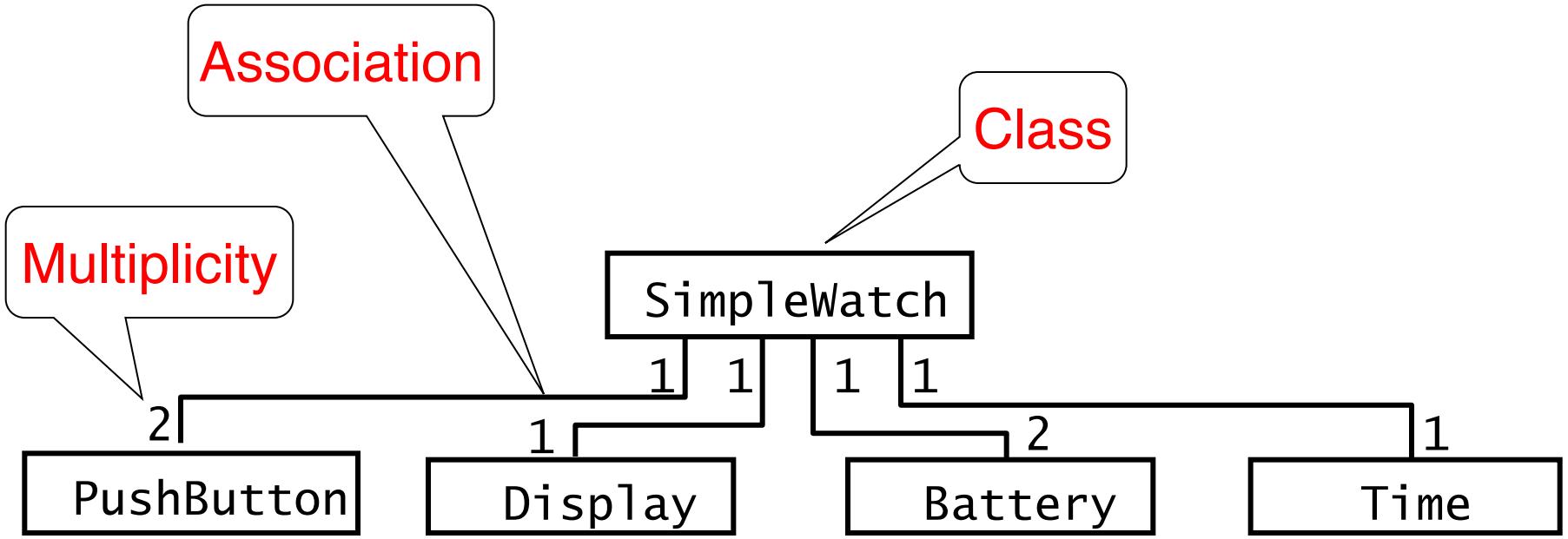
- All UML Diagrams denote graphs of nodes and edges
  - Nodes are entities and drawn as rectangles or ovals
  - Rectangles denote classes or instances
  - Ovals denote functions
- Names of Classes are not underlined
  - `SimpleWatch`
  - `Firefighter`
- Names of Instances are underlined
  - `myWatch:SimpleWatch`
  - `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

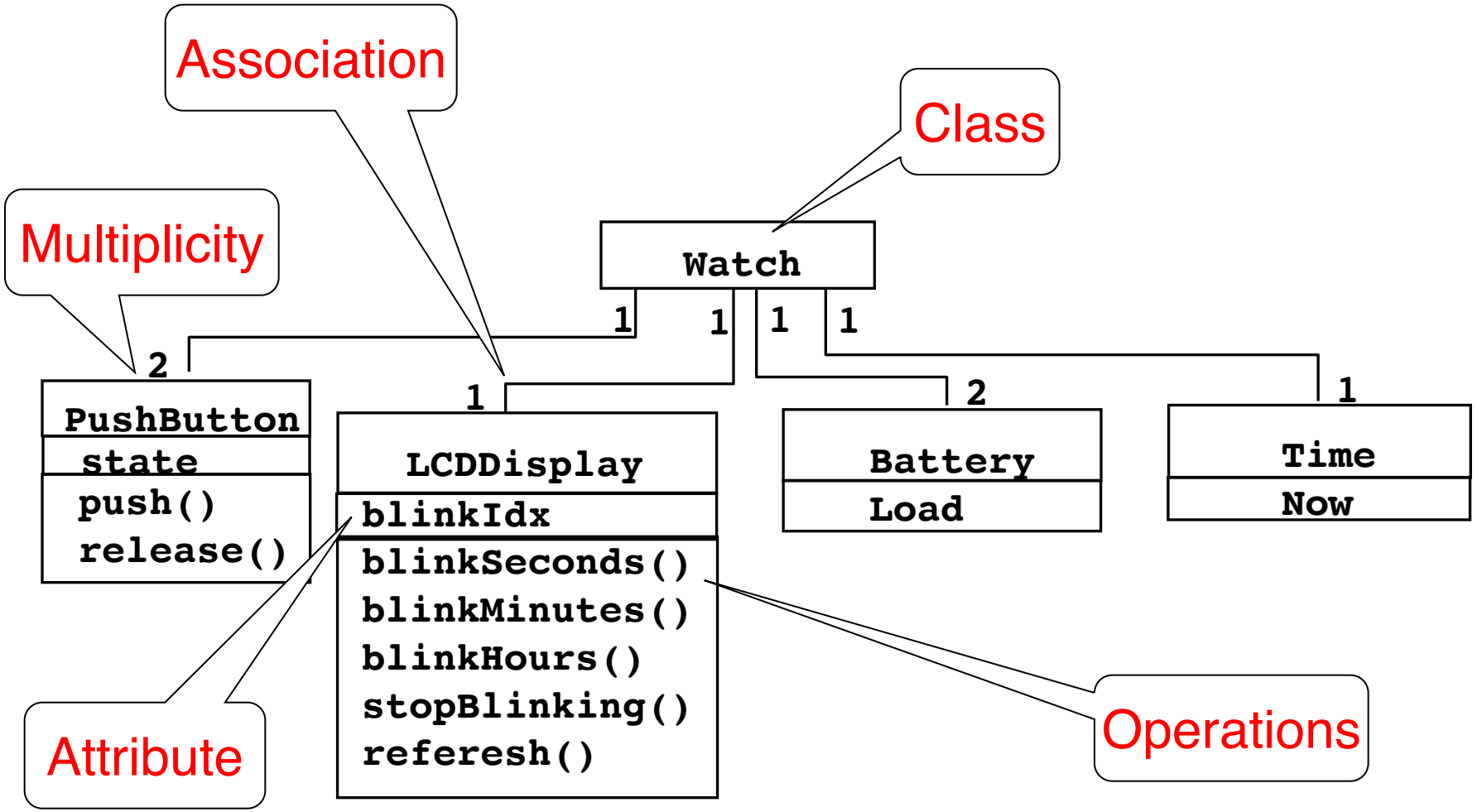
UML first pass: Class diagrams



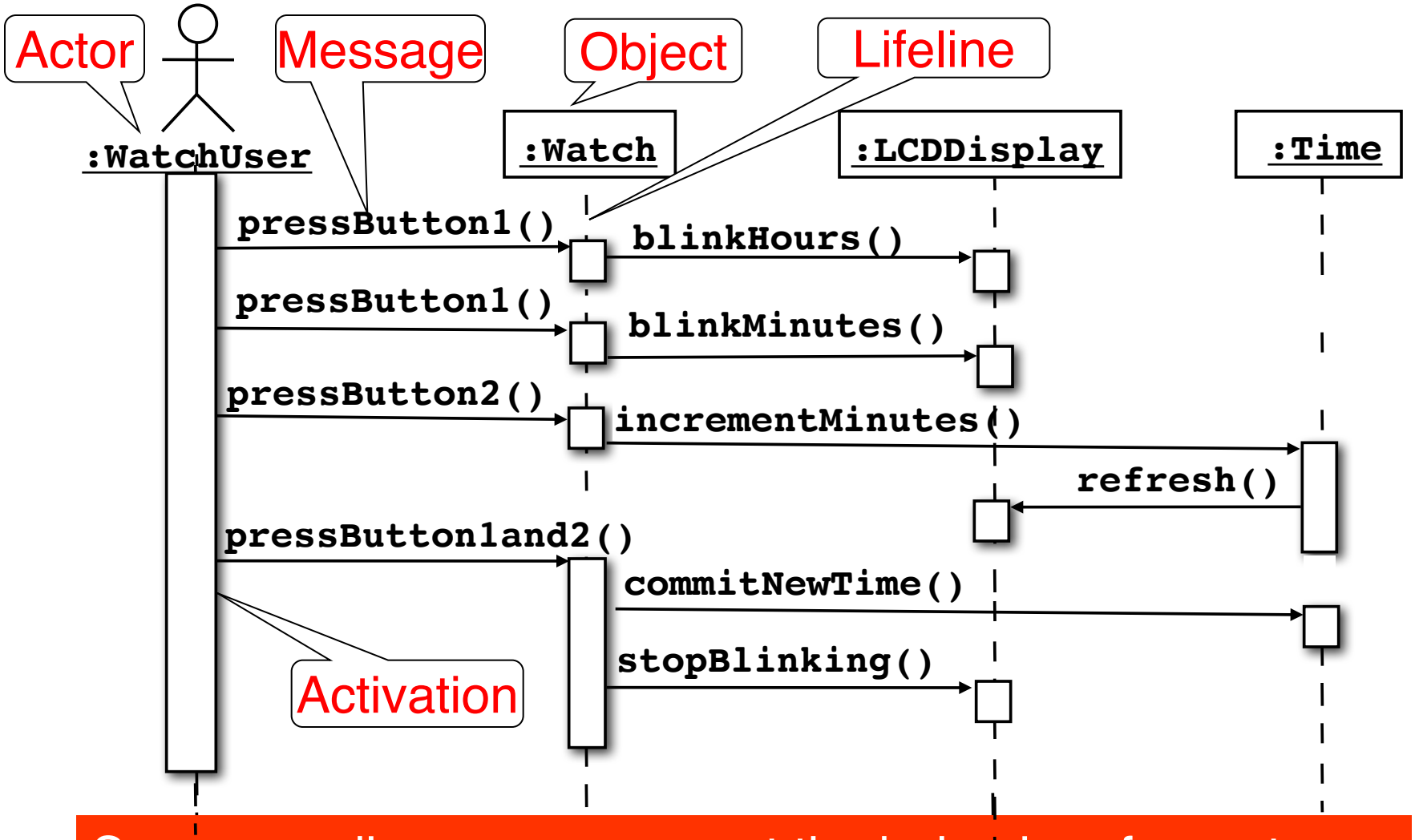
Class diagrams represent the structure of the system

UML first pass: Class diagrams

Class diagrams represent the structure of the system

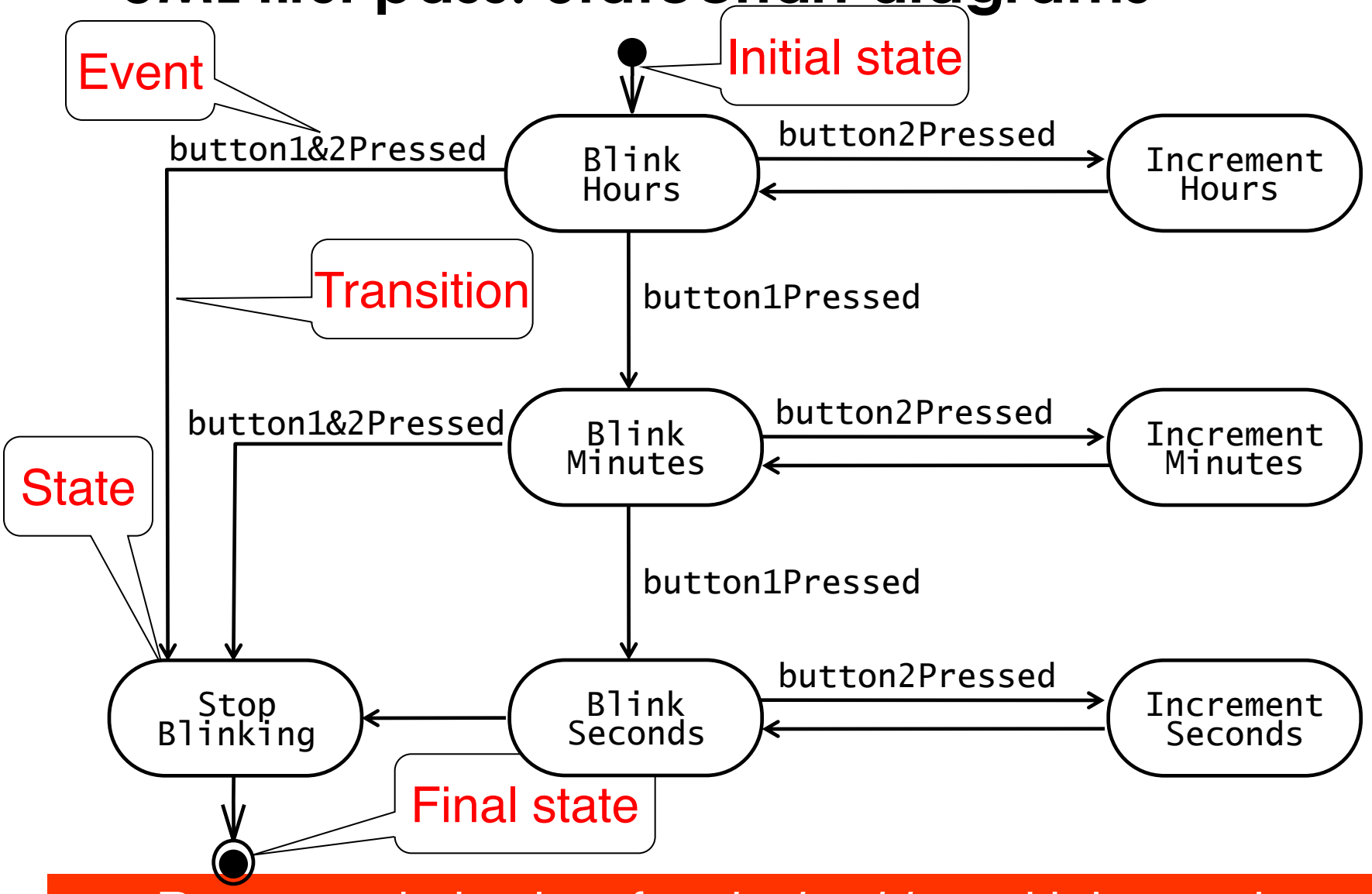


UML first pass: Sequence diagram



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

UML first pass: Statechart diagrams



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

Other UML Notations

UML provides many other notations, for example

- Deployment diagrams for modeling configurations
  - Useful for testing and for release management
- We introduce these and other notations as we go along in the lectures
  - OCL: A language for constraining UML models.

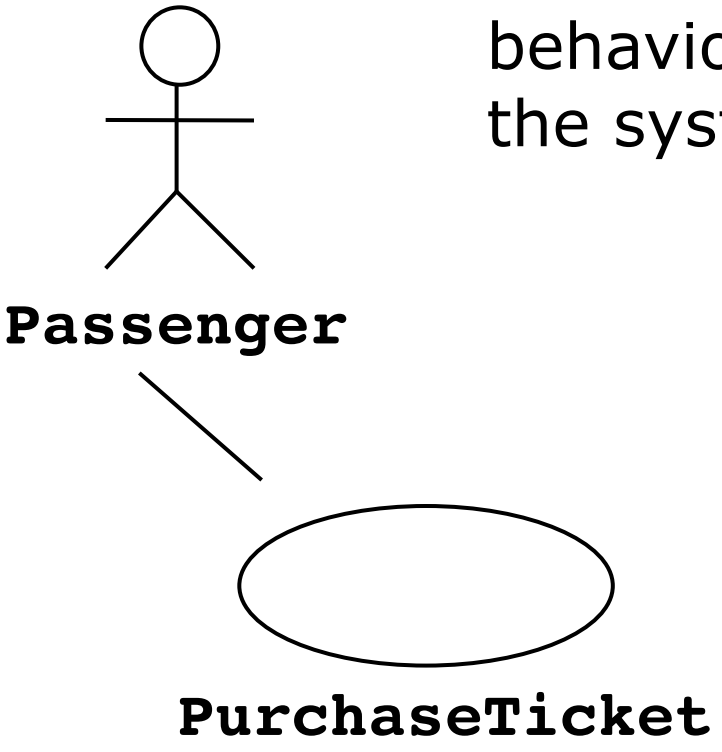
What should be done first? Coding or Modeling?

- It 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.

UML Basic Notation Summary

- UML provides a wide variety of notations for modeling many aspects of software systems
- Today we concentrate on a few notations:
  - Functional model: Use case diagram
  - Object model: Class diagram
  - Dynamic model: Sequence diagrams, statechart.

UML Use Case Diagrams: Second pass



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

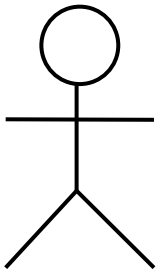
An ***Actor*** represents a role, that is, a type of user of the system

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



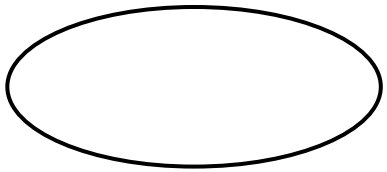
Passenger

- 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

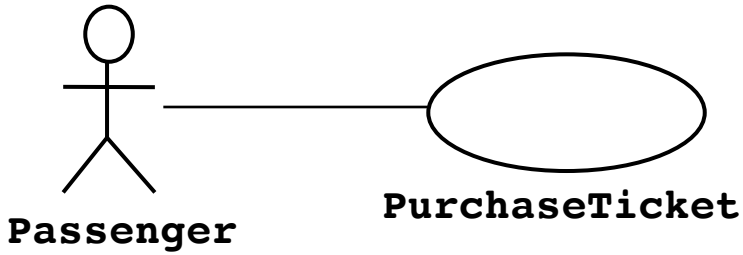
Use Case



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:

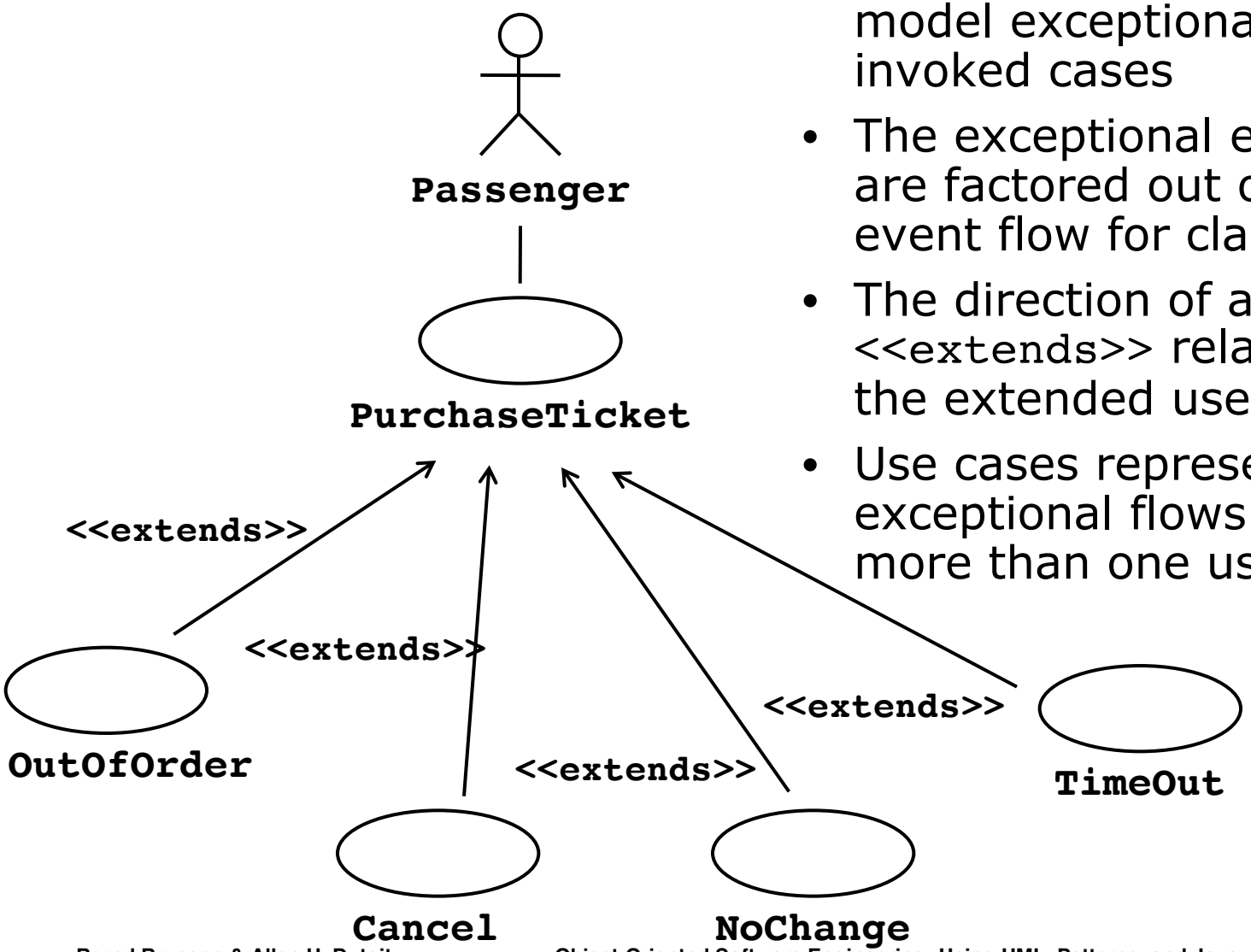
1. `Passenger` selects the number of zones to be traveled
2. Ticket Distributor displays the amount due
3. `Passenger` inserts money, at least the amount due
4. Ticket Distributor returns change
5. Ticket Distributor issues ticket

6. Special requirements:
None.

Uses Cases can be related

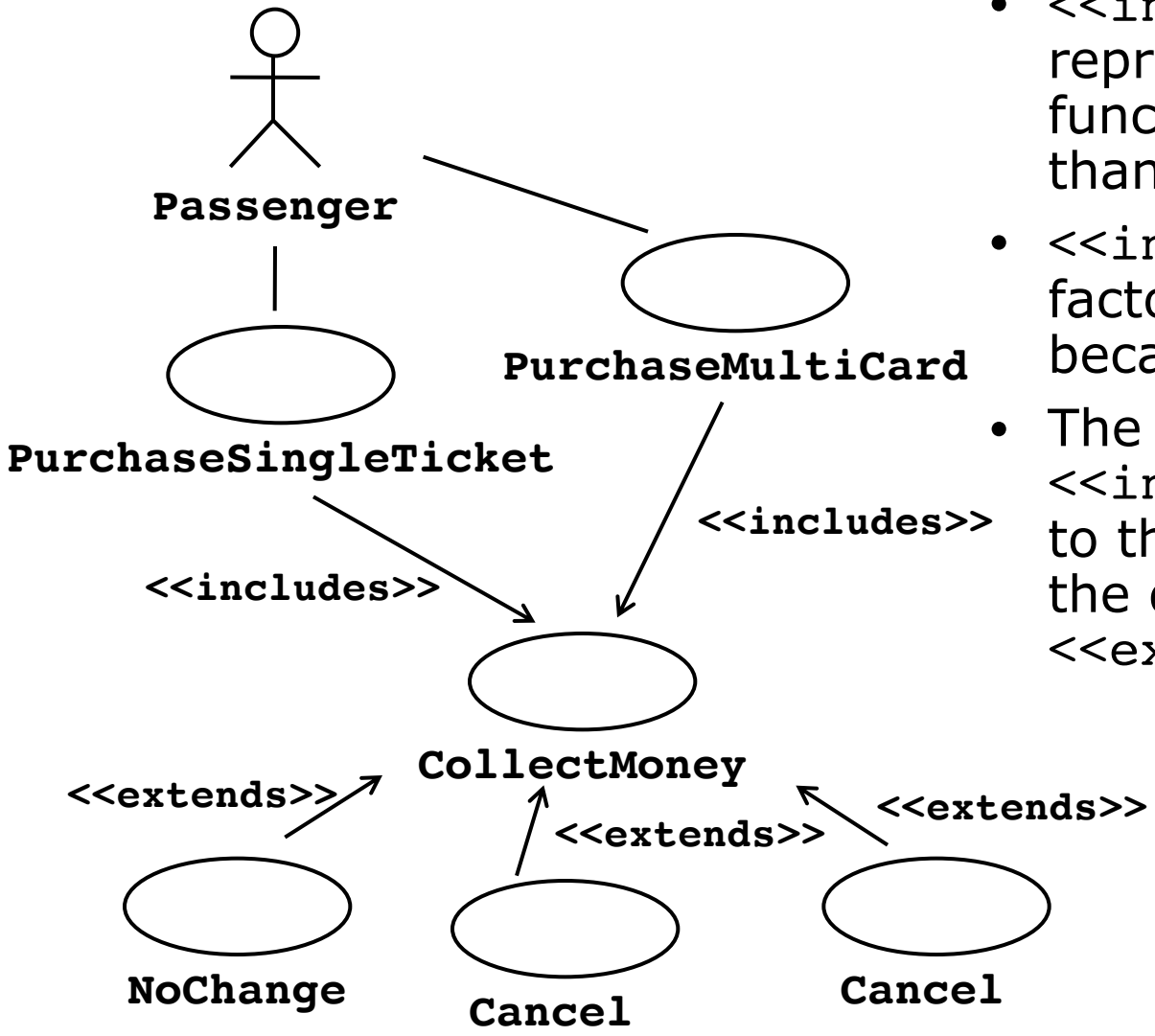
- Extends Relationship
  - To represent seldom invoked use cases or exceptional functionality
- Includes Relationship
  - To represent functional behavior common to more than one use case.

The <<*extends*>> Relationship



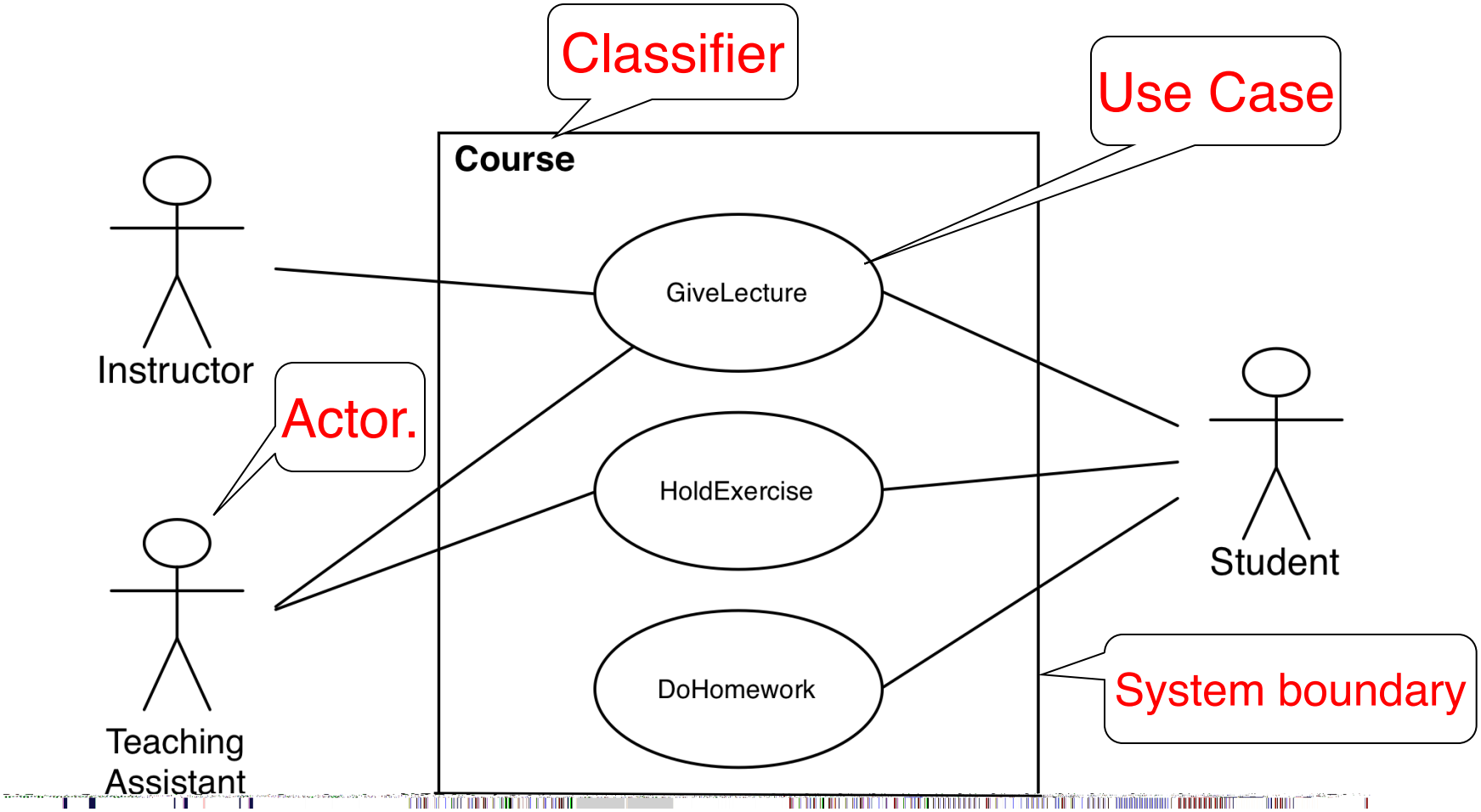
- `<<extends>>` relationships model exceptional or seldom invoked cases
- The exceptional event flows are factored out of the main event flow for clarity
- The direction of an `<<extends>>` relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use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

The <<*includes*>> Relationship

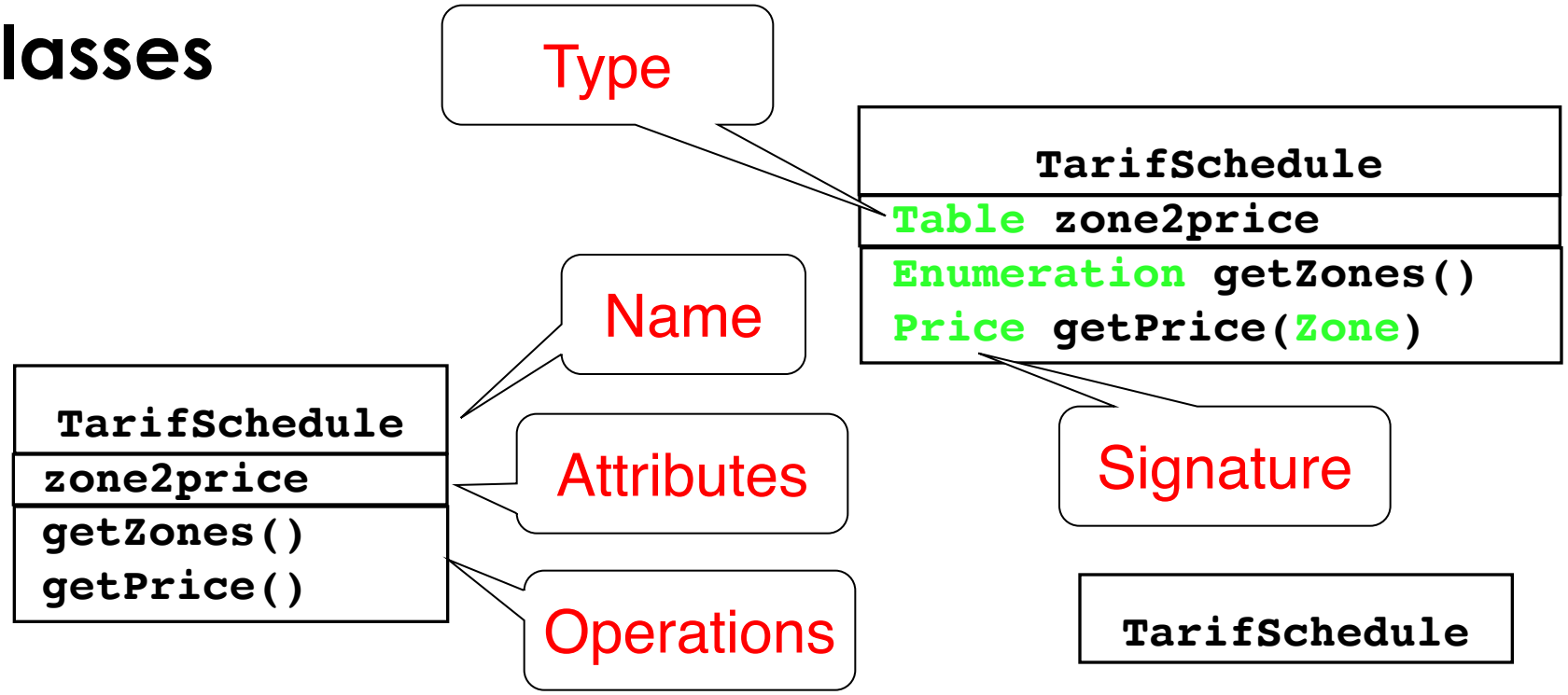


- <<includes>> relationship represents common functionality needed in more than one use case
- <<includes>> behavior is factored out for reuse, not because it is an exception
- The direction of a <<includes>> relationship is to the using use case (unlike the direction of the <<extends>> relationship).

Use Case Models can be packaged



Classes



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

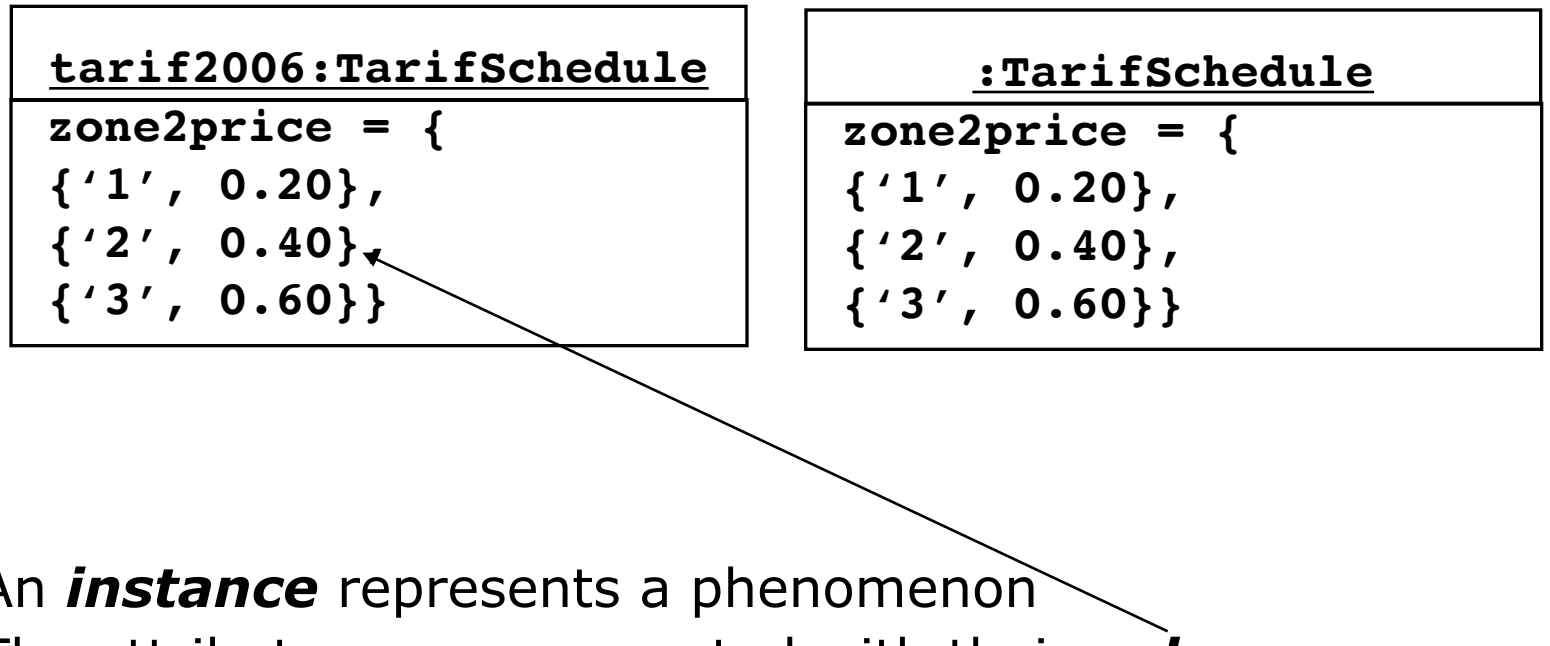
  Each attribute has a ***type***
  Each operation has a ***signature***

The class name is the only mandatory information

Actor vs Class vs Object

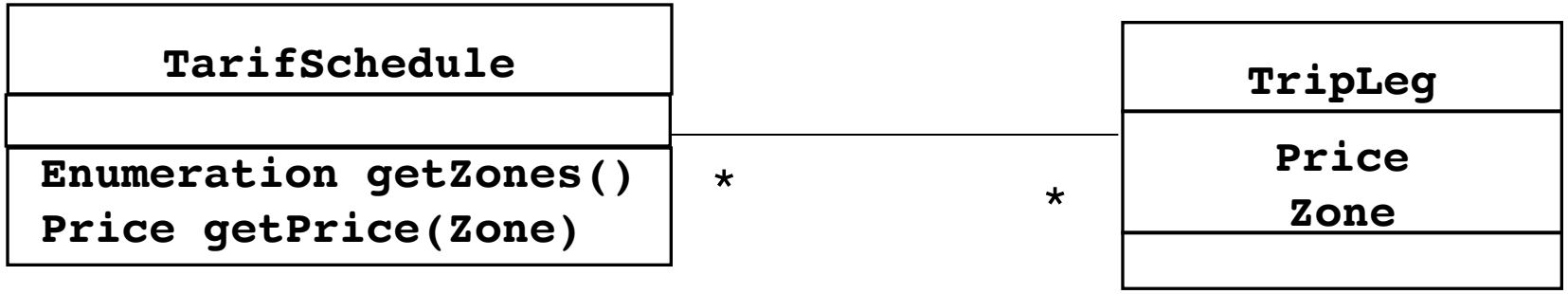
- **Actor**
  - An entity outside the system to be modeled, interacting with the system (“Passenger”)
- **Class**
  - An abstraction modeling an entity in the application or solution domain
  - The class is part of the system model (“User”, “Ticket distributor”, “Server”)
- **Object**
  - A specific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

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)

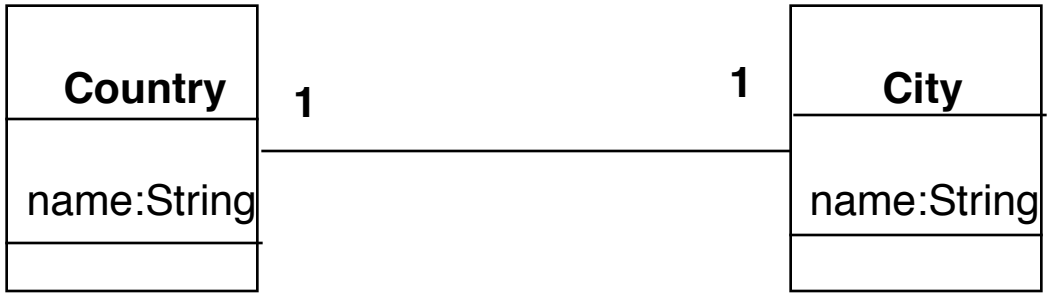
Associations



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

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

From Problem Statement To Object Model

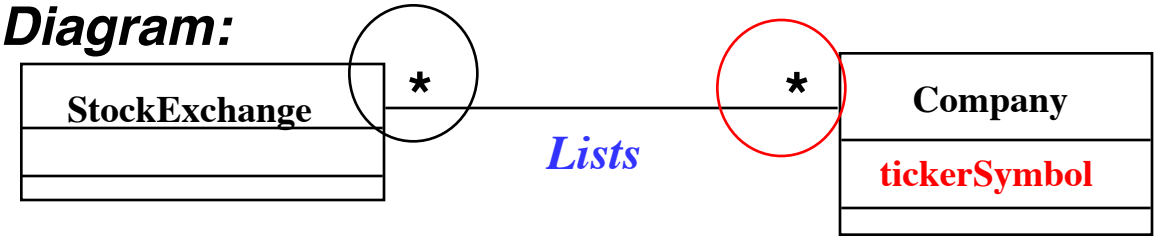
Problem Statement: A stock exchange lists many companies. Each company is uniquely identified by a ticker symbol

Class Diagram:

From Problem Statement to Code

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

Class Diagram:



Java Code

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

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

Associations are mapped to Attributes!

Model-driven development and architecture

1. Build a platform-independent model of an applications functionality and behavior
   a) Describe model in modeling notation (UML)
   b) Convert model into platform-specific model
2. Generate executable from platform-specific model

Advantages:

- Code is generated from model ("mostly")
- Portability and interoperability

- Model Driven Architecture effort:
  - http://www.omg.org/mda/

Additional References

- Martin Fowler
  - UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd ed., Addison-Wesley, 2003
- Grady Booch,James Rumbaugh,Ivar Jacobson
  - The Unified Modeling Language User Guide, Addison Wesley, 2nd edition, 2005
- Open Source UML tools
  - http://java-source.net/open-source/uml-modeling