Software Engineering

Object-Oriented Analysis and Design
and
Modeling with UML

Assoc. Prof. Marenglen Biba
MSc in Computer Science, UoG-UNYT
Foundation Programme

Material

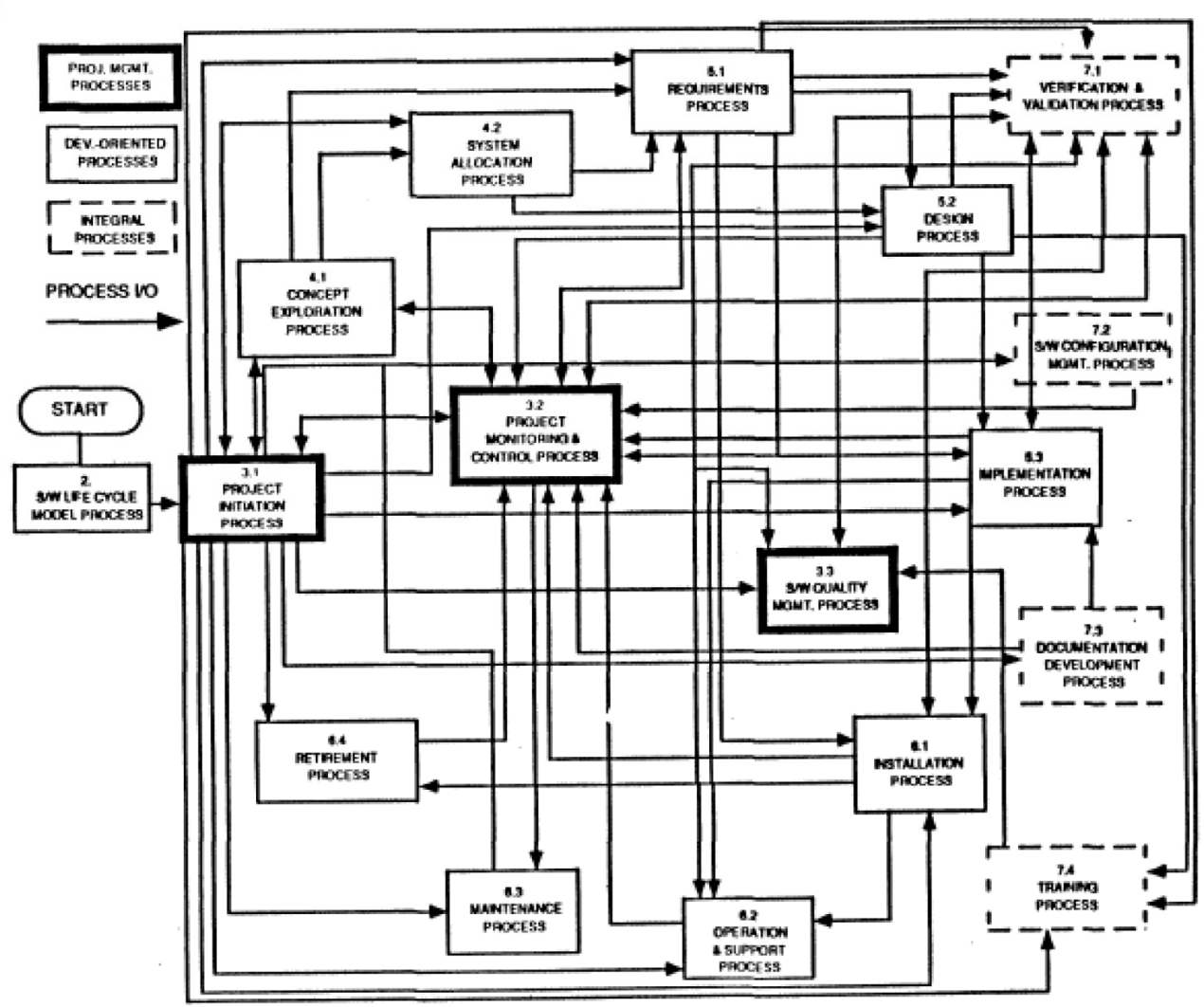
- Get the material from
  - http://www.marenglenbiba.net/foundprog/
  - Sufficient for FP exam purposes

- Other useful material
  - I. Sommerville. Software Engineering (in library)
  - R. Pressman. Software Engineering: A Practitioner's Approach (in library)
  - *B. Bruegge & A. H. Dutoit.* Object-Oriented Software Engineering: Using UML, Patterns, and Java, 2nd Edition.

Requirements Analysis and UML

- Abstraction and Modeling
- Decomposition
- Hierarchy
- Object-Oriented Modeling
- UML Diagrams
- Use case diagrams
- Case study in lab

What is the problem with this Drawing?



Abstraction

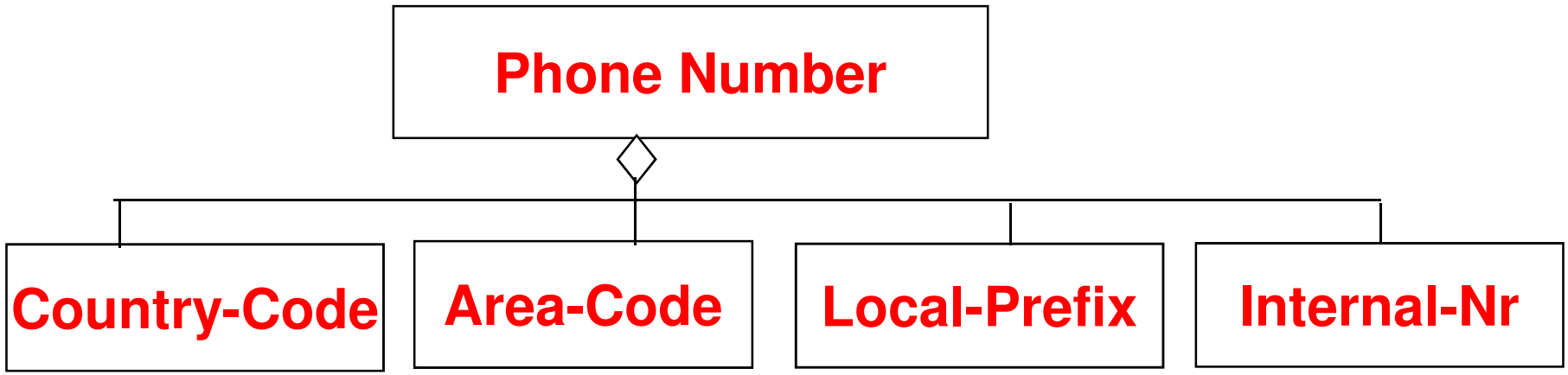
- Complex systems are hard to understand
  - The 7 +- 2 phenomena
    - Our short term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain
    - My Phone Number: 49892891820 4

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: 49892891820 4
- Chunking:
    - Group collection of objects to reduce complexity
    - 4 chunks:
      - State-code, Area-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 Phone Number: 49892891820 4
- Chunking:
    - Group collection of objects to reduce complexity
    - State-code, Area-code, Local Prefix, Internal-Nr



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

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

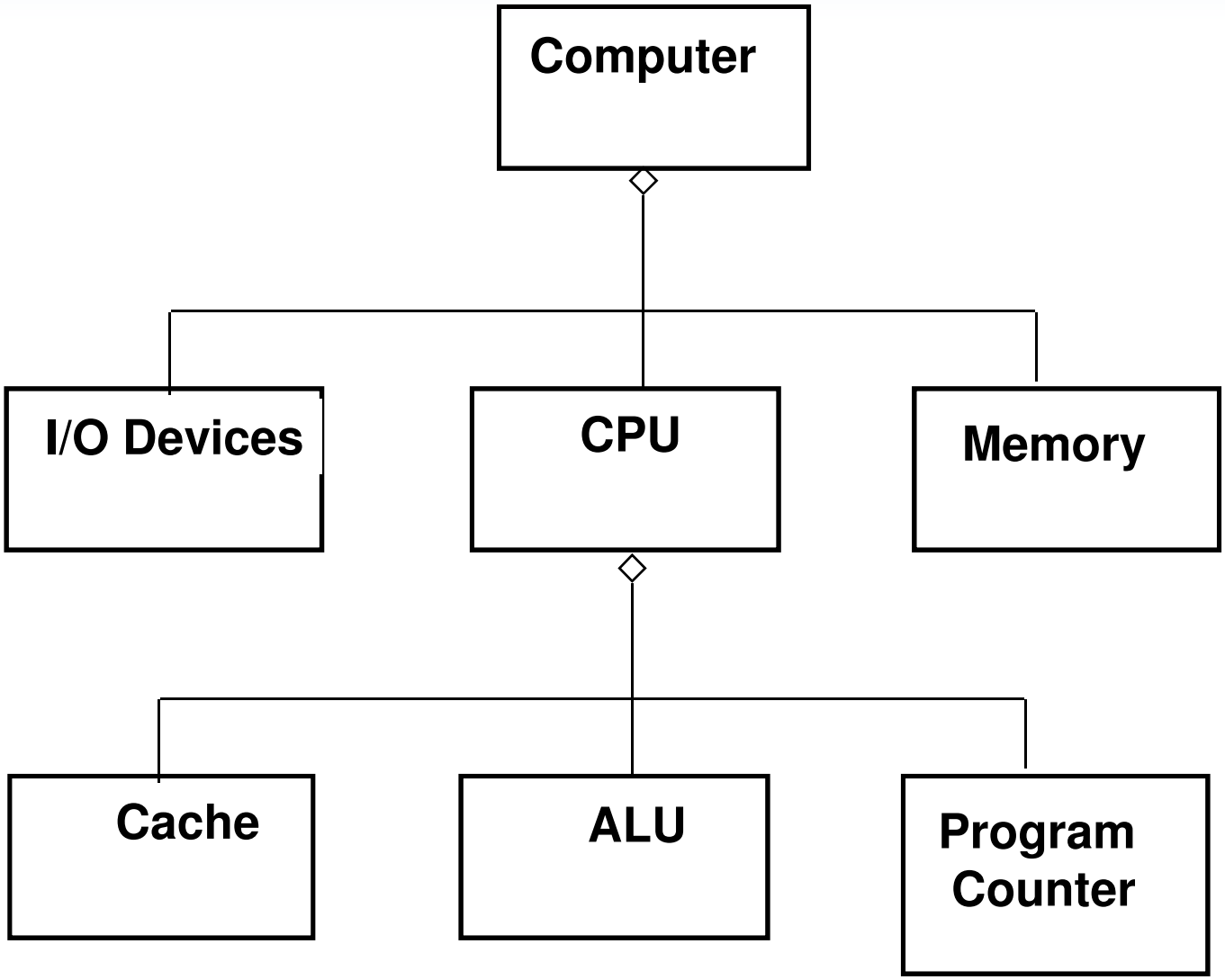
Class Identification

- **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.

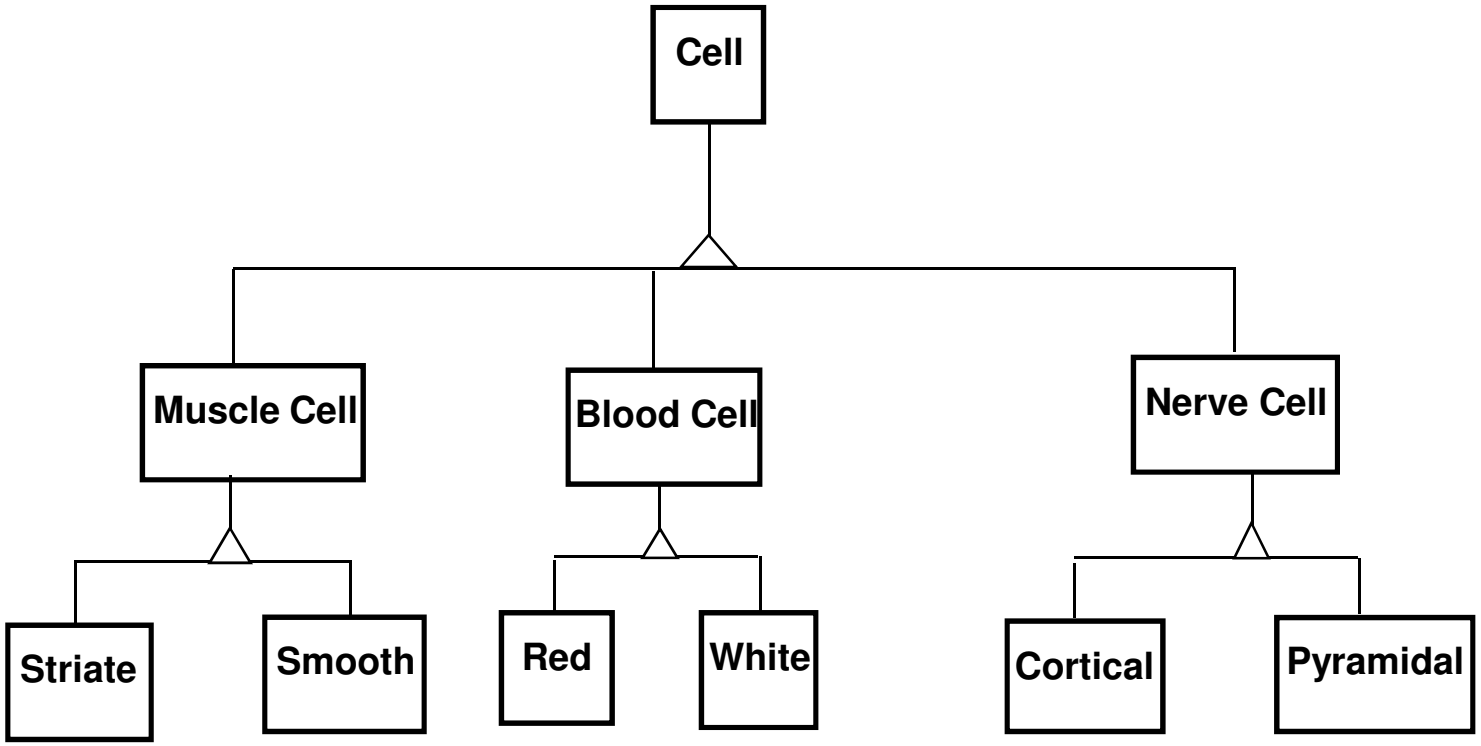
Hierarchy

- 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)



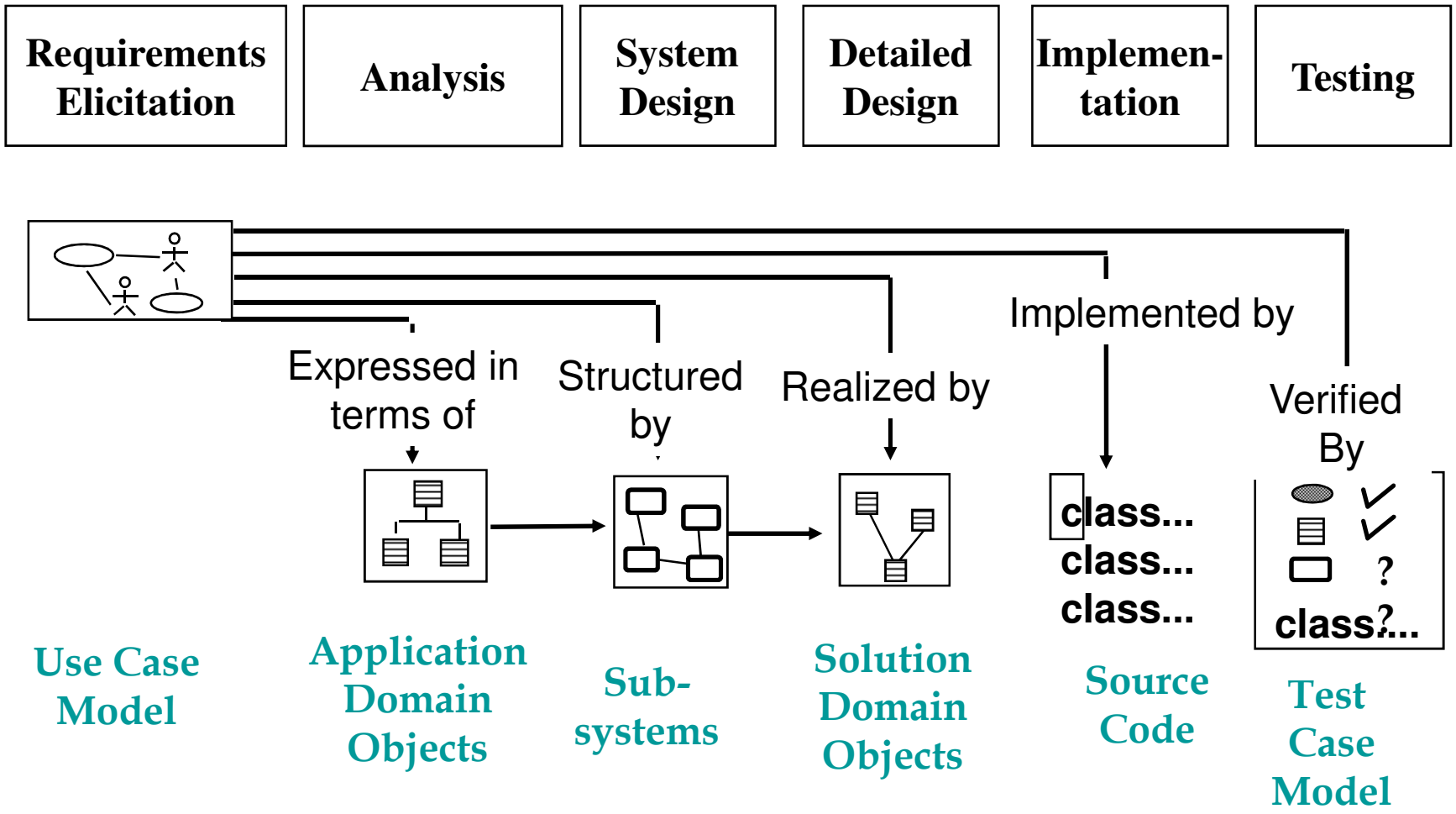
Where are we?

- 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

Requirements Elicitation

- UML diagrams
- Case Study: Use case diagrams with UML

Software Lifecycle Activities



First step in identifying the Requirements: System identification

- Two questions need to be answered:
  1. How can we identify the purpose of a system?
     - What are the requirements, what are the constraints?
  2. What is inside, what is outside the system?
- These two questions are answered during requirements elicitation and analysis
- Requirements elicitation:
  - Definition of the system in terms understood by the customer and/or user ("Requirements specification")
- Analysis:
  - Definition of the system in terms understood by the developer (Technical specification, "Analysis model")
- Requirements Process: Consists of the activities Requirements Elicitation and Analysis.

Techniques to elicit Requirements

- Bridging the gap between end user and developer:
  - **Questionnaires:** Asking the end user a list of pre-selected questions
  - **Task Analysis:** Observing end users in their operational environment
  - **Scenarios:** Describe the use of the system as a series of interactions between a specific end user and the system
  - **Use cases:** Abstractions that describe a class of scenarios.

- Job: Software Analyst, Business Analyst

What is UML?

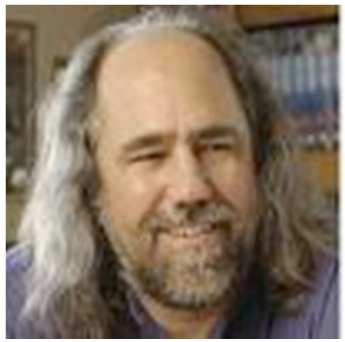
- UML (Unified Modeling Language)
  - Nonproprietary standard for modeling software systems, OMG
  - Convergence of notations used in object-oriented methods
    - OMT (James Rumbaugh and collegues)
    - Booch (Grady Booch)
    - OOSE (Ivar Jacobson)
- Current Version: UML 2.2
  - Information at the 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)

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



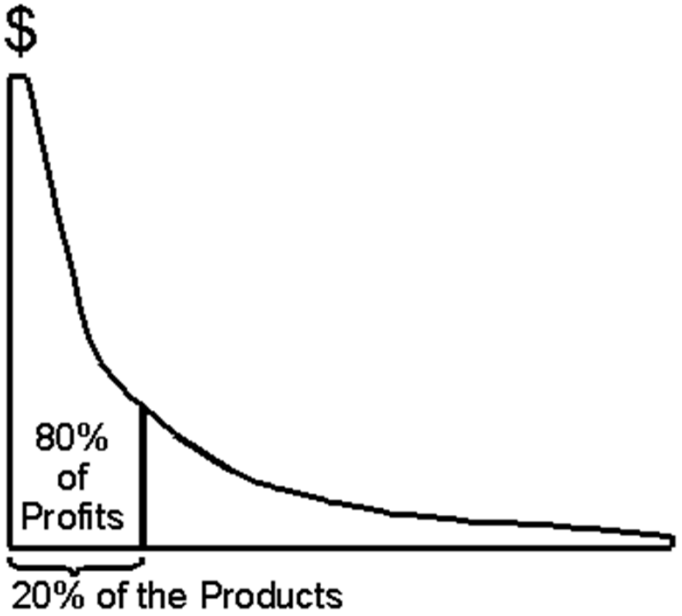
Developed the Booch method ("clouds"), ACM Fellow 1995, and IBM Fellow 2003 http://www.booch.com/

UML Basic Notation: First Summary

- UML provides a wide variety of notations for modeling many aspects of software systems
- Models:
  - Functional model: Use case diagrams
  - Object model: Class diagrams
  - Dynamic model: Sequence diagrams, statechart diagram

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.

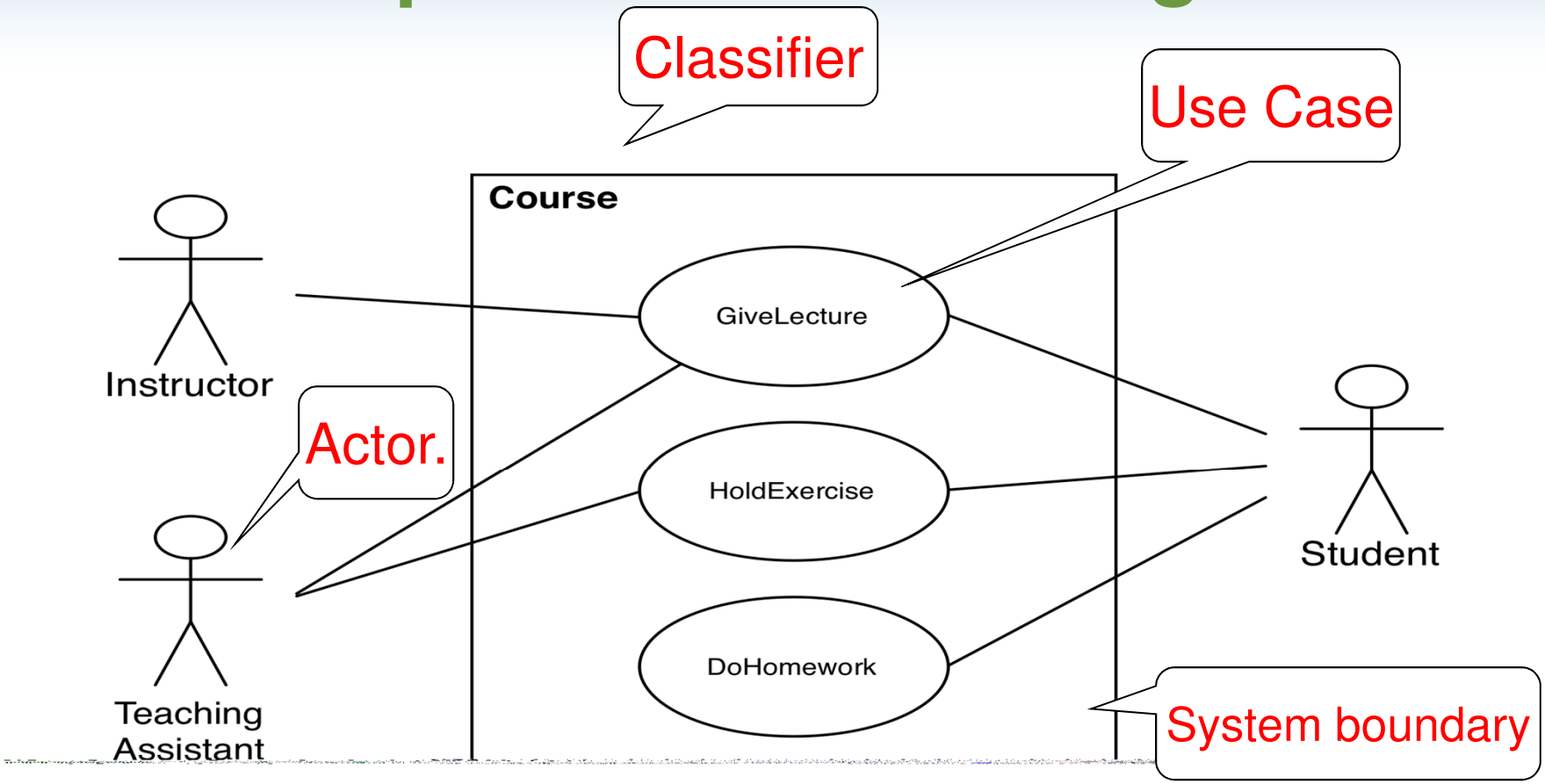
UML First Pass

- 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

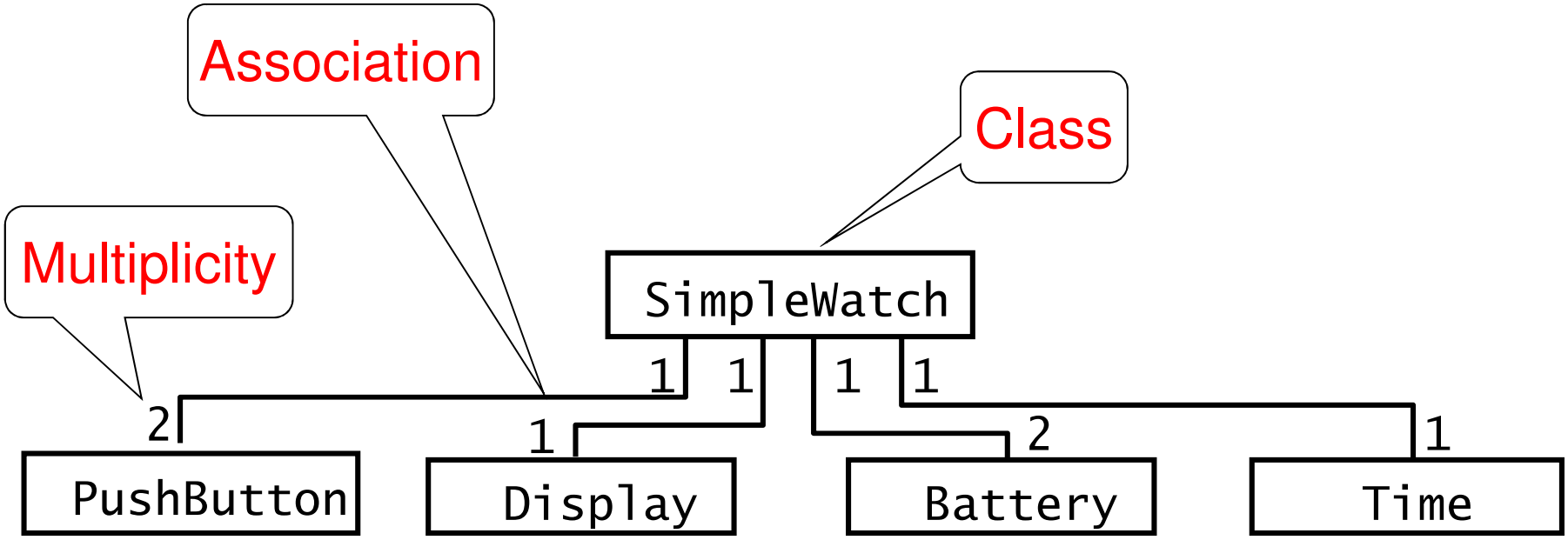
- All UML Diagrams denote graphs of nodes and edges
  - Nodes are entities and drawn as rectangles or ovals
  - Rectangles denote classes or instances
  - Ovals denote functions
  - Names of Classes are not underlined
    - `SimpleWatch`
    - `Firefighter`
  - Names of Instances are underlined
    -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

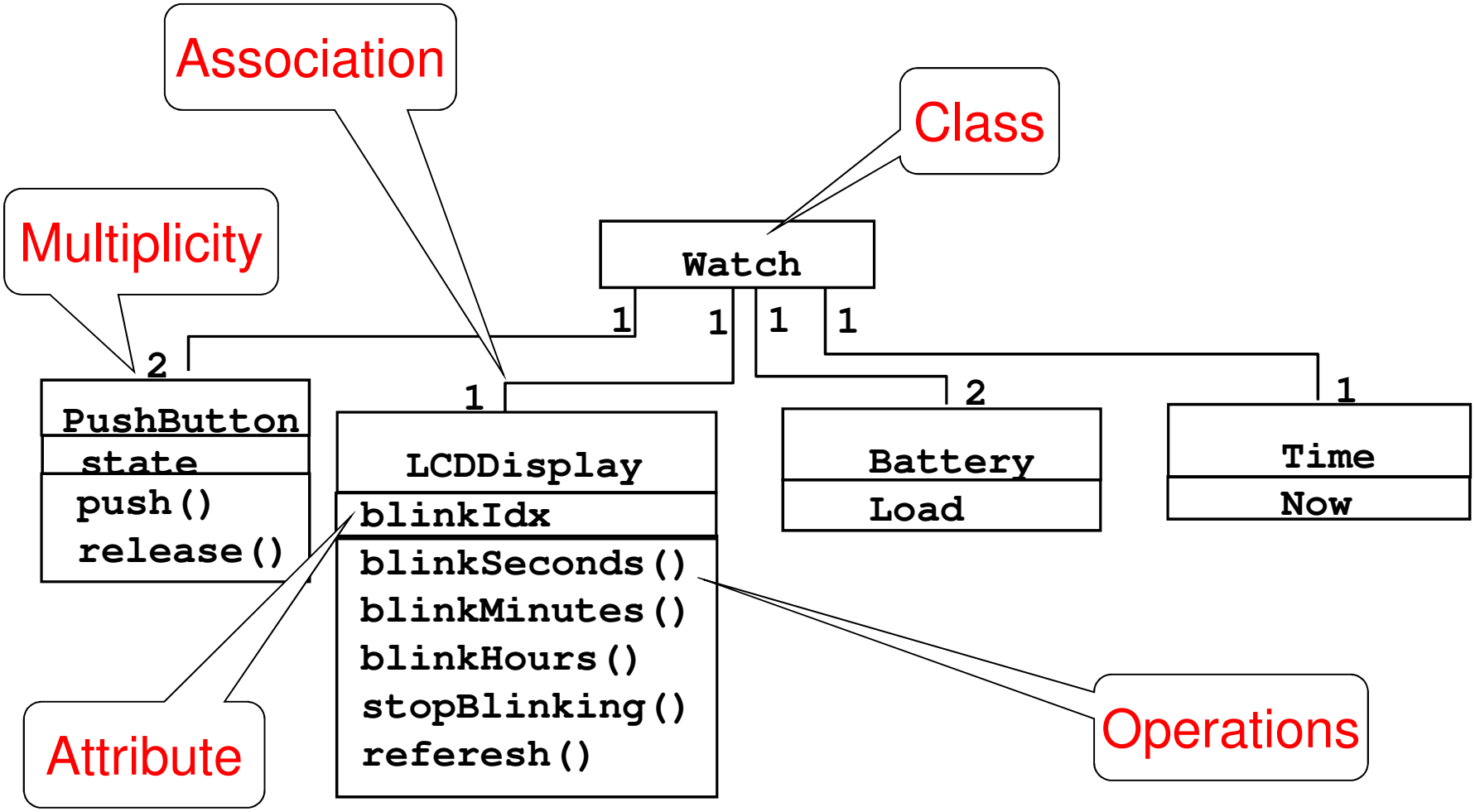
UML first pass: Class diagrams



Class diagrams represent the structure of the system

UML first pass: Class diagrams

Class diagrams represent the structure of the system



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

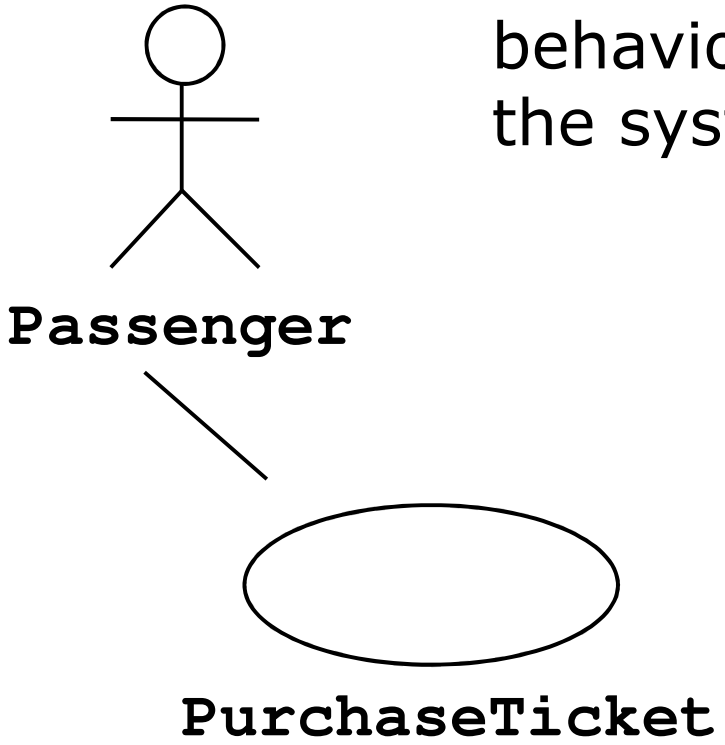
Use case diagrams

Use case diagrams

- 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 Use Case Diagrams

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

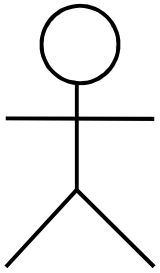


An ***Actor*** represents a role, that is, a type of user of the system

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



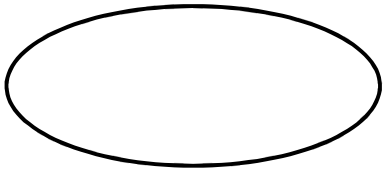
Passenger

- 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

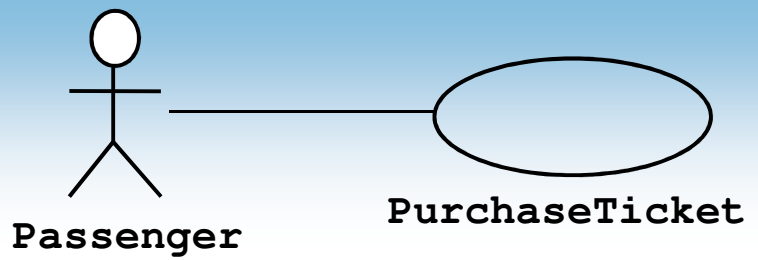
Use Case



`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:

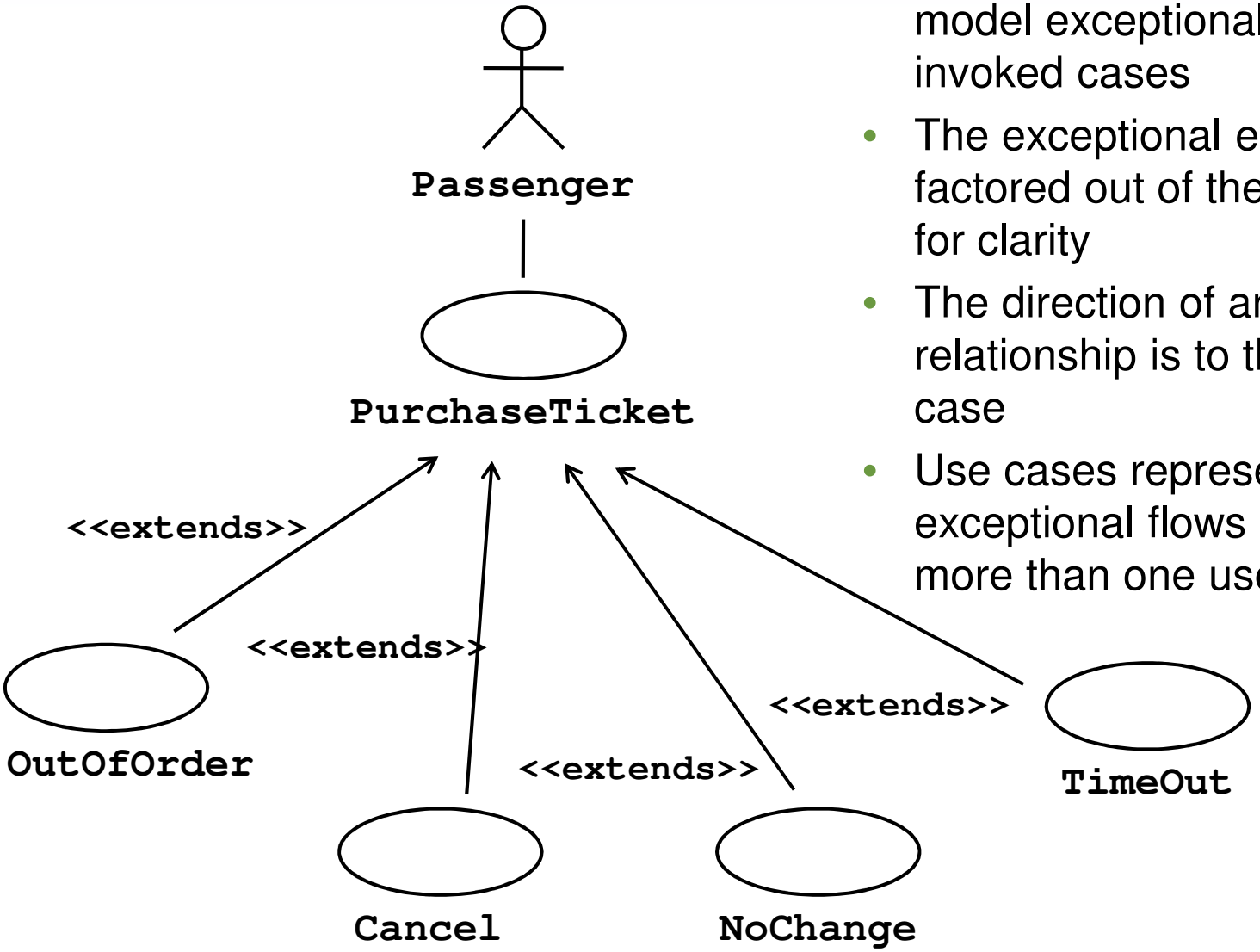
1. `Passenger` selects the number of zones to be traveled
2. Ticket Distributor displays the amount due
3. `Passenger` inserts money, at least the amount due
4. Ticket Distributor returns change
5. Ticket Distributor issues ticket

6. Special requirements: None.

Uses Cases can be related

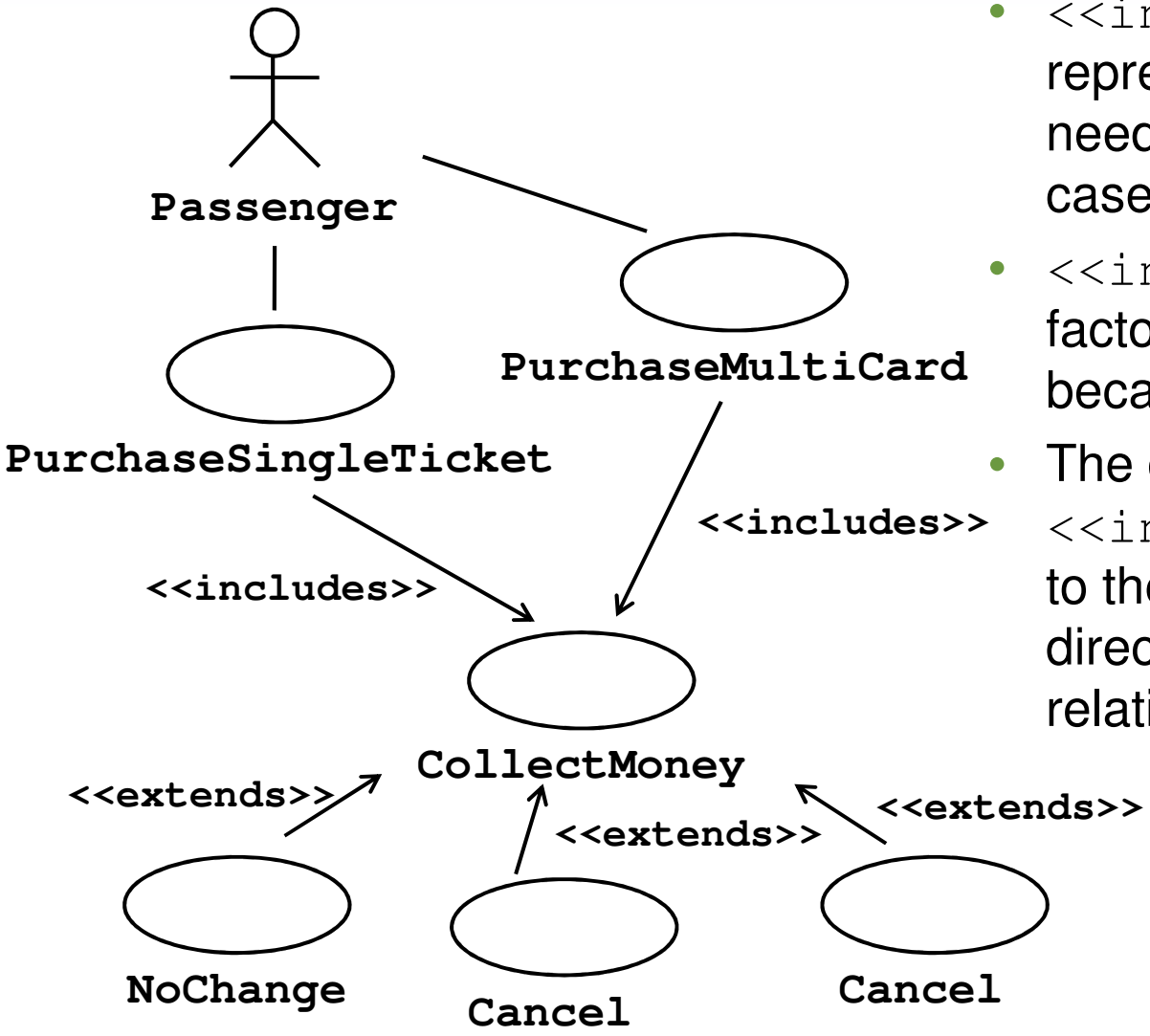
- Extends Relationship
  - To represent seldom invoked use cases or exceptional functionality
- Includes Relationship
  - To represent functional behavior common to more than one use case.

The *<<extends>>* Relationship



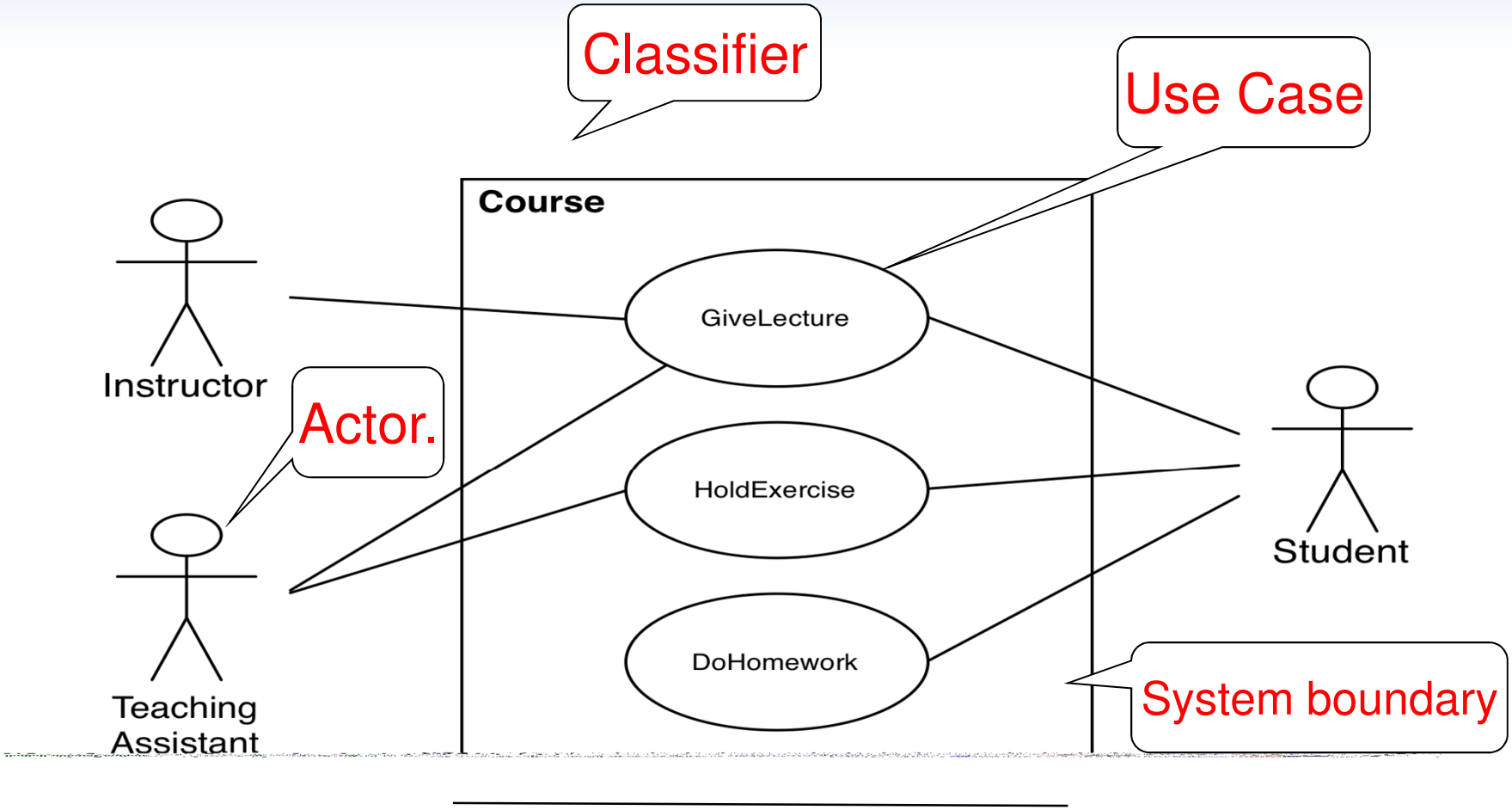
- `<<extends>>` relationships model exceptional or seldom invoked cases
- The exceptional event flows are factored out of the main event flow for clarity
- The direction of an `<<extends>>` relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use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



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”).

Class diagrams in UML: composition

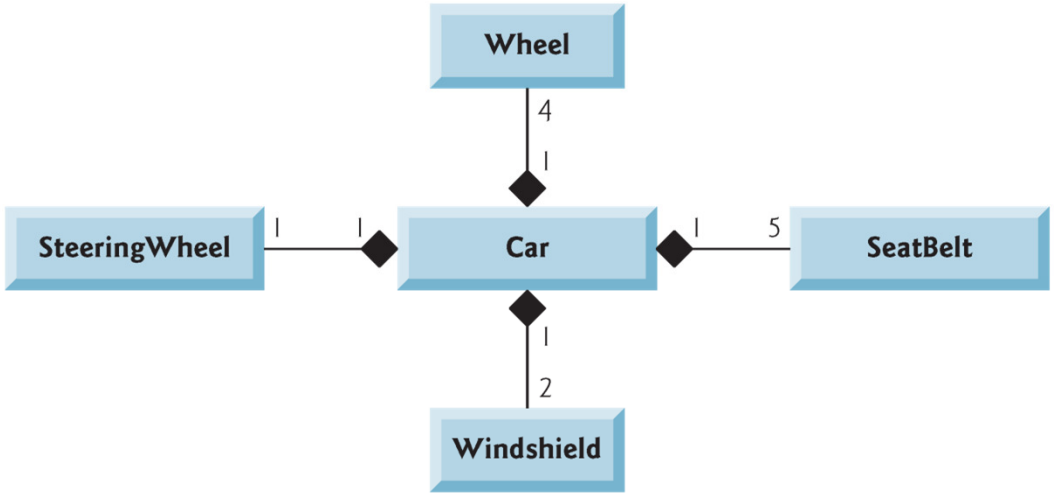


Fig. 12.27 | Class diagram showing composition relationships of a class `Car`.

Class diagrams in UML: inheritance

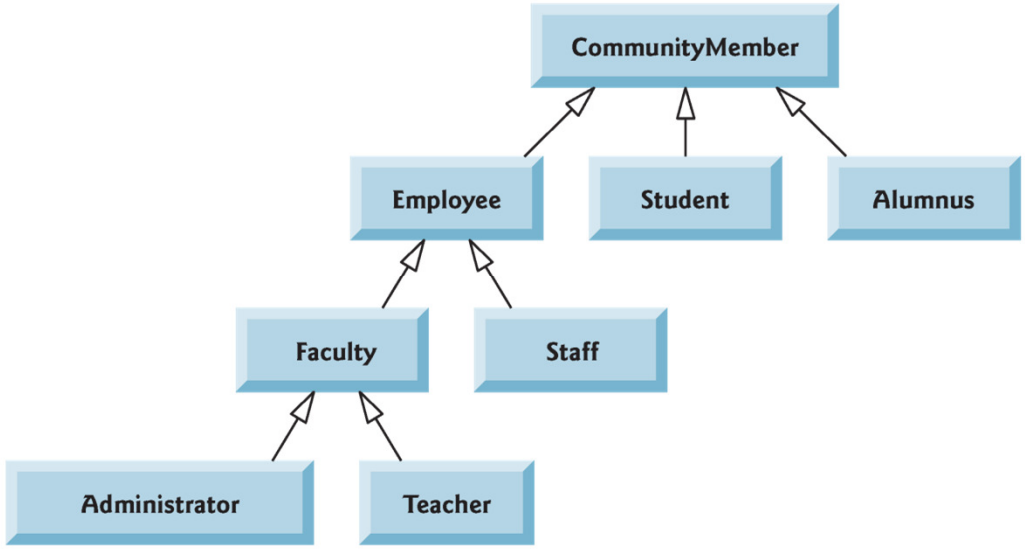


Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`.

Class diagrams in UML: inheritance

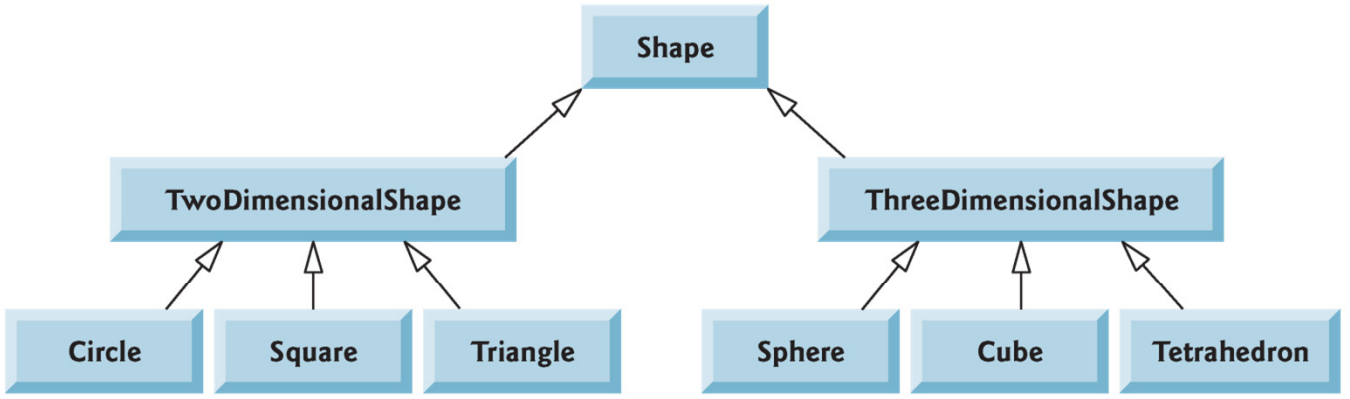


Fig. 9.3 | Inheritance hierarchy for Shapes.

Case Study for UML Diagrams

- Lab session
  - Work in StarUML

End of class