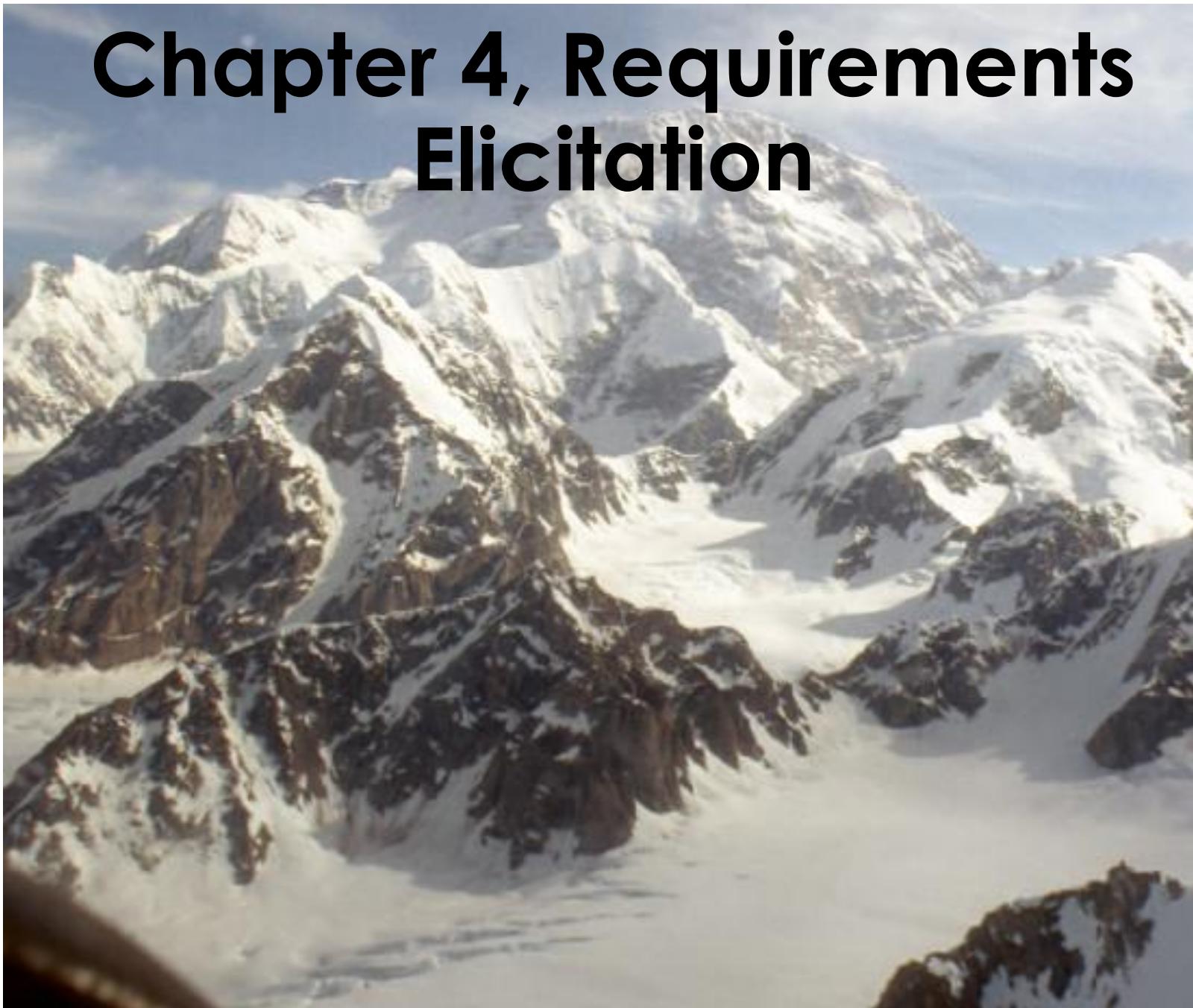


# **Object-Oriented Software Engineering**

Using UML, Patterns, and Java

## **Chapter 4, Requirements Elicitation**



# Outline

- *Motivation: Software Lifecycle*
- *Requirements elicitation challenges*
- *Problem statement*
- *Requirements specification*
  - *Types of requirements*
- *Validating requirements*
- *Summary*

# Software Lifecycle Definition

- *Software lifecycle*
  - *Models for the development of software*
    - *Set of activities and their dependency relationships to each other to support the development of a software system*
    - *Examples:*
      - *Analysis, Design, Implementation, Testing*
  - *Typical Lifecycle questions:*
    - *Which activities should I select when I develop software?*
    - *What are the dependencies between activities?*
    - *How should I schedule the activities?*

# A Typical Example of Software Lifecycle Activities



# Software Lifecycle Activities...and their models

Requirements  
Elicitation

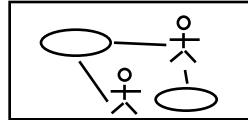
Analysis

System  
Design

Detailed  
Design

Implemen-  
tation

Testing



Use Case  
Model

# Software Lifecycle Activities...and their models

Requirements  
Elicitation

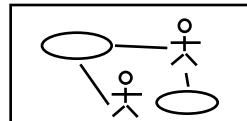
Analysis

System  
Design

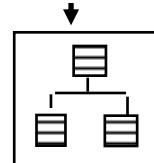
Detailed  
Design

Implemen-  
tation

Testing



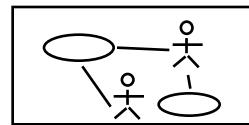
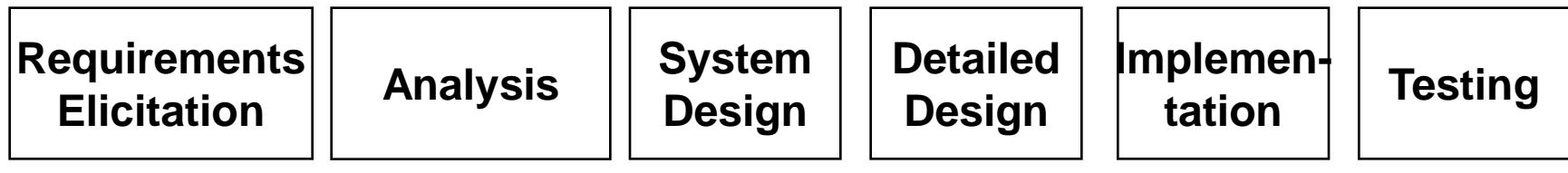
Expressed in  
terms of



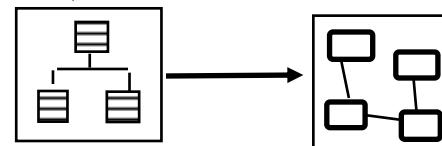
Use Case  
Model

Application  
Domain  
Objects

# Software Lifecycle Activities...and their models



Expressed in  
terms of

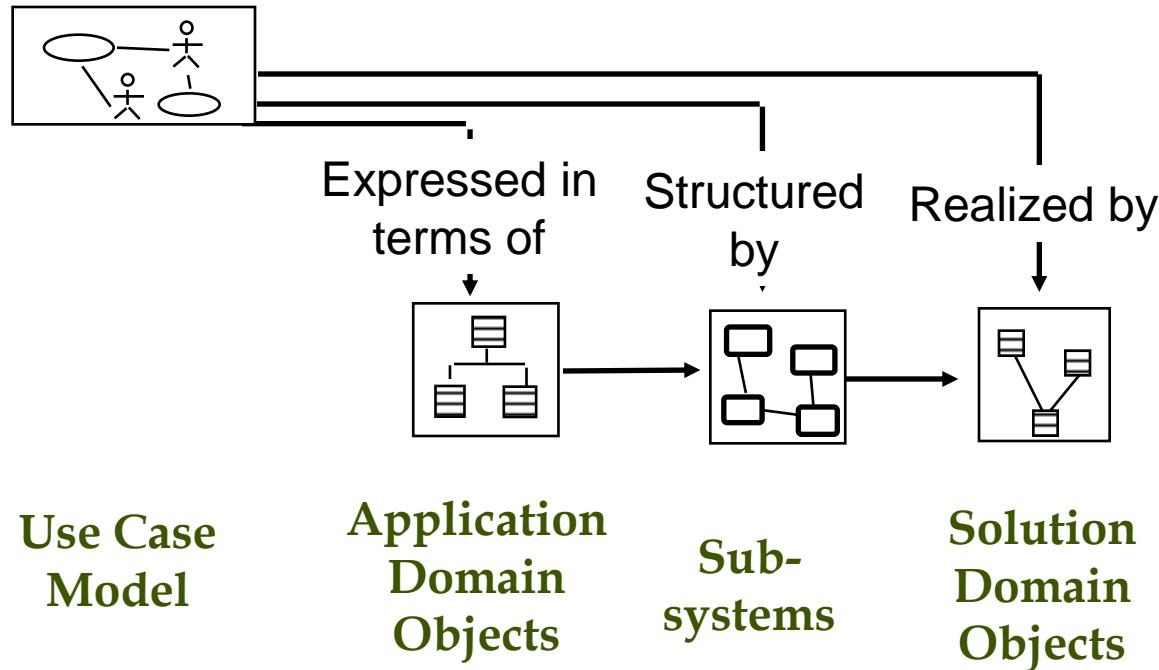
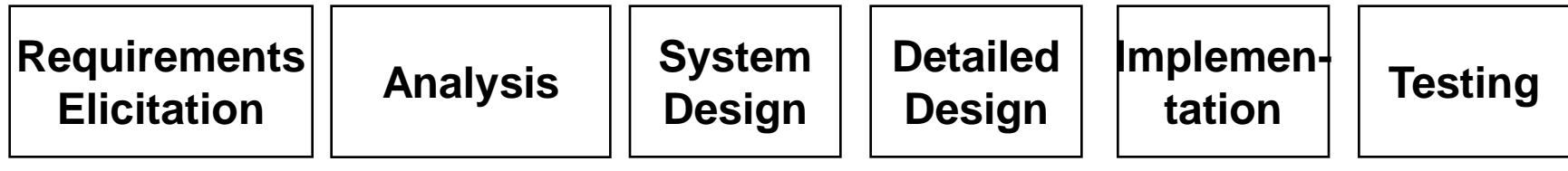


Use Case  
Model

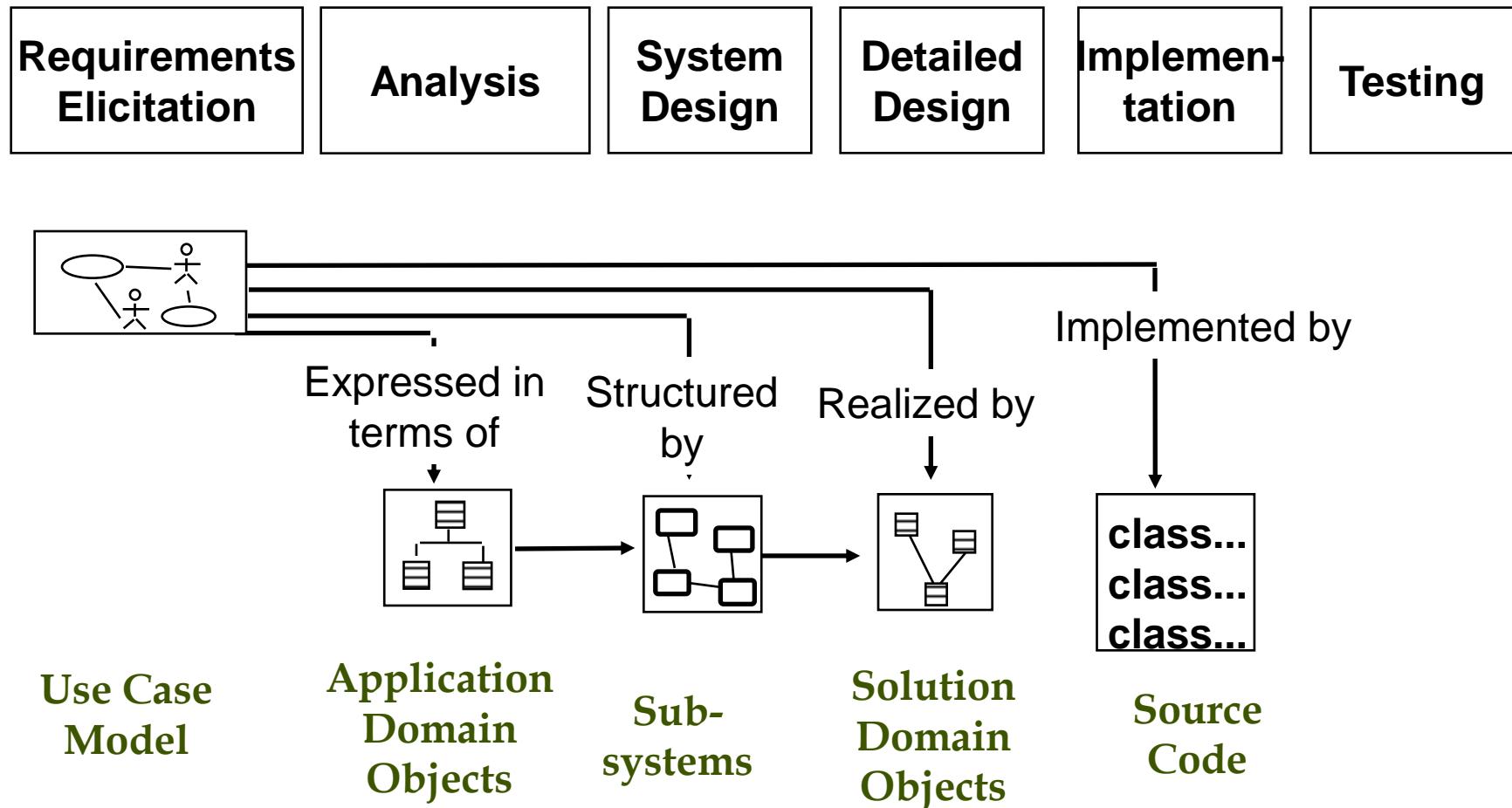
Application  
Domain  
Objects

Sub-  
systems

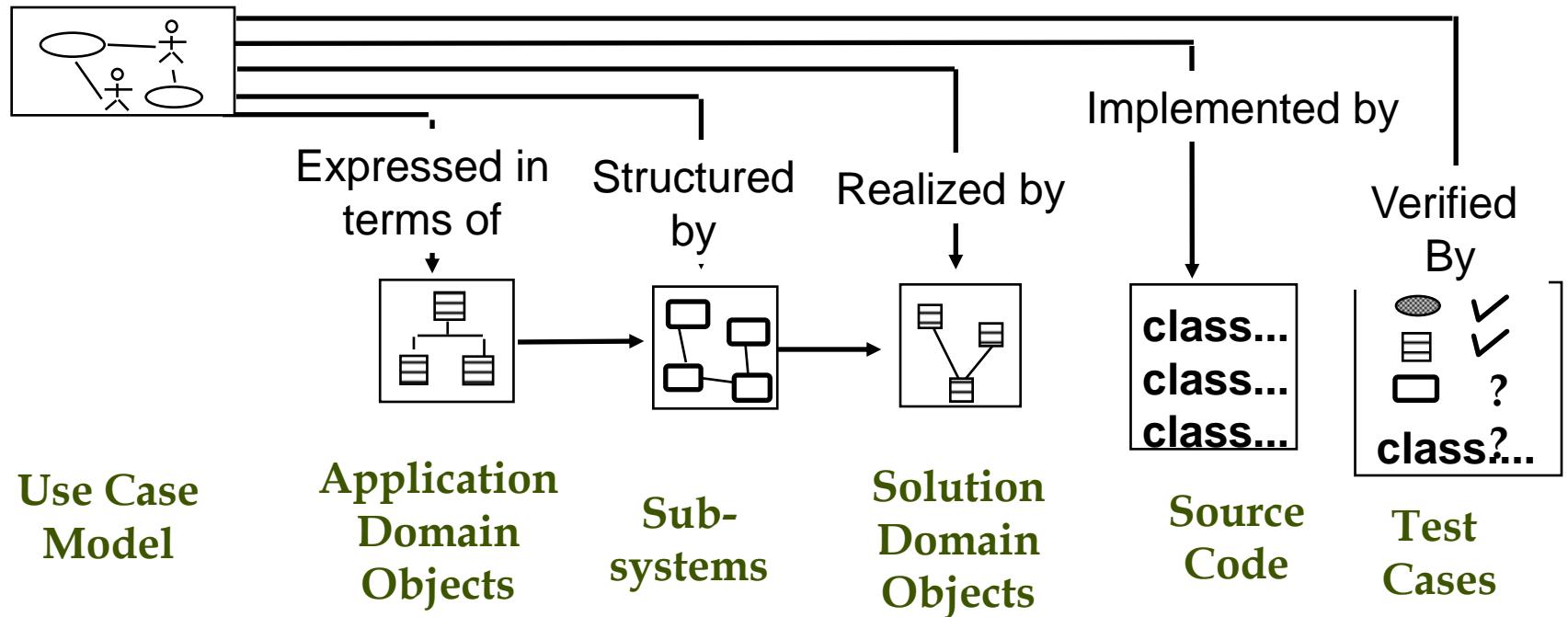
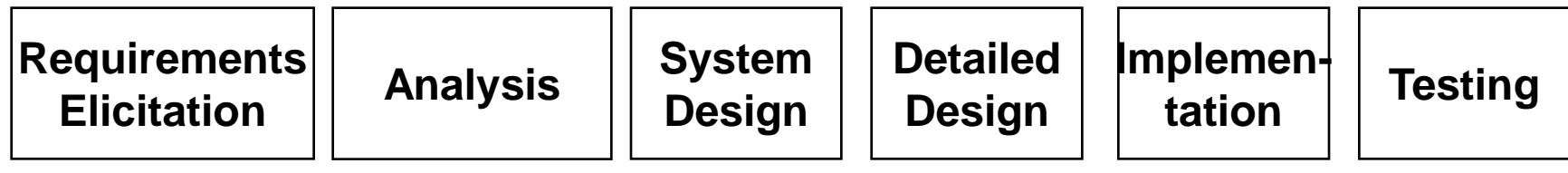
# Software Lifecycle Activities...and their models



# Software Lifecycle Activities...and their models



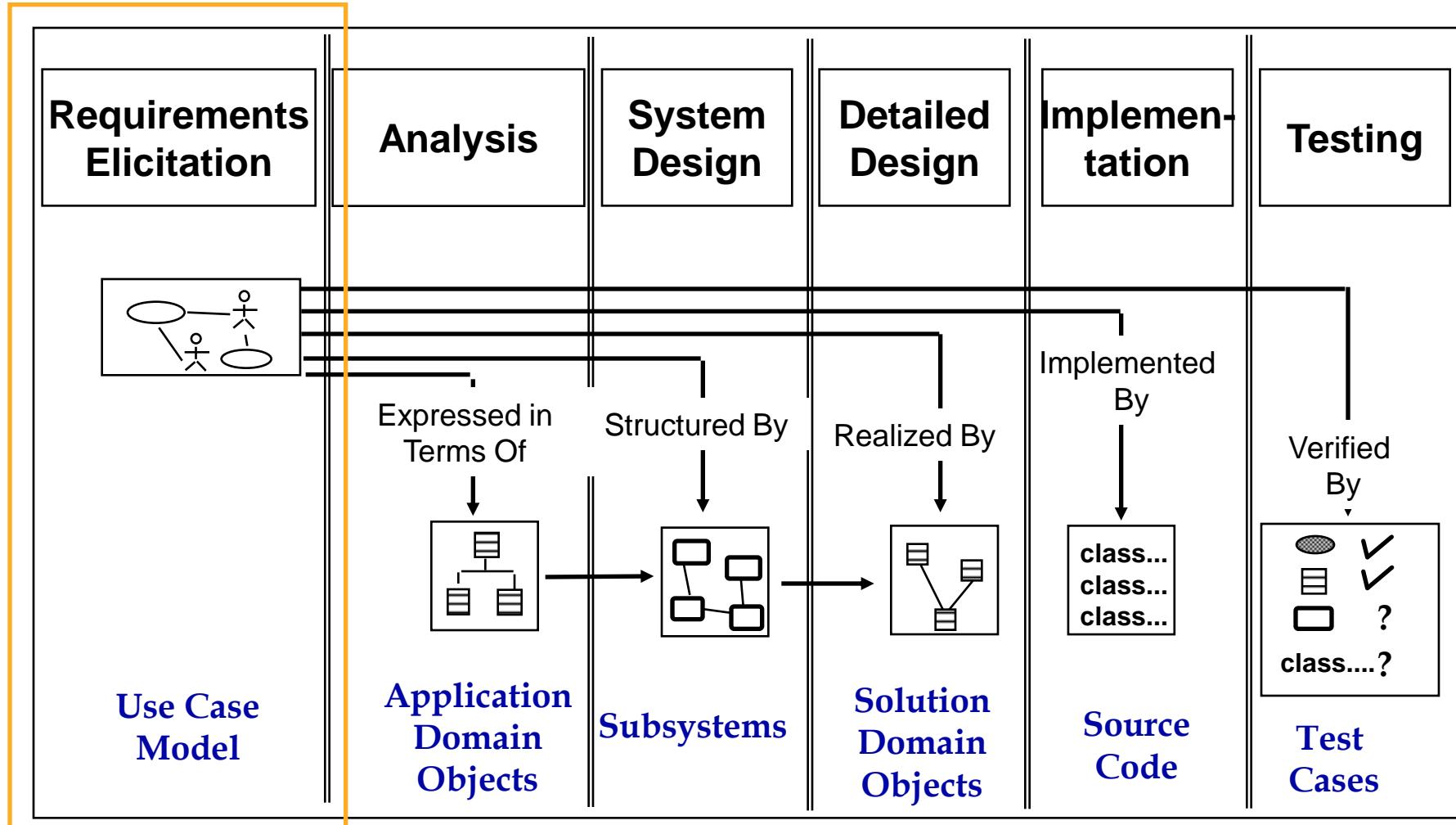
# Software Lifecycle Activities...and their models



# What is the best Software Lifecycle?

- *Answering this question is the topics of the lecture on software lifecycle modeling*
- *For now we assume we have a set of predefined activities:*
  - *Today we focus on the activity requirements elicitation*

# Software Lifecycle Activities



# What does the Customer say?



# First step in identifying the Requirements: System identification

- *Two questions need to be answered:*
  1. *How can we identify the purpose of a system?*
  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 ("Requirements specification")*
- **Analysis:**
  - *Definition of the system in terms understood by the developer (Technical specification, "Analysis model")*
- **Requirements Process:** Contains 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 concrete end user and the system
  - **Use cases:** Abstractions that describe a class of scenarios.

# Scenarios

- *Scenario (Italian: that which is pinned to the scenery)*
  - A synthetic description of an event or series of actions and events.
  - A textual description of the usage of a system. The description is written from an end user's point of view.
  - A scenario can include text, video, pictures and story boards. It usually also contains details about the work place, social situations and resource constraints.

# More Definitions

- *Scenario*: "A narrative description of what people do and experience as they try to make use of computer systems and applications" [M. Carroll, *Scenario-Based Design*, Wiley, 1995]
- A concrete, focused, informal description of a single feature of the system used by a single actor.

# Scenario-Based Design

*Scenarios can have many different uses during the software lifecycle*

- **Requirements Elicitation:** As-is scenario, visionary scenario
- **Client Acceptance Test:** Evaluation scenario
- **System Deployment:** Training scenario

*Scenario-Based Design: The use of scenarios in a software lifecycle activity*

- Scenario-based design is iterative
- Each scenario should be considered as a work document to be augmented and rearranged ("iterated upon") when the requirements, the client acceptance criteria or the deployment situation changes.

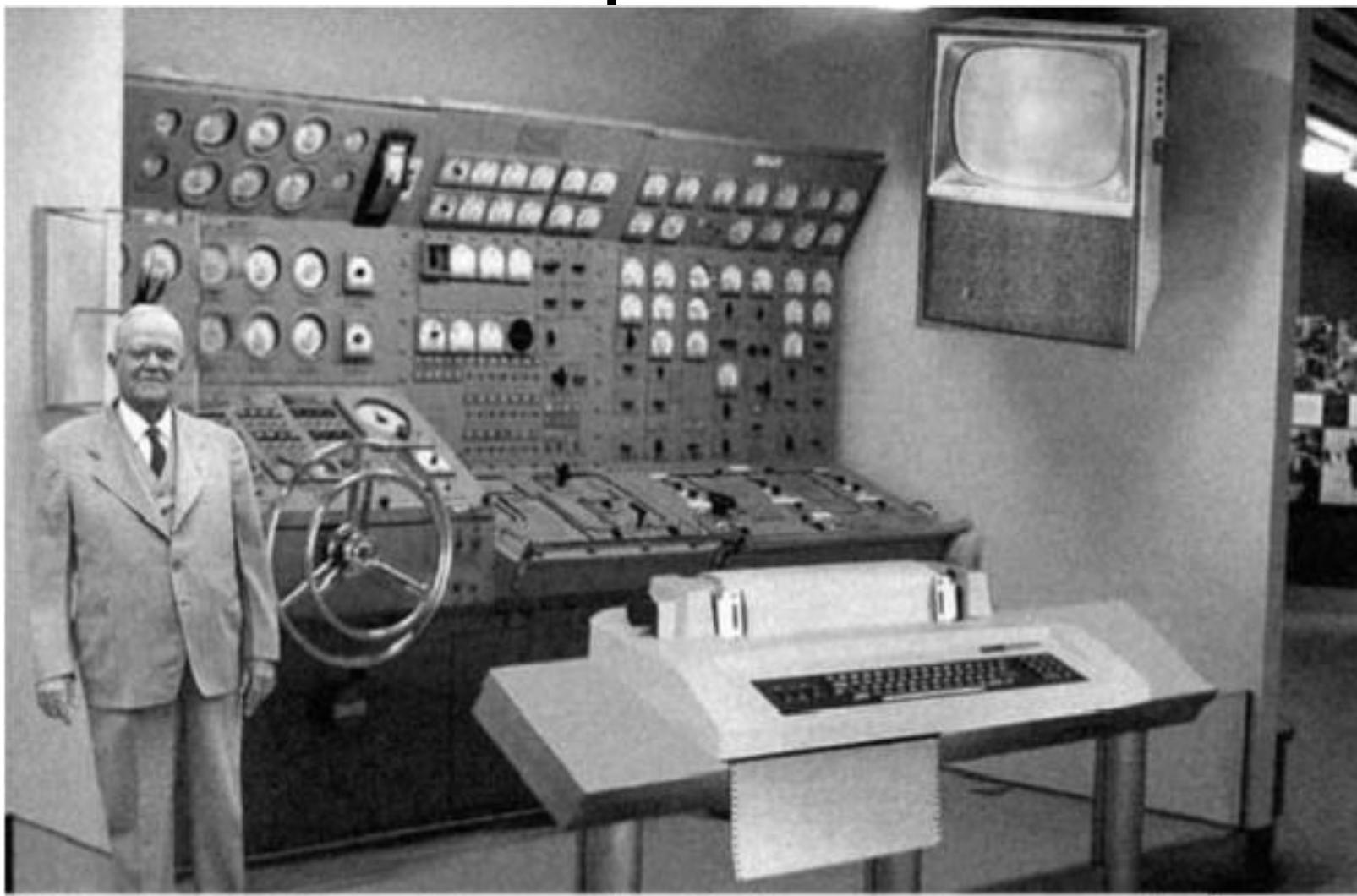
# Scenario-based Design

- *Focuses on concrete descriptions and particular instances, not abstract generic ideas*
- *It is work driven not technology driven*
- *It is open-ended, it does not try to be complete*
- *It is informal, not formal and rigorous*
- *Is about envisioned outcomes, not about specified outcomes.*

# Types of Scenarios

- *As-is scenario:*
  - Describes a current situation. Usually used in re-engineering projects. The user describes the system
    - Example: Description of Letter-Chess
- *Visionary scenario:*
  - Describes a future system. Usually used in greenfield engineering and reengineering projects
  - Can often not be done by the user or developer alone
    - Example: Description of an interactive internet-based Tic Tac Toe game tournament
    - Example: Description - in the year 1954 - of the Home Computer of the Future.

# A Visionary Scenario (1954): The Home Computer in 2004



Scientists from the RAND Corporation have created this model to illustrate how a "home computer" could look like in the year 2004. However the needed technology will not be economically feasible for the average home. Also the scientists readily admit that the computer will require not yet invented technology to actually work, but 50 years from now scientific progress is expected to solve these problems. With teletype interface and the Fortran language, the computer will be easy to use.

# Additional Types of Scenarios (2)

- *Evaluation scenario:*
  - *Description of a user task against which the system is to be evaluated.*
    - *Example:* Four users (two novice, two experts) play in a TicTac Toe tournament in ARENA.
- *Training scenario:*
  - *A description of the step by step instructions that guide a novice user through a system*
    - *Example:* How to play Tic Tac Toe in the ARENA Game Framework.

# How do we find scenarios?

- *Don't expect the client to be verbal if the system does not exist*
  - *Client understands problem domain, not the solution domain.*
- *Don't wait for information even if the system exists*
  - *"What is obvious does not need to be said"*
- *Engage in a dialectic approach*
  - *You help the client to formulate the requirements*
  - *The client helps you to understand the requirements*
  - *The requirements evolve while the scenarios are being developed*

# Heuristics for finding scenarios

- Ask yourself or the client the following questions:
  - What are the primary tasks that the system needs to perform?
  - What data will the actor create, store, change, remove or add in the system?
  - What external changes does the system need to know about?
  - What changes or events will the actor of the system need to be informed about?
- However, don't rely on *questions* and *questionnaires* alone
- Insist on *task observation* if the system already exists (*interface engineering or reengineering*)
  - Ask to speak to the end user, not just to the client
  - Expect resistance and try to overcome it.

# Scenario example: Warehouse on Fire

- *Bob, driving down main street in his patrol car notices smoke coming out of a warehouse. His partner, Alice, reports the emergency from her car.*
- *Alice enters the address of the building into her wearable computer , a brief description of its location (i.e., north west corner), and an emergency level.*
- *She confirms her input and waits for an acknowledgment.*
- *John, the dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and sends the estimated arrival time (ETA) to Alice.*
- *Alice received the acknowledgment and the ETA.*

# Observations about Warehouse on Fire Scenario

- *Concrete scenario*
  - *Describes a single instance of reporting a fire incident.*
  - *Does not describe all possible situations in which a fire can be reported.*
- *Participating actors*
  - *Bob, Alice and John*

# After the scenarios are formulated

- *Find all the use cases in the scenario that specify all instances of how to report a fire*
  - *Example: "Report Emergency" in the first paragraph of the scenario is a candidate for a use case*
- *Describe each of these use cases in more detail*
  - *Participating actors*
  - *Describe the entry condition*
  - *Describe the flow of events*
  - *Describe the exit condition*
  - *Describe exceptions*
  - *Describe nonfunctional requirements*
- *Functional Modeling (see next lecture)*

# Requirements Elicitation: Difficulties and Challenges

- *Communicate accurately about the domain and the system*
    - *People with different backgrounds must collaborate to bridge the gap between end users and developers*
      - Client and end users have *application domain knowledge*
      - Developers have *solution domain knowledge*
  - *Identify an appropriate system (Defining the system boundary)*
  - *Provide an unambiguous specification*
  - *Leave out unintended features*
- => 3 Examples.

# Defining the System Boundary is difficult

*What do you see here?*



# Defining the System Boundary is difficult

*What do you see now?*



# Defining the System Boundary is difficult

*What do you see now?*



# Example of an Ambiguous Specification

*During a laser experiment, a laser beam was directed from earth to a mirror on the Space Shuttle Discovery*

*The laser beam was supposed to be reflected back towards a mountain top 10,023 feet high*

*The operator entered the elevation as "10023"*

*The light beam never hit the mountain top  
What was the problem?*

*The computer interpreted the number in miles...*

# Example of an Unintended Feature

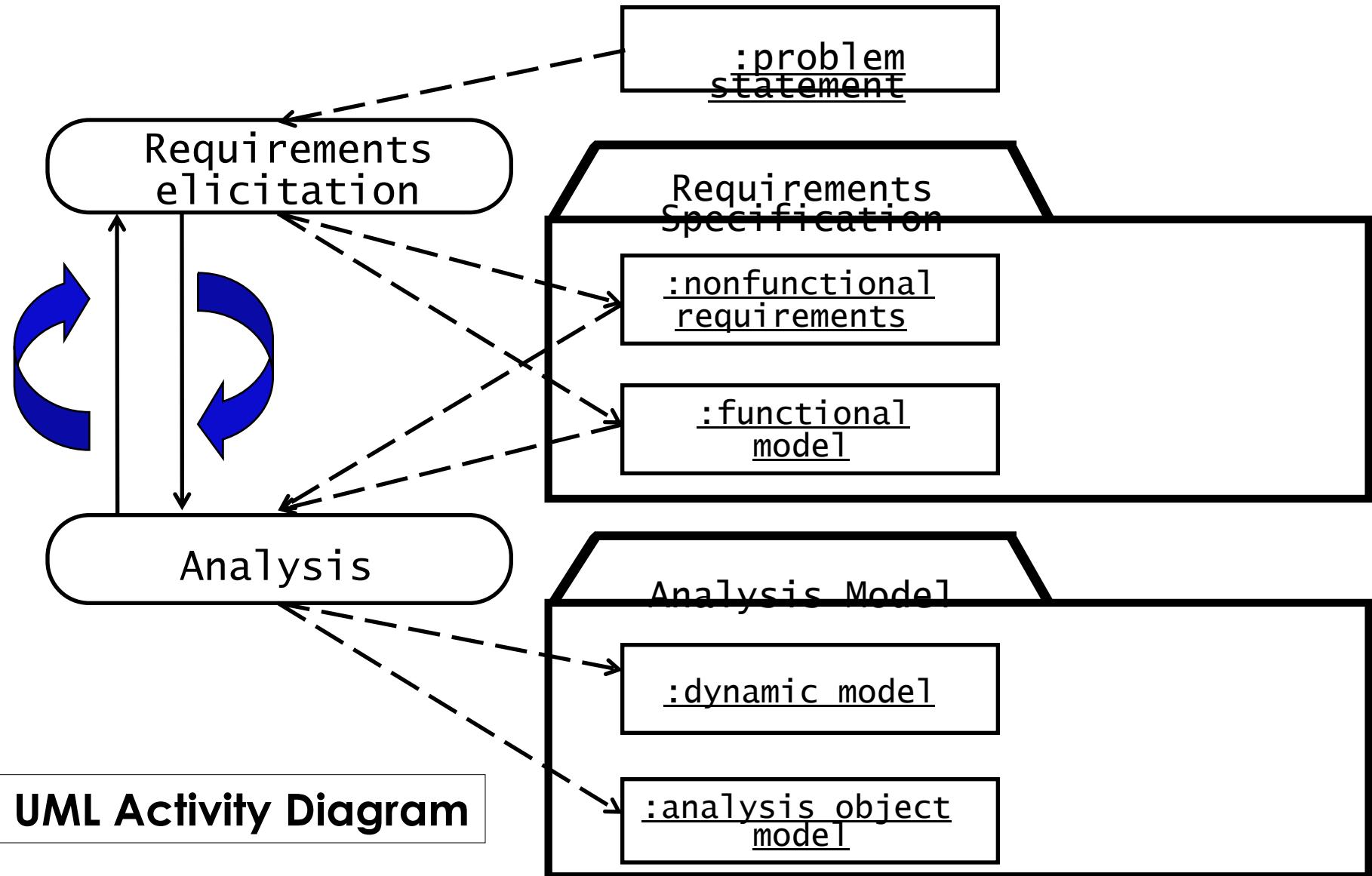
## From the News: London underground train leaves station without driver!

*What happened?*

- *A passenger door was stuck and did not close*
- *The driver left his train to close the passenger door*
  - *He left the driver door open*
  - *He relied on the specification that said the train does not move if at least one door is open*
- *When he shut the passenger door, the train left the station without him*
  - *The driver door was not treated as a door in the source code!*



# Requirements Process



# Requirements Specification vs Analysis Model

*Both focus on the requirements from the user's view of the system*

- *The requirements specification uses natural language (derived from the problem statement)*
- *The analysis model uses a formal or semi-formal notation*
  - We use UML.

# Types of Requirements

- *Functional requirements*
  - *Describe the interactions between the system and its environment independent from the implementation*  
"An operator must be able to define a new game."
- *Nonfunctional requirements*
  - *Aspects not directly related to functional behavior.*  
"The response time must be less than 1 second"
- *Constraints*
  - *Imposed by the client or the environment*
    - "The implementation language must be Java "
  - *Called "Pseudo requirements" in the text book.*

# Functional vs. Nonfunctional Requirements

## *Functional Requirements*

- *Describe user tasks that the system needs to support*
- *Phrased as actions*

*"Advertise a new league"*

*"Schedule tournament"*

*"Notify an interest group"*

## *Nonfunctional Requirements*

- *Describe properties of the system or the domain*
- *Phrased as constraints or negative assertions*

*"All user inputs should be acknowledged within 1 second"*

*"A system crash should not result in data loss".*

# Types of Nonfunctional Requirements

*Quality requirements*

*Constraints or  
Pseudo requirements*

# Types of Nonfunctional Requirements

- *Usability*
- *Reliability*
  - *Robustness*
  - *Safety*
- *Performance*
  - *Response time*
  - *Scalability*
  - *Throughput*
  - *Availability*
- *Supportability*
  - *Adaptability*
  - *Maintainability*

*Quality requirements*

*Constraints or  
Pseudo requirements*

# Types of Nonfunctional Requirements

- *Usability*
  - *Reliability*
    - *Robustness*
    - *Safety*
  - *Performance*
    - *Response time*
    - *Scalability*
    - *Throughput*
    - *Availability*
  - *Supportability*
    - *Adaptability*
    - *Maintainability*
- Quality requirements*
- *Implementation*
  - *Interface*
  - *Operation*
  - *Packaging*
  - *Legal*
    - *Licensing (GPL, LGPL)*
    - *Certification*
    - *Regulation*
- Constraints or  
Pseudo requirements*

# Task

- *Find definitions for all the nonfunctional requirements on the previous slide and learn them by heart*
- *Understand their meaning and scope (their applicability).*

# Some Quality Requirements Definitions

- **Usability**
  - The ease with which actors can use a system to perform a function
  - Usability is one of the most frequently misused terms ("The system is easy to use")
  - **Usability** must be **measurable**, otherwise it is **marketing**
    - Example: Specification of the number of steps – the measure! - to perform a internet-based purchase with a web browser
- **Robustness**: The ability of a system to maintain a function
  - even if the user enters a wrong input
  - even if there are changes in the environment
    - Example: The system can tolerate temperatures up to 90 C
- **Availability**: The ratio of the expected uptime of a system to the aggregate of the expected up and down time
  - Example: The system is down not more than 5 minutes per week.

# Nonfunctional Requirements: Examples

- "*Spectators must be able to watch a match without prior registration and without prior knowledge of the match.*"
  - *Usability Requirement*
- "*The system must support 10 parallel tournaments*"
  - *Performance Requirement*
- "*The operator must be able to add new games without modifications to the existing system.*"
  - *Supportability Requirement*

# What should not be in the Requirements?

- *System structure, implementation technology*
- *Development methodology*
  - *Parnas, How to fake the software development process*
- *Development environment*
- *Implementation language*
- *Reusability*
- *It is desirable that none of these above are constrained by the client.*

# Requirements Validation

*Requirements validation is a quality assurance step, usually performed after requirements elicitation or after analysis*

- **Correctness:**
  - *The requirements represent the client's view*
- **Completeness:**
  - *All possible scenarios, in which the system can be used, are described*
- **Consistency:**
  - *There are no requirements that contradict each other.*

# Requirements Validation (2)

- *Clarity:*
  - Requirements can only be interpreted in one way
- *Realism:*
  - Requirements can be implemented and delivered
- *Traceability:*
  - Each system behavior can be traced to a set of functional requirements
- *Problems with requirements validation:*
  - Requirements change quickly during requirements elicitation
  - Inconsistencies are easily added with each change
  - Tool support is needed!

# We can specify Requirements for “Requirements Management”

- *Functional requirements:*
  - *Store the requirements in a shared repository*
  - *Provide multi-user access to the requirements*
  - *Automatically create a specification document from the requirements*
  - *Allow change management of the requirements*
  - *Provide traceability of the requirements throughout the artifacts of the system.*

# Tools for Requirements Management (2)

## *DOORS* (Telelogic)

- *Multi-platform requirements management tool, for teams working in the same geographical location.*  
*DOORS XT for distributed teams*

## *RequisitePro* (IBM/Rational)

- *Integration with MS Word*
- *Project-to-project comparisons via XML baselines*

## *RD-Link* (<http://www.ring-zero.com>)

- *Provides traceability between RequisitePro & Telelogic DOORS*

## *Unicase* (<http://unicase.org>)

- *Research tool for the collaborative development of system models*
- *Participants can be geographically distributed.*

# Different Types of Requirements Elicitation

- *Greenfield Engineering*
  - *Development starts from scratch, no prior system exists, requirements come from end users and clients*
  - *Triggered by user needs*
- *Re-engineering*
  - *Re-design and/or re-implementation of an existing system using newer technology*
  - *Triggered by technology enabler*
- *Interface Engineering*
  - *Provision of existing services in a new environment*
  - *Triggered by technology enabler or new market needs*

# Prioritizing requirements

- *High priority*
  - *Addressed during analysis, design, and implementation*
  - *A high-priority feature must be demonstrated*
- *Medium priority*
  - *Addressed during analysis and design*
  - *Usually demonstrated in the second iteration*
- *Low priority*
  - *Addressed only during analysis*
  - *Illustrates how the system is going to be used in the future with not yet available technology*

# Requirements Analysis Document Template

- 1. Introduction*
- 2. Current system*
- 3. Proposed system*
  - 3.1 Overview*
  - 3.2 Functional requirements*
  - 3.3 Nonfunctional requirements*
  - 3.4 Constraints ("Pseudo requirements")*
  - 3.5 System models*
    - 3.5.1 Scenarios*
    - 3.5.2 Use case model*
    - 3.5.3 Object model*
      - 3.5.3.1 Data dictionary*
      - 3.5.3.2 Class diagrams*
    - 3.5.4 Dynamic models*
    - 3.5.5 User interface*
  - 4. Glossary*

# Section 3.3 Nonfunctional Requirements

*3.3.1 User interface and human factors*

*3.3.2 Documentation*

*3.3.3 Hardware considerations*

*3.3.4 Performance characteristics*

*3.3.5 Error handling and extreme conditions*

*3.3.6 System interfacing*

*3.3.7 Quality issues*

*3.3.8 System modifications*

*3.3.9 Physical environment*

*3.3.10 Security issues*

*3.3.11 Resources and management issues*

# Nonfunctional Requirements (Questions to overcome “Writers block”)

## *User interface and human factors*

- *What type of user will be using the system?*
- *Will more than one type of user be using the system?*
- *What training will be required for each type of user?*
- *Is it important that the system is easy to learn?*
- *Should users be protected from making errors?*
- *What input/output devices are available*

## *Documentation*

- *What kind of documentation is required?*
- *What audience is to be addressed by each document?*

# Nonfunctional Requirements (2)

## *Hardware considerations*

- *What hardware is the proposed system to be used on?*
- *What are the characteristics of the target hardware, including memory size and auxiliary storage space?*

## *Performance characteristics*

- *Are there speed, throughput, response time constraints on the system?*
- *Are there size or capacity constraints on the data to be processed by the system?*

## *Error handling and extreme conditions*

- *How should the system respond to input errors?*
- *How should the system respond to extreme conditions?*

# Nonfunctional Requirements (3)

## *System interfacing*

- *Is input coming from systems outside the proposed system?*
- *Is output going to systems outside the proposed system?*
- *Are there restrictions on the format or medium that must be used for input or output?*

## *Quality issues*

- *What are the requirements for reliability?*
- *Must the system trap faults?*
- *What is the time for restarting the system after a failure?*
- *Is there an acceptable downtime per 24-hour period?*
- *Is it important that the system be portable?*

# Nonfunctional Requirements (4)

## *System Modifications*

- *What parts of the system are likely to be modified?*
- *What sorts of modifications are expected?*

## *Physical Environment*

- *Where will the target equipment operate?*
- *Is the target equipment in one or several locations?*
- *Will the environmental conditions be ordinary?*

## *Security Issues*

- *Must access to data or the system be controlled?*
- *Is physical security an issue?*

# Nonfunctional Requirements (5)

## *Resources and Management Issues*

- *How often will the system be backed up?*
- *Who will be responsible for the back up?*
- *Who is responsible for system installation?*
- *Who will be responsible for system maintenance?*

# Example: Heathrow Luggage System

- *On April 5, 2008 a system update was performed to upgrade the baggage handling:*
  - *50 flights were canceled on the day of the update*
  - *A "Bag Backlog" of 20,000 bags was produced (Naomi Campbell had a fit and was arrested)*
  - *The bags were resorted in Italy and eventually sent to the passengers via Federal Express*
- *What happened? Initial explanation:*
  - *Computer failure in the high storage bay area in combination with shortage of personal*

# Exercise

- Reverse engineer the requirements for the Heathrow luggage system
  - Use the requirements analysis document template
  - Use available information on the internet.
- Questions to ask:
  - How are the bags stored after passengers have checked, but before they enter the plane?
  - How are the bags retrieved from the storage area?
  - What about existing luggage systems ("legacy systems")?
  - Scalability: How many users should the new luggage system support? How can this be tested before deployment?
  - Throughput: How many suit cases/hour need to be supported?

# Bonus Question

- *What changes to the requirements should have been done to avoid the Heathrow desaster?*

# Exercise Solution

- Available information on the internet. Examples
  - <http://blogs.zdnet.com/projectfailures/?p=610>
  - <http://www.bloomberg.com/apps/news?pid=conewsstory&refer=conews&tkr=FDX:US&sid=aY4IqhBRcytA>
- Examples of requirements:
  - Automate the processing of No-Show passengers
  - Use a high bay storage area ("high rack warehouse")
  - Provide a chaotic storage capability
  - Combine two existing luggage systems ("legacy systems"): Early (hours before) and last minute checkins
  - The system must be tested with 2500 volunteers
  - The throughput must be at least 12000 suitcases/hour.
  - 
  -

# Additional Readings

- *Scenario-Based Design*
  - *John M. Carroll, Scenario-Based Design: Envisioning Work and Technology in System Development, John Wiley, 1995*
  - *Usability Engineering: Scenario-Based Development of Human Computer Interaction, Morgan Kaufman, 2001*
- *Heathrow Luggage System:*
  - <http://blogs.zdnet.com/projectfailures/?p=610>
  - <http://www.bloomberg.com/apps/news?pid=conewsstory&refer=conews&tkr=FDX:US&sid=aY4IqhBRcytA>

*Additional Information about Heathrow (In German)*

*Panne auf Flughöhe Null*

*(Spiegel):*<http://www.spiegel.de/reise/aktuell/0,1518,544768,00.html>

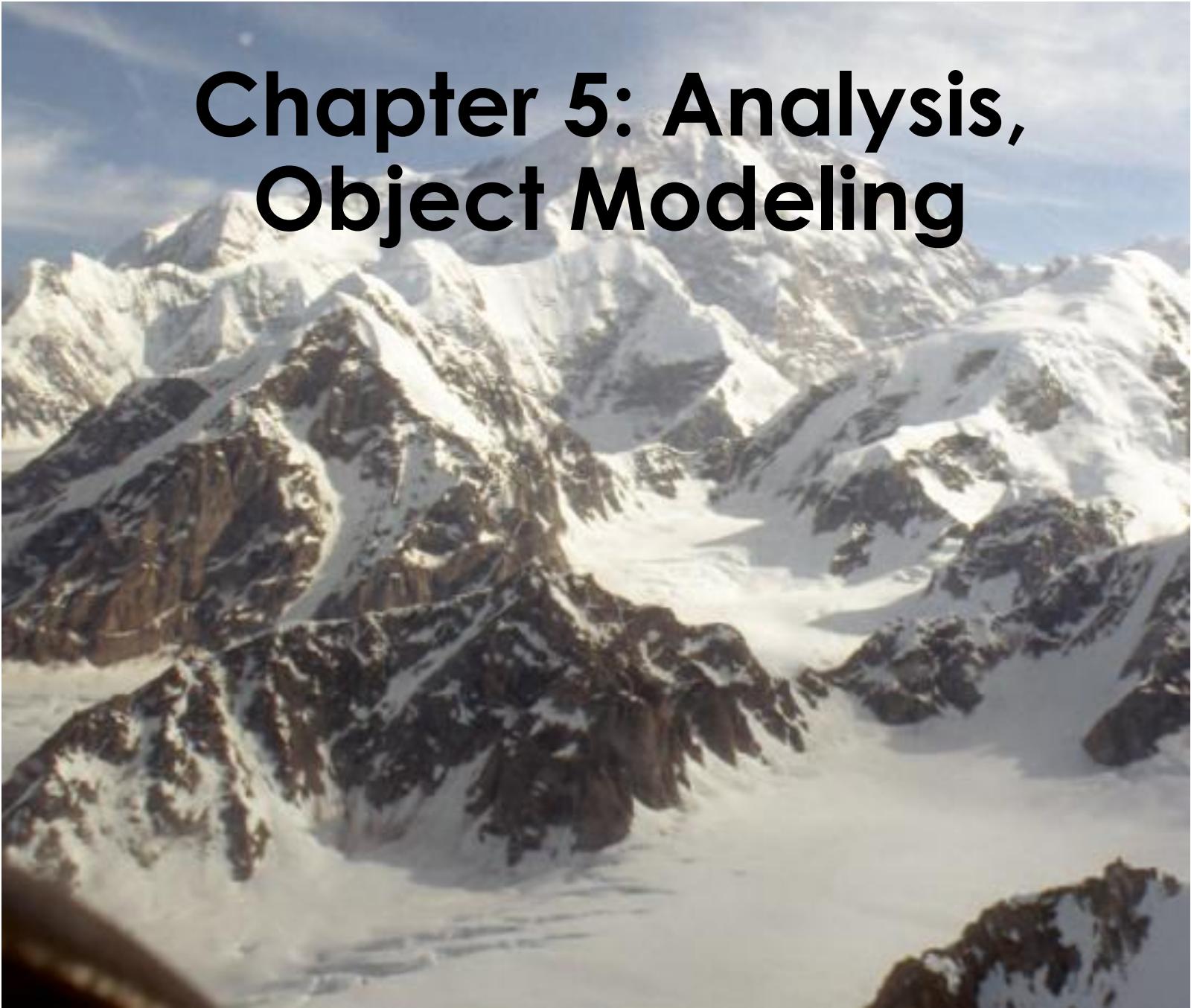
*Zurück in das rotierende Chaos (FAZ):*

<http://www.faz.net/s/Rub7F4BEE0E0C39429A8565089709B70C44/Doc~EC1120B27386C4E34A67A5EE8E5523433~ATpl~Ecommon~Scontent.html>

# **Object-Oriented Software Engineering**

Using UML, Patterns, and Java

## **Chapter 5: Analysis, Object Modeling**

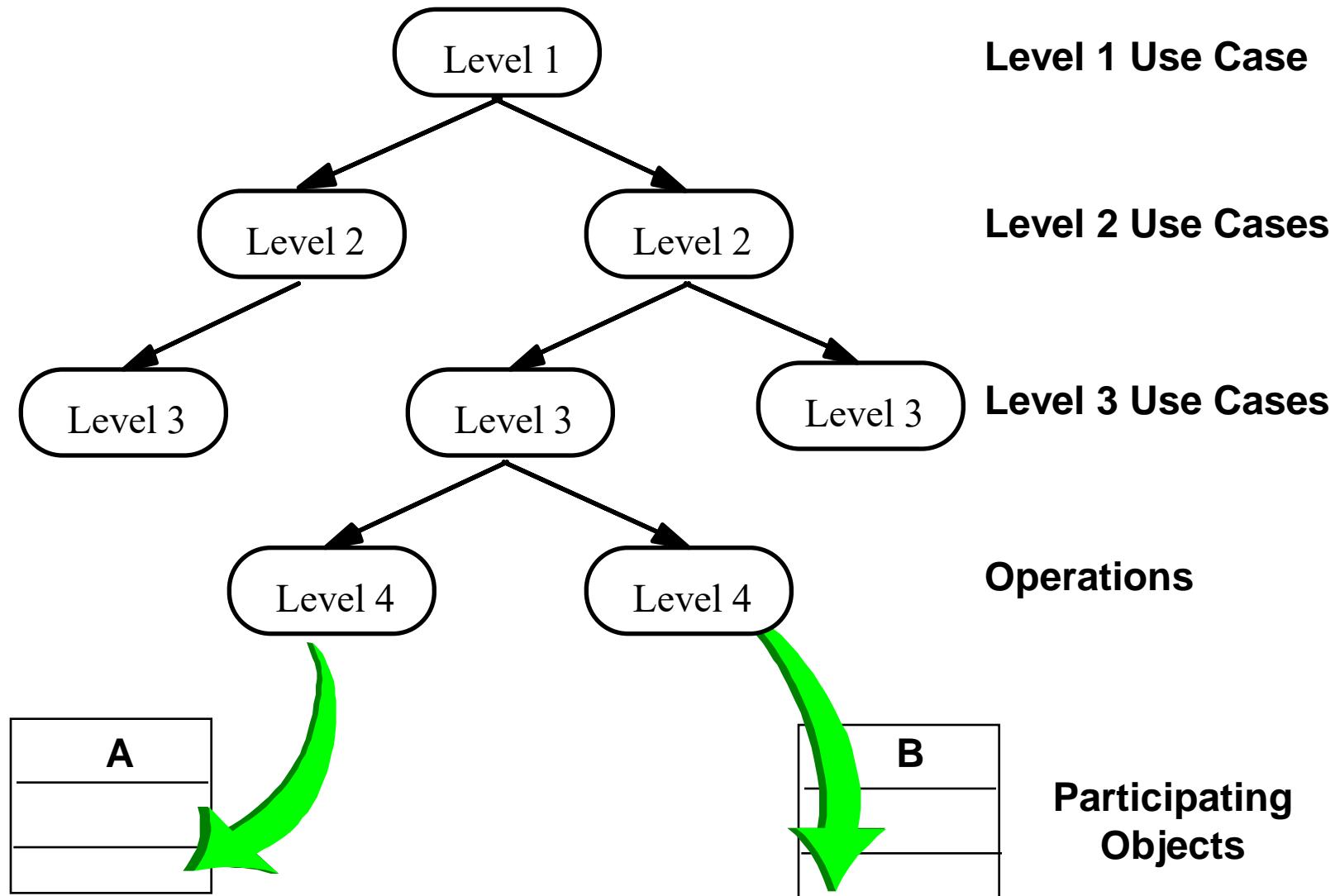


# Outline

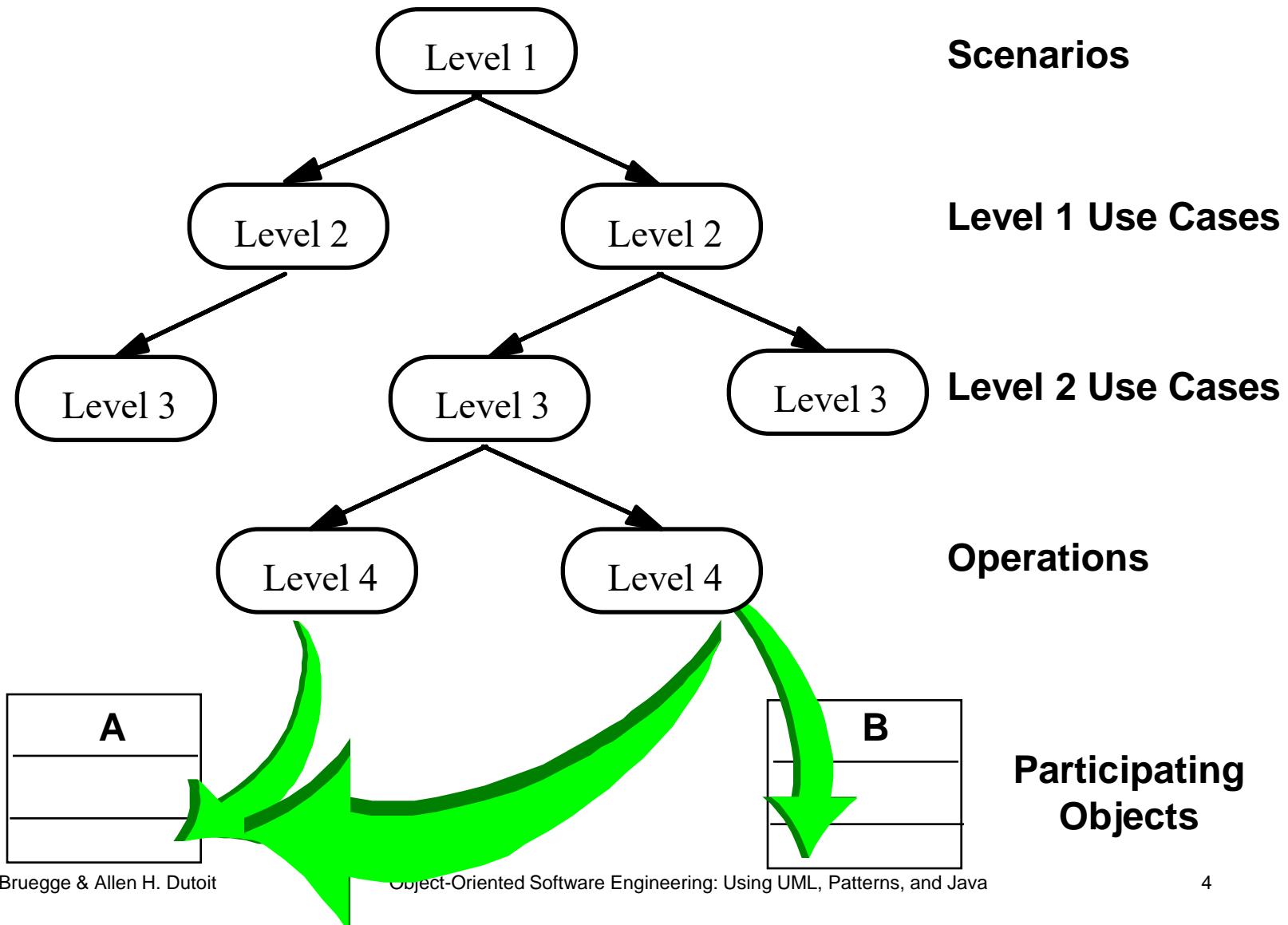
*Recall: System modeling = Functional modeling + Object modeling + Dynamic modeling*

- ✓ *Last week: Functional modeling*
- *Today: Object modeling*
  - *Activities during object modeling*
  - *Object identification*
  - *Object types*
    - *Entity, boundary and control objects*
  - *Stereotypes*
  - *Abott's technique*
    - *Helps in object identification.*

# From Use Cases to Objects



# From Use Cases to Objects: Why Functional Decomposition is not Enough



# Activities during Object Modeling

*Main goal: Find the important abstractions*

- *Steps during object modeling*

 *Class identification*

- *Based on the fundamental assumption that we can find abstractions*

2. *Find the attributes*

3. *Find the methods*

4. *Find the associations between classes*

- *Order of steps*

- *Goal: get the desired abstractions*

- *Order of steps secondary, only a heuristic*

- *What happens if we find the wrong abstractions?*

- *We iterate and revise the model*

# Class Identification

*Class identification is crucial to object-oriented modeling*

- *Helps to identify the important entities of a system*
- *Basic assumptions:*
  1. We can find the classes for a new software system  
*(Forward Engineering)*
  2. We can identify the classes in an existing system  
*(Reverse Engineering)*
- *Why can we do this?*
  - *Philosophy, science, experimental evidence.*

# Class Identification

- *Approaches*
  - *Application domain approach*
    - Ask application domain experts to identify relevant abstractions
  - *Syntactic approach*
    - Start with use cases
    - Analyze the text to identify the objects
    - Extract participating objects from flow of events
  - *Design patterns approach*
    - Use reusable design patterns
  - *Component-based approach*
    - Identify existing solution classes.

# Class identification is a Hard Problem

- *One problem: Definition of the system boundary:*
  - *Which abstractions are outside, which abstractions are inside the system boundary?*
    - *Actors are outside the system*
    - *Classes/Objects are inside the system.*
- *An other problem: Classes/Objects are not just found by taking a picture of a scene or domain*
  - *The application domain has to be analyzed*
  - *Depending on the purpose of the system different objects might be found*
    - *How can we identify the purpose of a system?*
    - *Scenarios and use cases => Functional model*

# There are different types of Objects

- *Entity Objects*
  - *Represent the persistent information tracked by the system (Application domain objects, also called "Business objects")*
- *Boundary Objects*
  - *Represent the interaction between the user and the system*
- *Control Objects*
  - *Represent the control tasks performed by the system.*

# Example: 2BWatch Modeling

*To distinguish different object types  
in a model we can use the  
UML Stereotype mechanism*

Year

Month

Day

ChangeDate

Button

LCDDisplay

*Entity Objects*

*Control Object*

*Boundary Objects*

# Naming Object Types in UML

- UML provides the *stereotype* mechanism to introduce *new types* of modeling elements
  - A stereotype is drawn as a name enclosed by angled double-quotes (“guillemets”) (`<<`, `>>`) and placed before the name of a UML element (class, method, attribute, ....)
  - Notation: `<<String>>Name`

`<<Entity>>`  
Year

`<<Control>>`  
ChangeDate

`<<Boundary>>`  
Button

`<<Entity>>`  
Month

`<<Boundary>>`  
LCDDisplay

`<<Entity>>`  
Day

*Entity Object*

*Control Object*

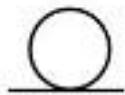
*Boundary Object*

# UML is an Extensible Language

- Stereotypes allow you to extend the vocabulary of the UML so that you can create new model elements, derived from existing ones
- Examples:
  - Stereotypes can also be used to classify method behavior such as <<constructor>>, <<getter>> or <<setter>>
  - To indicate the interface of a subsystem or system, one can use the stereotype <<interface>> (Lecture System Design)
- Stereotypes can be represented with icons and graphics:
  - *This can increase the readability of UML diagrams.*

# Icons for Stereotypes

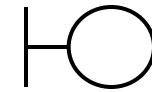
- One can use **icons** to identify a stereotype
  - When the stereotype is applied to a UML model element, the icon is displayed beside or above the name



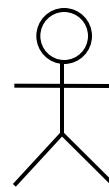
**Year**



**ChangeDate**



**Button**



**WatchUser**

*Entity Object*

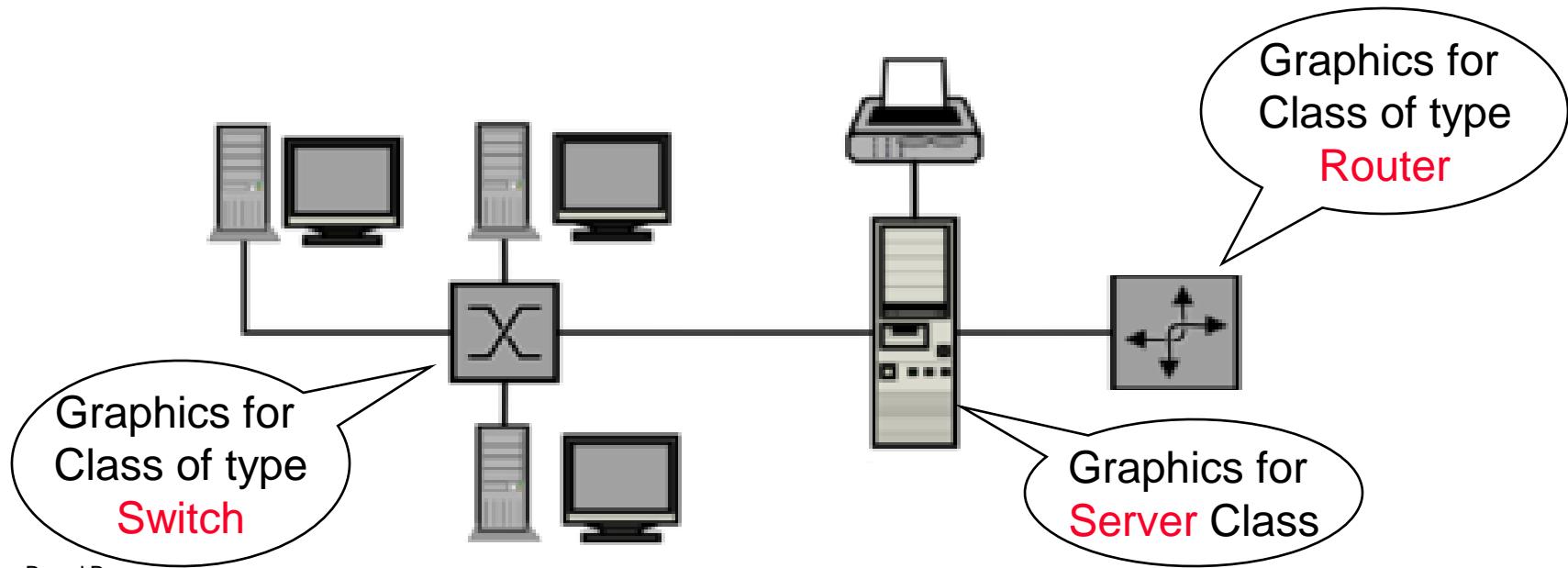
*Control Object*

*Boundary Object*

*Actor*

# Graphics for Stereotypes

- One can also use **graphical symbols** to identify a stereotype
  - When the stereotype is applied to a UML model element, the graphic replaces the default graphic for the diagram element.
- Example: When modeling a network, define graphics for representing classes of type **Switch**, **Server**, **Router**, **Printer**,etc.



# Pros and Cons of Stereotype Graphics

- Advantages:
  - UML diagrams may be easier to understand if they contain graphics and icons for stereotypes
    - This can increase the readability of the diagram, especially if the client is not trained in UML
    - And they are still UML diagrams!
- Disadvantages:
  - If developers are unfamiliar with the symbols being used, it can become much harder to understand what is going on
  - Additional symbols add to the burden of learning to read the diagrams.

# Object Types allow us to deal with Change

- *Having three types of object leads to models that are more resilient to change*
  - *The interface of a system changes more likely than the control*
  - *The way the system is controlled changes more likely than entities in the application domain*
- *Object types originated in Smalltalk:*
  - *Model, View, Controller (MVC)*  
*Model <-> Entity Object*  
*View <-> Boundary Object*  
*Controller <-> Control Object*
- *Next topic: Finding objects.*

# Finding Participating Objects in Use Cases

- *Pick a use case and look at flow of events*
- *Do a textual analysis (noun-verb analysis)*
  - *Nouns are candidates for objects/classes*
  - *Verbs are candidates for operations*
  - *This is also called Abbott's Technique*
- *After objects/classes are found, identify their types*
  - *Identify real world entities that the system needs to keep track of (FieldOfficer  $\bowtie$  Entity Object)*
  - *Identify real world procedures that the system needs to keep track of (EmergencyPlan  $\bowtie$  Control Object)*
  - *Identify interface artifacts (PoliceStation  $\bowtie$  Boundary Object).*

# Example for using the Technique

## Flow of Events:

- The customer enters the store to buy a toy.
- It has to be a toy that his daughter likes and it must cost less than 50 Euro.
- He tries a videogame, which uses a data glove and a head-mounted display. He likes it.
- An assistant helps him.
- The suitability of the game depends on the age of the child.
- His daughter is only 3 years old.
- The assistant recommends another type of toy, namely the boardgame "Monopoly".

# Mapping parts of speech to model components (Abbot's Technique)

<i>Example</i>	<i>Part of speech</i>	<i>UML model component</i>
“Monopoly”	Proper noun	object
Toy	Improper noun	class
Buy, recommend	Doing verb	operation
is-a	being verb	inheritance
has an	having verb	aggregation
must be	modal verb	constraint
dangerous	adjective	attribute
enter	transitive verb	operation
depends on	intransitive verb	Constraint, class, association

# Generating a Class Diagram from Flow of Events

Customer

store

enter()

daughter

age

suitable

Toy

price

buy()

like()

videogame

boardgame

*Flow of events:*

- The customer enters the store to store a toy. It has to be a toy that his daughter likes and it must cost less than 50 Euro. He tries a videogame, which less than 50 glove and a head-mounted videogame. He likes it.

An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely a boardgame. The customer buy the game and leaves the store

boardgame

# Ways to find Objects

- *Syntactical investigation with Abbot's technique:*
  - *Flow of events in use cases*
  - *Problem statement*
- *Use other knowledge sources:*
  - *Application knowledge: End users and experts know the abstractions of the application domain*
  - *Solution knowledge: Abstractions in the solution domain*
  - *General world knowledge: Your generic knowledge and intuition*

# Order of Activities for Object Identification

1. *Formulate a few scenarios with help from an end user or application domain expert*
2. *Extract the use cases from the scenarios, with the help of an application domain expert*
3. *Then proceed in parallel with the following:*
  - *Analyse the flow of events in each use case using Abbot's textual analysis technique*
  - *Generate the UML class diagram.*

# Steps in Generating Class Diagrams

1. *Class identification (textual analysis, domain expert)*
2. *Identification of attributes and operations (sometimes before the classes are found!)*
3. *Identification of associations between classes*
4. *Identification of multiplicities*
5. *Identification of roles*
6. *Identification of inheritance*

# Who uses Class Diagrams?

- *Purpose of class diagrams*
  - *The description of the static properties of a system*
- *The main users of class diagrams:*
  - *The application domain expert*
    - *uses class diagrams to model the application domain (including taxonomies)*
      - *during requirements elicitation and analysis*
    - *The developer*
      - *uses class diagrams during the development of a system*
        - *during analysis, system design, object design and implementation.*

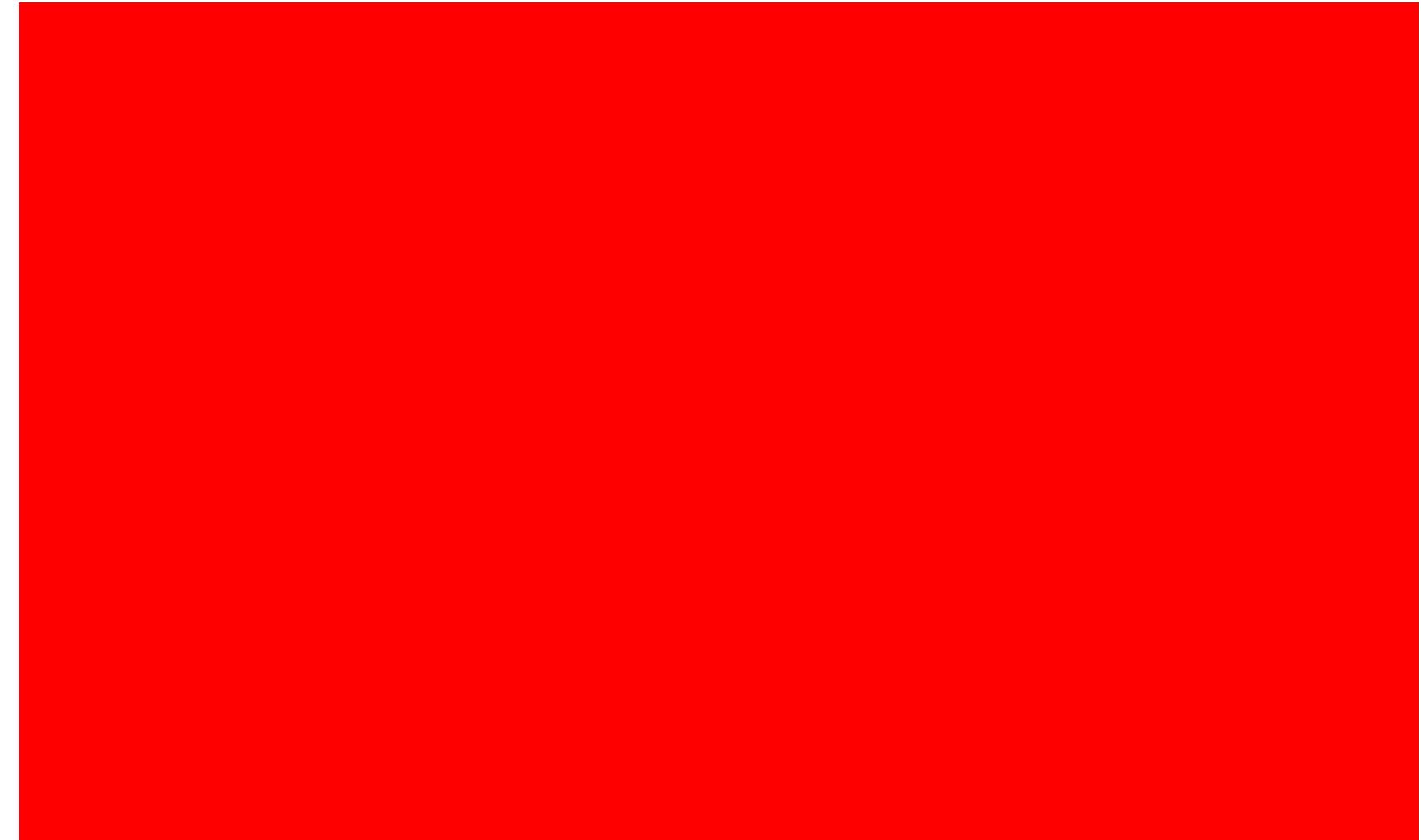
# Who does not use Class Diagrams?

- *The client and the end user are usually not interested in class diagrams*
  - *Clients focus more on project management issues*
  - *End users are more interested in the functionality of the system.*

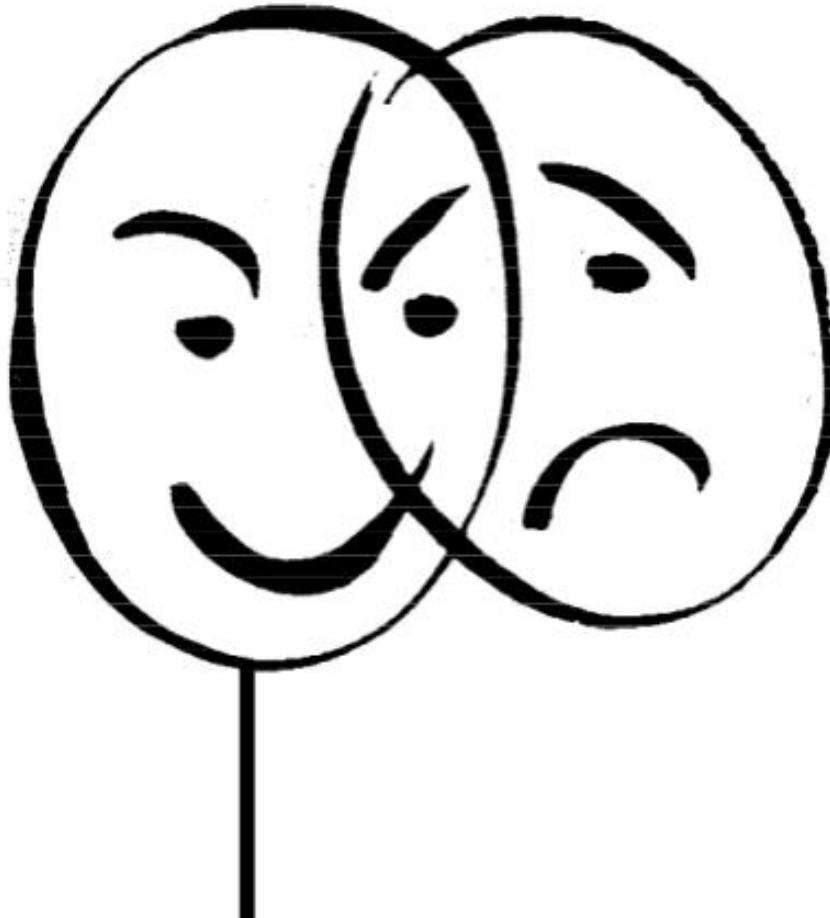
# Summary

- *System modeling*
  - *Functional modeling+object modeling+dynamic modeling*
- *Functional modeling*
  - *From scenarios to use cases to objects*
- *Object modeling is the central activity*
  - *Class identification is a major activity of object modeling*
  - *Easy syntactic rules to find classes and objects*
  - *Abbot's Technique*
- *Class diagrams are the “center of the universe” for the object-oriented developer*
  - *The end user focuses more on the functional model and and usability.*

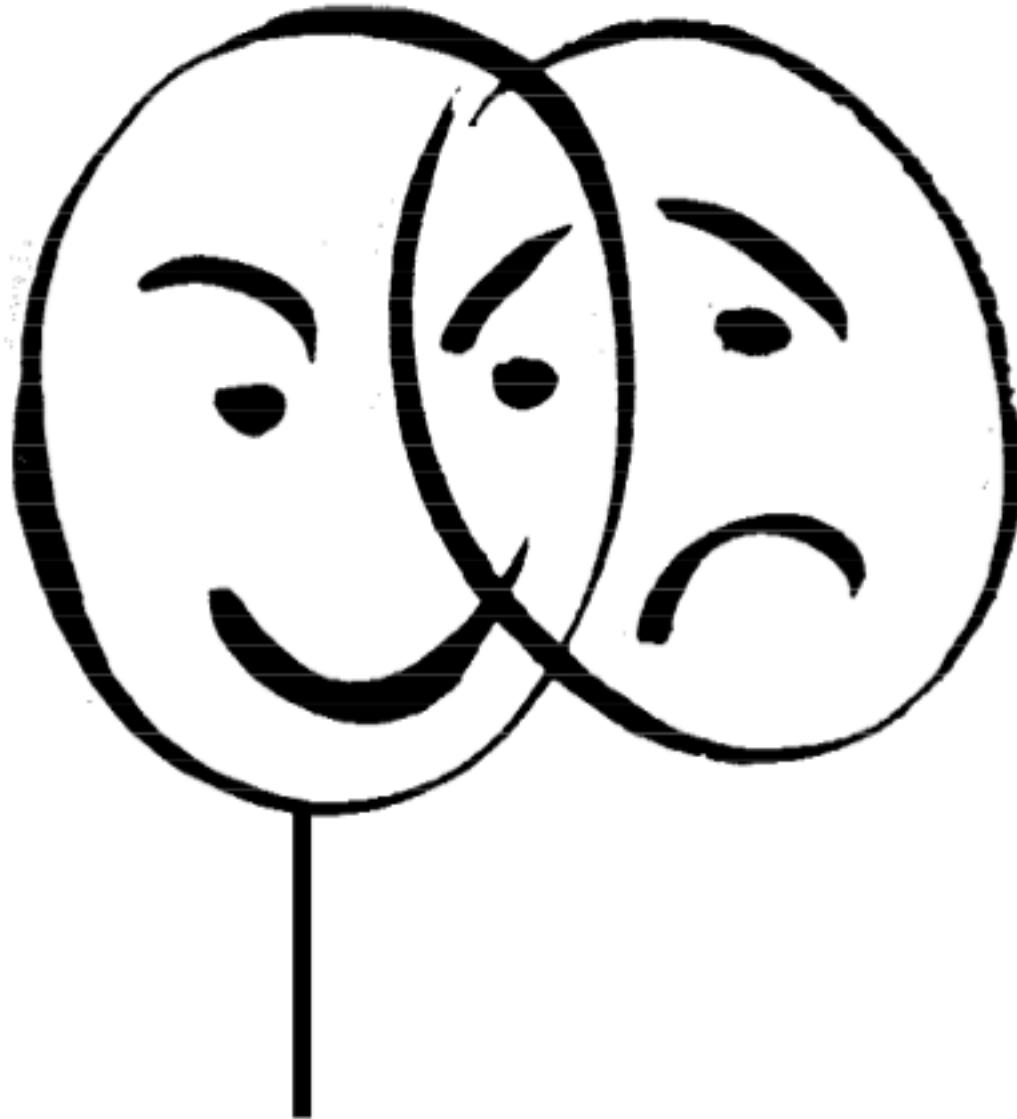
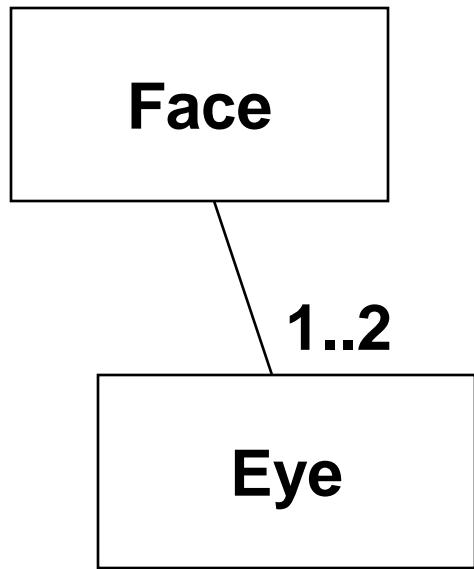
# Additional Slides



# What is This?



# What is This?



# Modeling in Action

- *If it is a Face*
  - *What are its Attributes?*
  - *Sad, Happy*
- *Or is it a Mask?*
- *Investigate the functional model*
  - *Who is using it? -> Actors*
    - *Art collector*
    - *Bankrobber*
    - *Carnival participant*
  - *How is it used? -> Event flow*
- *"Napkin design" of a Mask to be used at the Venetian Carnival*



# Pieces of an Object Model

- *Classes and their instances ("objects")*
- *Attributes*
- *Operations*
- *Associations between classes and objects*

# Associations

- *Types of Associations*
  - *Canonical associations*
    - *Part-of Hierarchy (Aggregation)*
    - *Kind-of Hierarchy (Inheritance)*
  - *Generic associations*

# Attributes

- *Detection of attributes is application specific*
- *Attributes in one system can be classes in another system*
- *Turning attributes to classes and vice versa*

# Operations

- *Source of operations*
  - *Use cases in the functional model*
  - *General world knowledge*
  - *Generic operations: Get/Set*
  - *Design Patterns*
  - *Application domain specific operations*
  - *Actions and activities in the dynamic model*

# Object vs Class

- *Object (instance): Exactly one thing*
  - *This lecture on object modeling*
- *A class describes a group of objects with similar properties*
  - *Game, Tournament, mechanic, car, database*
- *Object diagram: A graphical notation for modeling objects, classes and their relationships*
  - *Class diagram: Template for describing many instances of data. Useful for taxonomies, patterns, schemata...*
  - *Instance diagram: A particular set of objects relating to each other. Useful for discussing scenarios, test cases and examples*

# Developers have different Views on Class Diagrams

- *According to the development activity, a developer plays different roles:*
  - *Analyst*
  - *System Designer*
  - *Object Designer*
  - *Implementor*
- *Each of these roles has a different view about the class diagram (the object model).*

# The View of the Analyst

- *The analyst is interested*
  - *in application classes: The associations between classes are relationships between abstractions in the application domain*
  - *operations and attributes of the application classes (difference to E/R models!)*
- *The analyst uses inheritance in the model to reflect the taxonomies in the application domain*
  - *Taxonomy: An is-a-hierarchy of abstractions in an application domain*
- *The analyst is not interested*
  - *in the exact signature of operations*
  - *in solution domain classes*

# The View of the Designer

- *The designer focuses on the solution of the problem, that is, the solution domain*
- *The associations between classes are now references (pointers) between classes in the application or solution domain*
- *An important design task is the specification of interfaces:*
  - *The designer describes the interface of classes and the interface of subsystems*
  - *Subsystems originate from modules (term often used during analysis):*
    - *Module: a collection of classes*
    - *Subsystem: a collection of classes with an interface*
- *Subsystems are modeled in UML with a package.*

# Goals of the Designer

- *The most important design goals for the designer are design usability and design reusability*
- *Design usability:* the interfaces are usable from as many classes as possible within in the system
- *Design reusability:* The interfaces are designed in a way, that they can also be reused by other (future) software systems
  - => Class libraries
  - => Frameworks
  - => Design patterns.

# The View of the Implementor

- *Class implementor*
  - Must realize the interface of a class in a programming language
  - Interested in appropriate data structures (for the attributes) and algorithms (for the operations)
- *Class extender*
  - Interested in how to extend a class to solve a new problem or to adapt to a change in the application domain
- *Class user*
  - The class user is interested in the signatures of the class operations and conditions, under which they can be invoked
  - The class user is not interested in the implementation of the class.

# Why do we distinguish different Users of Class Diagrams?

- *Models often don't distinguish between application classes and solution classes*
  - **Reason:** Modeling languages like UML allow the use of both types of classes in the same model
    - "address book", "array"
  - **Preferred:** No solution classes in the analysis model
- *Many systems don't distinguish between the specification and the implementation of a class*
  - **Reason:** Object-oriented programming languages allow the simultaneous use of specification and implementation of a class
  - **Preferred:** We distinguish between analysis model and object design model. The analysis design model does not contain any implementation specification.

# Analysis Model vs. Object Design model

- *The analysis model is constructed during the analysis phase*
  - *Main stakeholders: End user, customer, analyst*
  - *The class diagrams contains only application domain classes*
- *The object design model (sometimes also called specification model) is created during the object design phase*
  - *Main stakeholders: class specifiers, class implementors, class users and class extenders*
  - *The class diagrams contain application domain as well as solution domain classes.*

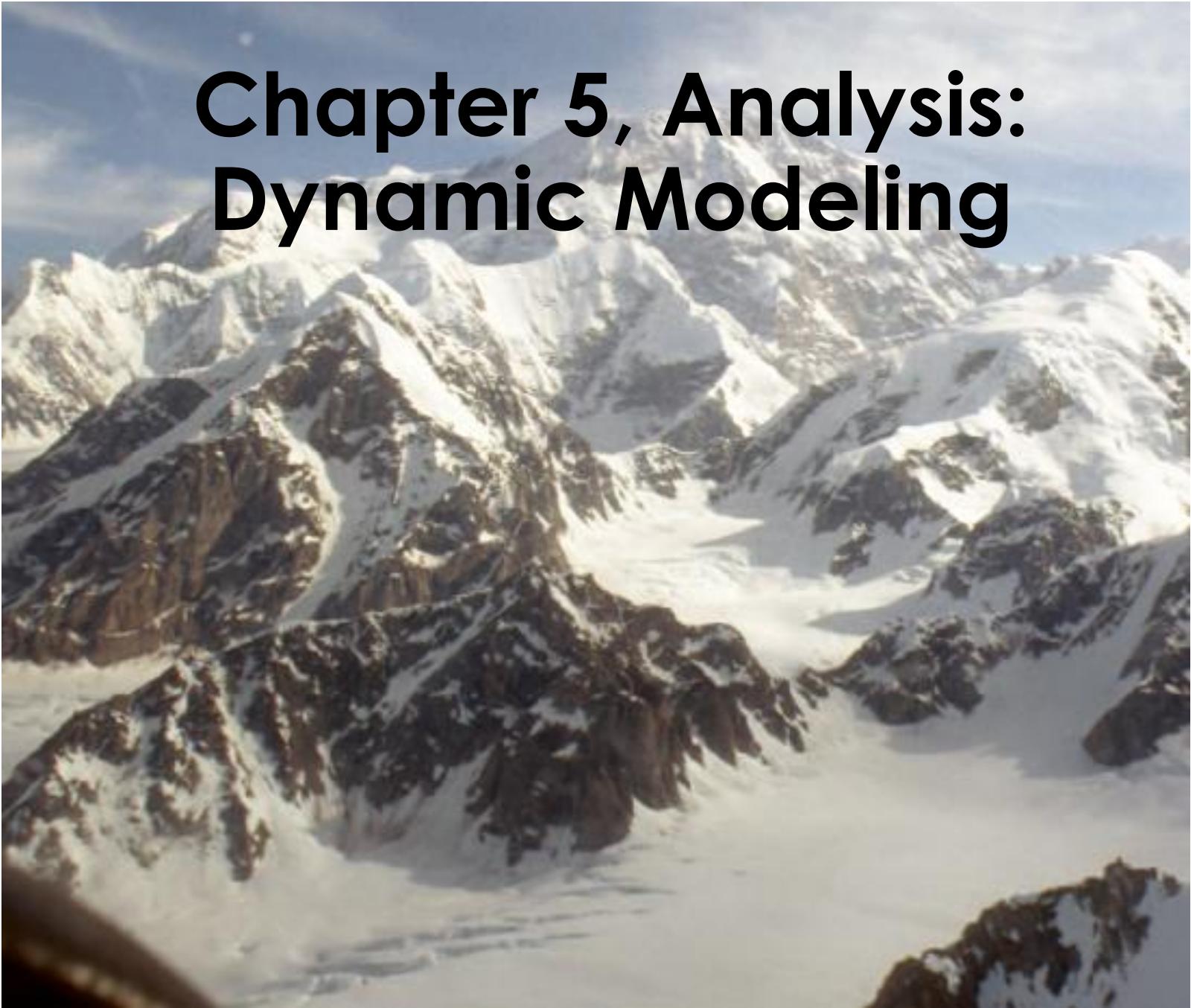
# Analysis Model vs Object Design Model (2)

- *The analysis model is the basis for communication between analysts, application domain experts and end users.*
- *The object design model is the basis for communication between designers and implementors.*

# Summary 2

- *System modeling*
  - *Functional modeling+object modeling+dynamic modeling*
- *Functional modeling*
  - *From scenarios to use cases to objects*
- *Object modeling is the central activity*
  - *Class identification is a major activity of object modeling*
  - *Easy syntactic rules to find classes and objects*
  - *Abbot's Technique*
- *Analysts, designers and implementors have different modeling needs*
- *There are three types of implementors with different roles during*
  - *Class user, class implementor, class extender.*

# Chapter 5, Analysis: Dynamic Modeling



# Reverse Engineering Challenge

- *Date: Next Tuesday, May 15th*
- **Rules:**
  - We start at exactly at 12:50
    - 10 Minutes introduction to the problem
    - 35 minutes development time for the solution
  - You have to do the development in the lecture hall
  - Bring your own laptop (well charged!)
  - Collaboration is ok: Team work is encouraged
    - Pair programming (solo and triple also ok)
- **First Prize:**
  - First person, who finishes a solution that executes and can be demonstrated on lecturer laptop
- **Second prize:**
  - Lottery among all the solutions submitted by the end of the lecture (by 13:35).

# How can you prepare for this event?

- *Visit the Challenge-Website (see lecture portal)*
- *Prepare your development environment:*
  - *Download Eclipse*
  - *Download the Bumper system*
  - *Compile and run Bumpers*
  - *Inspect the source code*
- *Work through the video tutorials.*

# Bumpers-Demo



# I cannot follow the lectures. Where are we?

- We have covered Ch 1 - 4
- We are in the middle of Chapter 5
  - Functional modeling: Read again Ch 2, pp. 46 - 51
  - Structural modeling: Read again Ch 2, pp. 52 - 59
- From use cases to class diagrams
  - Identify participatory objects in flow of events descriptions
    - Exercise: Apply Abbot's technique to Fig. 5-7, p. 181
  - Identify entity, control and boundary objects
    - Heuristics to find these types: Ch 5, Section 5.4
- We are now moving into dynamic modeling
- Notations for dynamic models are
  - Interaction-, Collaboration-, Statechart-, Activity diagrams
  - Reread Ch. 2, pp. 59-67

# Outline of the Lecture

- *Dynamic modeling*
  - *Interaction Diagrams*
    - *Sequence diagrams*
    - *Collaboration diagrams*
  - *State diagrams*
- *Using dynamic modeling for the design of user interfaces*
- *Requirements analysis model validation*
- *Design Patterns*
  - *Reuse of design knowledge*
  - *A first design pattern: the composite pattern.*

# How do you find classes?

- We have already established several sources for class identification:
  - *Application domain analysis:* We find classes by talking to the client and identify abstractions by observing the end user
  - *General world knowledge and intuition*
  - *Textual analysis* of event flow in use cases (Abbot)
- Today we identify classes from dynamic models
- Two good heuristics:
  - Actions and activities in state chart diagrams are candidates for public operations in classes
  - Activity lines in sequence diagrams are candidates for objects.

# Dynamic Modeling with UML

- *Two UML diagrams types for dynamic modeling:*
  - *Interaction diagrams describe the dynamic behavior between objects*
  - *Statechart diagrams describe the dynamic behavior of a single object.*

# UML Interaction Diagrams

- *Two types of interaction diagrams:*
  - *Sequence Diagram:*
    - Describes the dynamic behavior of several objects over time
    - Good for real-time specifications
  - *Collaboration Diagram:*
    - Shows the temporal relationship among objects
    - Position of objects is based on the position of the classes in the UML class diagram.
    - Does not show time,

# UML State Chart Diagram

- *State Chart Diagram:*
  - A state machine that describes the response of an object of a given class to the receipt of outside stimuli (Events).
- *Activity Diagram:*
  - A special type of state chart diagram, where all states are action states (Moore Automaton).

# Dynamic Modeling

- *Definition of a dynamic model:*
  - Describes the components of the system that have interesting dynamic behavior
- *The dynamic model is described with*
  - *State diagrams: One state diagram for each class with interesting dynamic behavior*
    - *Classes without interesting dynamic behavior are not modeled with state diagrams*
  - *Sequence diagrams: For the interaction between classes*
- *Purpose:*
  - *Detect and supply operations for the object model.*

# How do we detect Operations?

- *We look for objects, who are interacting and extract their “protocol”*
- *We look for objects, who have interesting behavior on their own*
- *Good starting point: Flow of events in a use case description*
- *From the flow of **events** we proceed to the sequence diagram to find the **participating objects**.*

# What is an Event?

- *Something that happens at a point in time*
- *An event sends information from one object to another*
- *Events can have associations with each other:*
  - *Causally related:*
    - *An event happens always before another event*
    - *An event happens always after another event*
  - *Causally unrelated:*
    - *Events that happen concurrently*
- *Events can also be grouped in event classes with a hierarchical structure => Event taxonomy*

# The term ‘Event’ is often used in two ways

- *Instance of an event class:*
  - “*Slide 14 shown on Thursday May 9 at 8:50*”.
  - *Event class “Lecture Given”, Subclass “Slide Shown”*
- *Attribute of an event class*
  - *Slide Update(7:27 AM, 05/07/2009)*
  - *Train\_Leaves(4:45pm, Manhattan)*
  - *Mouse button down(button#, tablet-location)*

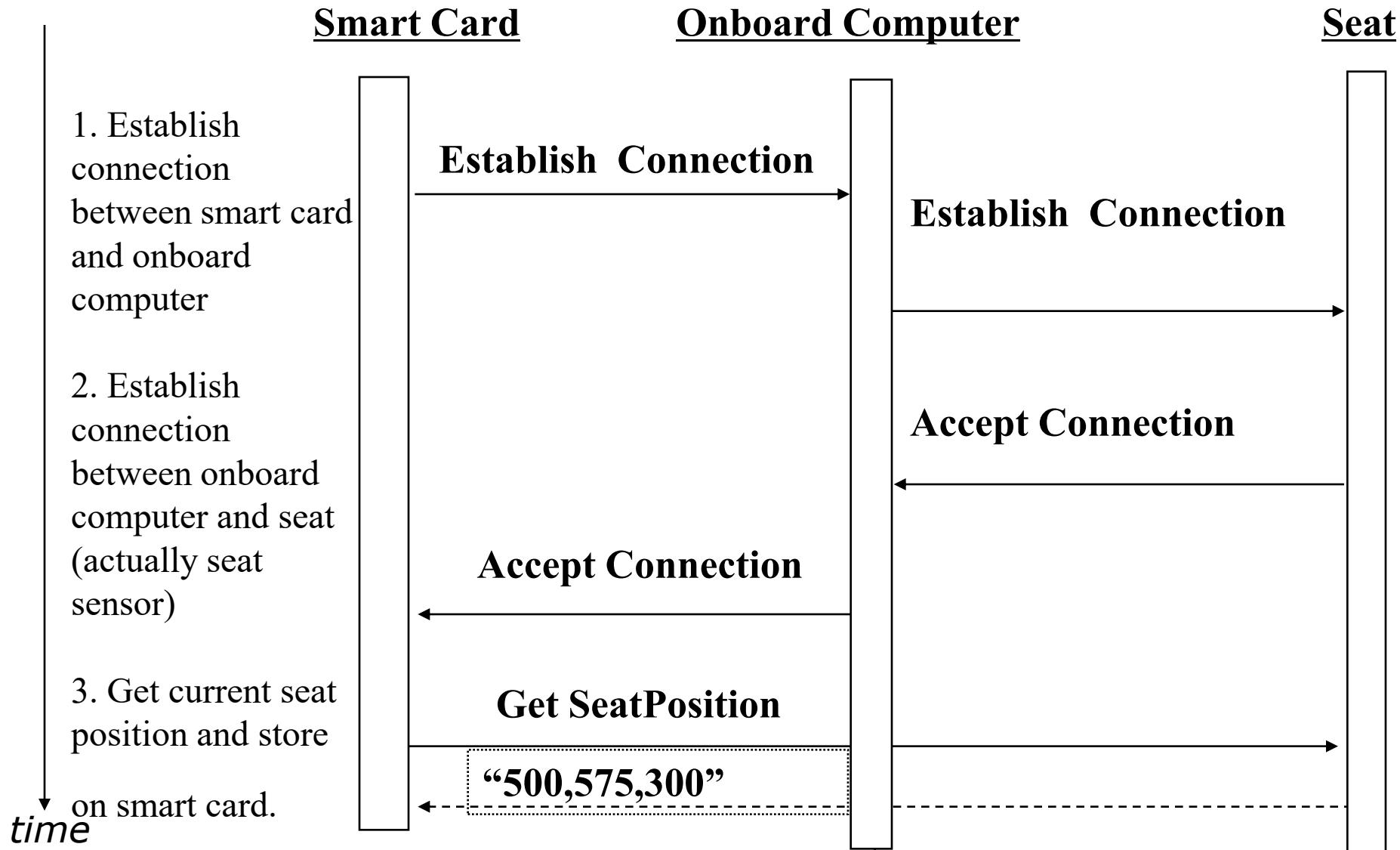
# Sequence Diagram

- A *sequence diagram* is a graphical description of the objects participating in a use case using a DAG notation
- Heuristic for finding participating objects:
  - A event always has a sender and a receiver
  - Find them for each event => These are the objects participating in the use case.

# An Example

- *Flow of events in "Get SeatPosition" use case :*
  1. *Establish connection between smart card and onboard computer*
  2. *Establish connection between onboard computer and sensor for seat*
  3. *Get current seat position and store on smart card*
- *Where are the objects?*

# Sequence Diagram for “Get SeatPosition”



# Heuristics for Sequence Diagrams

- *Layout:*

- 1st column: Should be the **actor** of the use case*

- 2nd column: Should be a **boundary object***

- 3rd column: Should be the **control object** that manages the rest of the use case*

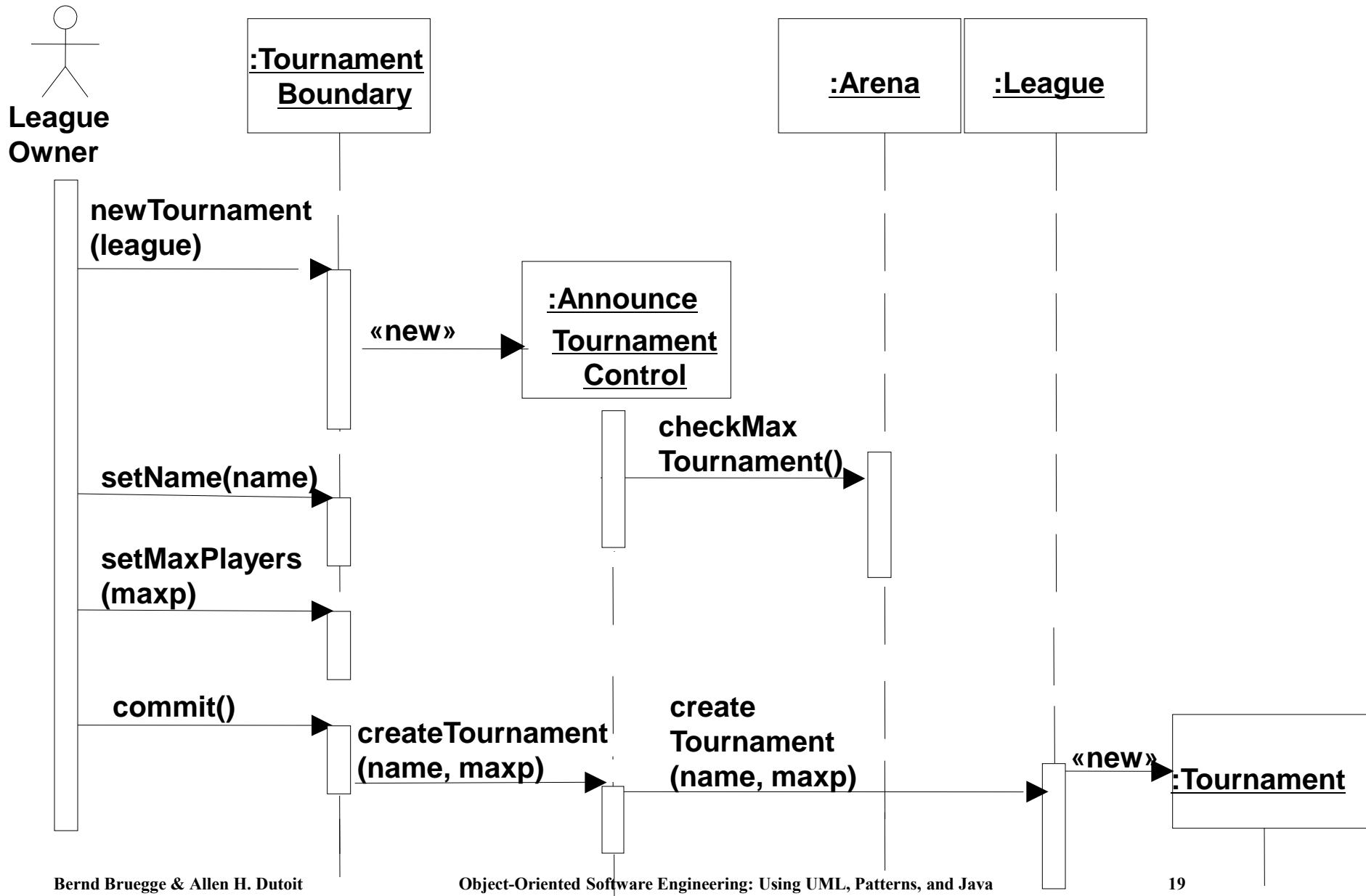
- *Creation of objects:*

- Create control objects at beginning of event flow*
  - The control objects create the boundary objects*

- *Access of objects:*

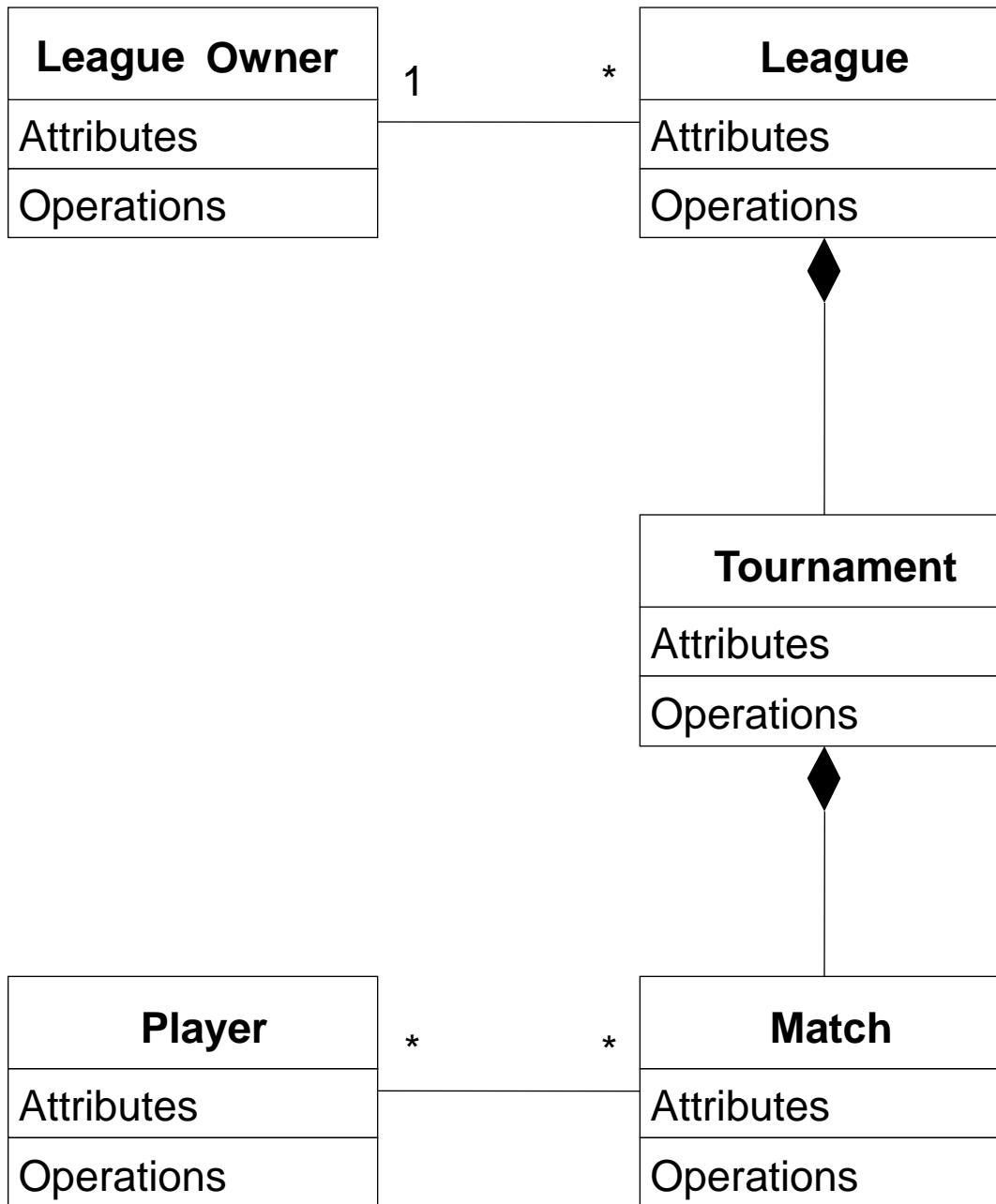
- Entity objects can be accessed by control and boundary objects*
  - Entity objects should not access boundary or control objects.*

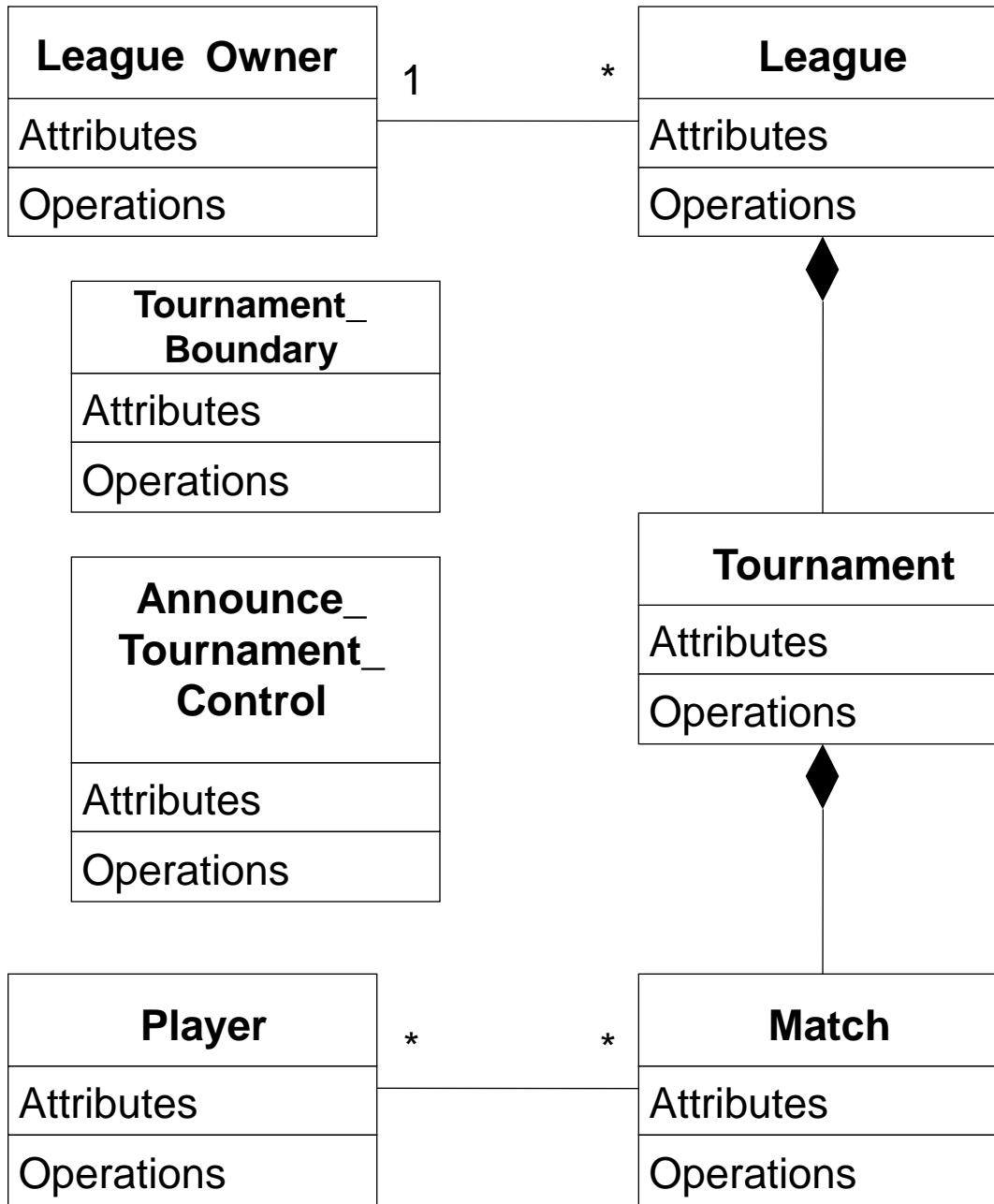
# ARENA Sequence Diagram: Create Tournament



# Impact on ARENA's Object Model

- *Let's assume ARENA's object model contains - at this modeling stage - the objects*
  - ▶ League Owner, Arena, League, Tournament, Match and Player
- *The Sequence Diagram identifies 2 new Classes*
  - ▶ Tournament Boundary, Announce\_Tournament\_Control

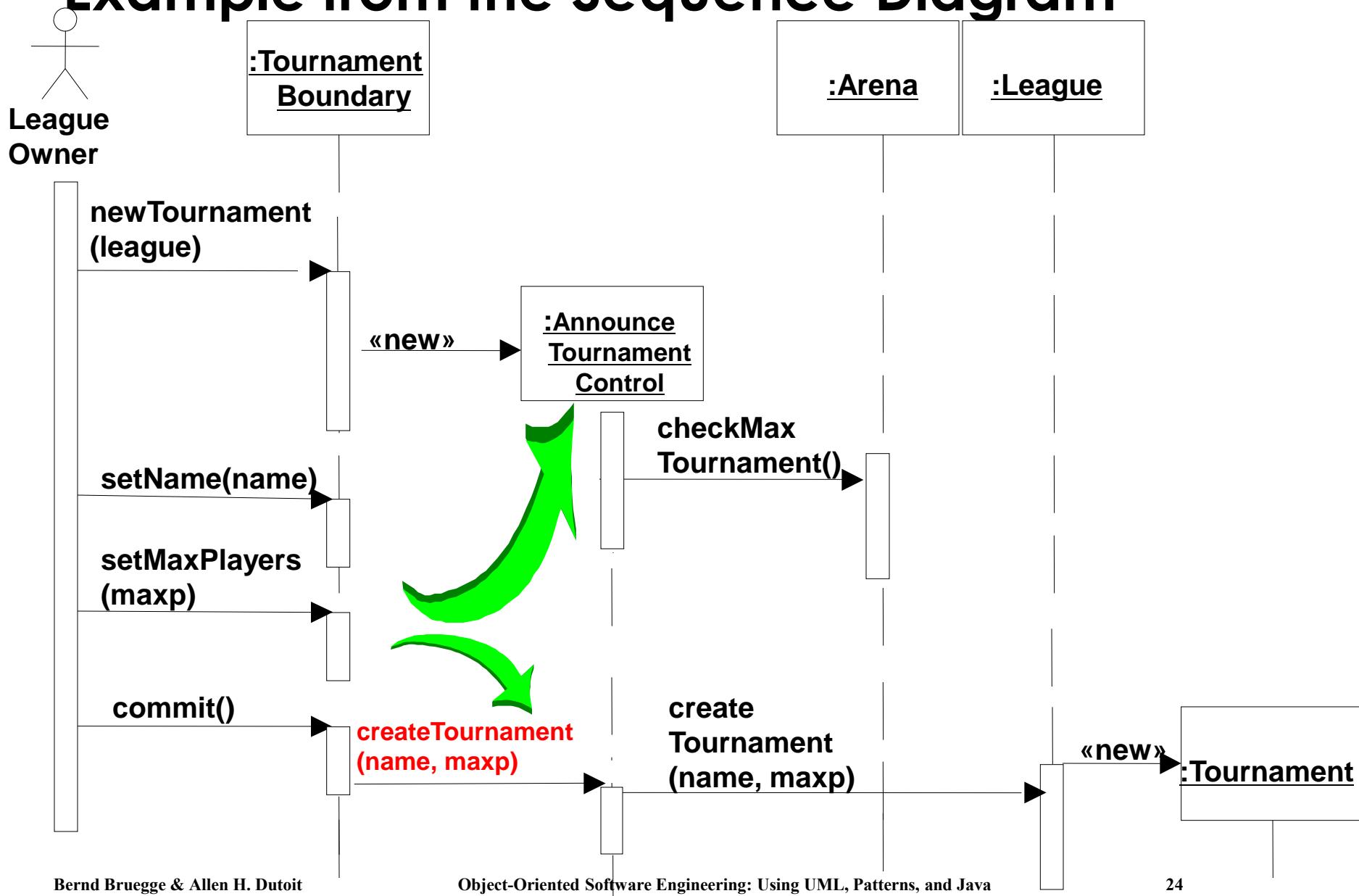


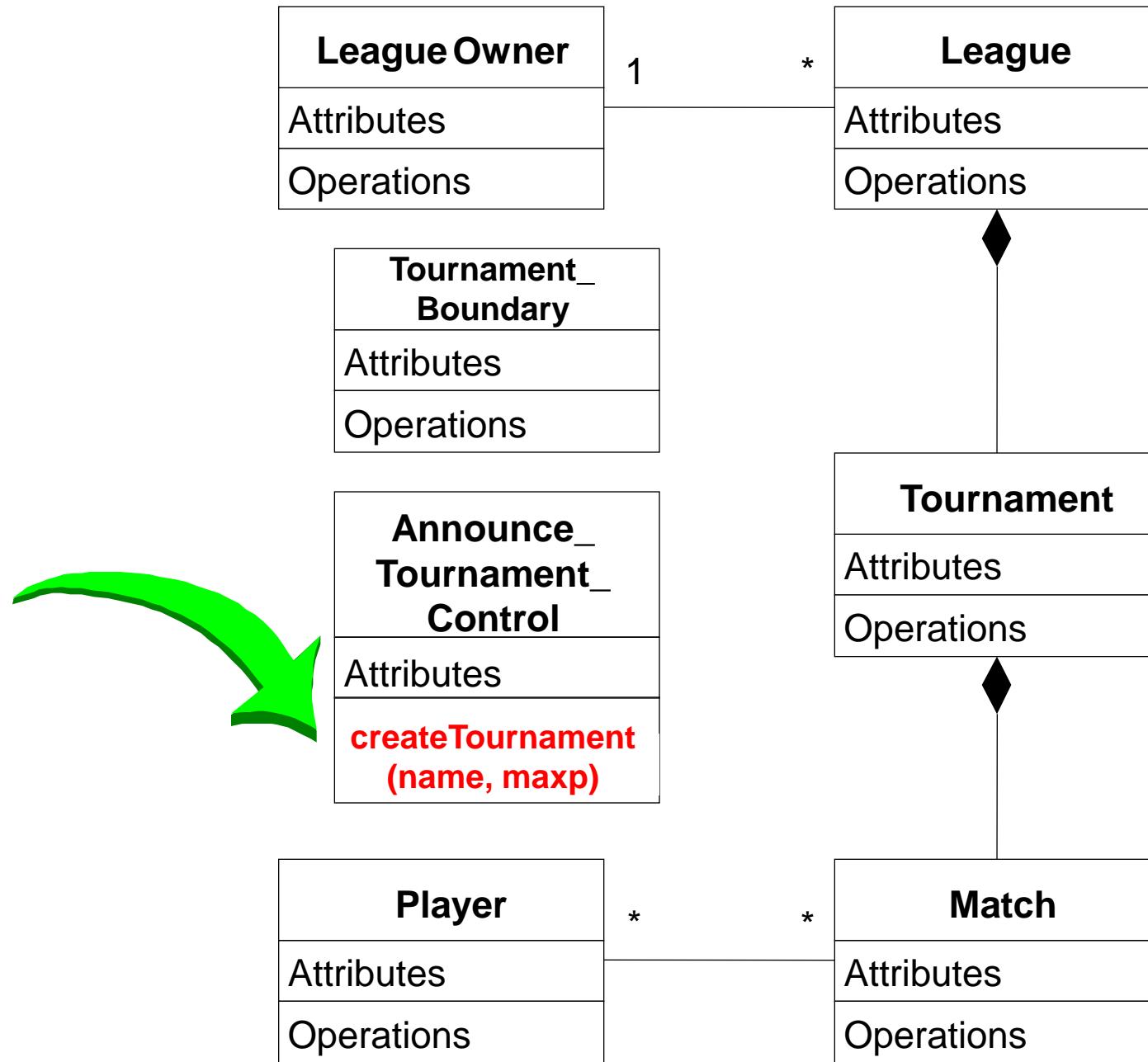


# Impact on ARENA's Object Model (2)

- *The sequence diagram also supplies us with many new events*
  - *newTournament(league)*
  - *setName(name)*
  - *setMaxPlayers(max)*
  - *commit*
  - *checkMaxTournament()*
  - *createTournament*
- *Question:*
  - *Who owns these events?*
- *Answer:*
  - *For each object that receives an event there is a public operation in its associated class*
  - *The name of the operation is usually the name of the event.*

# Example from the Sequence Diagram



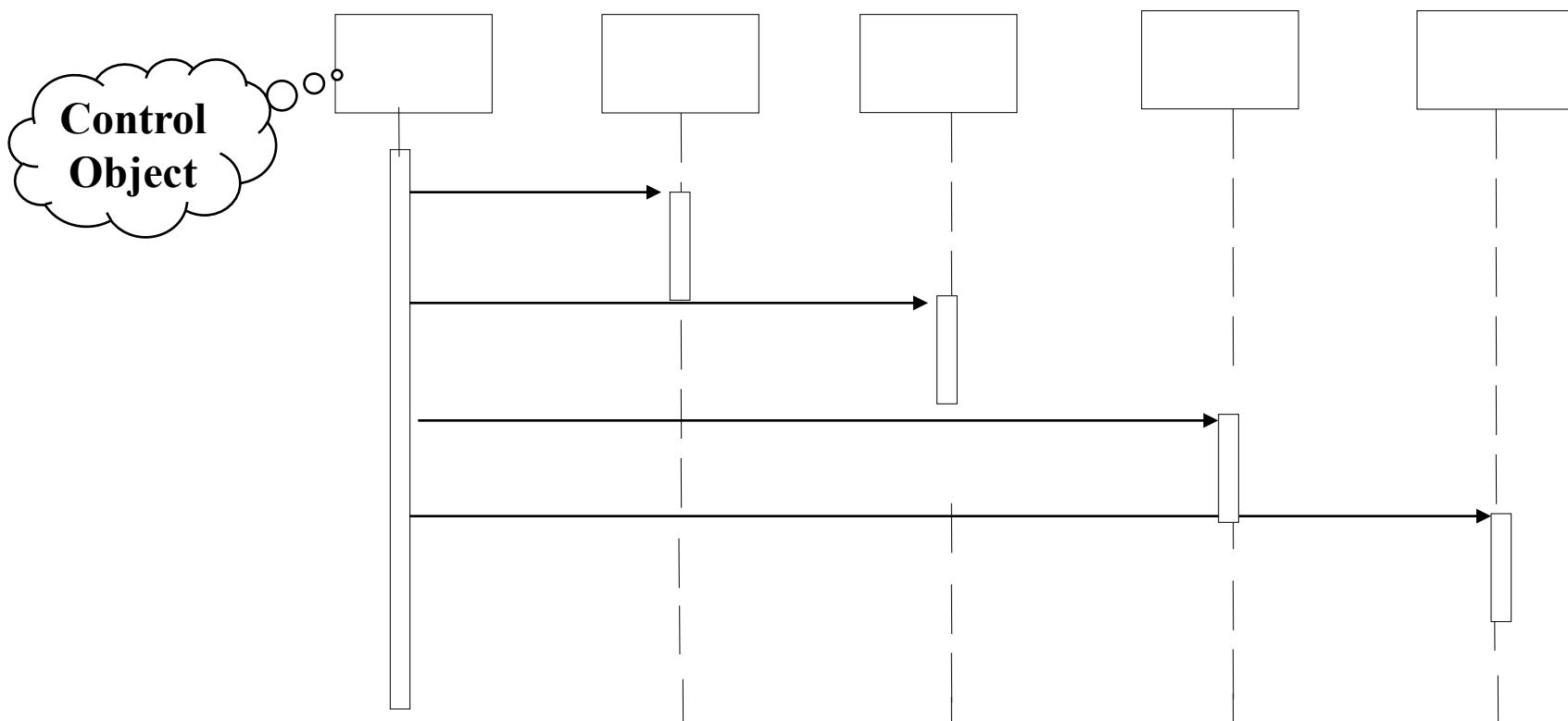


# What else can we get out of Sequence Diagrams?

- *Sequence diagrams are derived from use cases*
- *The structure of the sequence diagram helps us to determine how decentralized the system is*
- *We distinguish two structures for sequence diagrams*
  - *Fork Diagrams and Stair Diagrams (Ivar Jacobson)*

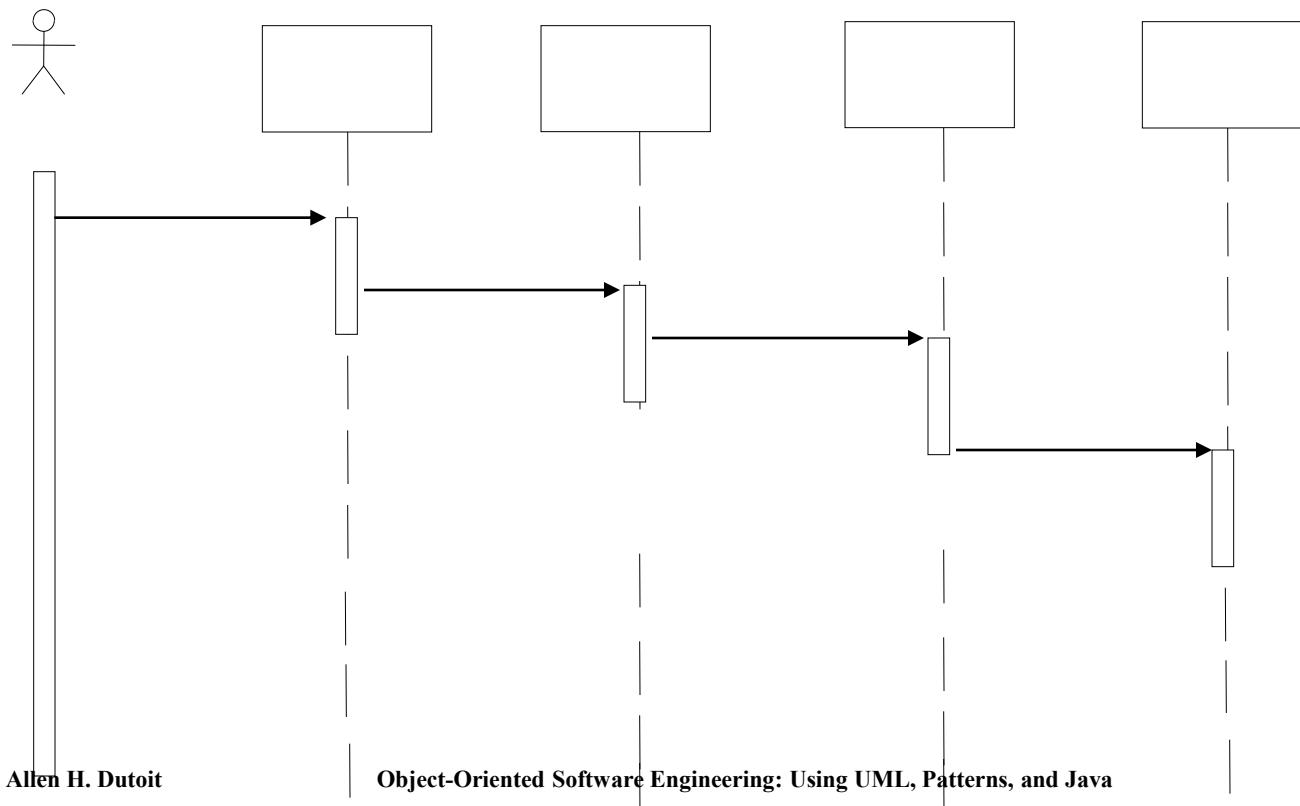
# Fork Diagram

- *The dynamic behavior is placed in a single object, usually a control object*
  - *It knows all the other objects and often uses them for direct questions and commands*



# Stair Diagram

- *The dynamic behavior is distributed. Each object delegates responsibility to other objects*
  - *Each object knows only a few of the other objects and knows which objects can help with a specific behavior*



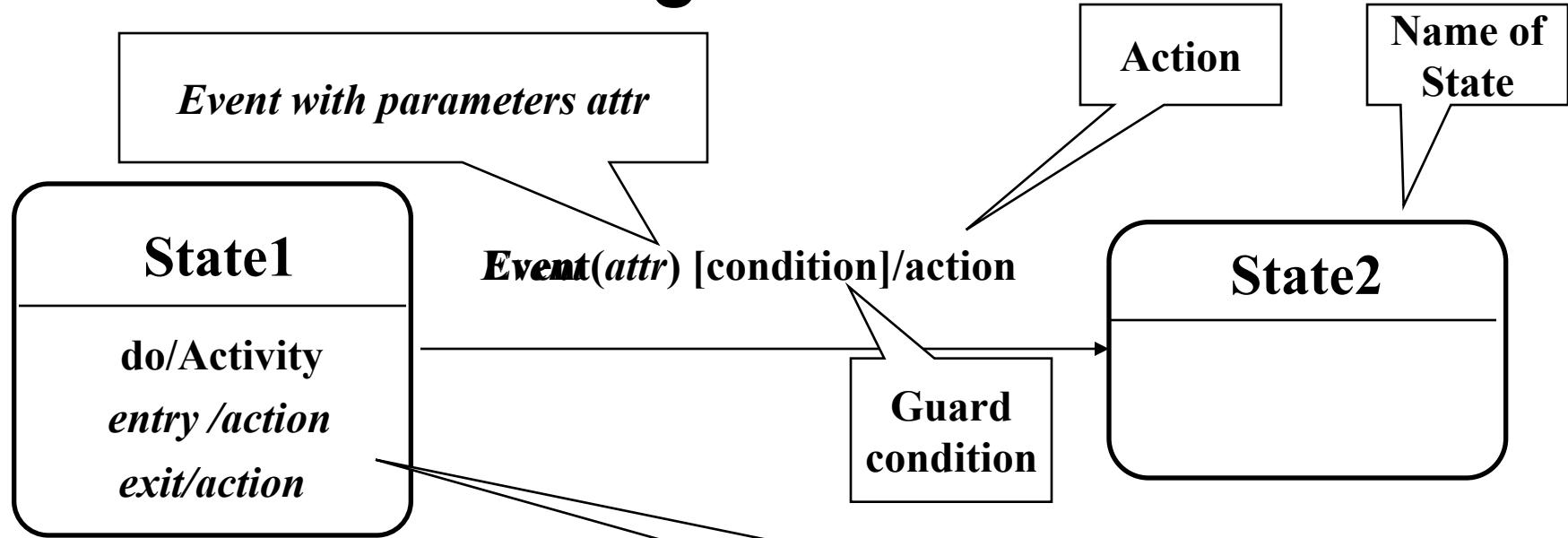
# Fork or Stair?

- *Object-oriented supporters claim that the stair structure is better*
- *Modeling Advice:*
  - *Choose the stair - a decentralized control structure - if*
    - *The operations have a strong connection*
    - *The operations will always be performed in the same order*
  - *Choose the fork - a centralized control structure - if*
    - *The operations can change order*
    - *New operations are expected to be added as a result of new requirements.*

# Dynamic Modeling

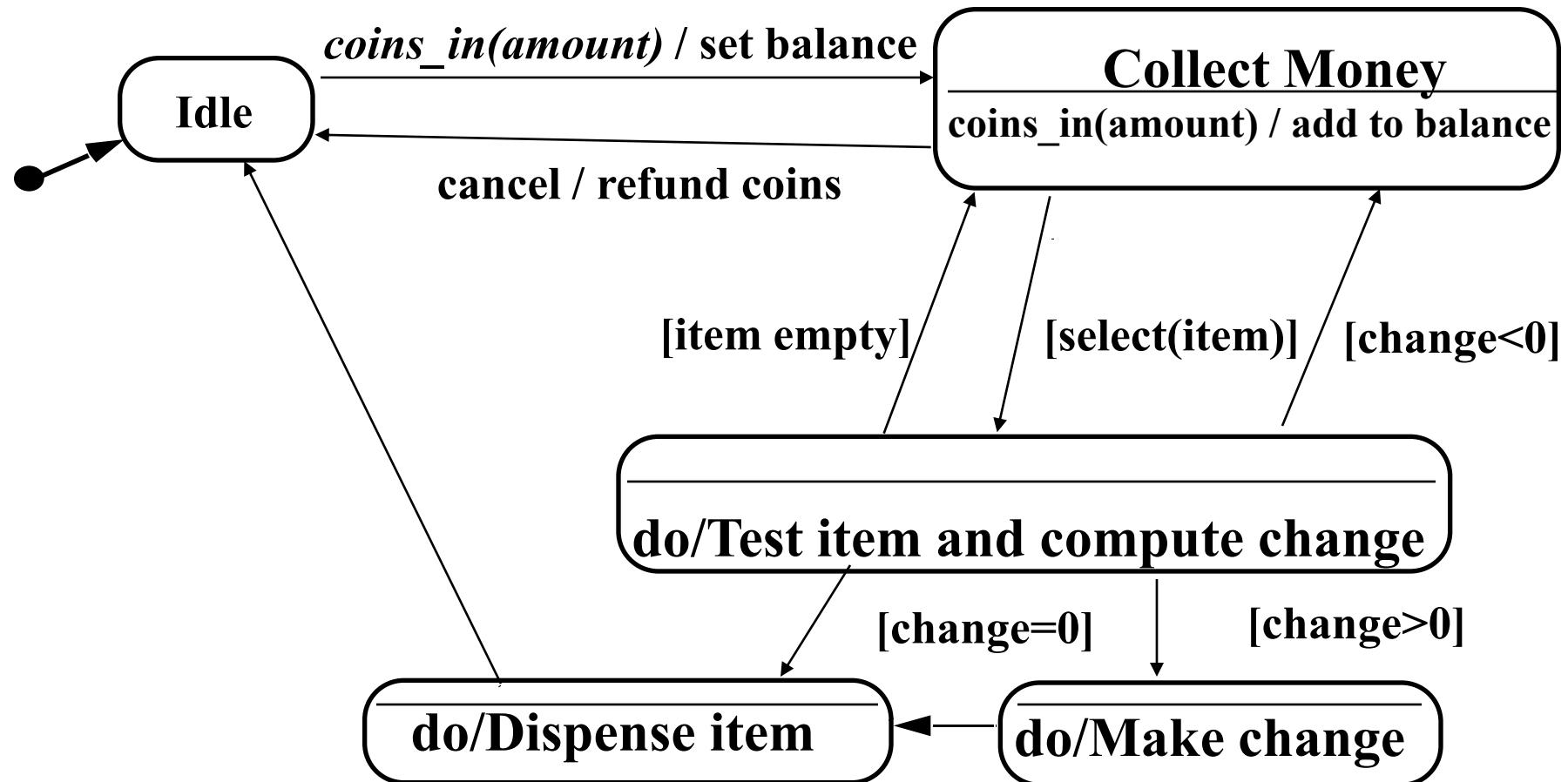
- *We distinguish between two types of operations:*
  - *Activity: Operation that takes time to complete*
    - associated with states
  - *Action: Instantaneous operation*
    - associated with events
- *A state chart diagram relates events and states for one class*
- *An object model with several classes with interesting behavior has a set of state diagrams*

# UML Statechart Diagram Notation



- **Note:**
  - *Events are italics*
  - *Conditions are enclosed with brackets: []*
  - *Actions and activities are prefixed with a slash /*
- *Notation is based on work by Harel*
- *Added are a few object-oriented modifications.*

# Example of a StateChart Diagram



# State

- *An abstraction of the attributes of a class*
  - *State is the aggregation of several attributes a class*
- *A state is an equivalence class of all those attribute values and links that do no need to be distinguished*
  - *Example: State of a bank*
- *State has duration*

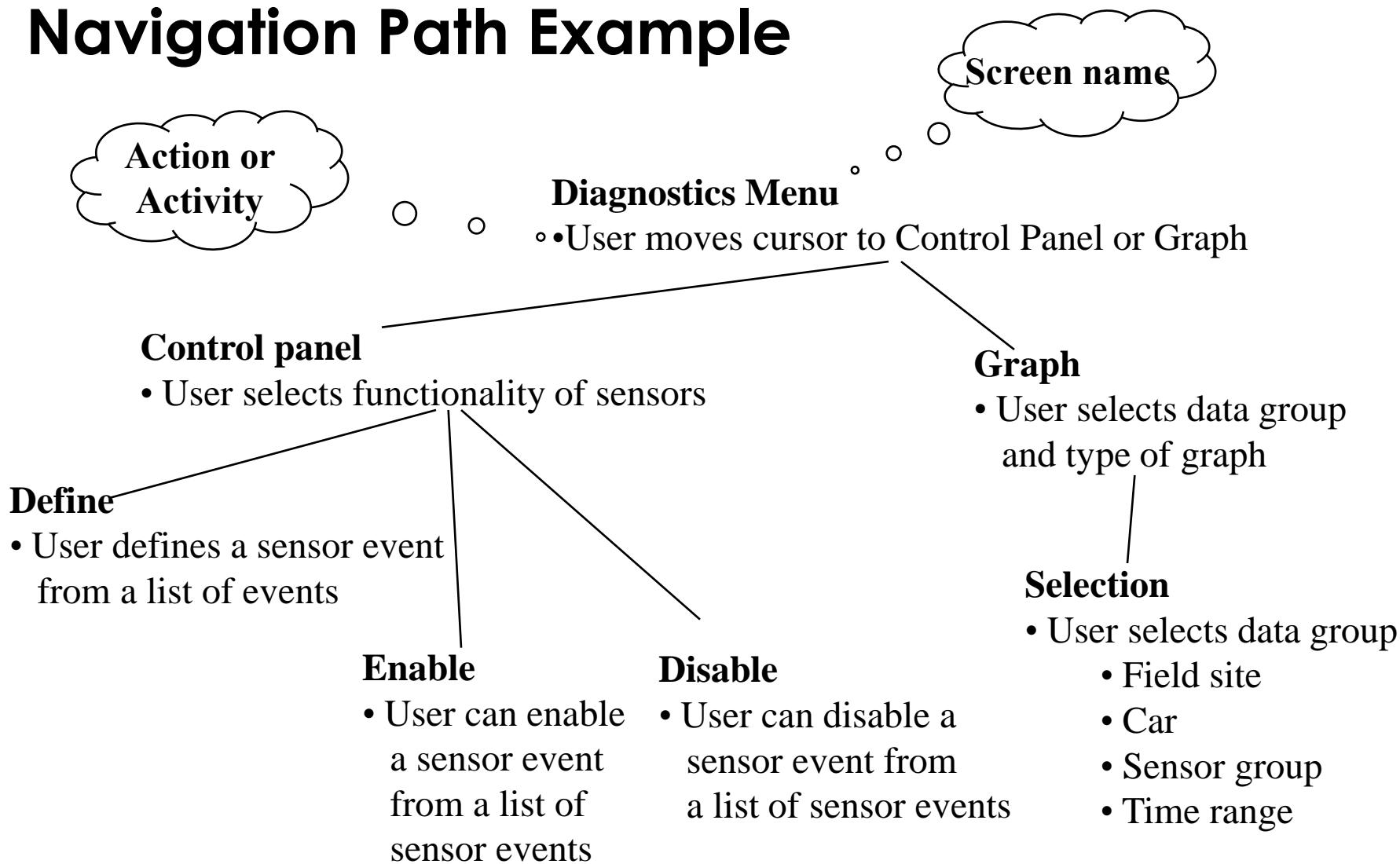
# State Chart Diagram vs Sequence Diagram

- *State chart diagrams help to identify:*
  - *Changes to an individual object over time*
- *Sequence diagrams help to identify:*
  - *The temporal relationship of between objects over time*
  - *Sequence of operations as a response to one ore more events.*

# Dynamic Modeling of User Interfaces

- *Statechart diagrams can be used for the design of user interfaces*
- *States: Name of screens*
- *Actions or activities are shown as bullets under the screen name*

# Navigation Path Example



# Practical Tips for Dynamic Modeling

- *Construct dynamic models only for classes with significant dynamic behavior*
  - Avoid “analysis paralysis”
- *Consider only relevant attributes*
  - Use abstraction if necessary
- *Look at the granularity of the application when deciding on actions and activities*
- *Reduce notational clutter*
  - Try to put actions into superstate boxes (look for identical actions on events leading to the same state).

# Let's Do Analysis: A Toy Example

- *Analyze the problem statement*
  - *Identify functional requirements*
  - *Identify nonfunctional requirements*
  - *Identify constraints (pseudo requirements)*
- *Build the functional model:*
  - *Develop use cases to illustrate functional requirements*
- *Build the dynamic model:*
  - *Develop sequence diagrams to illustrate the interaction between objects*
  - *Develop state diagrams for objects with interesting behavior*
- *Build the object model:*
  - *Develop class diagrams for the structure of the system*

# Problem Statement: Direction Control for a Toy Car

- *Power is turned on*
  - *Car moves forward and car headlight shines*
- *Power is turned off*
  - *Car stops and headlight goes out.*
- *Power is turned on*
  - *Headlight shines*
- *Power is turned off*
  - *Headlight goes out*
- *Power is turned on*
  - *Car runs backward with its headlight shining*
- *Power is turned off*
  - *Car stops and headlight goes out*
- *Power is turned on*
  - *Headlight shines*
- *Power is turned off*
  - *Headlight goes out*
- *Power is turned on*
  - *Car runs forward with its headlight shining*

# Find the Functional Model: Use Cases

- Use case 1: System Initialization
  - *Entry condition: Power is off, car is not moving*
  - *Flow of events:*
    1. *Driver turns power on*
  - *Exit condition: Car moves forward, headlight is on*
- Use case 2: Turn headlight off
  - *Entry condition: Car moves forward with headlights on*
  - *Flow of events:*
    1. *Driver turns power off, car stops and headlight goes out.*
    2. *Driver turns power on, headlight shines and car does not move.*
    3. *Driver turns power off, headlight goes out*
  - *Exit condition: Car does not move, headlight is out*

# Use Cases continued

- Use case 3: Move car backward
  - *Entry condition: Car is stationary, headlights off*
  - *Flow of events:*
    1. Driver turns power on
  - *Exit condition: Car moves backward, headlight on*
- Use case 4: Stop backward moving car
  - *Entry condition: Car moves backward, headlights on*
  - *Flow of events:*
    1. Driver turns power off, car stops, headlight goes out.
    2. Power is turned on, headlight shines and car does not move.
    3. Power is turned off, headlight goes out.
  - *Exit condition: Car does not move, headlight is out*

# Use Cases Continued

- Use case 5: Move car forward
  - *Entry condition: Car does not move, headlight is out*
  - *Flow of events*
    1. *Driver turns power on*
  - *Exit condition:*
    - *Car runs forward with its headlight shining*

# Use Case Pruning

- *Do we need use case 5?*
- *Let us compare use case 1 and use case 5:*

## Use case 1: System Initialization

- *Entry condition: Power is off, car is not moving*
- *Flow of events:*
  1. Driver turns power on
- *Exit condition: Car moves forward, headlight is on*

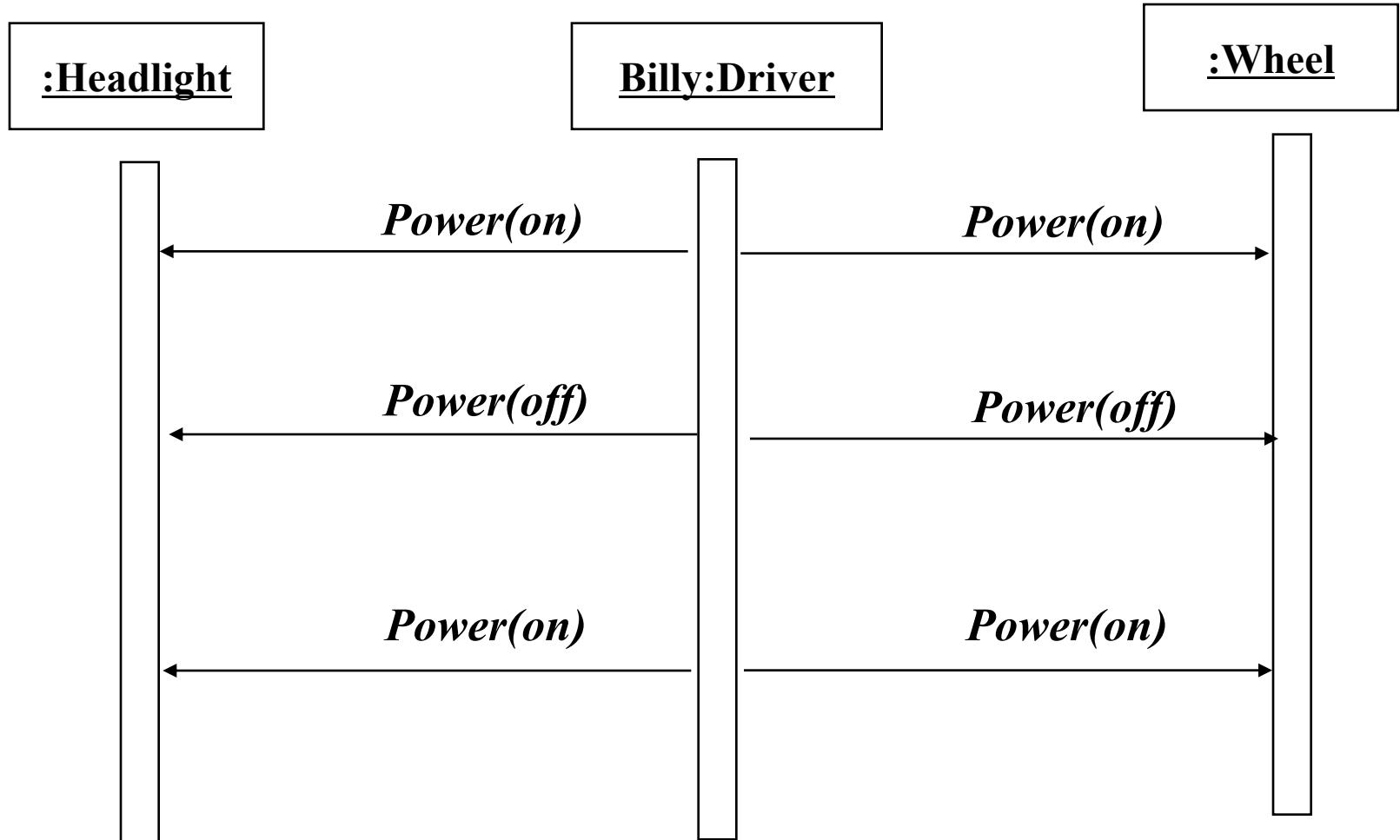
## Use case 5: Move car forward

- *Entry condition: Car does not move, headlight is out*
- *Flow of events*
  1. Driver turns power on
- *Exit condition:*
  - *Car runs forward with its headlight shining*

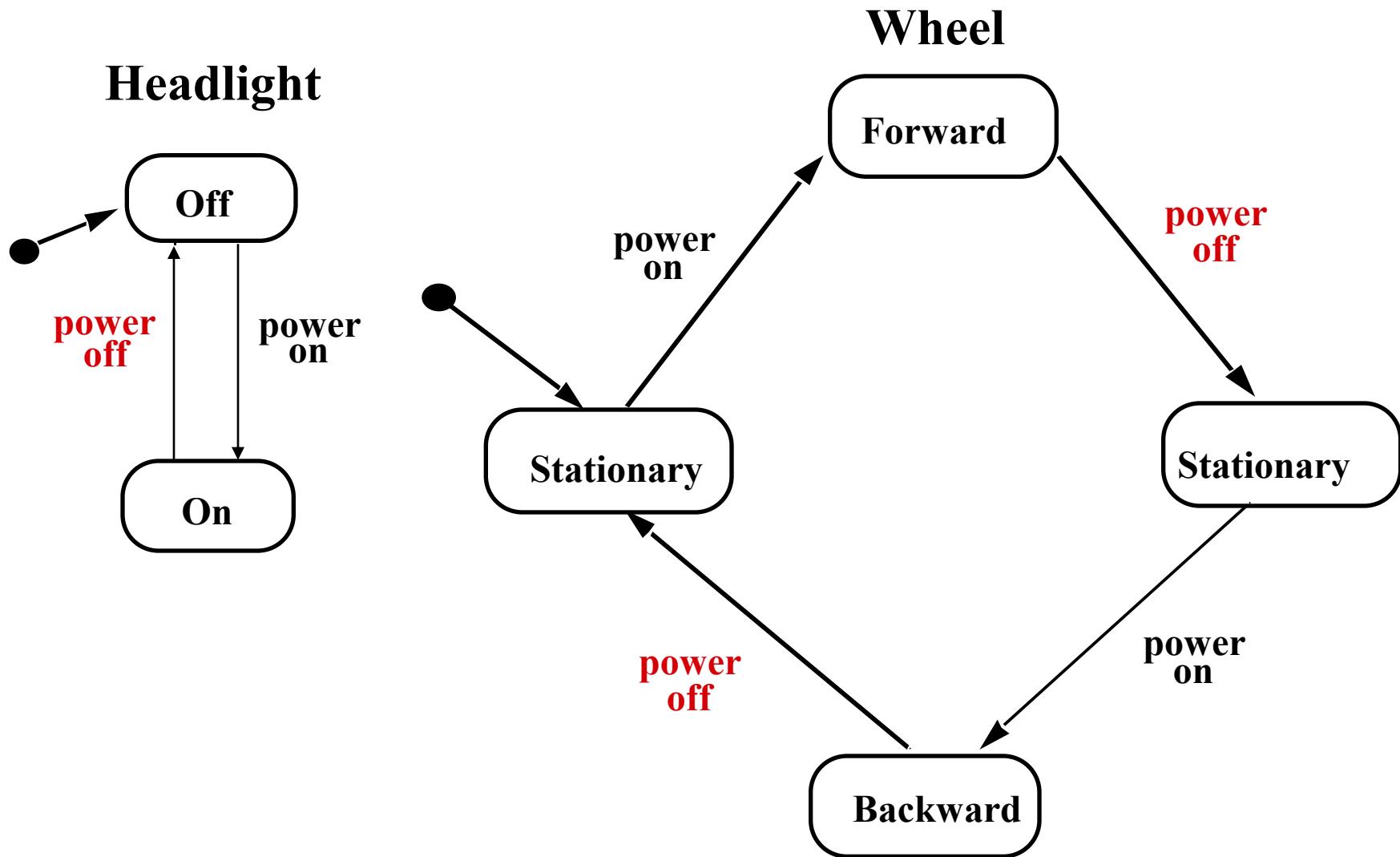
# Dynamic Modeling: Create the Sequence Diagram

- *Name: Drive Car*
- *Sequence of events:*
  - *Billy turns power on*
  - *Headlight goes on*
  - *Wheels starts moving forward*
  - *Wheels keeps moving forward*
  - *Billy turns power off*
  - *Headlight goes off*
  - *Wheels stops moving*
  - . . .

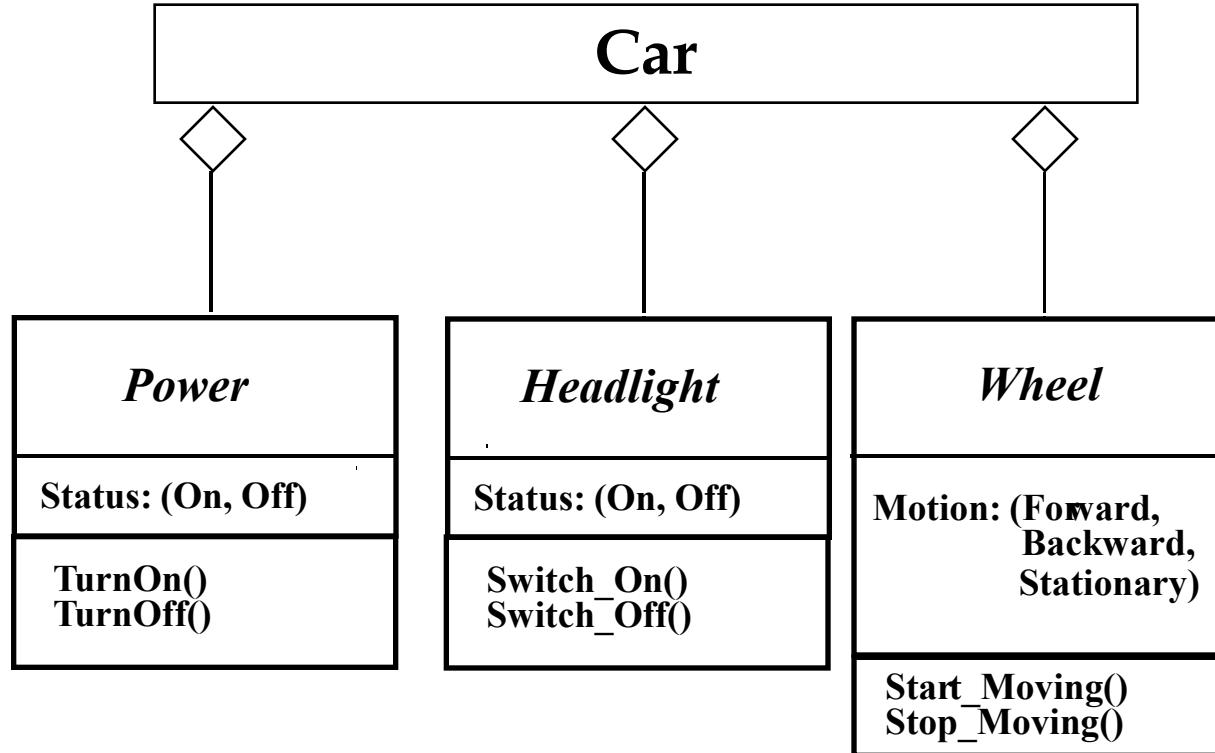
# Sequence Diagram for Drive Car Scenario



# Toy Car: Dynamic Model



# Toy Car: Object Model



# Outline of the Lecture

■ *Dynamic modeling*

- *Sequence diagrams*

- *State diagrams*

■ *Using dynamic modeling for the design of user interfaces*

■ *Analysis example*

→ *Requirements analysis model validation*

# Model Validation and Verification

- *Verification is an equivalence check between the transformation of two models*
- *Validation is the comparison of the model with reality*
  - *Validation is a critical step in the development process Requirements should be validated with the client and the user.*
  - *Techniques: Formal and informal reviews (Meetings, requirements review)*
- *Requirements validation involves several checks*
  - *Correctness, Completeness, Ambiguity, Realism*

# Checklist for a Requirements Review

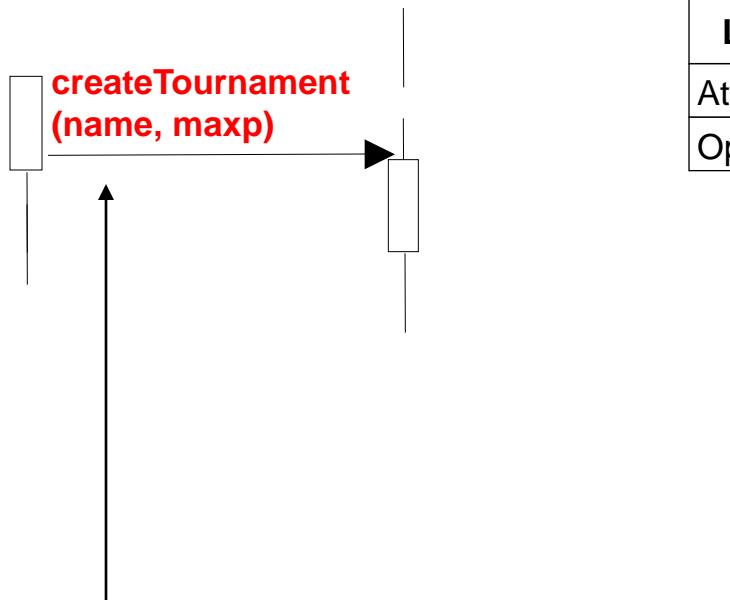
- *Is the model correct?*
  - *A model is correct if it represents the client's view of the system*
- *Is the model complete?*
  - *Every scenario is described*
- *Is the model consistent?*
  - *The model does not have components that contradict each other*
- *Is the model unambiguous?*
  - *The model describes one system, not many*
- *Is the model realistic?*
  - *The model can be implemented*

# Examples for syntactical Problems

- *Different spellings in different UML diagrams*
- *Omissions in diagrams*

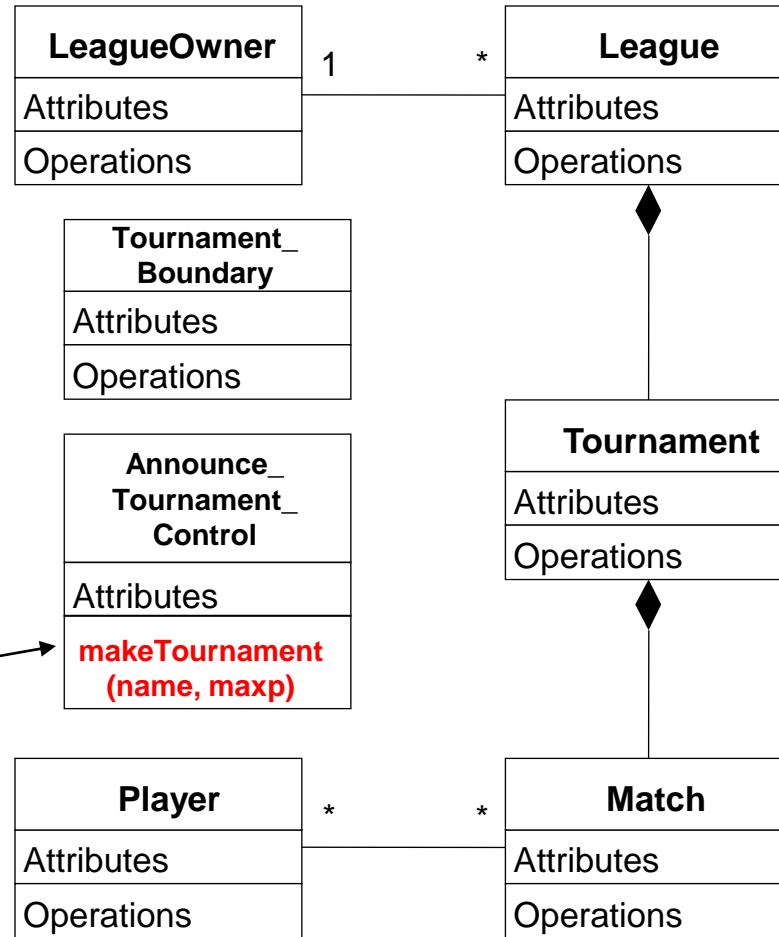
# Different spellings in different UML diagrams

*UML Sequence Diagram*



*Different spellings  
in different models  
for the same operation*

*UML Class Diagram*

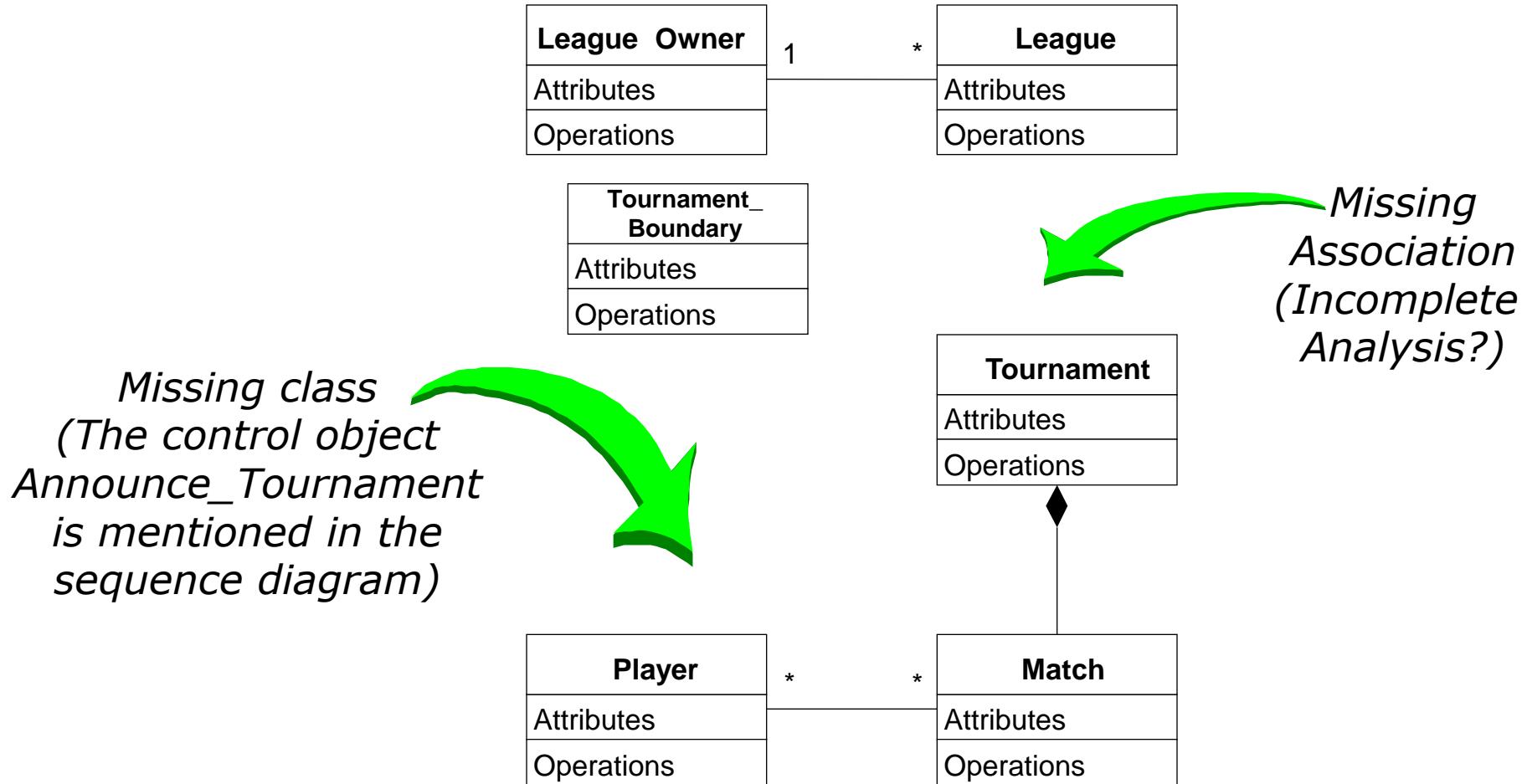


# Checklist for the Requirements Review (2)

- *Syntactical check of the models*
  - *Check for consistent naming of classes, attributes, methods in different subsystems*
  - *Identify dangling associations ("pointing to nowhere")*
  - *Identify double-defined classes*
  - *Identify missing classes (mentioned in one model but not defined anywhere)*
  - *Check for classes with the same name but different meanings*

# Omissions in some UML Diagrams

## *Class Diagram*



# When is a Model Dominant?

- *Object model:*
  - *The system has classes with nontrivial states and many relationships between the classes*
- *Dynamic model:*
  - *The model has many different types of events: Input, output, exceptions, errors, etc.*
- *Functional model:*
  - *The model performs complicated transformations (eg. computations consisting of many steps).*
- *Which model is dominant in these applications?*
  - *Compiler*
  - *Database system*
  - *Spreadsheet program*

# Examples of Dominant Models

- *Compiler:*
  - *The functional model is most important*
  - *The dynamic model is trivial because there is only one type input and only a few outputs*
    - *Is that true for IDEs?*
- *Database systems:*
  - *The object model most important*
  - *The functional model is trivial, because the purpose of the functions is to store, organize and retrieve data*
- *Spreadsheet program:*
  - *The functional model most important*
  - *The dynamic model is interesting if the program allows computations on a cell*
  - *The object model is trivial.*

# Requirements Analysis Document Template

- 1. Introduction*
- 2. Current system*
- 3. Proposed system*
  - 3.1 Overview*
  - 3.2 Functional requirements*
  - 3.3 Nonfunctional requirements*
  - 3.4 Constraints ("Pseudo requirements")*
  -  ***3.5 System models***
    - 3.5.1 Scenarios*
    - 3.5.2 Use case model*
    - 3.5.3 Object model*
      - 3.5.3.1 Data dictionary*
      - 3.5.3.2 Class diagrams*
    - 3.5.4 Dynamic models*
    - 3.5.5 User interface*
  - 4. Glossary*

# Section 3.5 System Model

## 3.5.1 Scenarios

- *As-is scenarios, visionary scenarios*

## 3.5.2 Use case model

- *Actors and use cases*

## 3.5.3 Object model

- *Data dictionary*
- *Class diagrams (classes, associations, attributes and operations)*

## 3.5.4 Dynamic model

- *State diagrams for classes with significant dynamic behavior*
- *Sequence diagrams for collaborating objects (protocol)*

## 3.5.5 User Interface

- *Navigational Paths, Screen mockups*

# Requirements Analysis Questions

1. What are the transformations?



**Functional Modeling**

*Create scenarios and use case diagrams*

- Talk to client, observe, get historical records

2. What is the structure of the system?



**Object Modeling**

*Create class diagrams*

- Identify objects.
- What are the associations between them?
- What is their multiplicity?
- What are the attributes of the objects?
- What operations are defined on the objects?

3. What is its behavior?



**Dynamic Modeling**

*Create sequence diagrams*

- Identify senders and receivers
- Show sequence of events exchanged between objects.
- Identify event dependencies and event concurrency.

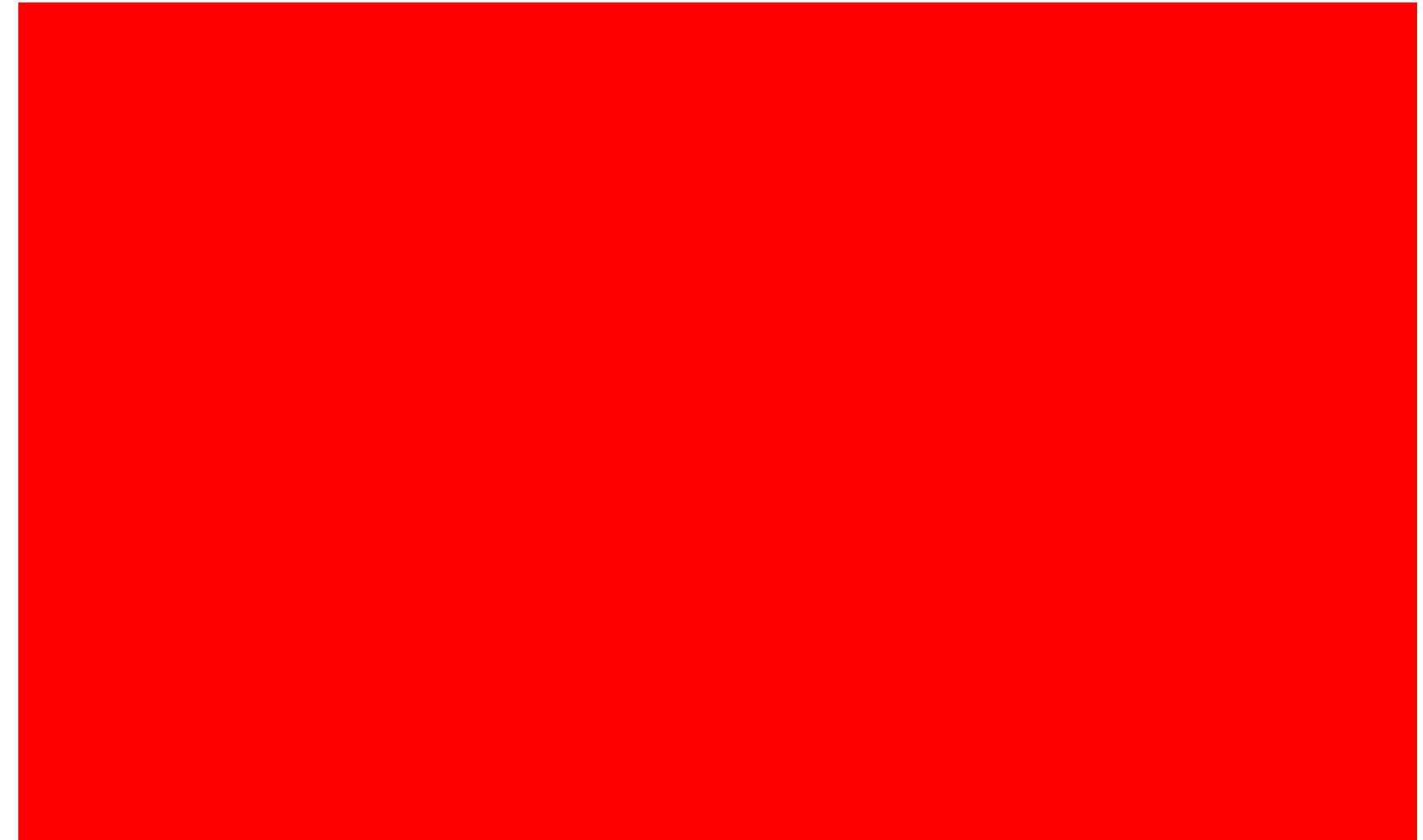
*Create state diagrams*

- Only for the dynamically interesting objects.

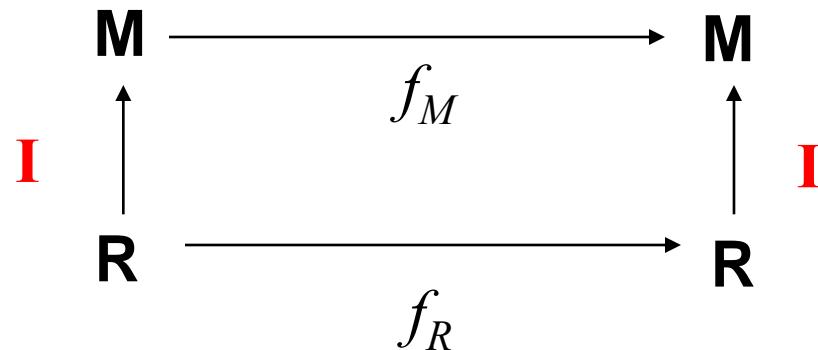
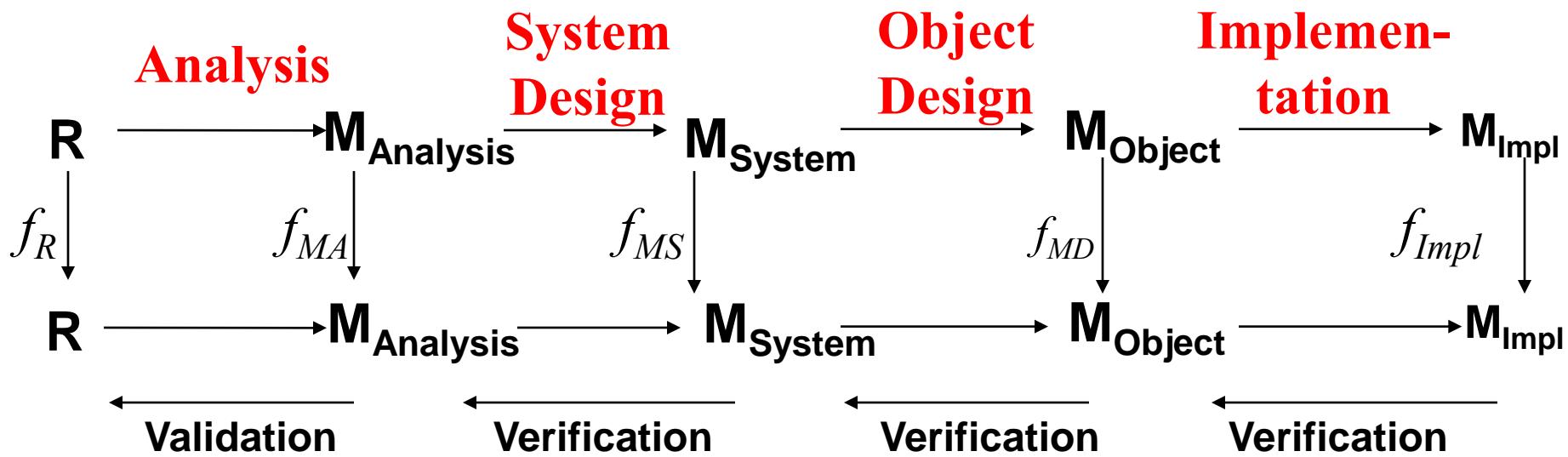
# Summary

- *In this lecture, we reviewed the construction of the dynamic model from use case and object models. In particular, we described:*
- *Sequence and statechart diagrams for identifying new classes and operations.*
- *In addition, we described the requirements analysis document and its components*

# Backup Slides



# Verification vs Validation of models



# Modeling Concurrency of Events

*Two types of concurrency:*

## *1. System concurrency*

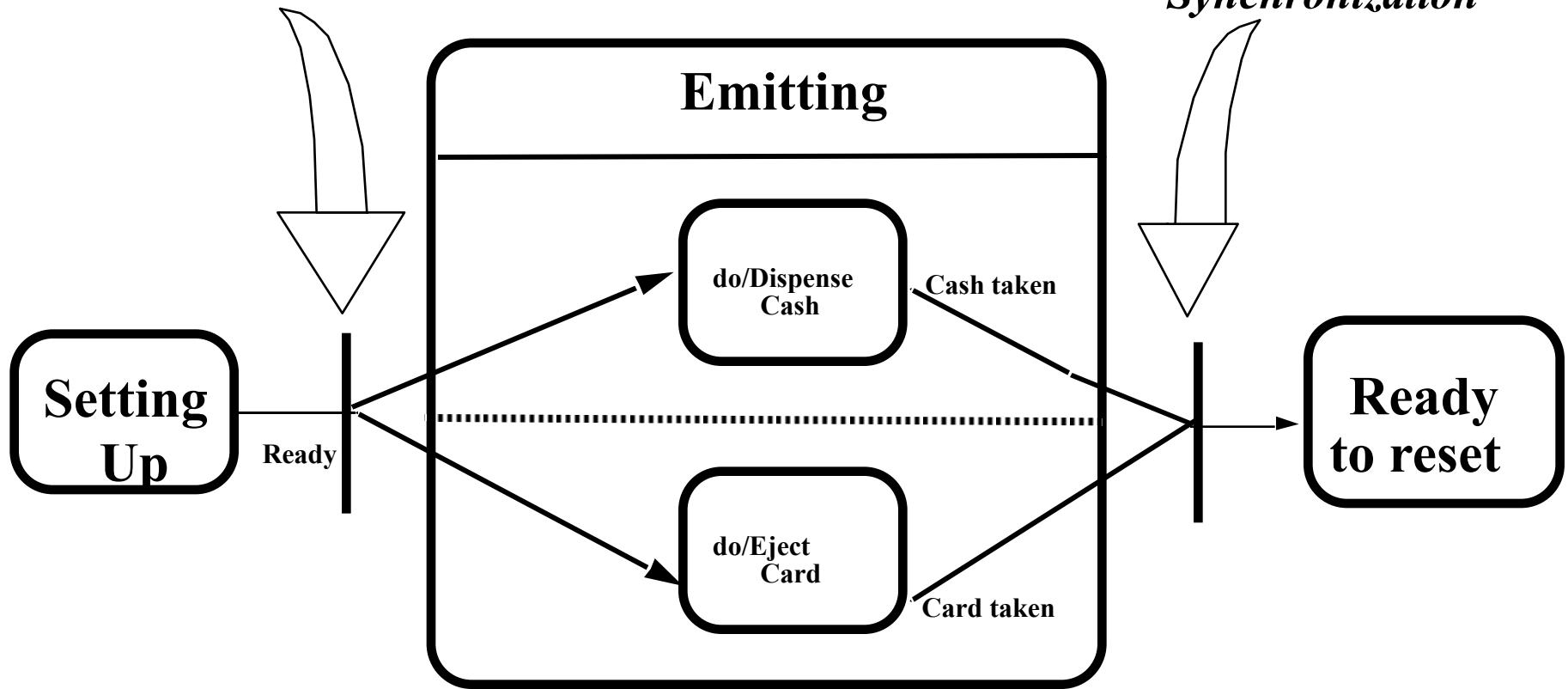
- *The overall system is modeled as the aggregation of state diagrams*
- *Each state diagram is executing concurrently with the others.*

## *2. Concurrency within an object*

- *An object can issue concurrent events*
- *Two problems:*
  - *Show how control is split*
  - *Show how to synchronize when moving to a state without object concurrency*

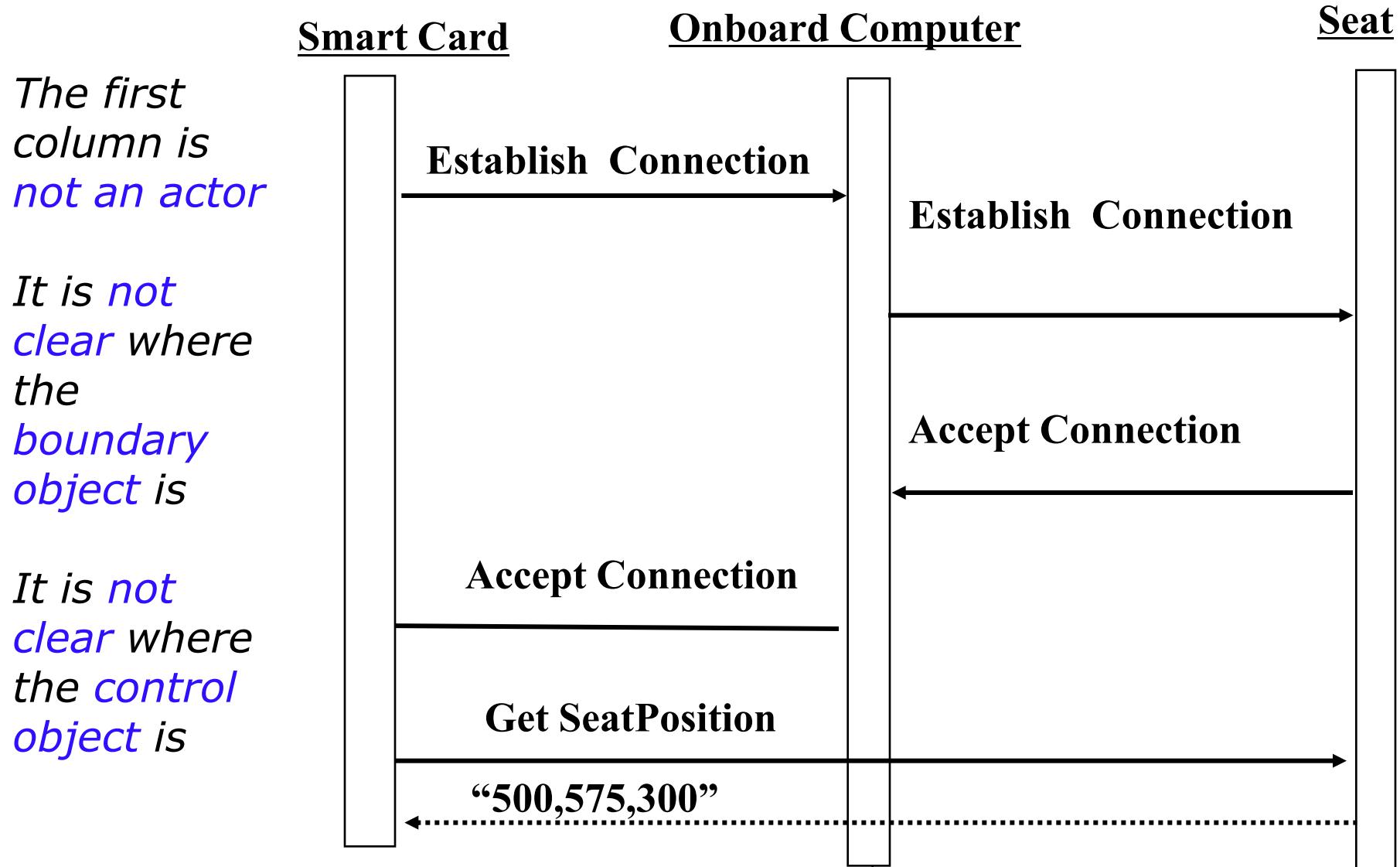
# Example of Concurrency within an Object

*Splitting control*



*Synchronization*

# Is this a good Sequence Diagram?

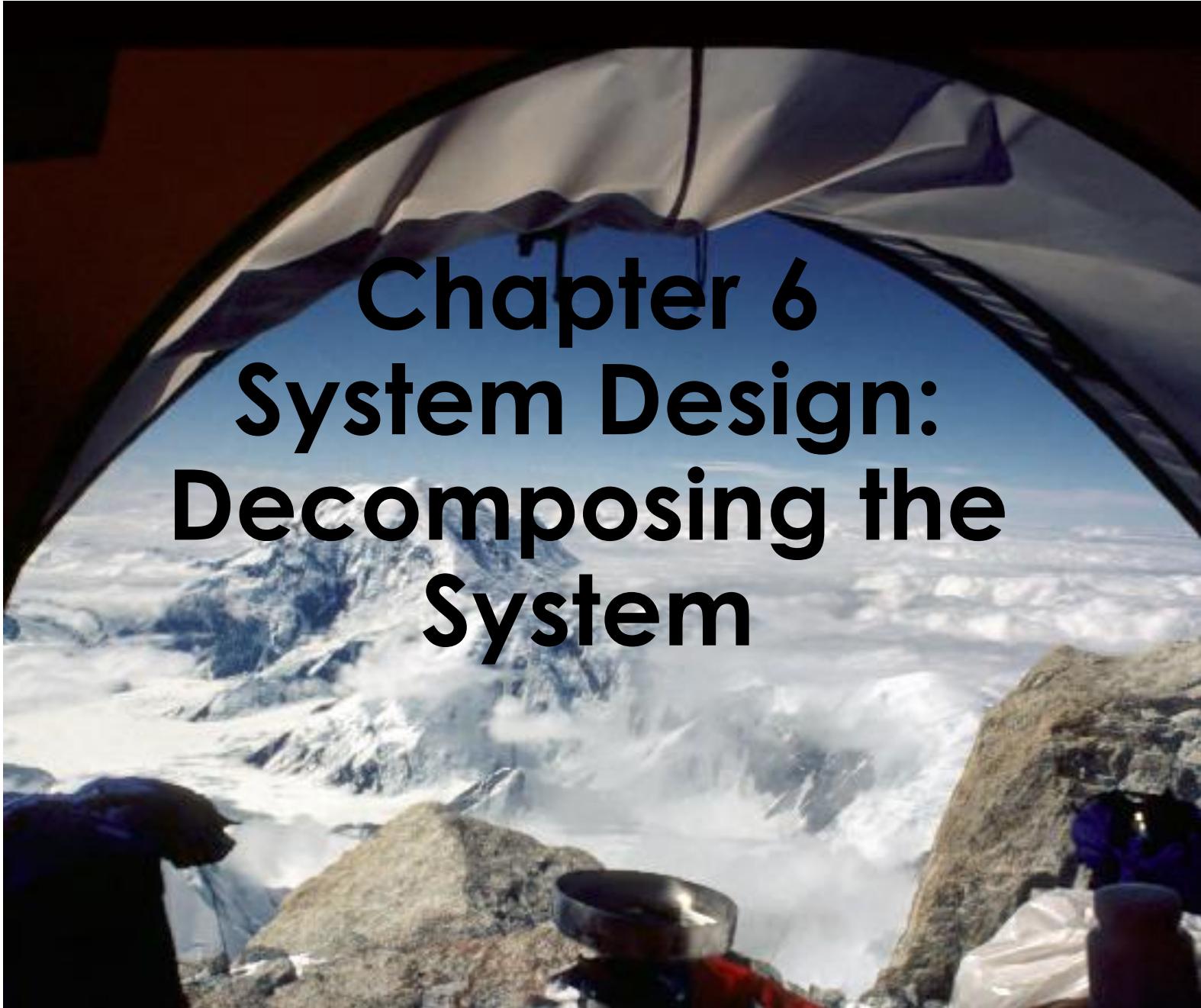


# Object-Oriented Software Engineering

Using UML, Patterns, and Java

## Chapter 6

# System Design: Decomposing the System



# Where are we?

- *We have covered Testing (Ch 11), Chapter on Object Design (Ch 9), Requirements Elicitations (Ch 2), Analysis (Ch 3).*
- *We are moving to Chapter 5 (System Design) and 6 (Addressing Design Goals).*

# Announcements

- Mid-term exam:
- Date, Time and Location:
- Programming assignments in exercises will start next week
  - Please bring your laptop to the exercise sessions
  - Please visit website and install prerequisites.

# Design is Difficult

- *There are two ways of constructing a software design (Tony Hoare):*
  - *One way is to make it so simple that there are obviously no deficiencies*
  - *The other way is to make it so complicated that there are no obvious deficiencies.”*
- *Corollary (Jostein Gaarder):*
  - *If our brain would be so simple that we can understand it, we would be too stupid to understand it.*



Sir Antony Hoare, \*1934

- Quicksort
- Hoare logic for verification
- CSP ([Communicating Sequential Processes](#)): modeling language for concurrent [processes](#) (basis for Occam).



Jostein Gardner, \*1952, writer

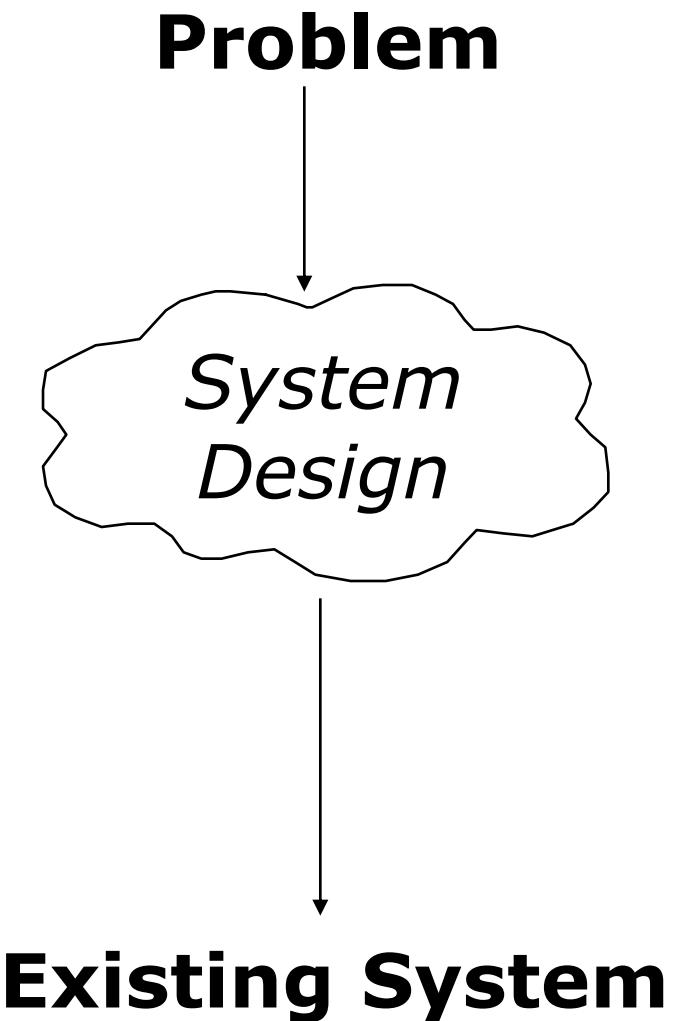
Uses metafiction in his stories:  
Fiction which uses the device of fiction  
- Best known for: „Sophie’s World“.

# Why is Design so Difficult?

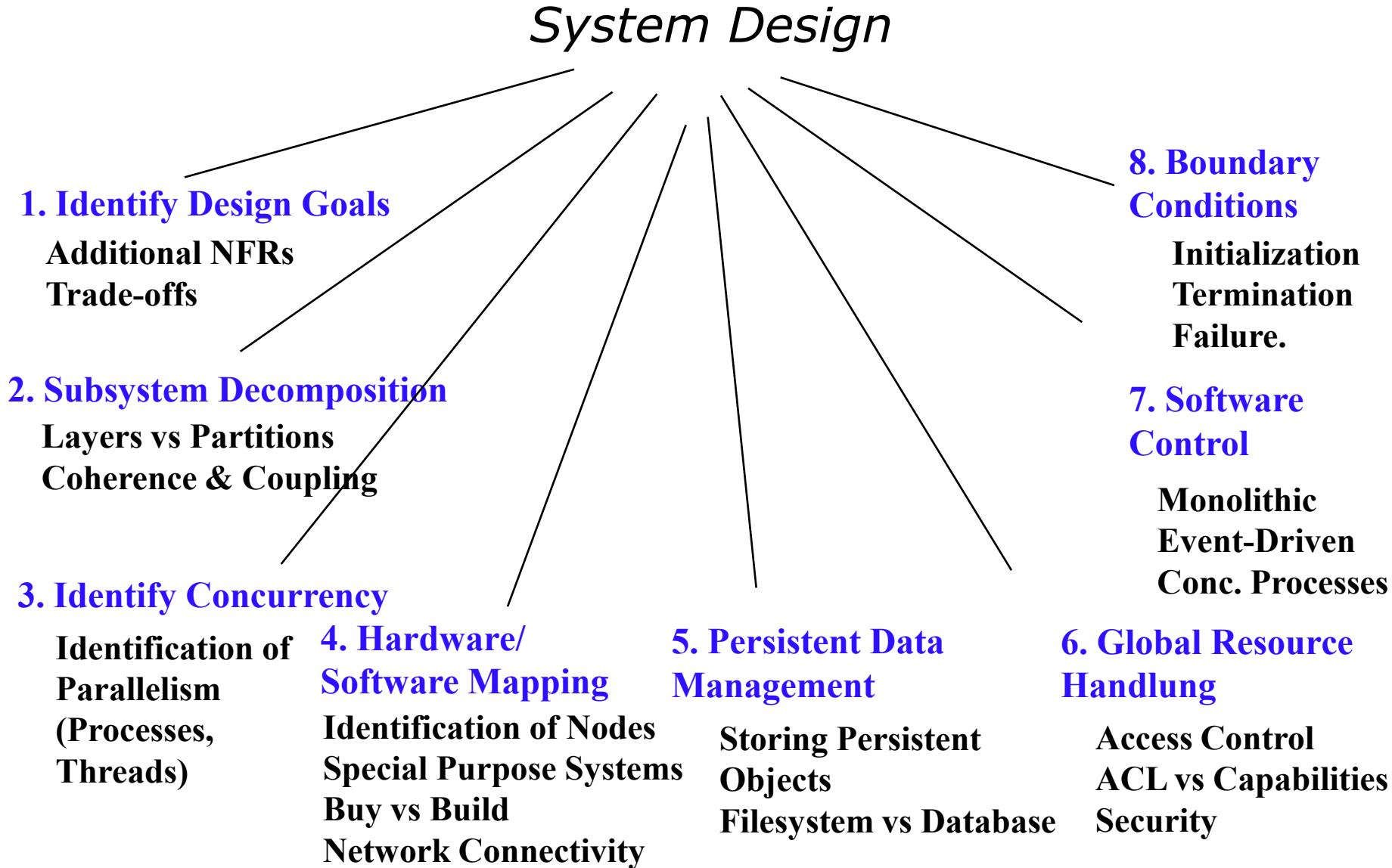
- *Analysis: Focuses on the application domain*
- *Design: Focuses on the solution domain*
  - *The solution domain is changing very rapidly*
    - *Halftime knowledge in software engineering: About 3-5 years*
    - *Cost of hardware rapidly sinking*
  - *Design knowledge is a moving target*
- *Design window: Time in which design decisions have to be made.*

# The Scope of System Design

- *Bridge the gap*
  - *between a problem and an existing system in a manageable way*
- *How?*
- *Use Divide & Conquer:*
  - 1) *Identify design goals*
  - 2) *Model the new system design as a set of subsystems*
  - 3-8) *Address the major design goals.*



# System Design: Eight Issues



# Overview

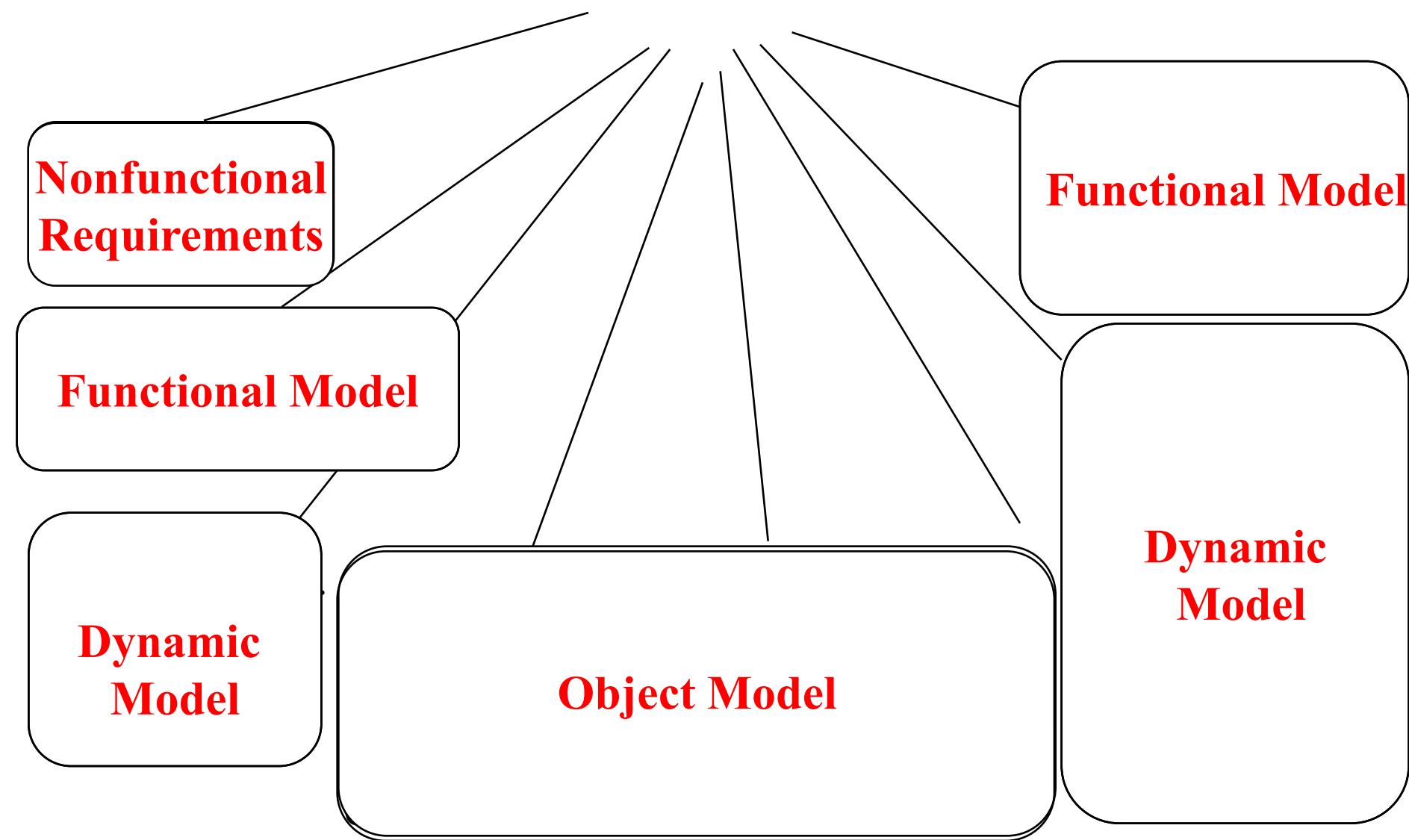
## *System Design I (This Lecture)*

0. Overview of System Design
1. Design Goals
2. Subsystem Decomposition, Software Architecture

## *System Design II (Next Lecture)*

3. Concurrency: Identification of parallelism
4. Hardware/Software Mapping:  
*Mapping subsystems to processors*
5. Persistent Data Management: Storage for entity objects
6. Global Resource Handling & Access Control:  
*Who can access what?)*
7. Software Control: Who is in control?
8. Boundary Conditions: Administrative use cases.

# *Analysis Sources: Requirements and System Model*



# How the Analysis Models influence System Design

- *Nonfunctional Requirements*  
    => *Definition of Design Goals*
- *Functional model*  
    => *Subsystem Decomposition*
- *Object model*  
    => *Hardware/Software Mapping, Persistent Data Management*
- *Dynamic model*  
    => *Identification of Concurrency, Global Resource Handling, Software Control*
- *Finally: Hardware/Software Mapping*  
    => *Boundary conditions*

# *From Analysis to System Design*

## Nonfunctional Requirements

### 1. Design Goals

Definition  
Trade-offs

## Functional Model

### 2. System Decomposition

Layers vs Partitions  
Coherence/Coupling

## Dynamic Model

### 3. Concurrency

Identification of  
Threads

## Object Model

### 4. Hardware/ Software Mapping

Special Purpose Systems  
Buy vs Build  
Allocation of Resources  
Connectivity

### 5. Data Management

Persistent Objects  
Filesystem vs  
Database

## Functional Model

### 8. Boundary Conditions

Initialization  
Termination  
Failure

## Dynamic Model

### 7. Software Control

Monolithic  
Event-Driven  
Conc. Processes

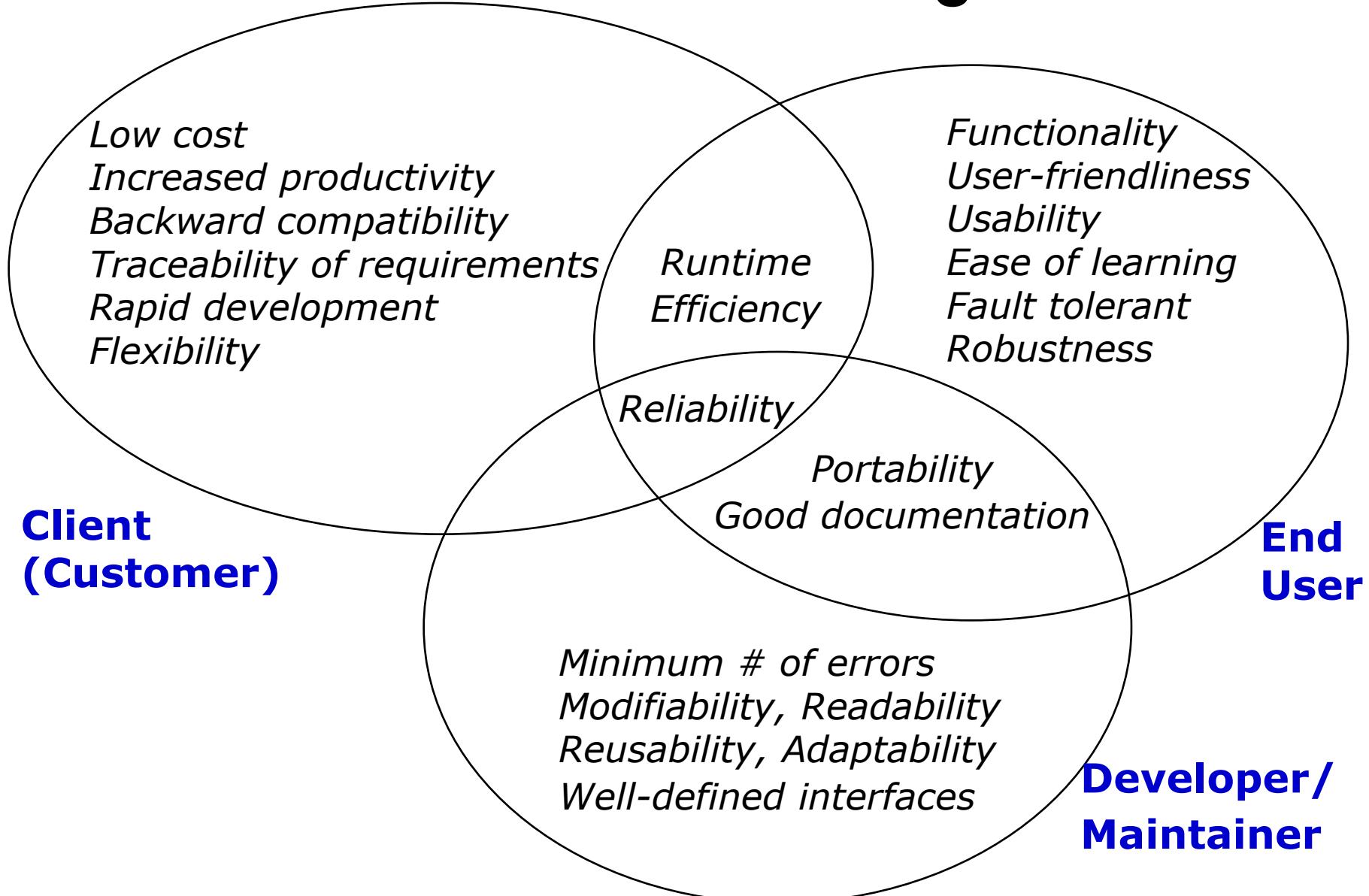
### 6. Global Resource Handling

Access Control List  
vs Capabilities  
Security

# Example of Design Goals

- *Reliability*
  - *Modifiability*
  - *Maintainability*
  - *Understandability*
  - *Adaptability*
  - *Reusability*
  - *Efficiency*
  - *Portability*
  - *Traceability of requirements*
  - *Fault tolerance*
  - *Backward-compatibility*
  - *Cost-effectiveness*
  - *Robustness*
  - *High-performance*
- ❖ *Good documentation*
  - ❖ *Well-defined interfaces*
  - ❖ *User-friendliness*
  - ❖ *Reuse of components*
  - ❖ *Rapid development*
  - ❖ *Minimum number of errors*
  - ❖ *Readability*
  - ❖ *Ease of learning*
  - ❖ *Ease of remembering*
  - ❖ *Ease of use*
  - ❖ *Increased productivity*
  - ❖ *Low-cost*
  - ❖ *Flexibility*

# Stakeholders have different Design Goals



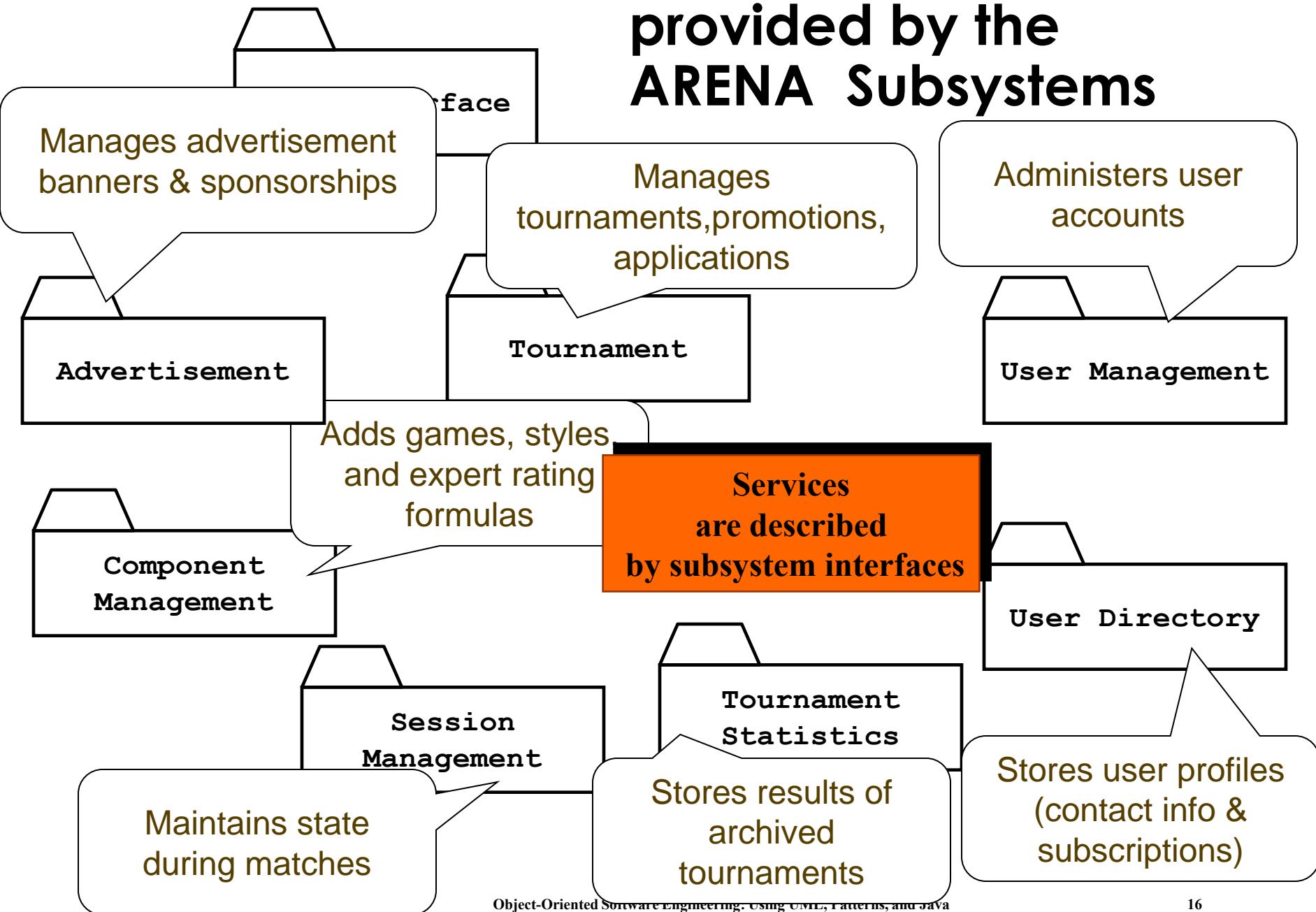
# Typical Design Trade-offs

- *Functionality v. Usability*
- *Cost v. Robustness*
- *Efficiency v. Portability*
- *Rapid development v. Functionality*
- *Cost v. Reusability*
- *Backward Compatibility v. Readability*

# Subsystem Decomposition

- *Subsystem*
  - *Collection of classes, associations, operations, events and constraints that are closely interrelated with each other*
  - *The objects and classes from the object model are the "seeds" for the subsystems*
  - *In UML subsystems are modeled as packages*
- *Service*
  - *A set of named operations that share a common purpose*
  - *The origin ("seed") for services are the use cases from the functional model*
- *Services are defined during system design.*

# Example: Services provided by the ARENA Subsystems



# Subsystem Interfaces vs API

- *Subsystem interface: Set of fully typed UML operations*
  - *Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem*
  - *Refinement of service, should be well-defined and small*
  - *Subsystem interfaces are defined during object design*
- *Application programmer's interface (API)*
  - *The API is the specification of the subsystem interface in a specific programming language*
  - *APIs are defined during implementation*
- *The terms subsystem interface and API are often confused with each other*
  - *The term API should not be used during system design and object design, but only during implementation.*

# Example: Notification subsystem

- *Service provided by Notification Subsystem*
  - *LookupChannel()*
  - *SubscribeToChannel()*
  - *SendNotice()*
  - *UnsubscribeFromChannel()*
- *Subsystem Interface of Notification Subsystem*
  - *Set of fully typed UML operations*
    - *Left as an Exercise*
- *API of Notification Subsystem*
  - *Implementation in Java*
  - *Left as an Exercise.*

# Subsystem Interface Object

- *Good design: The subsystem interface object describes all the services of the subsystem interface*
- *Subsystem Interface Object*
  - *The set of public operations provided by a subsystem*

*Subsystem Interface Objects can be realized with the Façade pattern (=> lecture on design patterns).*

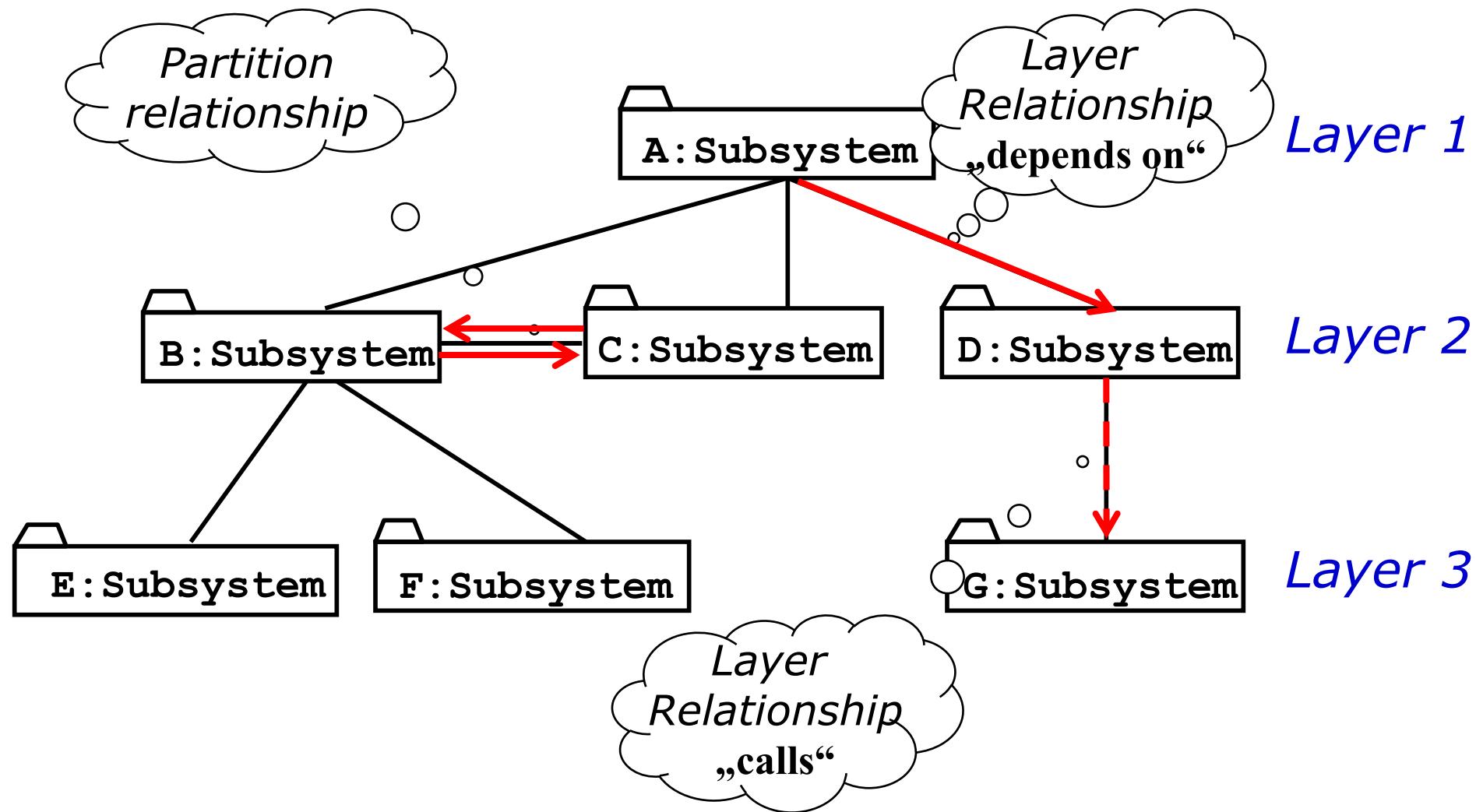
# Properties of Subsystems: Layers and Partitions

- A *layer* is a subsystem that provides a service to another subsystem with the following restrictions:
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers
- A layer can be divided horizontally into several independent subsystems called *partitions*
  - Partitions provide services to other partitions on the same layer
  - Partitions are also called "weakly coupled" subsystems.

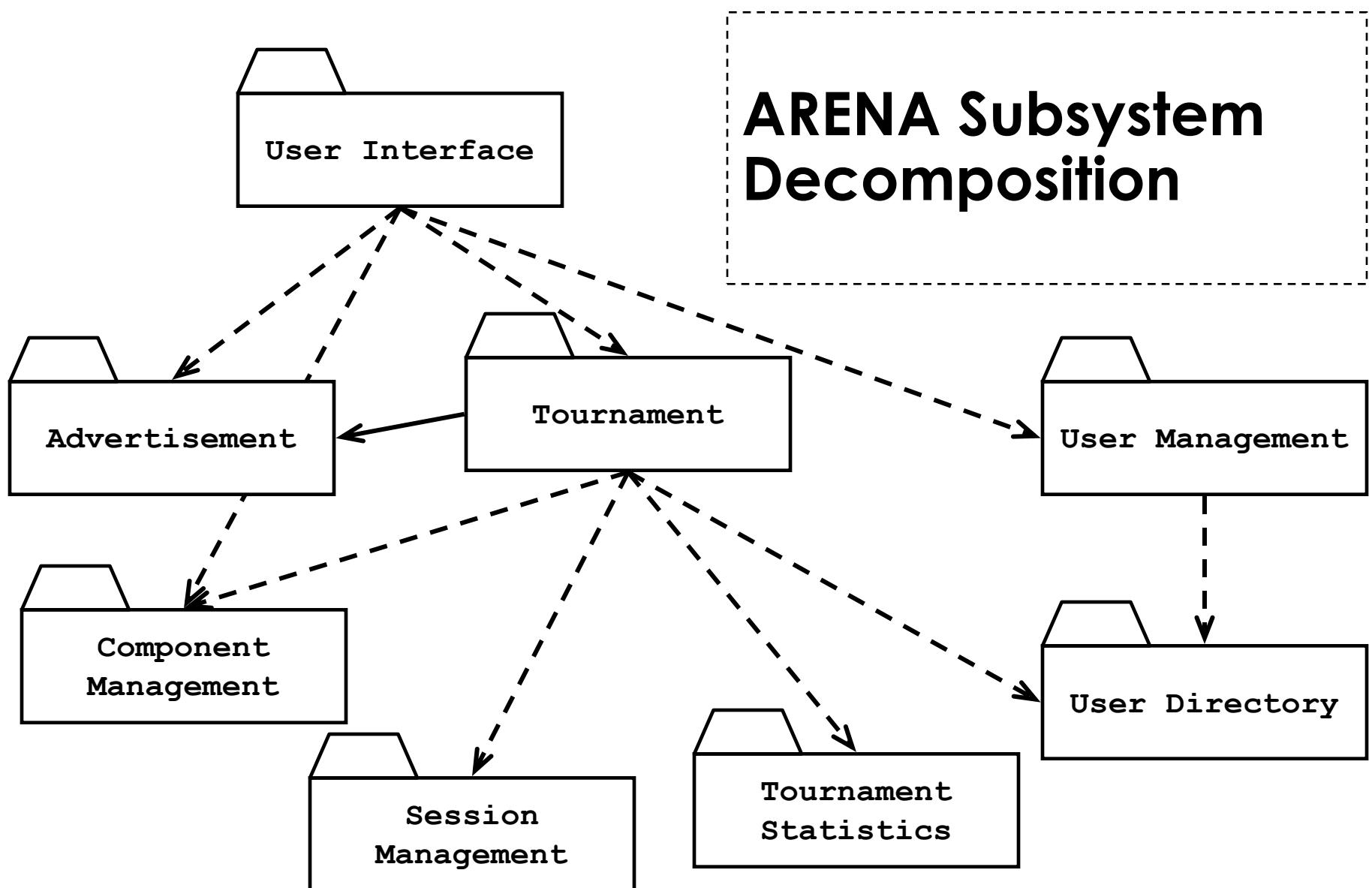
# Relationships between Subsystems

- *Two major types of Layer relationships*
  - *Layer A "depends on" Layer B (compile time dependency)*
    - *Example: Build dependencies (make, ant, maven)*
  - *Layer A "calls" Layer B (runtime dependency)*
    - *Example: A web browser calls a web server*
    - *Can the client and server layers run on the same machine?*
      - *Yes, they are layers, not processor nodes*
      - *Mapping of layers to processors is decided during the Software/hardware mapping!*
- *Partition relationship*
  - *The subsystems have mutual knowledge about each other*
    - *A calls services in B; B calls services in A (Peer-to-Peer)*
- *UML convention:*
  - *Runtime dependencies are associations with dashed lines*
  - *Compile time dependencies are associations with solid lines.*

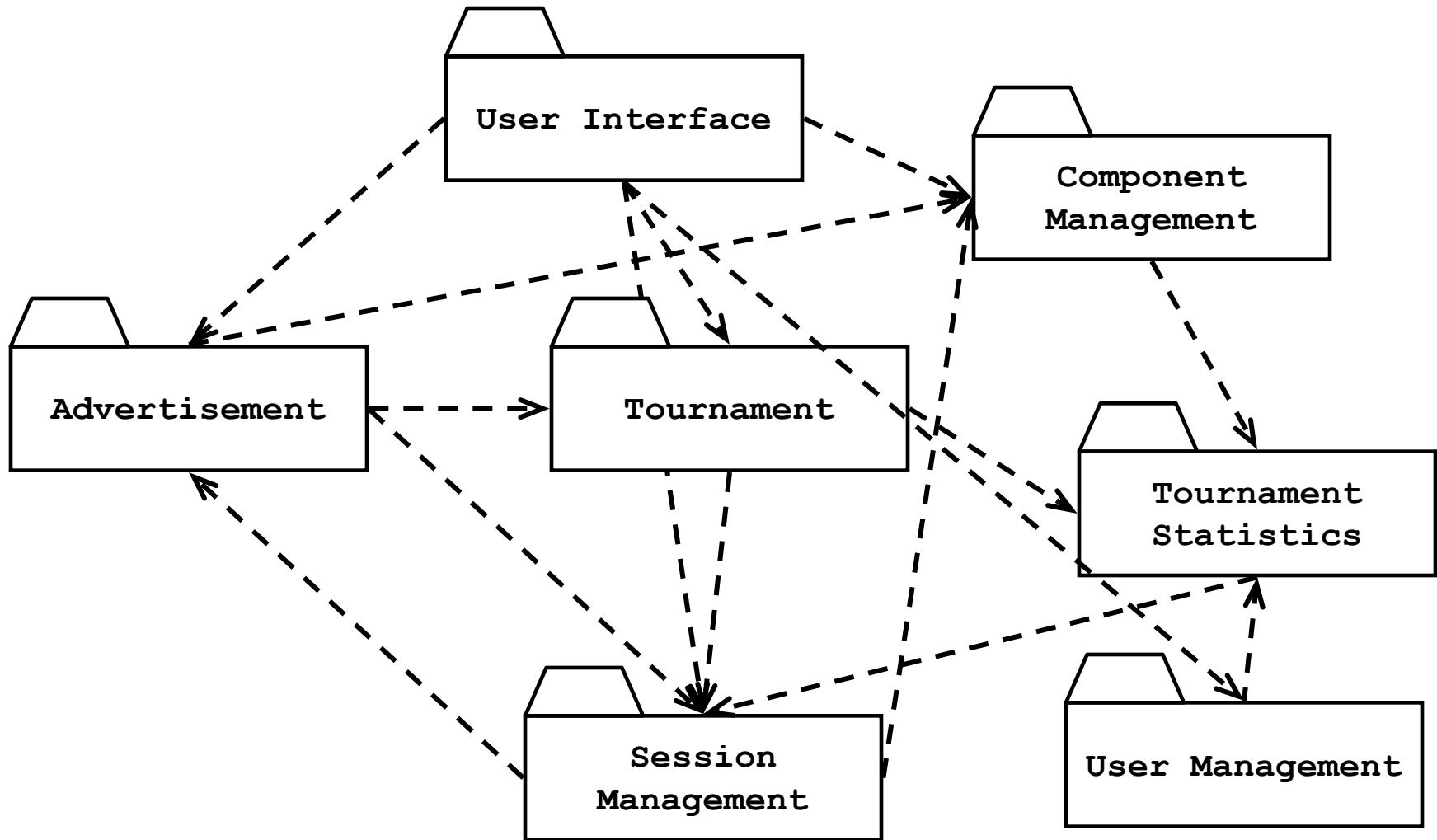
# Example of a Subsystem Decomposition



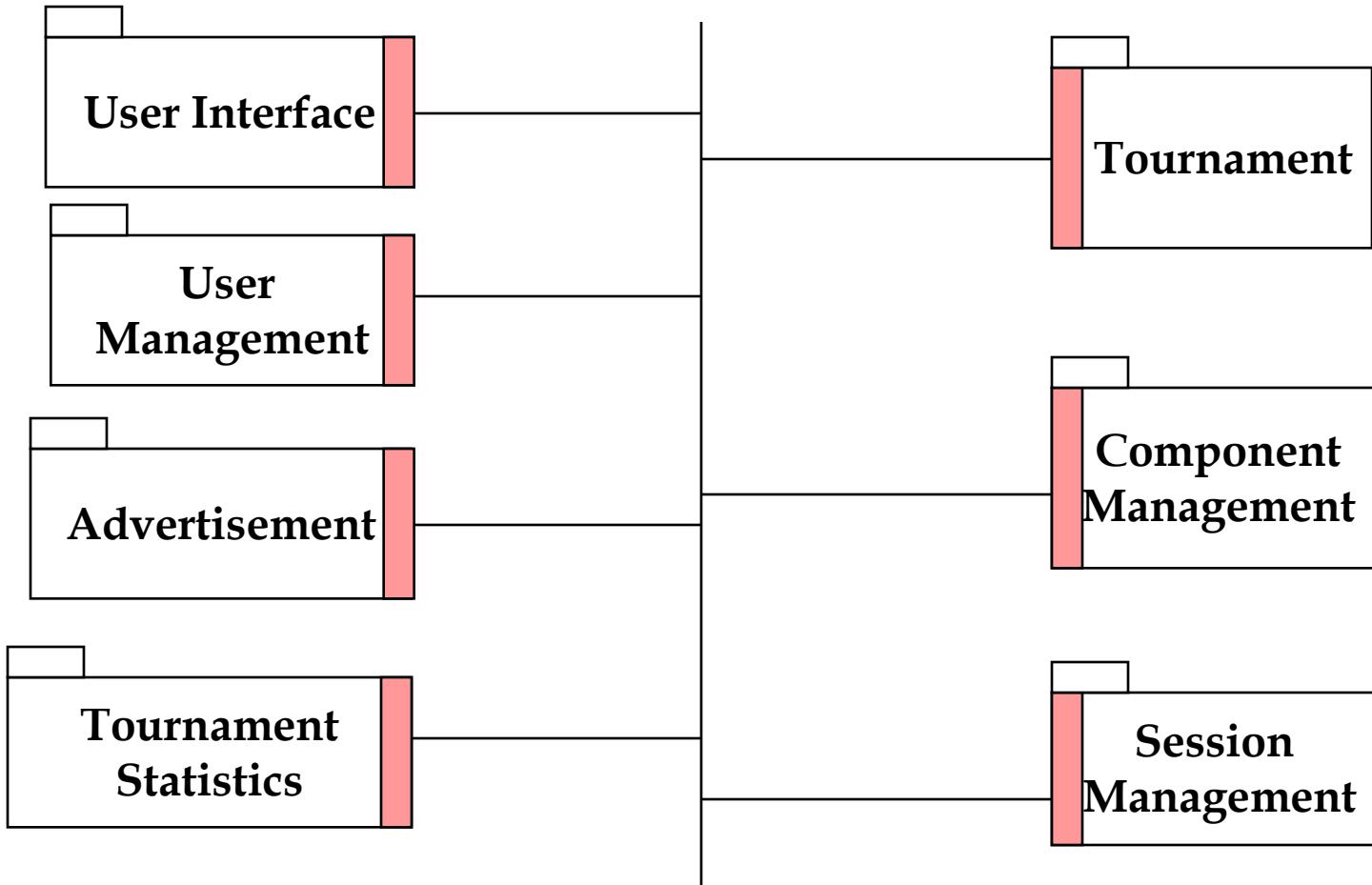
# ARENA Subsystem Decomposition



# Example of a Bad Subsystem Decomposition



# Good Design: The System as set of Interface Objects



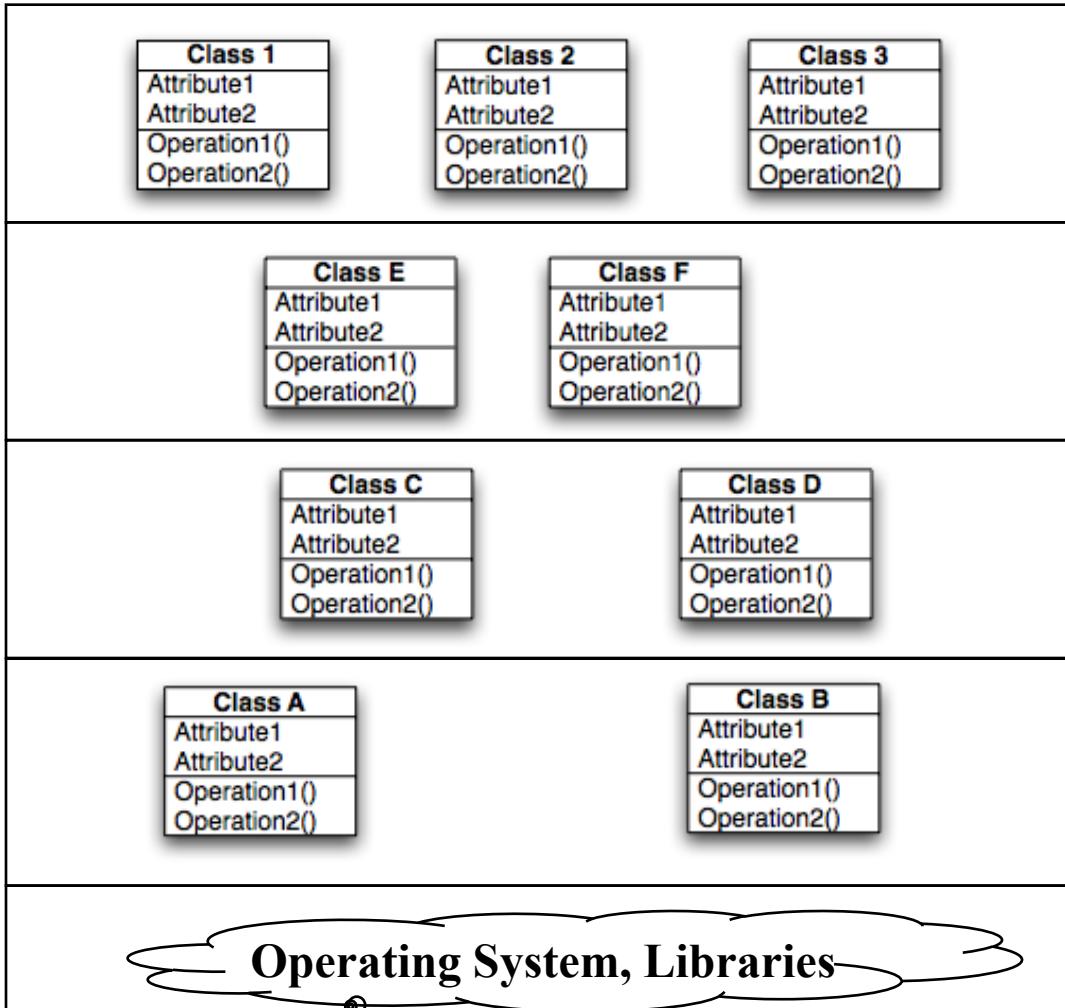
*Subsystem Interface Objects*

# Virtual Machine

- A *virtual machine* is a subsystem connected to higher and lower level virtual machines by "provides services for" associations
- A *virtual machine* is an abstraction that provides a set of attributes and operations
- The terms *layer* and *virtual machine* can be used interchangeably
  - Also sometimes called "level of abstraction".

# Building Systems as a Set of Virtual Machines

A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.



Virtual Machine 4 .

Virtual Machine 3

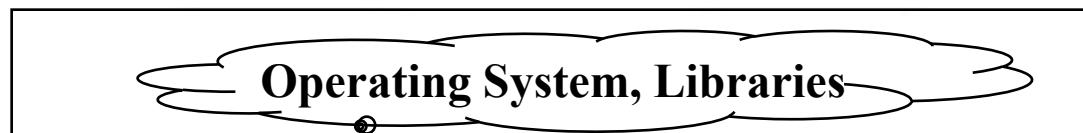
Virtual Machine 2

Virtual Machine 1

Existing System

# Building Systems as a Set of Virtual Machines

*A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.*

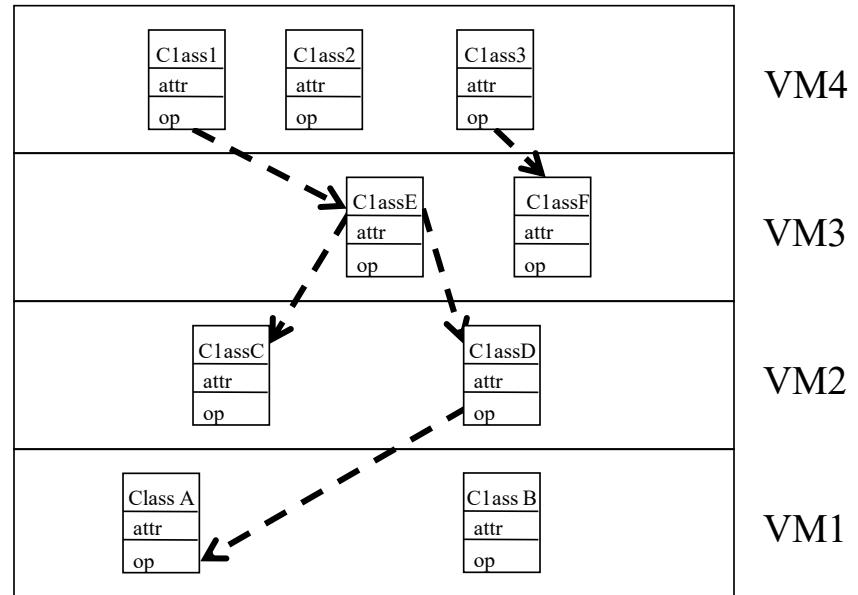


Existing System

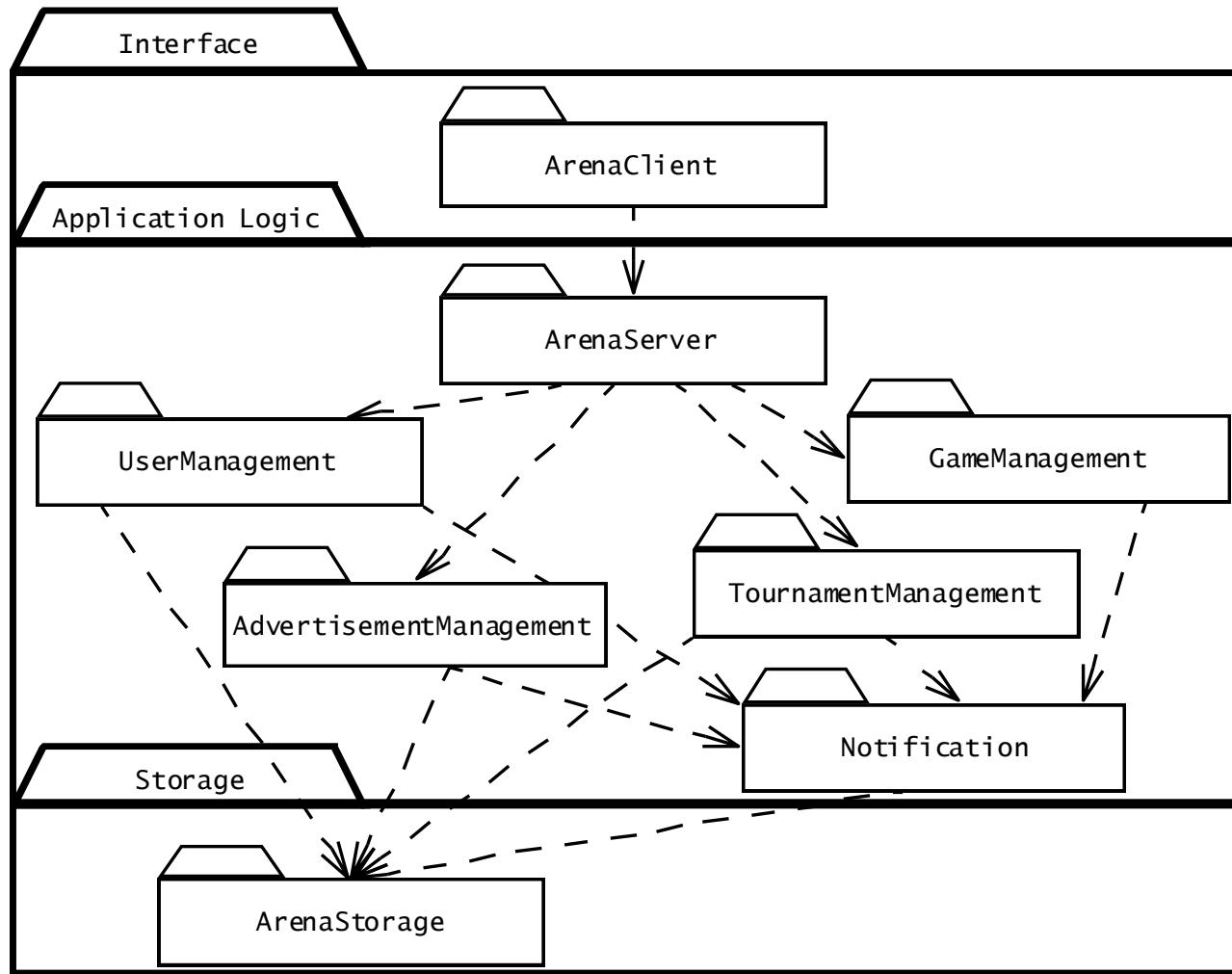
# Closed Architecture (Opaque Layering)

- *Each virtual machine can only call operations from the layer below*

*Design goals:*  
Maintainability,  
flexibility.



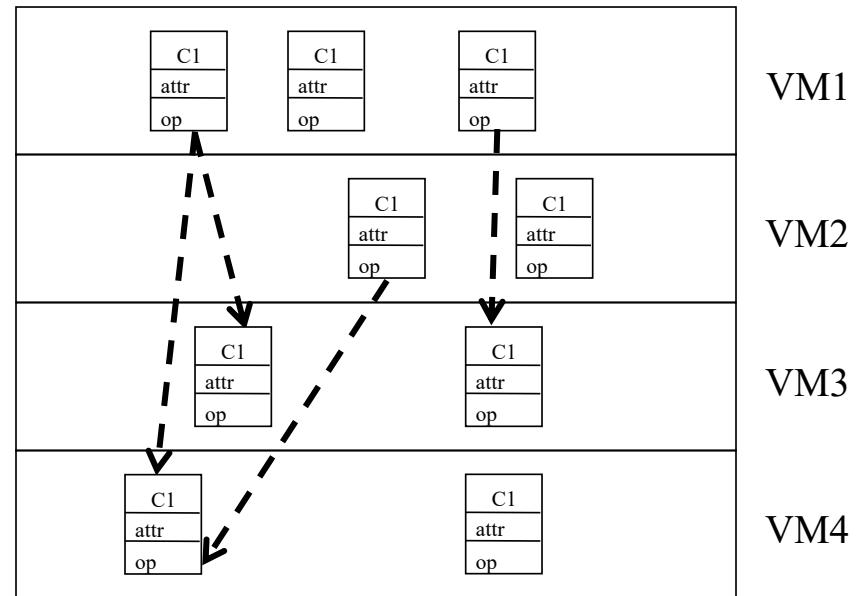
# Opaque Layering in ARENA



# Open Architecture (Transparent Layering)

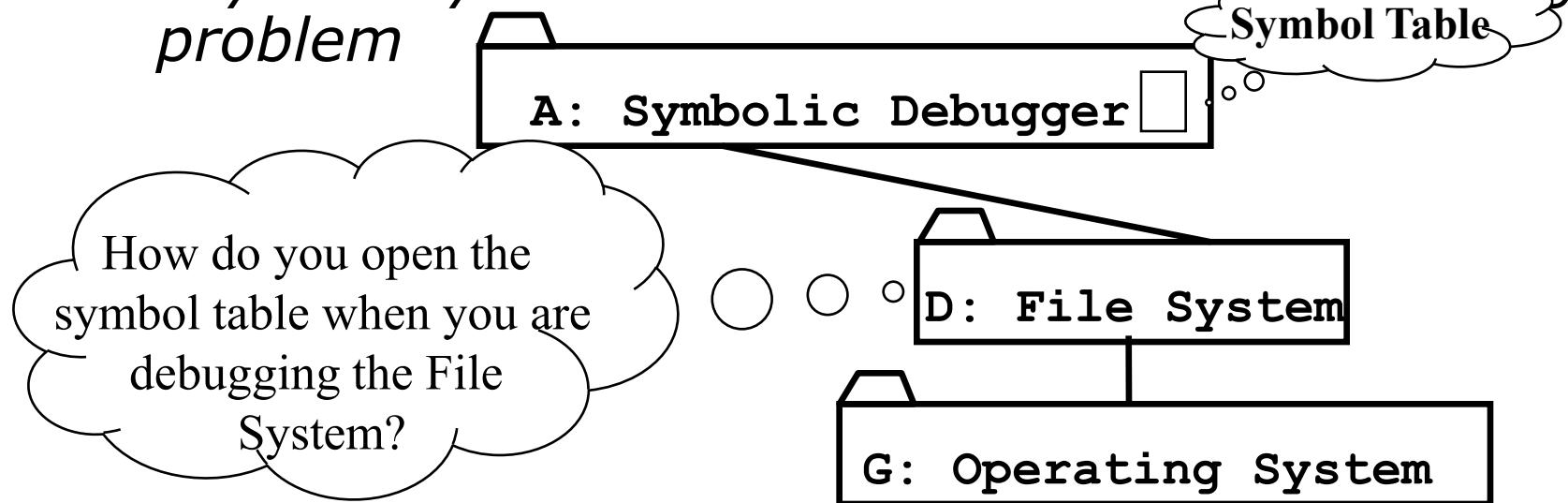
- *Each virtual machine can call operations from any layer below*

*Design goal:*  
*Runtime efficiency*



# Properties of Layered Systems

- *Layered systems are hierarchical. This is a desirable design, because hierarchy reduces complexity*
  - *low coupling*
- *Closed architectures are more portable*
- *Open architectures are more efficient*
- *Layered systems often have a chicken-and-egg problem*



# Coupling and Coherence of Subsystems

- *Goal: Reduce system complexity while allowing change*
- **Coherence** measures dependency among classes
  - *High coherence: The classes in the subsystem perform similar tasks and are related to each other via many associations*
  - *Low coherence: Lots of miscellaneous and auxiliary classes, almost no associations*
- **Coupling** measures dependency among subsystems
  - *High coupling: Changes to one subsystem will have high impact on the other subsystem*
  - *Low coupling: A change in one subsystem does not affect any other subsystem.*

# Coupling and Coherence of Subsystems

## Good Design

- *Goal: Reduce system complexity while allowing change*
  - *Coherence measures dependency among classes*
    - *High coherence: The classes in the subsystem perform similar tasks and are related to each other via associations*
      - *Low coherence: Lots of miscellaneous and auxiliary classes, no associations*
  - *Coupling measures dependency among subsystems*
    - *High coupling: Changes to one subsystem will have high impact on the other subsystem*
- *Low coupling: A change in one subsystem does not affect any other subsystem*

# How to achieve high Coherence

- *High coherence can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries*
- *Questions to ask:*
  - *Does one subsystem always call another one for a specific service?*
    - *Yes: Consider moving them together into the same subsystem.*
  - *Which of the subsystems call each other for services?*
    - *Can this be avoided by restructuring the subsystems or changing the subsystem interface?*
    - *Can the subsystems even be hierarchically ordered (in layers)?*

# How to achieve Low Coupling

- *Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (**Principle of information hiding**, Parnas)*
- *Questions to ask:*
  - Does the calling class really have to know any attributes of classes in the lower layers?
  - Is it possible that the calling class calls only operations of the lower level classes?

*David Parnas, \*1941,  
Developed the concept of  
modularity in design.*



# Architectural Style vs Architecture

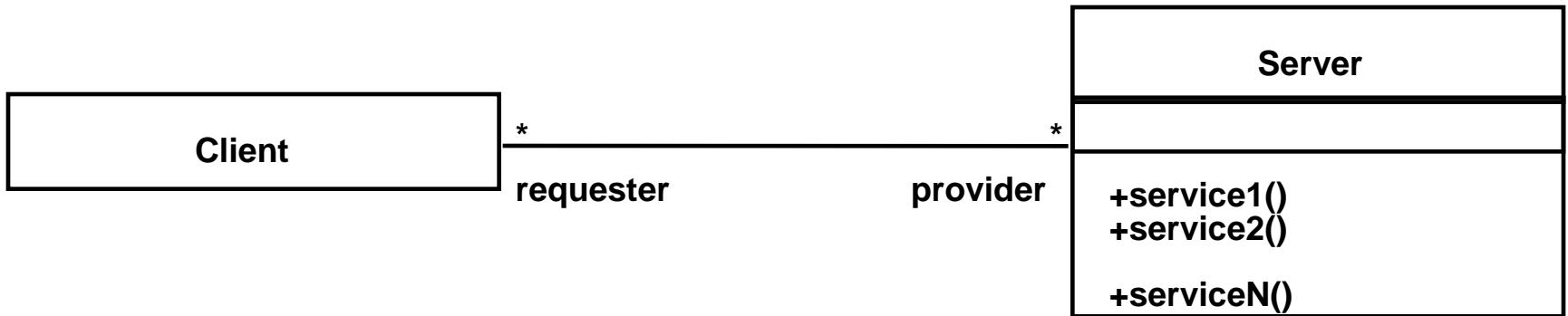
- *Subsystem decomposition: Identification of subsystems, services, and their association to each other (hierarchical, peer-to-peer, etc)*
- *Architectural Style: A pattern for a subsystem decomposition*
- *Software Architecture: Instance of an architectural style.*

# Examples of Architectural Styles

- *Client/Server*
- *Peer-To-Peer*
- *Repository*
- *Model/View/Controller*
- *Three-tier, Four-tier Architecture*
- *Service-Oriented Architecture (SOA)*
- *Pipes and Filters*

# Client/Server Architectural Style

- One or many *servers* provide services to instances of subsystems, called *clients*
- Each client calls on the server, which performs some service and returns the result
  - The clients know the interface of the server
  - The server does not need to know the interface of the client
- The response in general is immediate
- End users interact only with the client.



# Client/Server Architectures

- Often used in the design of database systems
  - Front-end: User application (client)
  - Back end: Database access and manipulation (server)
- Functions performed by client:
  - Input from the user (Customized user interface)
  - Front-end processing of input data
- Functions performed by the database server:
  - Centralized data management
  - Data integrity and database consistency
  - Database security

# Design Goals for Client/Server Architectures

## *Service Portability*

*Server runs on many operating systems and many networking environments*

## *Location-Transparency*

*Server might itself be distributed, but provides a single "logical" service to the user*

## *High Performance*

*Client optimized for interactive display-intensive tasks; Server optimized for CPU-intensive operations*

## *Scalability*

*Server can handle large # of clients*

## *Flexibility*

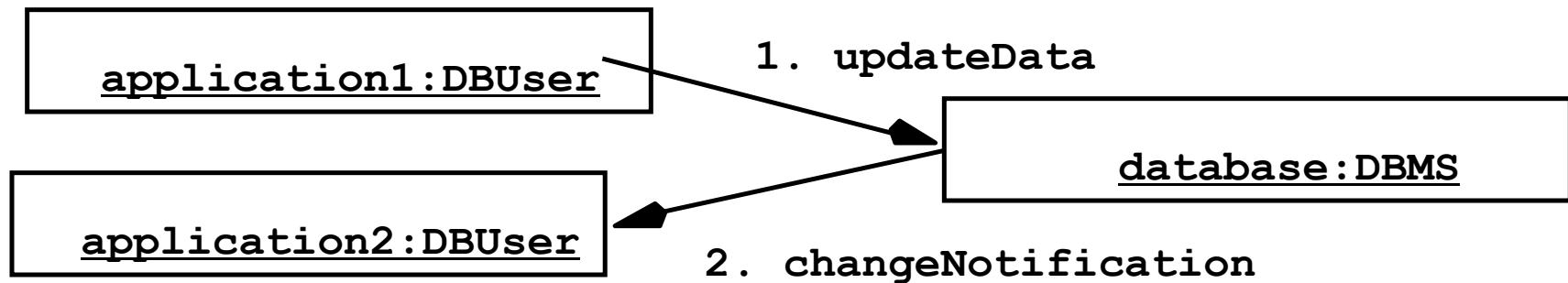
*User interface of client supports a variety of end devices (PDA, Handy, laptop, wearable computer)*

## *Reliability*

A measure of success with which the observed behavior of a system confirms to the specification of its behavior (Chapter 11: Testing)

# Problems with Client/Server Architectures

- *Client/Server systems do not provide peer-to-peer communication*
- *Peer-to-peer communication is often needed*
- *Example:*
  - *Database must process queries from application and should be able to send notifications to the application when data have changed*



# Peer-to-Peer Architectural Style

*Generalization of Client/Server Architectural Style*

**"Clients can be servers and servers can be clients"**

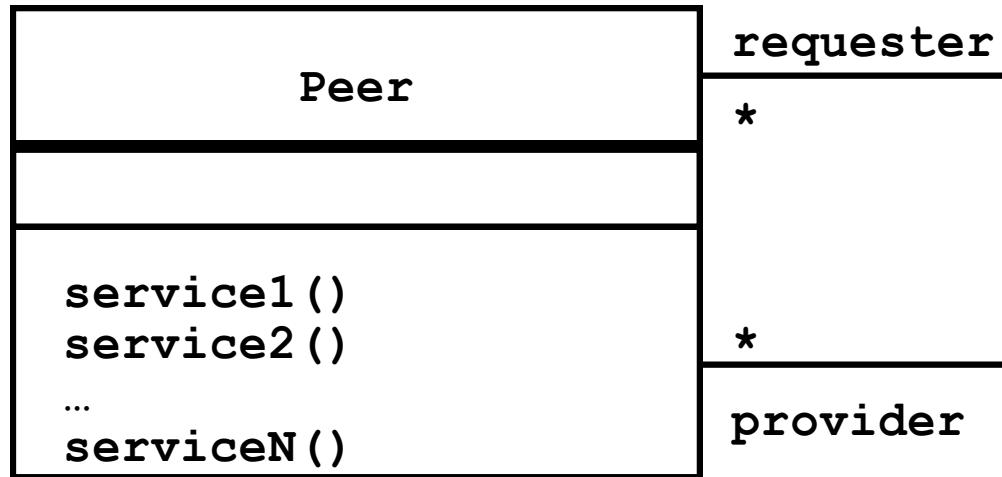
*Introduction a new abstraction: Peer*

**"Clients and servers can be both peers"**

*How do we model this statement? With Inheritance?*

*Proposal 1:* "A peer can be either a client or a server"

*Proposal 2:* "A peer can be a client as well as a server".



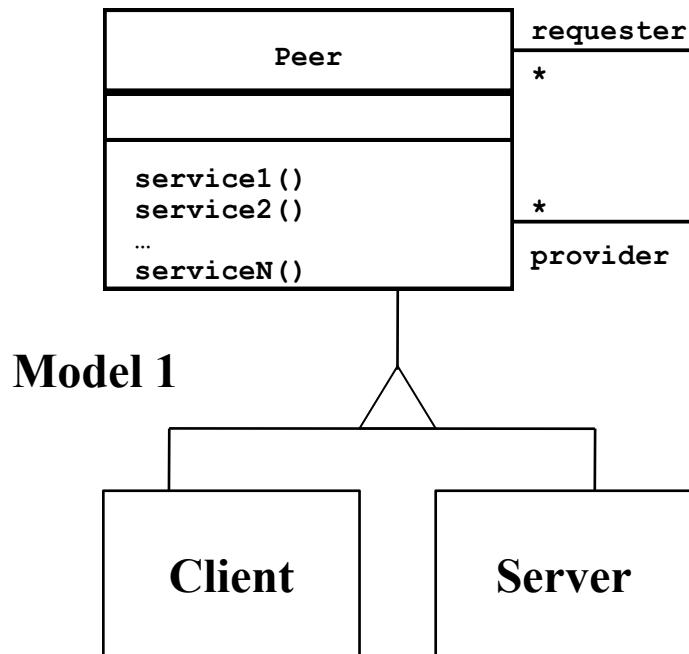
# Relationship Client/Server & Peer-to-Peer

*Problem statement "Clients can be servers and servers can be clients"*

*Which model is correct?*

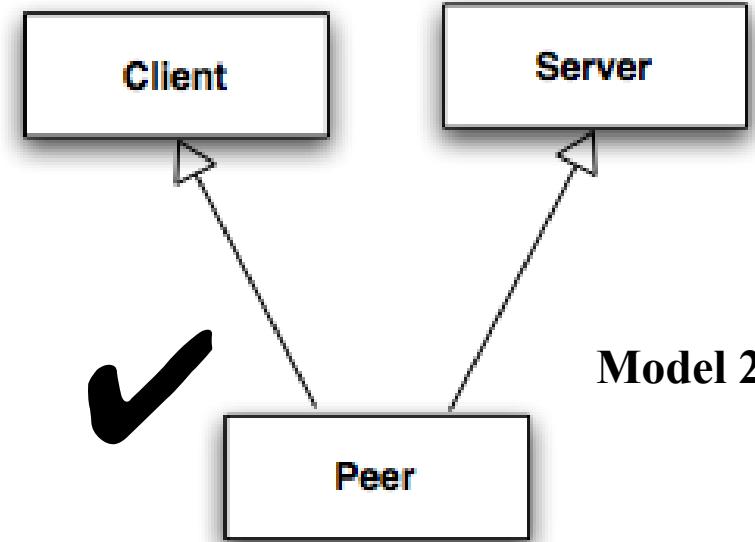
*Model 1: "A peer can be either a client or a server"*

*Model 2: "A peer can be a client as well as a server"*



**Model 1**

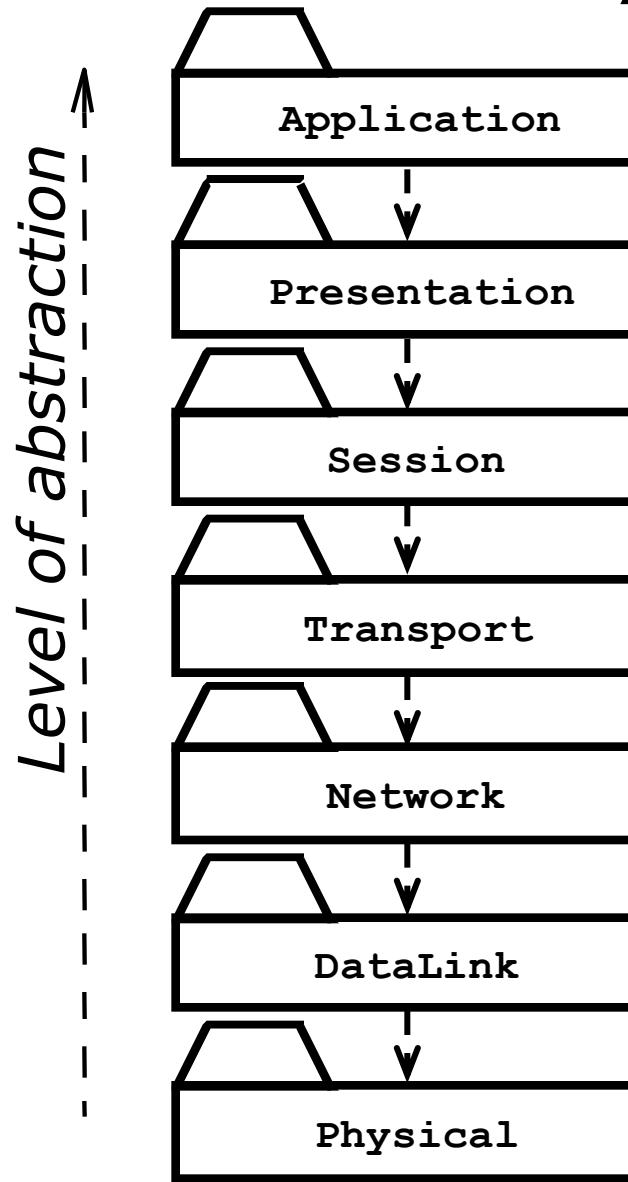
?



**Model 2**

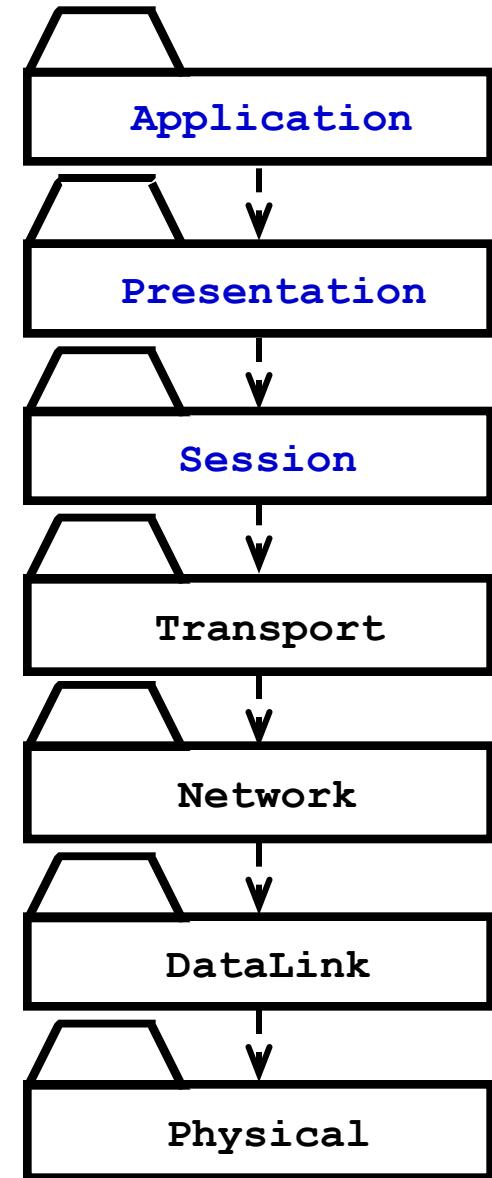
# Example: Peer-to-Peer Architectural Style

- ISO's OSI Reference Model
  - ISO = International Standard Organization
  - OSI = Open System Interconnection
- Reference model which defines 7 layers and communication protocols between the layers



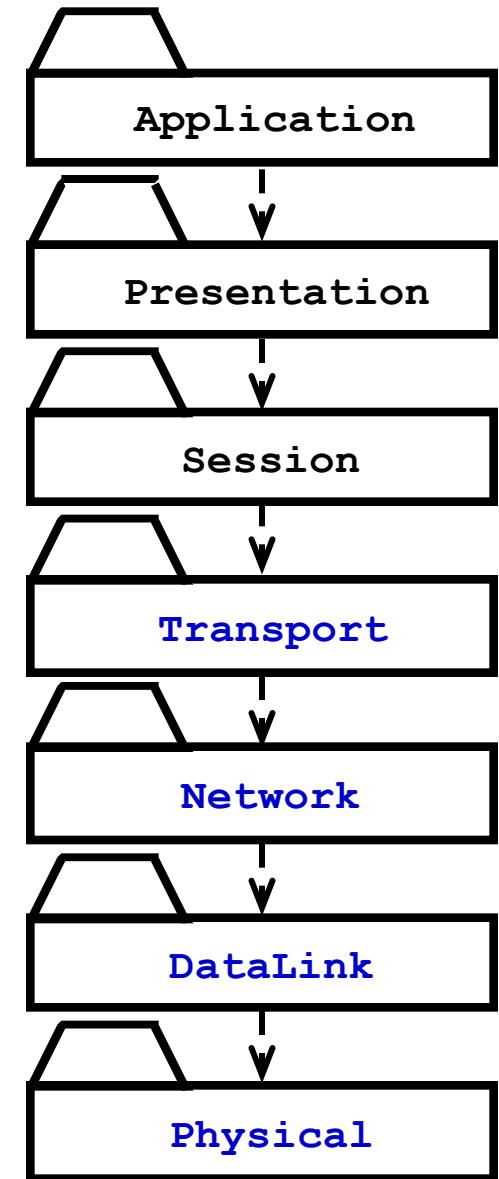
# OSI Model Layers and Services

- *The Application layer is the system you are building (unless you build a protocol stack)*
  - ! • *The application layer is usually layered itself*
- *The Presentation layer performs data transformation services, such as byte swapping and encryption*
- *The Session layer is responsible for initializing a connection, including authentication*

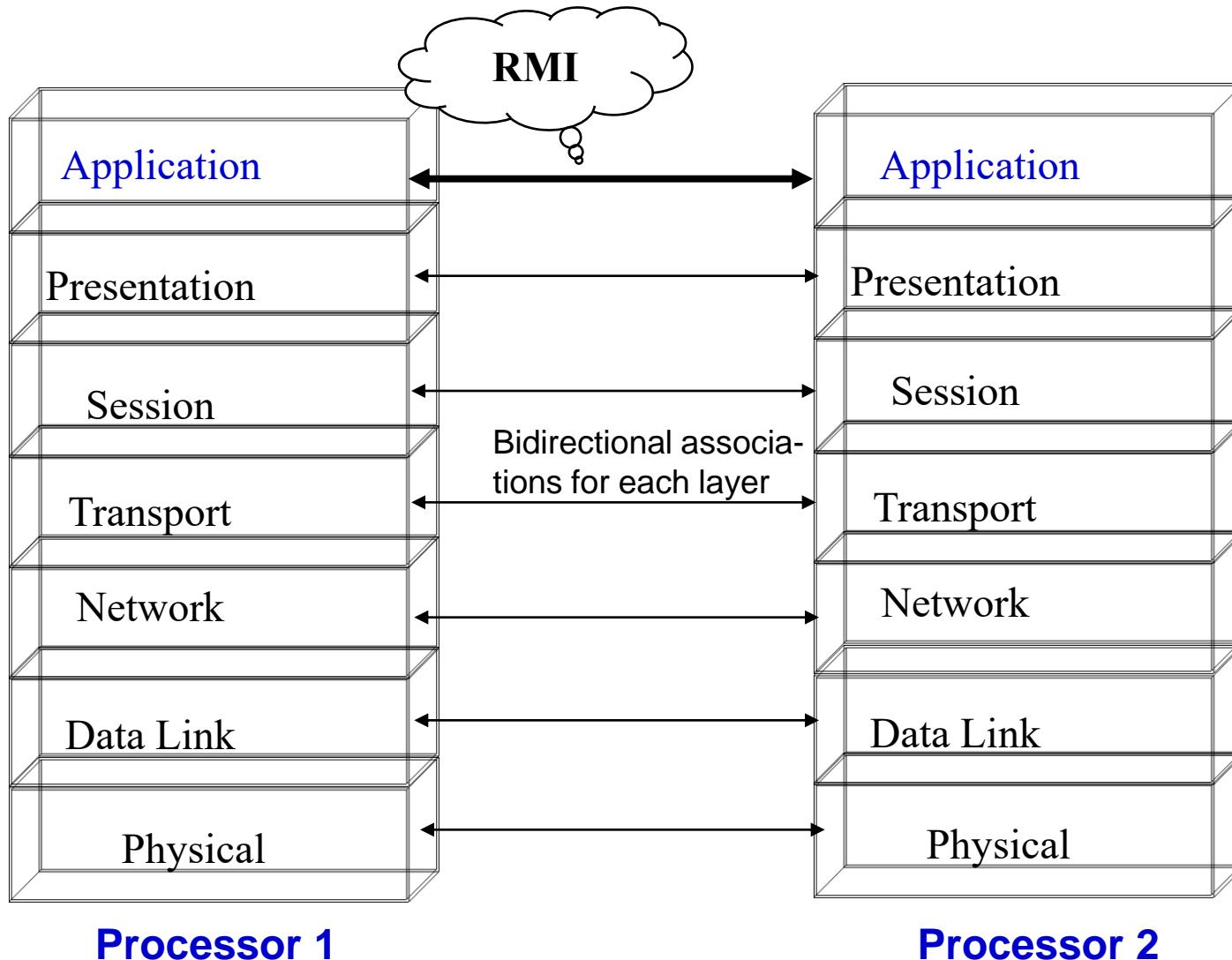


# OSI Model Layers and their Services

- The *Transport layer* is responsible for reliably transmitting messages
  - Used by Unix programmers who transmit messages over TCP/IP sockets
- The *Network layer* ensures transmission and routing
  - Services: Transmit and route data within the network
- The *Datalink layer* models frames
  - Services: Transmit frames without error
- The *Physical layer* represents the hardware interface to the network
  - Services: `sendBit()` and `receiveBit()`

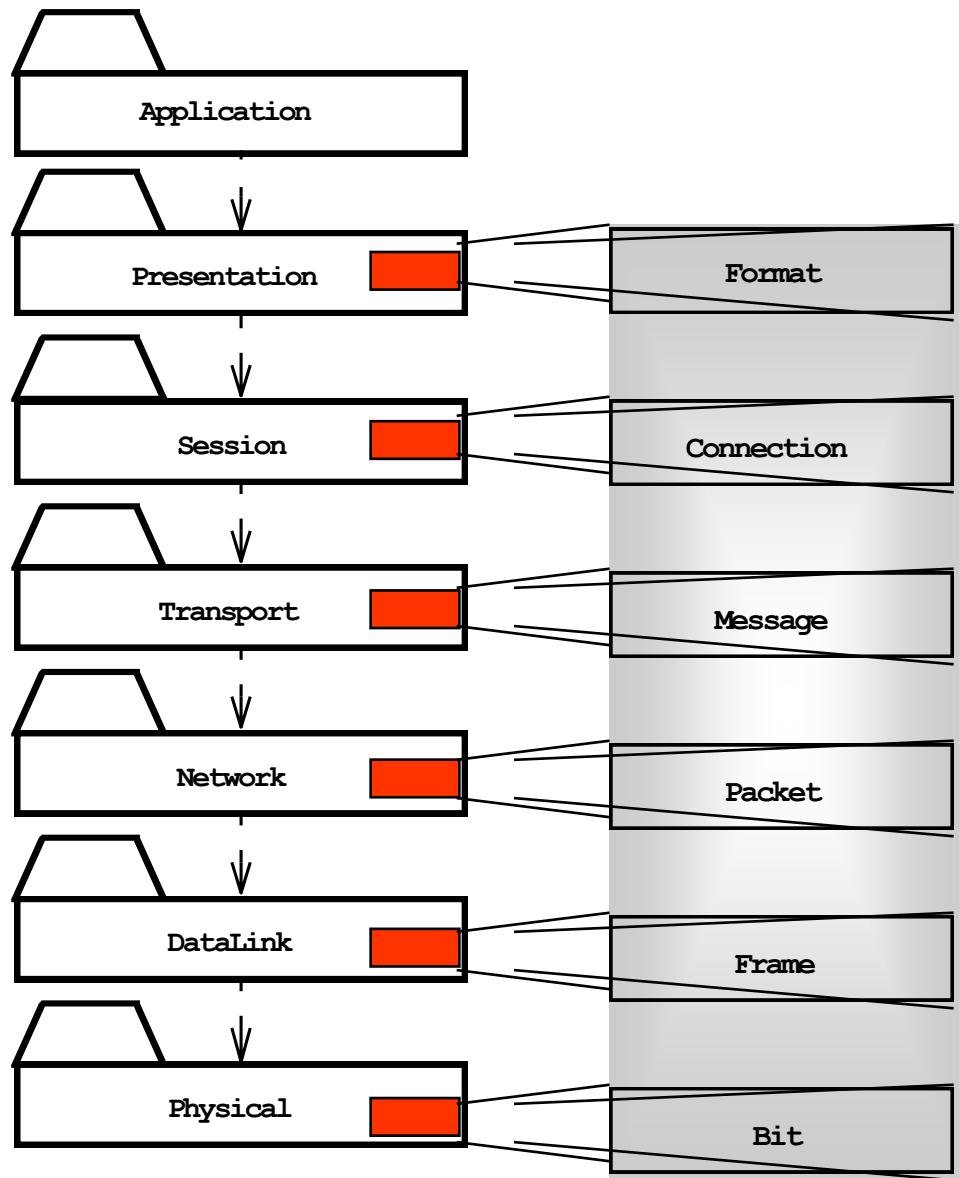


# The Application Layer Provides the Abstractions of the “New System”



# An Object-Oriented View of the OSI Model

- *The OSI Model is a closed software architecture (i.e., it uses opaque layering)*
- *Each layer can be modeled as a UML package containing a set of classes available for the layer above*

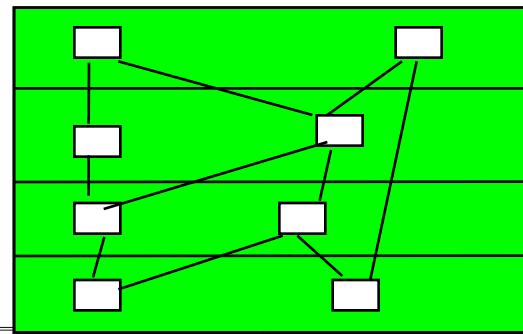


**Layer 1**

**Layer 2**

**Layer 3**

**Layer 4**



**Layer 1**

**Layer 2**

**Layer 3**

Application Layer

Presentation Layer

Session Layer

Transport Layer

Network Layer

Data Link Layer

Physical

Application Layer

Presentation Layer

Session Layer

Transport Layer

Network Layer

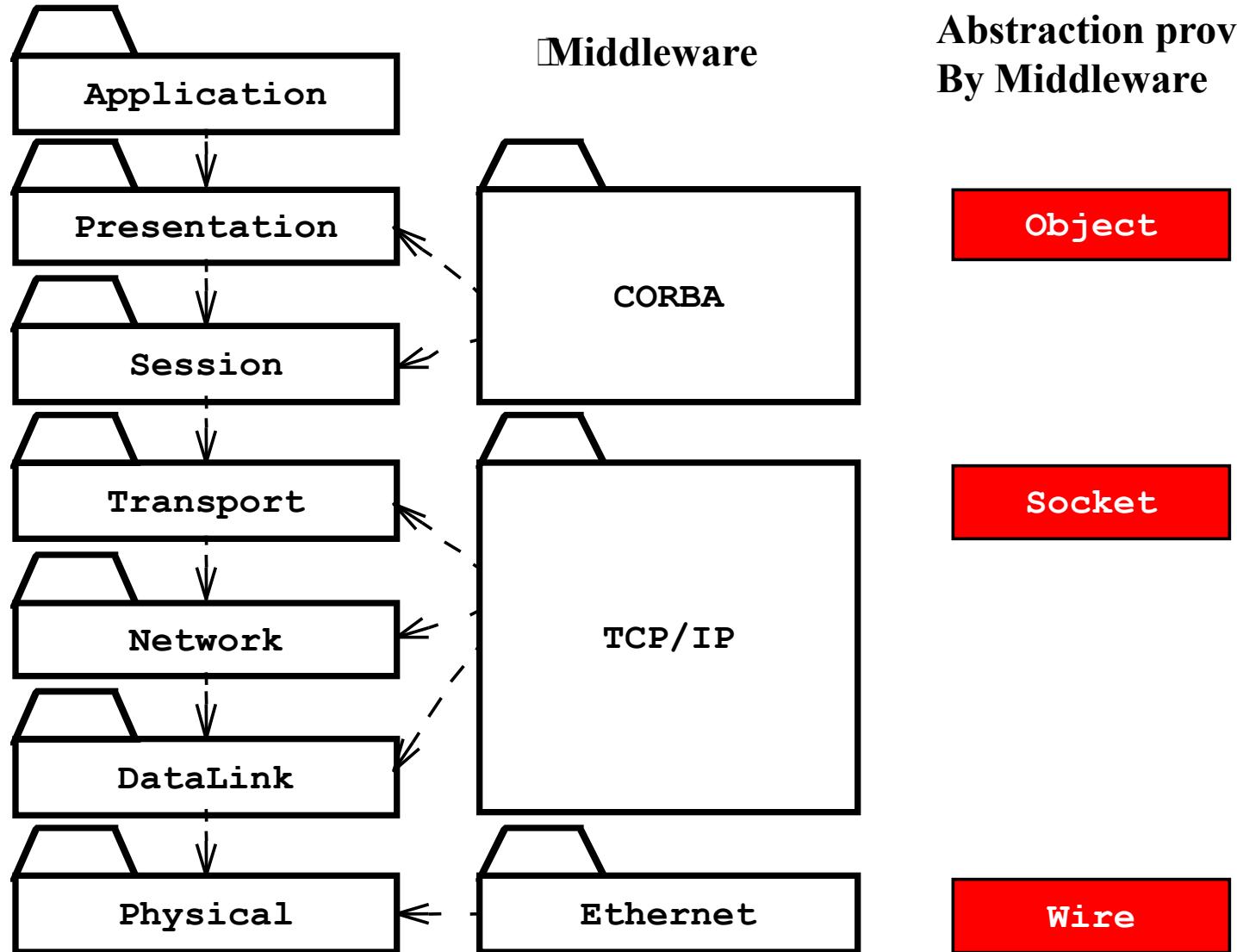
Data Link Layer

Physical

**Processor 1**

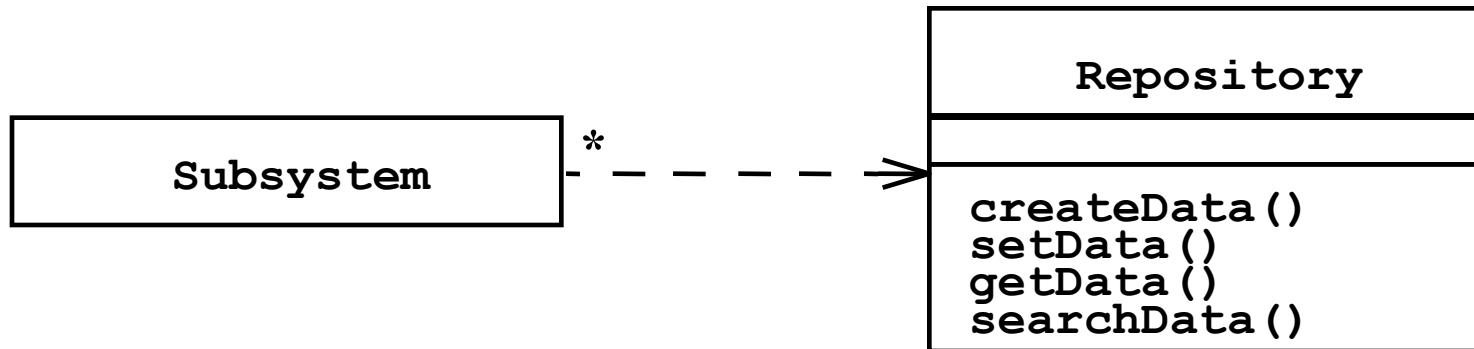
**Processor 2**

# Middleware Allows Focus On Higher Layers



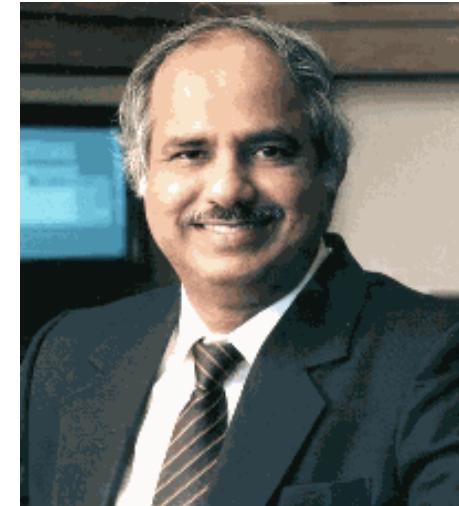
# Repository Architectural Style

- Subsystems access and modify data from a single data structure called the *repository*
- Historically called *blackboard architecture* (Erman, Hayes-Roth and Reddy 1980)
- Subsystems are loosely coupled (interact only through the repository)
- Control flow is dictated by the repository through triggers or by the subsystems through locks and synchronization primitives



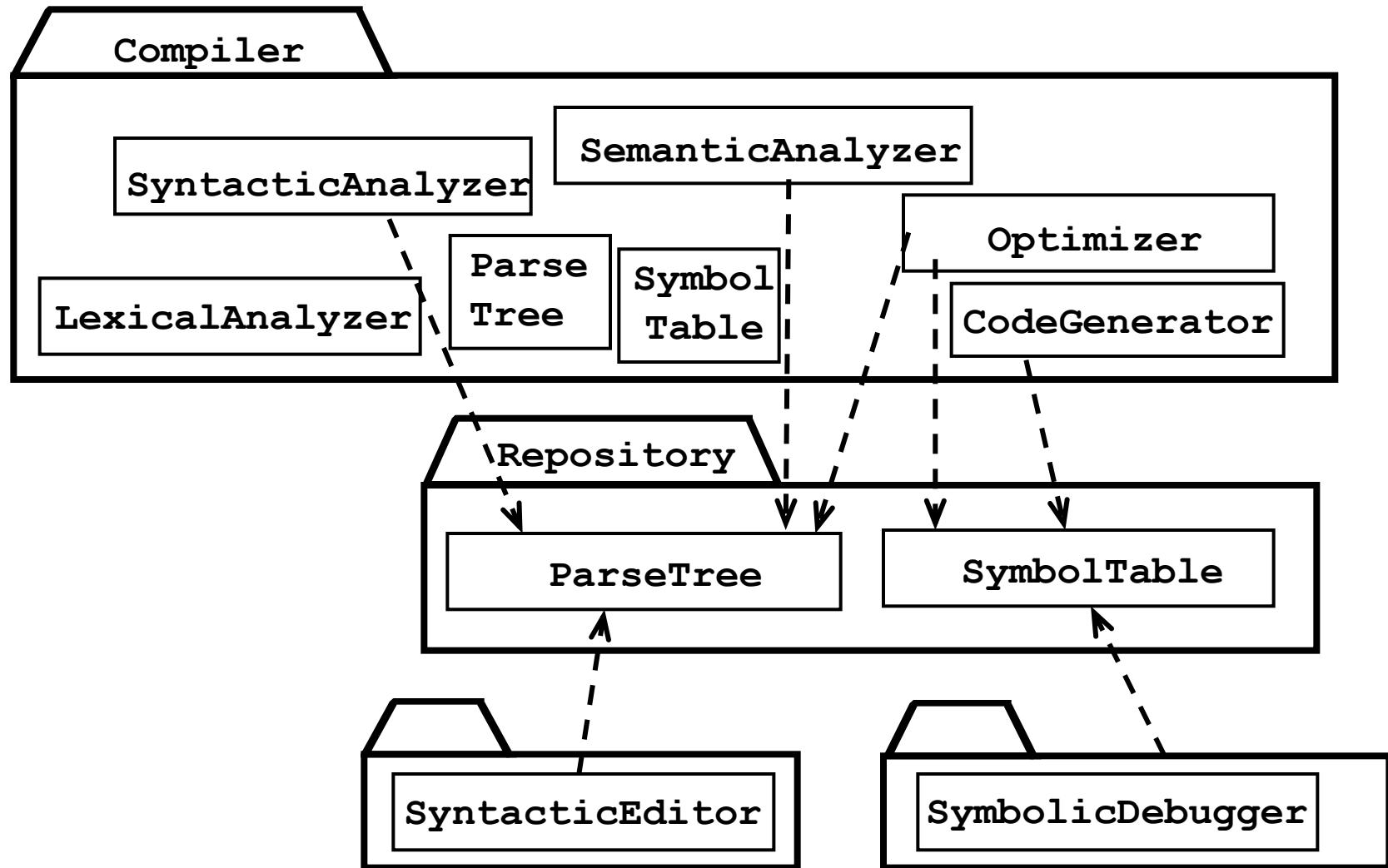
# Blackboard Subsystem Decomposition

- A *blackboard-system* consists of three major components
  - The **blackboard**. A shared repository of problems, partial solutions and new information.
  - The **knowledge sources** (KSs). Each knowledge source embodies specific expertise. It reads the information placed on the blackboard and places new information on the blackboard.
  - The **control shell**. It controls the flow of problem-solving activity in the system, in particular how the knowledge sources get notified of new information put into the blackboard.



Raj Reddy, \*1937, AI pioneer  
- Major contributions to speech, vision, robotics, e.g. Hearsay and Harpy  
- Founding Director of Robotics Institute, HCII, Center for Machine Learning, etc  
1994: Turing Award (with Ed Feigenbaum).

# Repository Architecture Example: Incremental Development Environment (IDE)



# Providing Consistent Views

- **Problem:** In systems with high coupling changes to the user interface (boundary objects) often force changes to the entity objects (data)
  - The user interface cannot be reimplemented without changing the representation of the entity objects
  - The entity objects cannot be reorganized without changing the user interface
- **Solution: Decoupling!** The model-view-controller architectural style decouples data access (entity objects) and data presentation (boundary objects)
  - The Data Presentation subsystem is called the **View**
  - The Data Access subsystem is called the **Model**
  - The **Controller** subsystem mediates between View (data presentation) and Model (data access)
- Often called **MVC**.

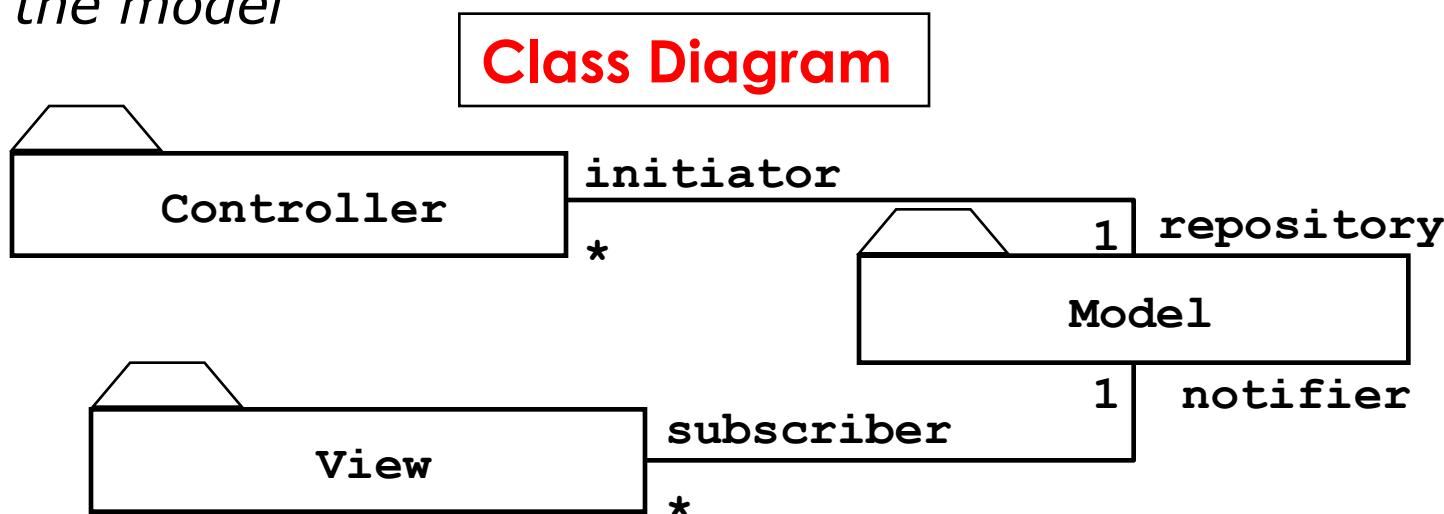
# Model-View-Controller Architectural Style

- Subsystems are classified into 3 different types

*Model subsystem:* Responsible for application domain knowledge

*View subsystem:* Responsible for displaying application domain objects to the user

*Controller subsystem:* Responsible for sequence of interactions with the user and notifying views of changes in the model



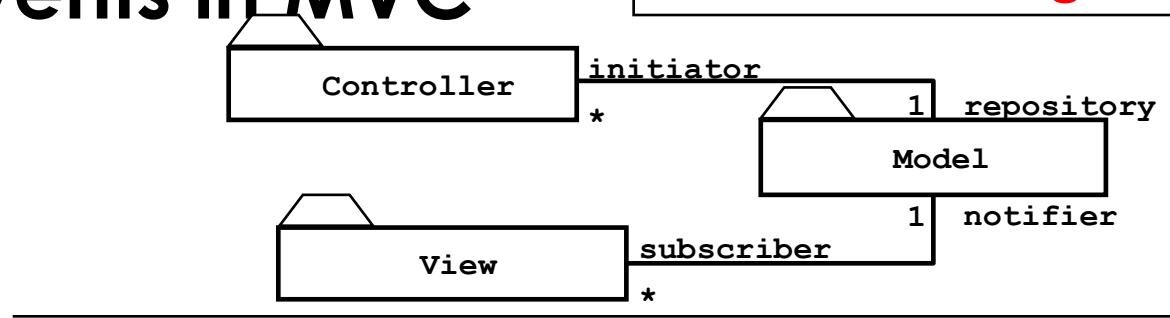
**Better understanding with a Collaboration Diagram**

# UML Collaboration Diagram

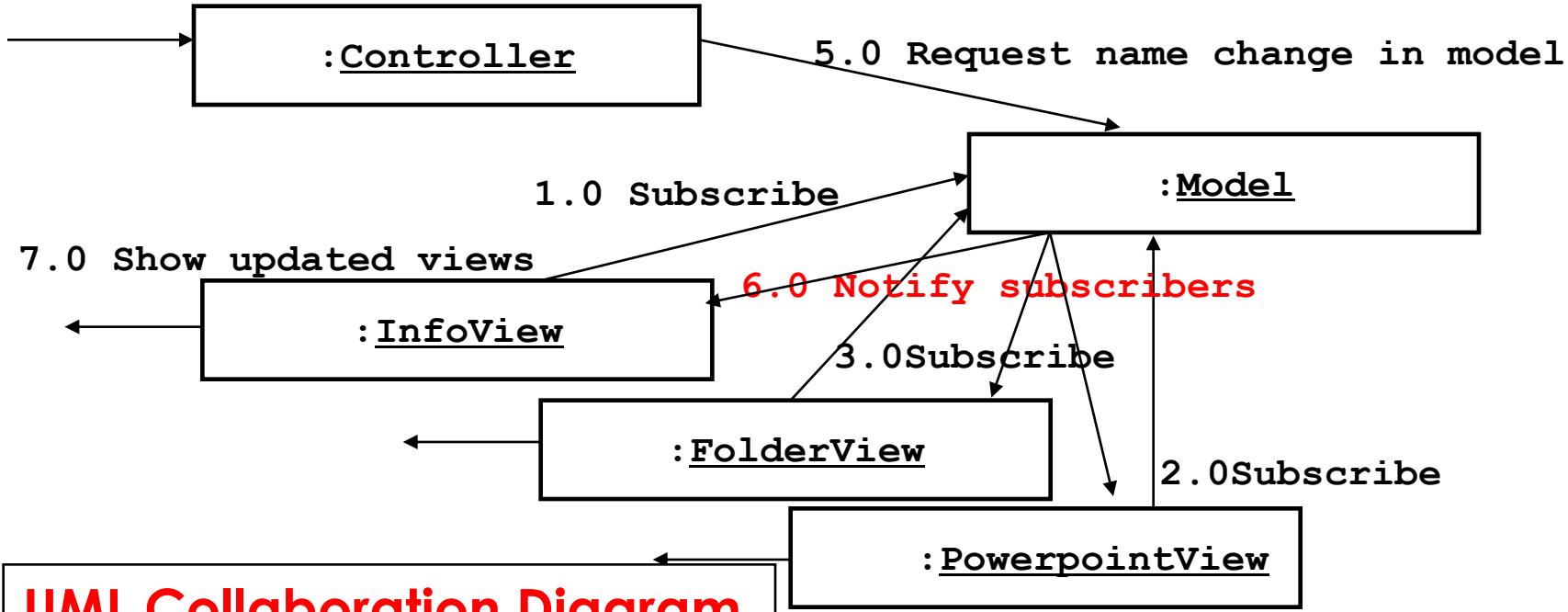
- A **Collaboration Diagram** is an instance diagram that visualizes the interactions between objects as a flow of messages. Messages can be events or calls to operations
- Communication diagrams **describe the static structure as well as the dynamic behavior of a system:**
  - The static structure is obtained from the UML class diagram
    - Collaboration diagrams reuse the layout of classes and associations in the class diagram
  - The dynamic behavior is obtained from the dynamic model (UML sequence diagrams and UML statechart diagrams)
    - Messages between objects are labeled with a chronological number and placed near the link the message is sent over
- Reading a collaboration diagram involves starting at message 1.0, and following the messages from object to object.

# Example: Modeling the Sequence of Events in MVC

## UML Class Diagram



4.0 User types new filename



## UML Collaboration Diagram

# 3-Layer-Architectural Style

## 3-Tier Architecture

### Definition: 3-Layer Architectural Style

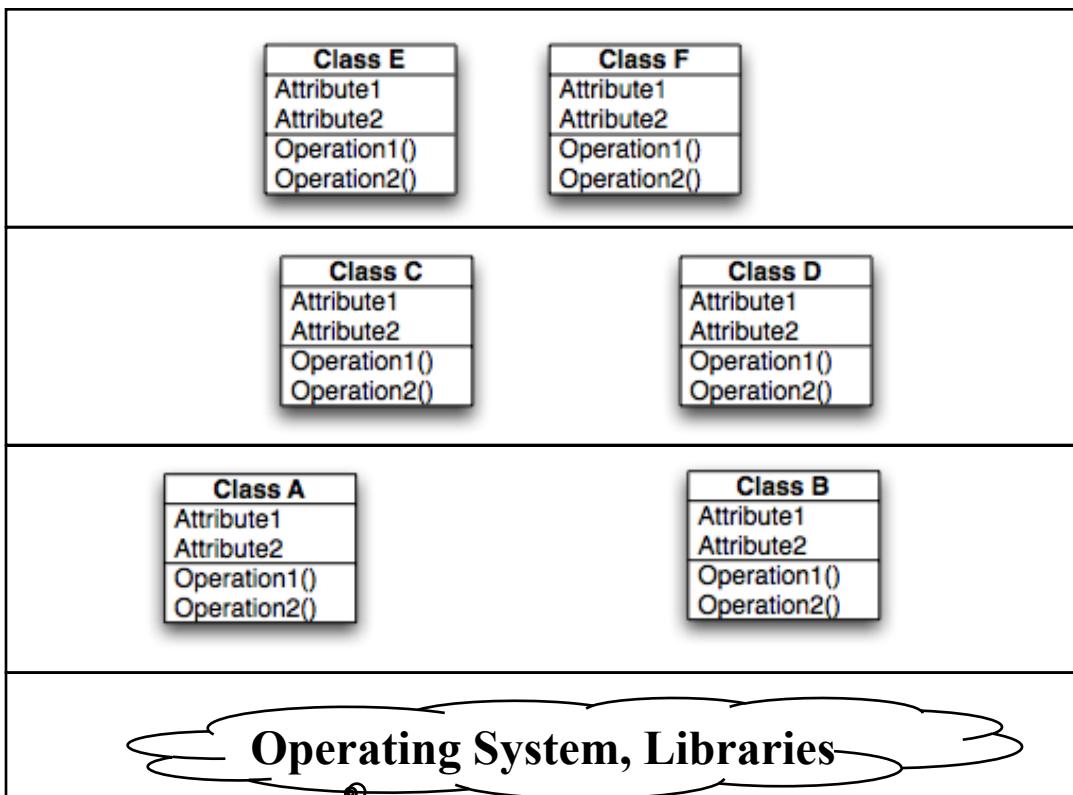
- An architectural style, where an application consists of 3 hierarchically ordered subsystems
  - A user interface, middleware and a database system
  - The middleware subsystem services data requests between the user interface and the database subsystem

### Definition: 3-Tier Architecture

- A software architecture where the 3 layers are allocated on 3 separate hardware nodes
- Note: **Layer** is a type (e.g. class, subsystem) and **Tier** is an instance (e.g. object, hardware node)
- Layer and Tier are often used interchangeably.

# Virtual Machines in 3-Layer Architectural Style

*A 3-Layer Architectural Style is a hierarchy of 3 virtual machines usually called presentation, application and data layer*



Presentation Layer  
(Client Layer)

Application Layer  
(Middleware,  
Business Logic)

Data Layer

Existing System

# Example of a 3-Layer Architectural Style

- Three-Layer architectural style are often used for the development of Websites:
  1. The **Web Browser** implements the user interface
  2. The **Web Server** serves requests from the web browser
  3. The **Database** manages and provides access to the persistent data.

# Example of a 4-Layer Architectural Style

4-Layer-architectural styles (4-Tier Architectures) are usually used for the development of electronic commerce sites. The layers are

1. The **Web Browser**, providing the user interface
2. A **Web Server**, serving static HTML requests
3. An **Application Server**, providing session management (for example the contents of an electronic shopping cart) and processing of dynamic HTML requests
4. A back end **Database**, that manages and provides access to the persistent data
  - In current 4-tier architectures, this is usually a relational Database management system (RDBMS).

# MVC vs. 3-Tier Architectural Style

- The **MVC** architectural style is **nonhierarchical** (triangular):
  - View subsystem sends updates to the Controller subsystem
  - Controller subsystem updates the Model subsystem
  - View subsystem is updated directly from the Model subsystem
- The **3-tier** architectural style is **hierarchical** (linear):
  - The presentation layer never communicates directly with the data layer (opaque architecture)
  - All communication must pass through the middleware layer
- **History:**
  - MVC (1970-1980): Originated during the development of modular graphical applications for a single graphical workstation at Xerox Parc
  - 3-Tier (1990s): Originated with the appearance of Web applications, where the client, middleware and data layers ran on physically separate platforms.

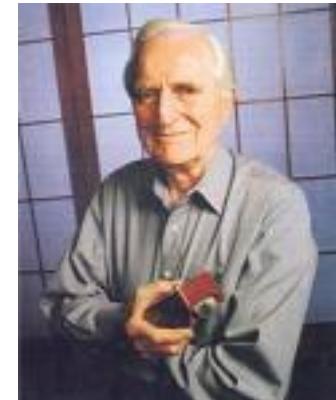
# History at Xerox Parc



## Xerox PARC (Palo Alto Research Center)

Founded in 1970 by Xerox, since 2002 a separate company PARC (wholly owned by Xerox). Best known for the invention of

- Laser printer (1973, Gary Starkweather)
- Ethernet (1973, Bob Metcalfe)
- Modern personal computer (1973, Alto, Bravo)
- Graphical user interface (GUI) based on WIMP
  - Windows, icons, menus and pointing device
    - Based on Doug Engelbart's invention of the mouse in 1965
- Object-oriented programming (Smalltalk, 1970s, Adele Goldberg)
- Ubiquitous computing (1990, Mark Weiser).



# Pipes and Filters

- A *pipeline* consists of a chain of processing elements (processes, threads, etc.), arranged so that the output of one element is the input to the next element
  - Usually some amount of buffering is provided between consecutive elements
  - The information that flows in these pipelines is often a stream of records, bytes or bits.

# Pipes and Filters Architectural Style

- An architectural style that consists of two subsystems called pipes and filters
  - **Filter:** A subsystem that does a processing step
  - **Pipe:** A Pipe is a connection between two processing steps
- Each filter has an input pipe and an output pipe.
  - The data from the input pipe are processed by the filter and then moved to the output pipe
- Example of a Pipes-and-Filters architecture: Unix
  - *Unix shell command: **ls -a** / **cat***



# Additional Readings

- *E.W. Dijkstra (1968)*
  - *The structure of the T.H.E Multiprogramming system, Communications of the ACM, 18(8), pp. 453-457*
- *D. Parnas (1972)*
  - *On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058*
- *L.D. Erman, F. Hayes-Roth (1980)*
  - *The Hearsay-II-Speech-Understanding System, ACM Computing Surveys, Vol 12. No. 2, pp 213-253*
- *J.D. Day and H. Zimmermann (1983)*
  - *The OSI Reference Model, Proc. IEEE, Vol.71, 1334-1340*
- *Jostein Gaarder (1991)*
  - *Sophie's World: A Novel about the History of Philosophy.*

# Summary

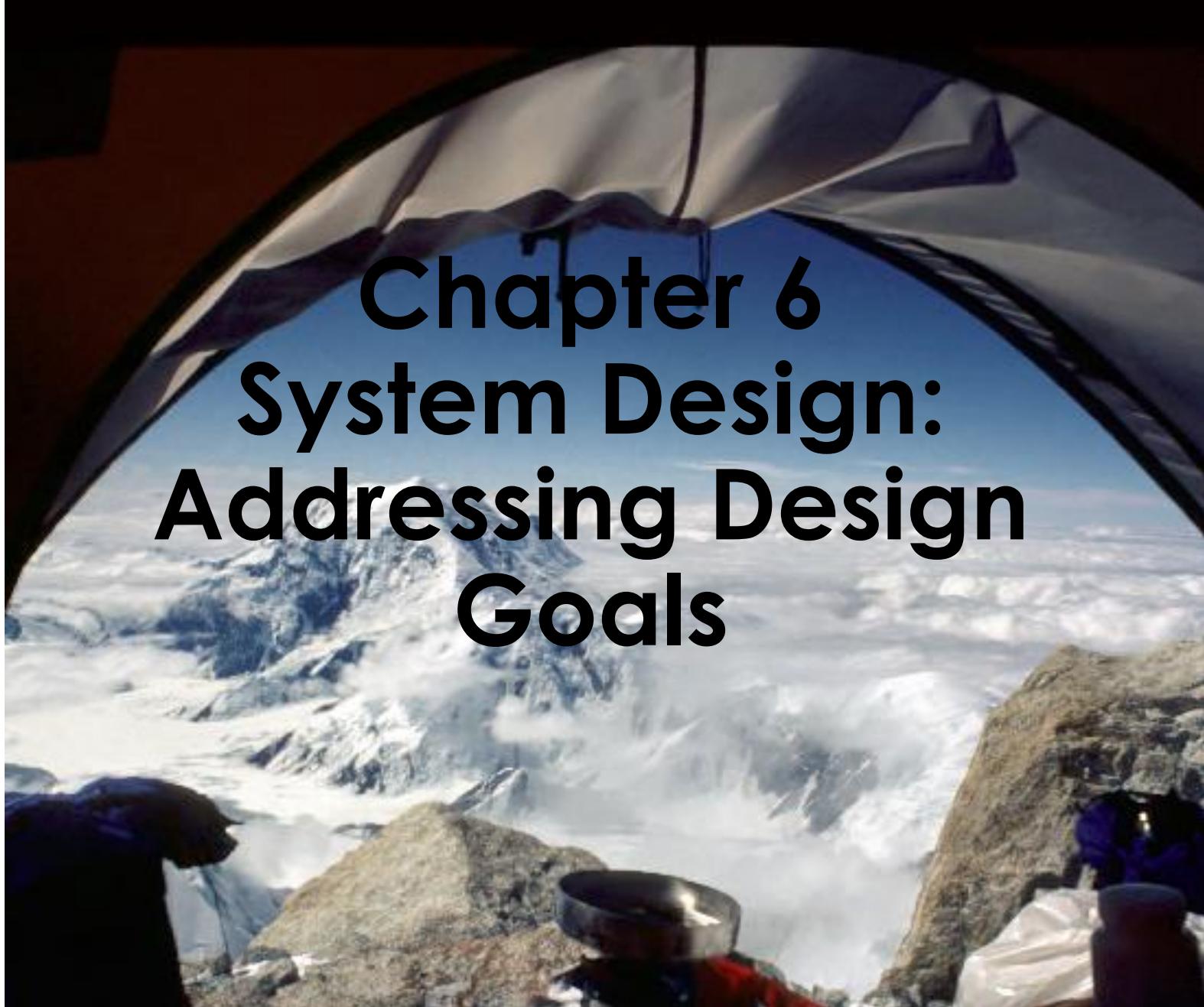
- *System Design*
  - *An activity that reduces the gap between the problem and an existing (virtual) machine*
- *Design Goals Definition*
  - *Describes the important system qualities*
  - *Defines the values against which options are evaluated*
- *Subsystem Decomposition*
  - *Decomposes the overall system into manageable parts by using the principles of cohesion and coherence*
- *Architectural Style*
  - *A pattern of a typical subsystem decomposition*
- *Software architecture*
  - *An instance of an architectural style*
  - *Client Server, Peer-to-Peer, Model-View-Controller.*

# Object-Oriented Software Engineering

Using UML, Patterns, and Java

## Chapter 6

# System Design: Addressing Design Goals



# Overview

## *System Design I*

- ✓ *0. Overview of System Design*
- ✓ *1. Design Goals*
- ✓ *2. Subsystem Decomposition*
  - ✓ *Architectural Styles*

## *System Design II*

- 3. Concurrency*
- 4. Hardware/Software Mapping*
- 5. Persistent Data Management*
- 6. Global Resource Handling and Access Control*
- 7. Software Control*
- 8. Boundary Conditions*

# System Design

## ✓ 1. Design Goals

Definition  
Trade-offs

## ✓ 2. Subsystem Decomposition

Layers vs Partitions  
Coherence/Coupling

## → 3. Concurrency

Identification of  
Threads

## 4. Hardware/ Software Mapping

Special Purpose  
Buy vs Build  
Allocation of Resources  
Connectivity

## 5. Data Management

Persistent Objects  
File system vs Database

## 6. Global Resource Handling

Access Control List  
vs Capabilities  
Security

## 8. Boundary Conditions

Initialization  
Termination  
Failure

## 7. Software Control

Monolithic  
Event-Driven  
Conc. Processes

# Concurrency

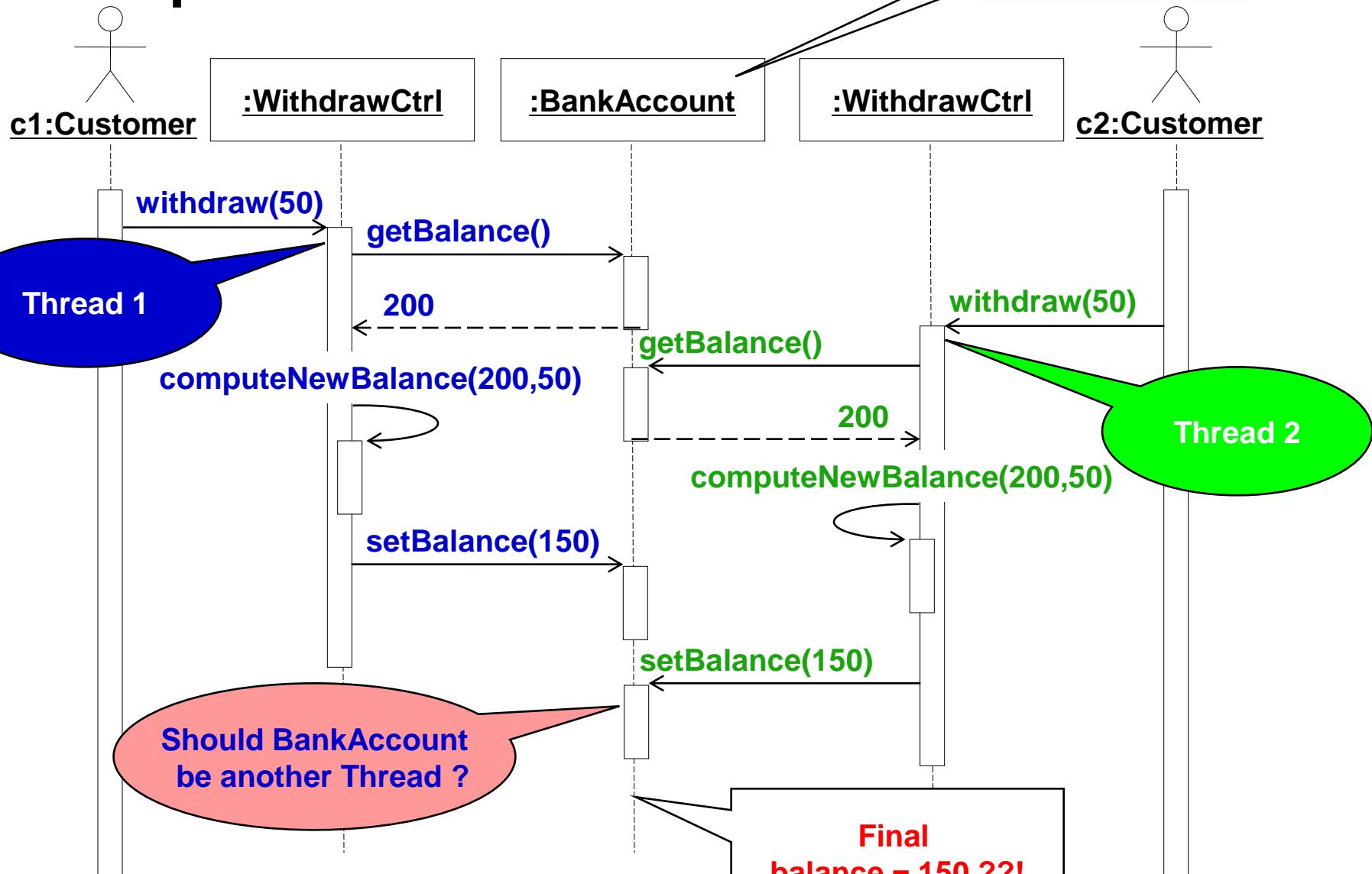
- *Nonfunctional Requirements to be addressed: Performance, Response time, latency, availability.*
- *Two objects are **inherently concurrent** if they can receive events at the same time without interacting*
  - *Source for identification: Objects in a sequence diagram that can simultaneously receive events*
    - *Unrelated events, instances of the same event*
- *Inherently concurrent objects can be assigned to different threads of control*
- *Objects with **mutual exclusive activity** could be folded into a single thread of control*

# Thread of Control

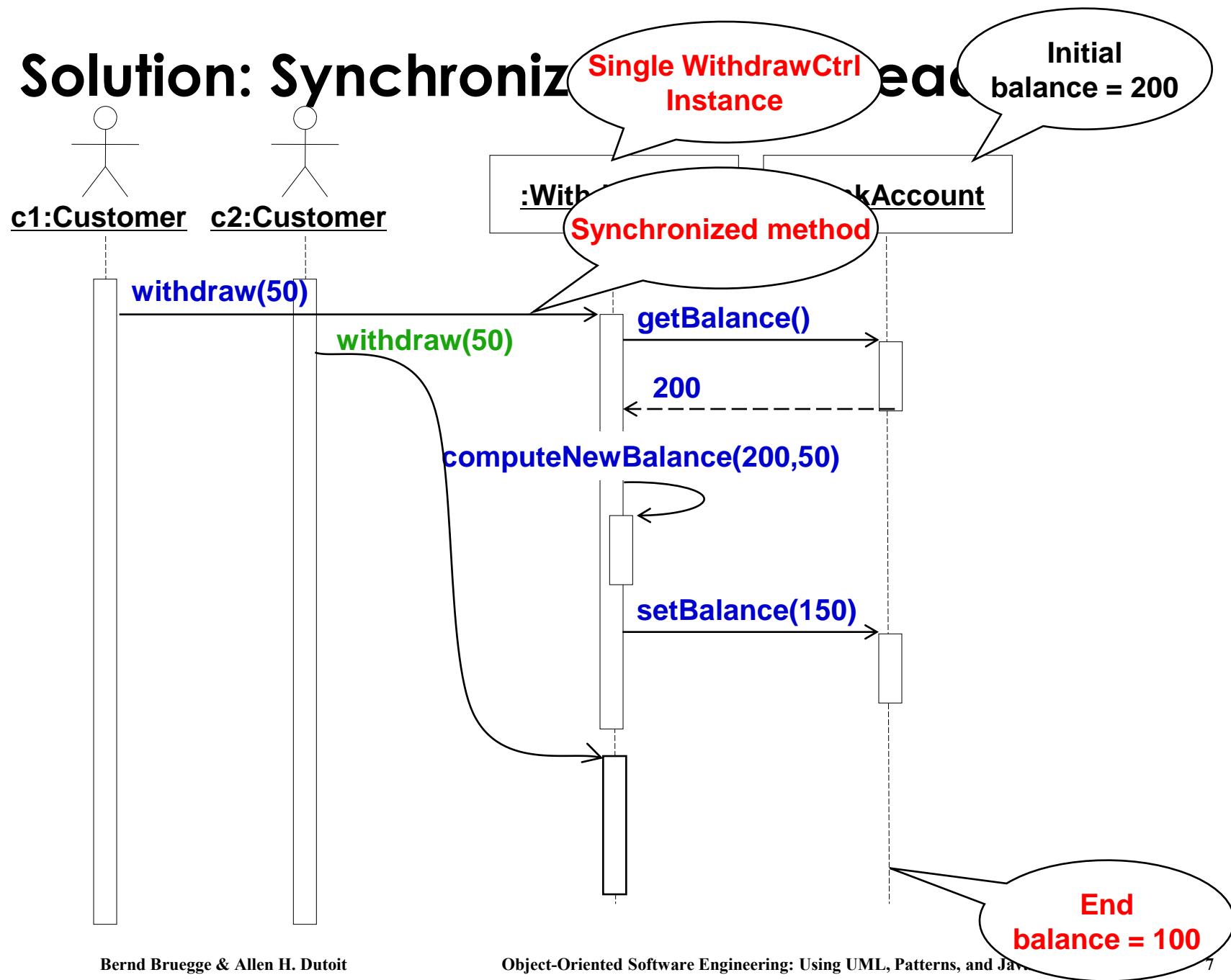
- A *thread of control* is a path through a set of state diagrams on which a single object is active at a time
  - A *thread remains within a state diagram until an object sends an event to different object and waits for another event*
  - *Thread splitting:* Object does a non-blocking send of an event to another object.
- Concurrent threads can lead to race conditions.
- A *race condition* (also *race hazard*) is a design flaw where the output of a process depends on the specific sequence of other events.
  - The name originated in digital circuit design: Two signals racing each other to influence the output.

# Example: Problem with threads

Assume: Initial balance = 200



# Solution: Synchronization



# Concurrency Questions

- *To identify threads for concurrency we ask the following questions:*
  - *Does the system provide access to multiple users?*
  - *Which entity objects of the object model can be executed independently from each other?*
  - *What kinds of control objects are identifiable?*
  - *Can a single request to the system be decomposed into multiple requests? Can these requests be handled in parallel? (Example: a distributed query)*

# Implementing Concurrency

- *Concurrent systems can be implemented on any system that provides*
  - *Physical concurrency: Threads are provided by hardware or*
  - *Logical concurrency: Threads are provided by software*
- *Physical concurrency is provided by multiprocessors and computer networks*
- *Logical concurrency is provided by threads packages.*

# Implementing Concurrency (2)

- *In both cases, - physical concurrency as well as logical concurrency - we have to solve the scheduling of these threads:*
  - *Which thread runs when?*
- *Today's operating systems provide a variety of scheduling mechanisms:*
  - *Round robin, time slicing, collaborating processes, interrupt handling*
- *General question addresses starvation, deadlocks, fairness -> Topic for researchers in operating systems*
- *Sometimes we have to solve the scheduling problem ourselves*
  - *Topic addressed by software control (system design topic 7).*

# System Design

## ✓ 1. Design Goals

Definition  
Trade-offs

## ✓ 2. Subsystem Decomposition

Layers vs Partitions  
Coherence/Coupling

## ✓ 3. Concurrency

Identification of  
Threads

## 4. Hardware/ Software Mapping

Special Purpose  
Buy vs Build  
Allocation of Resources  
Connectivity

## 8. Boundary Conditions

Initialization  
Termination  
Failure

## 7. Software Control

Monolithic  
Event-Driven  
Conc. Processes

## 5. Data Management

Persistent Objects  
Filesystem vs Database

## 6. Global Resource Handling

Access Control List  
vs Capabilities  
Security

# 4. Hardware Software Mapping

- *This system design activity addresses two questions:*
  - *How shall we realize the subsystems: With hardware or with software?*
  - *How do we map the object model onto the chosen hardware and/or software?*
    - *Mapping the Objects:*
      - Processor, Memory, Input/Output
    - *Mapping the Associations:*
      - Network connections

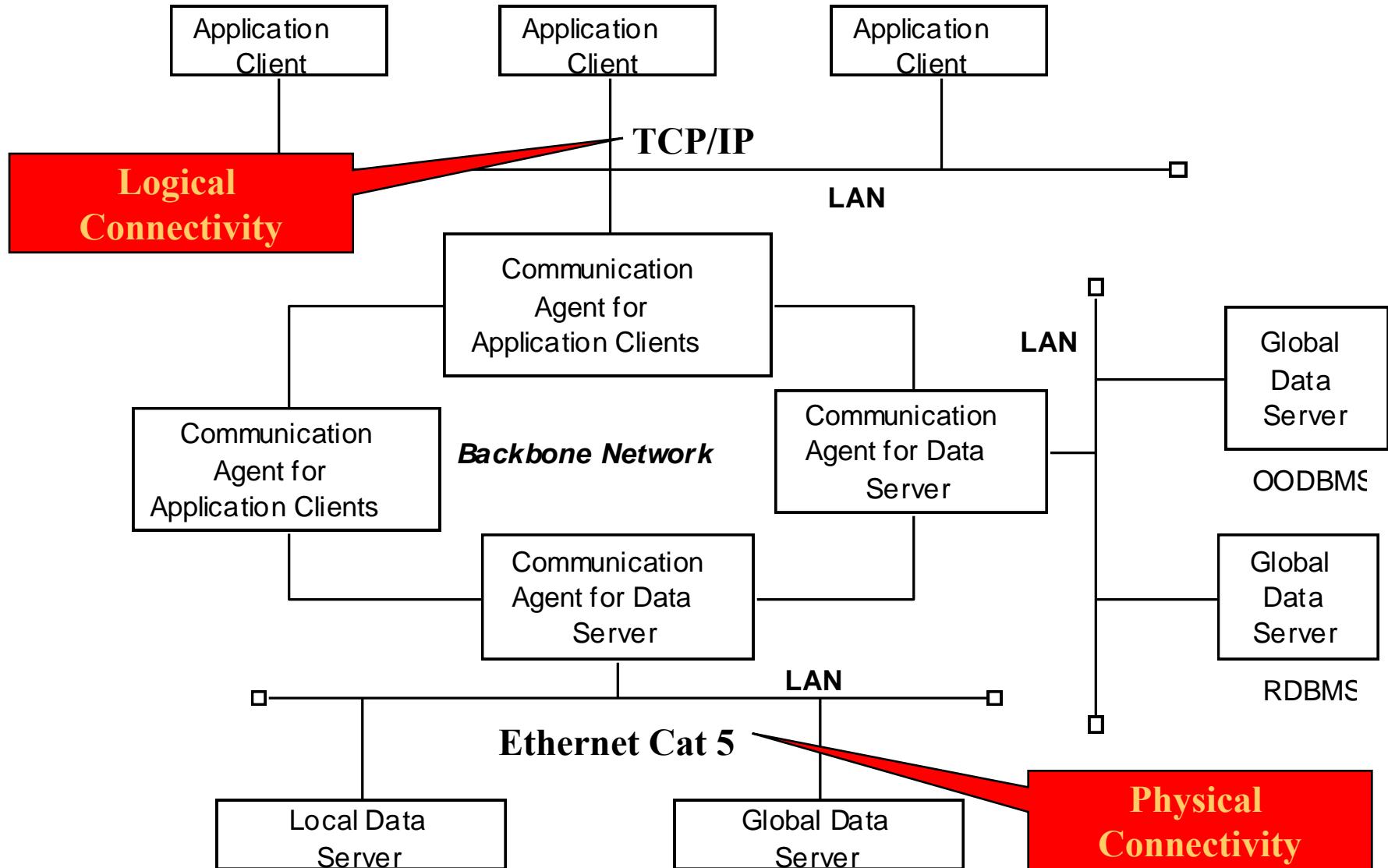
# Mapping Objects onto Hardware

- *Control Objects -> Processor*
  - *Is the computation rate too demanding for a single processor?*
  - *Can we get a speedup by distributing objects across several processors?*
  - *How many processors are required to maintain a steady state load?*
- *Entity Objects -> Memory*
  - *Is there enough memory to buffer bursts of requests?*
- *Boundary Objects -> Input/Output Devices*
  - *Do we need an extra piece of hardware to handle the data generation rates?*
  - *Can the desired response time be realized with the available communication bandwidth between subsystems?*

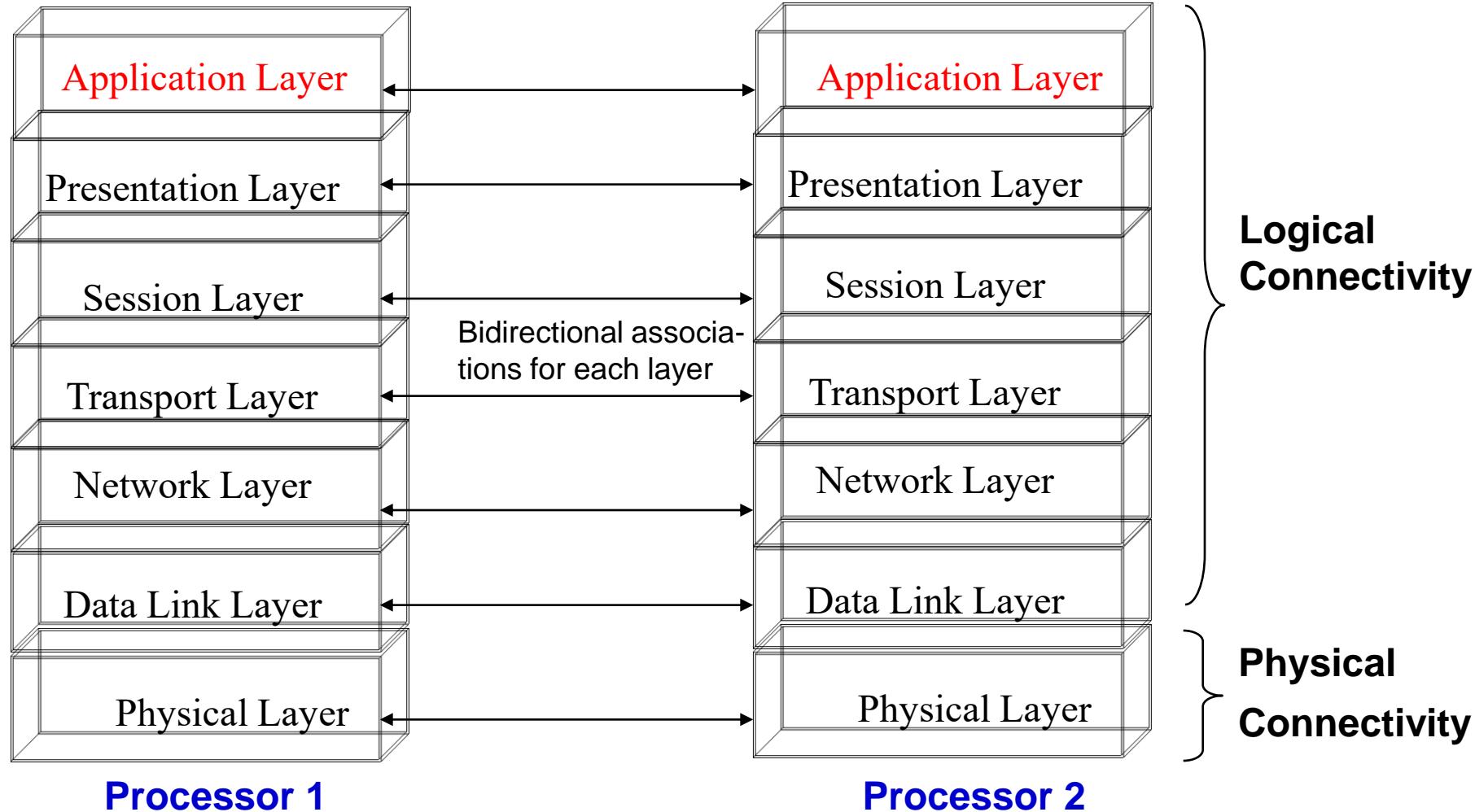
# Mapping the Associations: Connectivity

- *Describe the physical connectivity*
  - ("physical layer in the OSI Reference Model")
    - Describes which associations in the object model are mapped to physical connections.
- *Describe the logical connectivity (subsystem associations)*
  - Associations that do not directly map into physical connections.
  - In which layer should these associations be implemented?
- *Informal connectivity drawings often contain both types of connectivity*
  - Practiced by many developers, sometimes confusing.

# Example: Informal Connectivity Drawing



# Logical vs Physical Connectivity and the relationship to Subsystem Layering



Processor 1

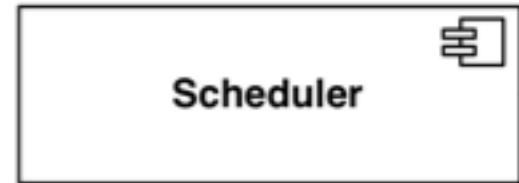
Processor 2

# Hardware-Software Mapping Difficulties

- *Much of the difficulty of designing a system comes from addressing externally-imposed hardware and software constraints*
  - *Certain tasks have to be at specific locations*
    - *Example: Withdrawing money from an ATM machine*
  - *Some hardware components have to be used from a specific manufacturer*
    - *Example: To send DVB-T signals, the system has to use components from a company that provides DVB-T transmitters.*

# Hardware/Software Mappings in UML

- A *UML component* is a building block of the system.  
It is represented as a rectangle with a tabbed rectangle symbol inside
- Components have different lifetimes:
  - Some exist only at design time
    - Classes, associations
  - Others exist until compile time
    - Source code, pointers
  - Some exist at link or only at runtime
    - Linkable libraries, executables, addresses
- The Hardware/Software Mapping addresses dependencies and distribution issues of UML components during system design.

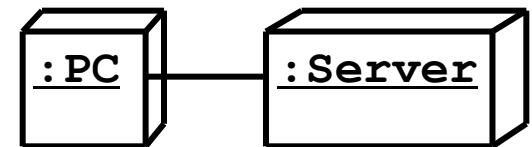


# Two New UML Diagram Types

- *Deployment Diagram:*
  - *Illustrates the distribution of components at run-time.*
  - *Deployment diagrams use nodes and connections to depict the physical resources in the system.*
- *Component Diagram:*
  - *Illustrates dependencies between components at design time, compilation time and runtime*

# Deployment Diagram

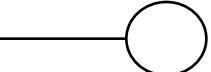
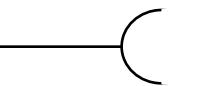
- Deployment diagrams are useful for showing a system design after these system design decisions have been made:
  - Subsystem decomposition
  - Concurrency
  - Hardware/Software Mapping
- A *deployment diagram* is a graph of nodes and connections ("communication associations")
  - Nodes are shown as 3-D boxes
  - Connections between nodes are shown as solid lines
  - Nodes may contain components
    - Components can be connected by "lollipops" and "grabbers"
    - Components may contain objects (indicating that the object is part of the component).



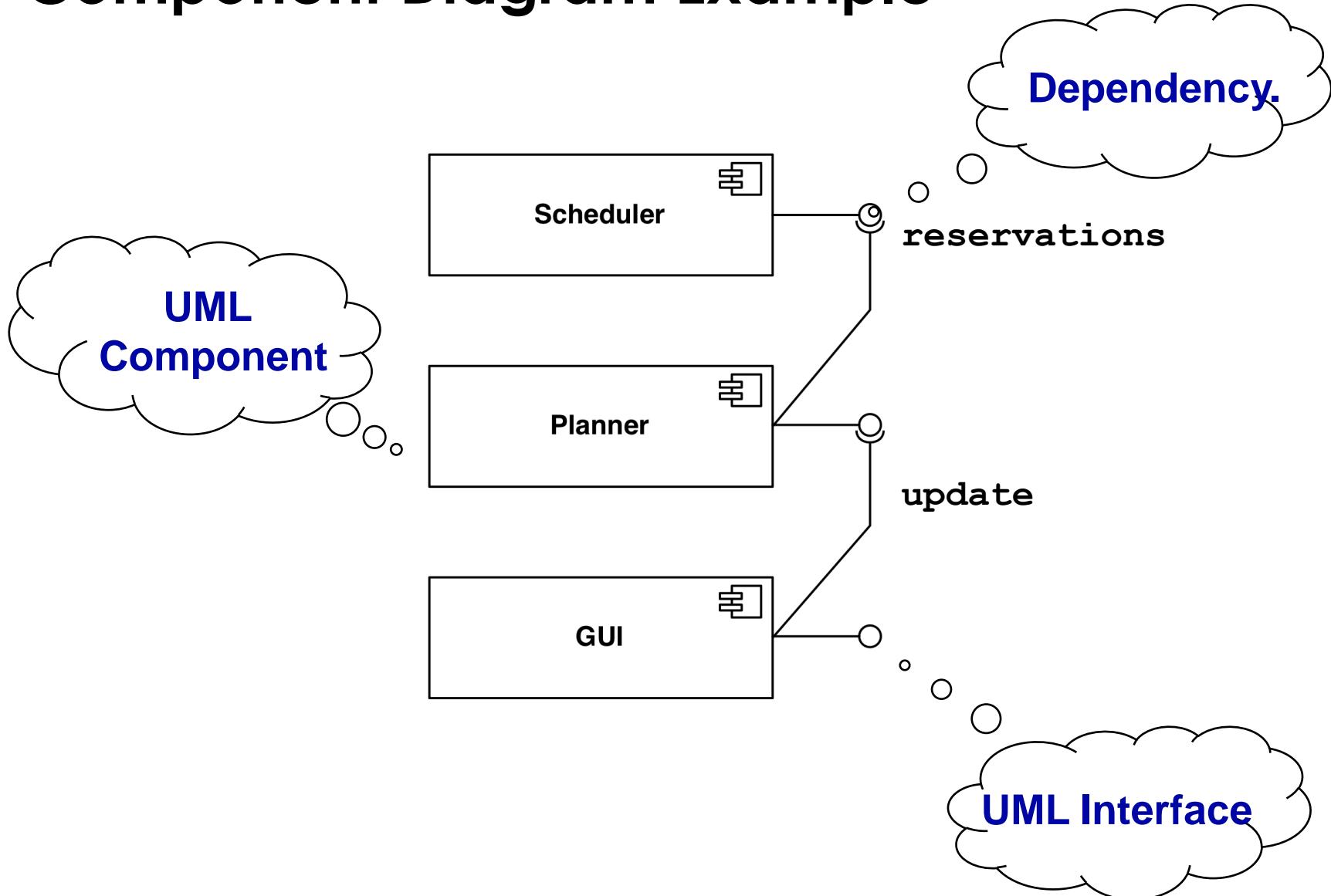
# UML Component Diagram

- *Used to model the top-level view of the system design in terms of components and dependencies among the components. Components can be*
  - *source code, linkable libraries, executables*
- *The dependencies (edges in the graph) are shown as dashed lines with arrows from the client component to the supplier component:*
  - *The lines are often also called connectors*
  - *The types of dependencies are implementation language specific*
- *Informally also called "software wiring diagram" because it show how the software components are wired together in the overall application.*

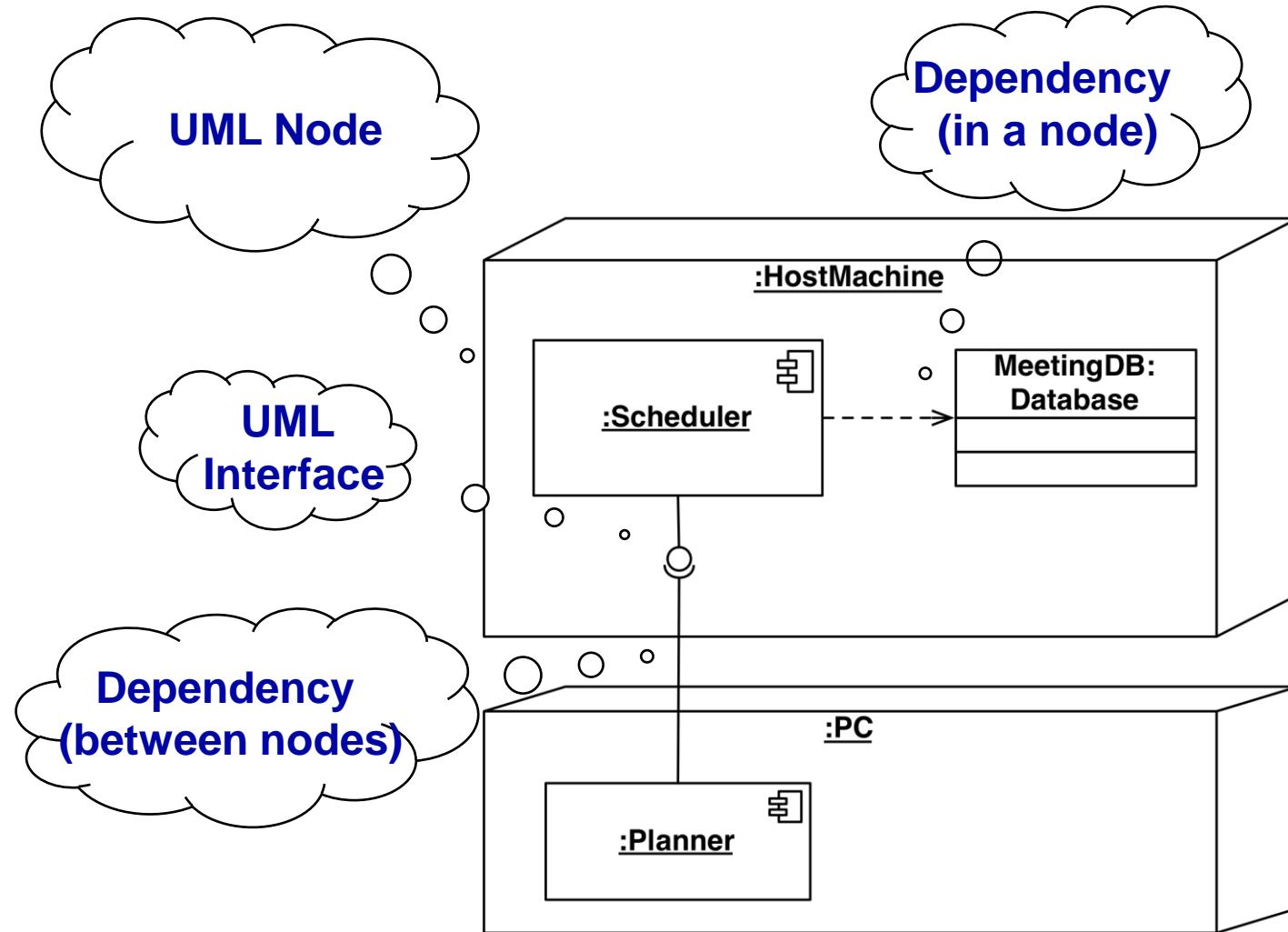
# UML Interfaces: Lollipops and Sockets

- A UML interface describes a group of operations used or created by UML components.
  - There are two types of interfaces: provided and required interfaces.
    - A *provided interface* is modeled using the lollipop notation 
    - A *required interface* is modeled using the socket notation 
- A port specifies a distinct interaction point between the component and its environment.
  - Ports are depicted as small squares on the sides of classifiers.

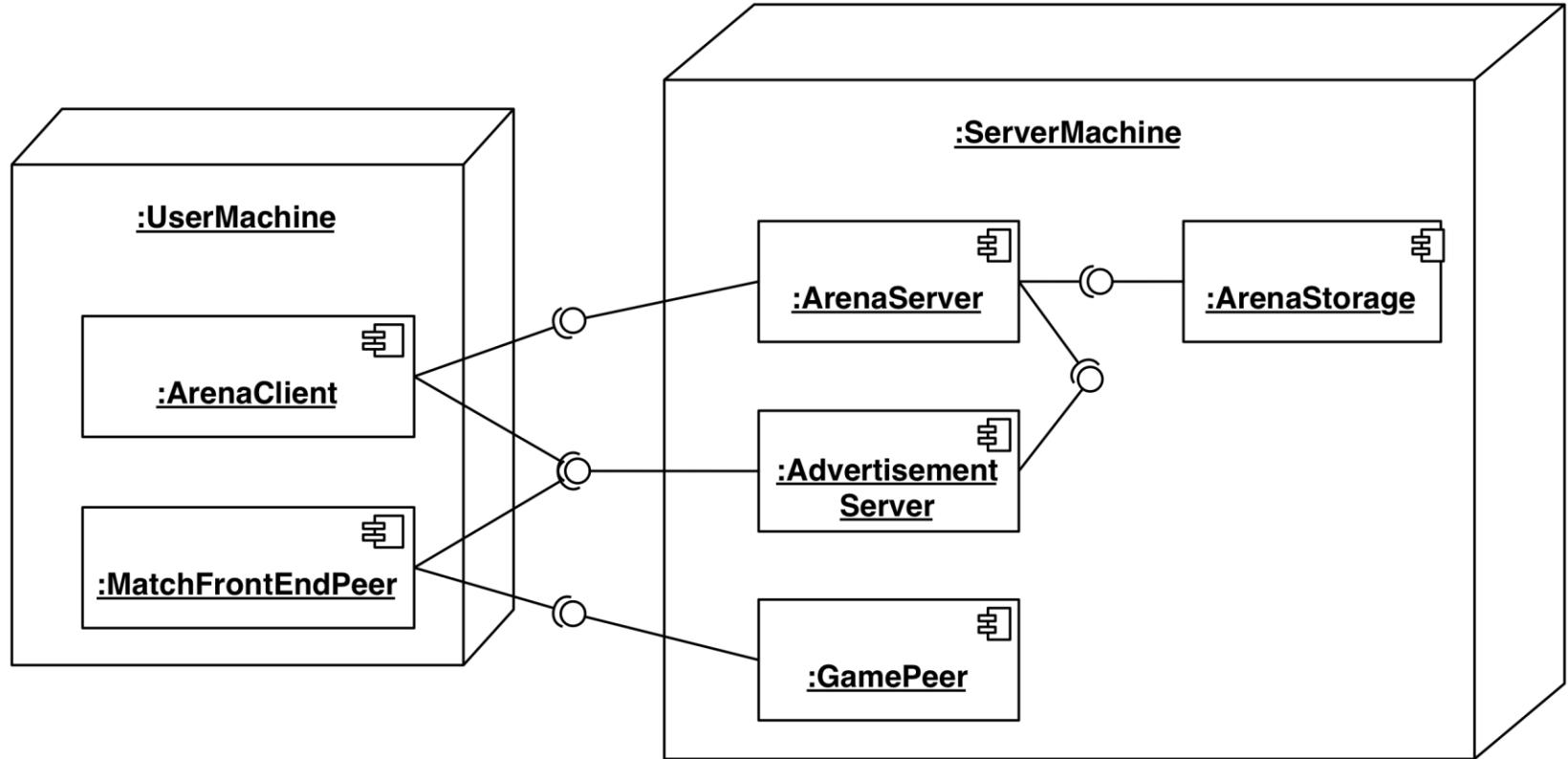
# Component Diagram Example



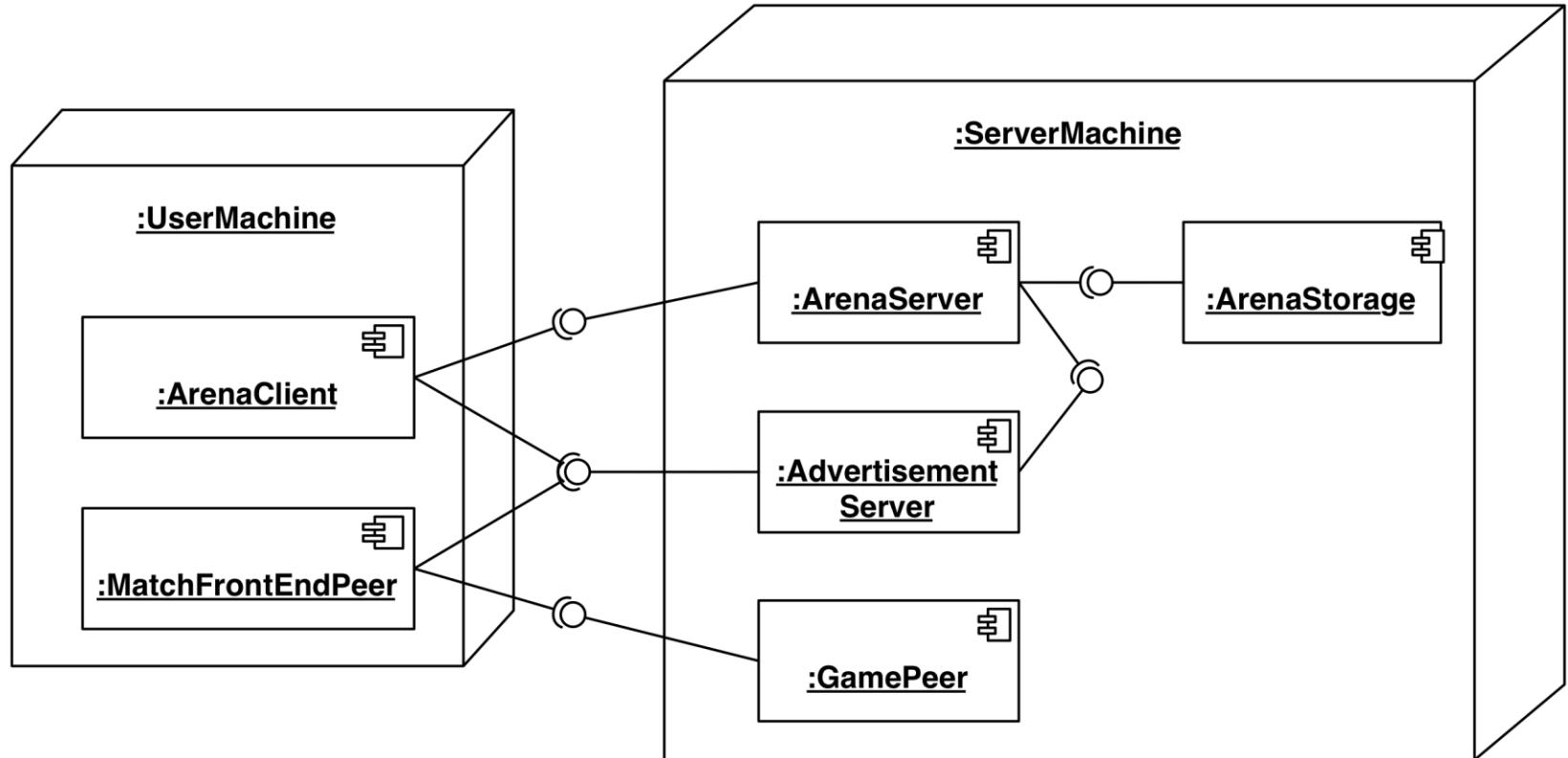
# Deployment Diagram Example



# ARENA Deployment Diagram



# Another ARENA Deployment Diagram



# 5. Data Management

- *Some objects in the system model need to be persistent:*
  - *Values for their attributes have a lifetime longer than a single execution*
- *A persistent object can be realized with one of the following mechanisms:*
  - *Filesystem:*
    - *If the data are used by multiple readers but a single writer*
  - *Database:*
    - *If the data are used by concurrent writers and readers.*

# Data Management Questions

- *How often is the database accessed?*
  - *What is the expected request (query) rate? The worst case?*
  - *What is the size of typical and worst case requests?*
- *Do the data need to be archived?*
- *Should the data be distributed?*
  - *Does the system design try to hide the location of the databases (location transparency)?*
- *Is there a need for a single interface to access the data?*
  - *What is the query format?*
- *Should the data format be extensible?*

# Mapping Object Models

- *UML object models can be mapped to relational databases*
- *The mapping:*
  - *Each class is mapped to its own table*
  - *Each class attribute is mapped to a column in the table*
  - *An instance of a class represents a row in the table*
  - *One-to-many associations are implemented with a buried foreign key*
  - *Many-to-many associations are mapped to their own tables*
- *Methods are not mapped*
- *More details in Lecture: Mapping Models to Relational Schema*

# 6. Global Resource Handling

- *Discusses access control*
- *Describes access rights for different classes of actors*
- *Describes how object guard against unauthorized access.*

# Defining Access Control

- *In multi-user systems different actors usually have different access rights to different functionality and data*
- *How do we model these accesses?*
  - *During analysis we model them by associating different use cases with different actors*
  - *During system design we model them determining which objects are shared among actors.*

# Access Matrix

- We model access on classes with an *access matrix*:
  - The rows of the matrix represents the actors of the system
  - The column represent classes whose access we want to control
- *Access Right*: An entry in the access matrix. It lists the operations that can be executed on instances of the class by the actor.

# Access Matrix Example

Diagram illustrating an Access Matrix Example:

The diagram shows a matrix where rows represent **Actors** (Operator, LeagueOwner, Player, Spectator) and columns represent **Classes** (Arena, League, Tournament, Match). A third dimension, **Access Rights**, is represented by the text in each cell.

**Actors** (Rows): Operator, LeagueOwner, Player, Spectator

**Classes** (Columns): Arena, League, Tournament, Match

**Access Rights** (Matrix Cells):

	Arena	League	Tournament	Match
<b>Operator</b>	<<create>> createUser() view ()	<<create>> archive()		
<b>LeagueOwner</b>	view ()	edit ()	<<create>> archive() schedule() view()	<<create>> end()
<b>Player</b>	view() applyForOwner()	view() subscribe()	applyFor() view()	play() forfeit()
<b>Spectator</b>	view() applyForPlayer()	view() subscribe()	view()	view() replay()

# Access Matrix Implementations

- *Global access table: Represents explicitly every cell in the matrix as a triple (actor, class, operation)*

**LeagueOwner**, **Arena**, **view()**

**LeagueOwner**, **League**, **edit()**

**LeagueOwner**, **Tournament**, <<create>>

**LeagueOwner**, **Tournament**, **view()**

**LeagueOwner**, **Tournament**, **schedule()**

**LeagueOwner**, **Tournament**, **archive()**

**LeagueOwner**, **Match**, <<create>>

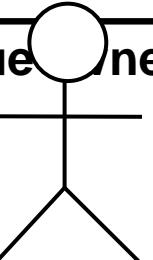
**LeagueOwner**, **Match**, **end()**

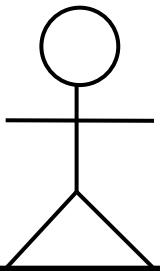
.

# Better Access Matrix Implementations

- *Access control list*
  - Associates a *list of (actor,operation) pairs* with each class to be accessed.
  - Every time an instance of this class is accessed, the access list is checked for the corresponding actor and operation.
- *Capability*
  - Associates a *(class,operation) pair* with an actor.
  - A capability provides an actor to gain control access to an object of the class described in the capability.

# Access Matrix Example

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
League Owner 	view ()	edit ()	<<create>> archive() schedule() view()	<<create>> end()
Player	view() applyForOwner()	view() subscribe()	applyFor() view()	play() forfeit()
Spectator	view() applyForPlayer()	view() subscribe()	view()	view() replay()

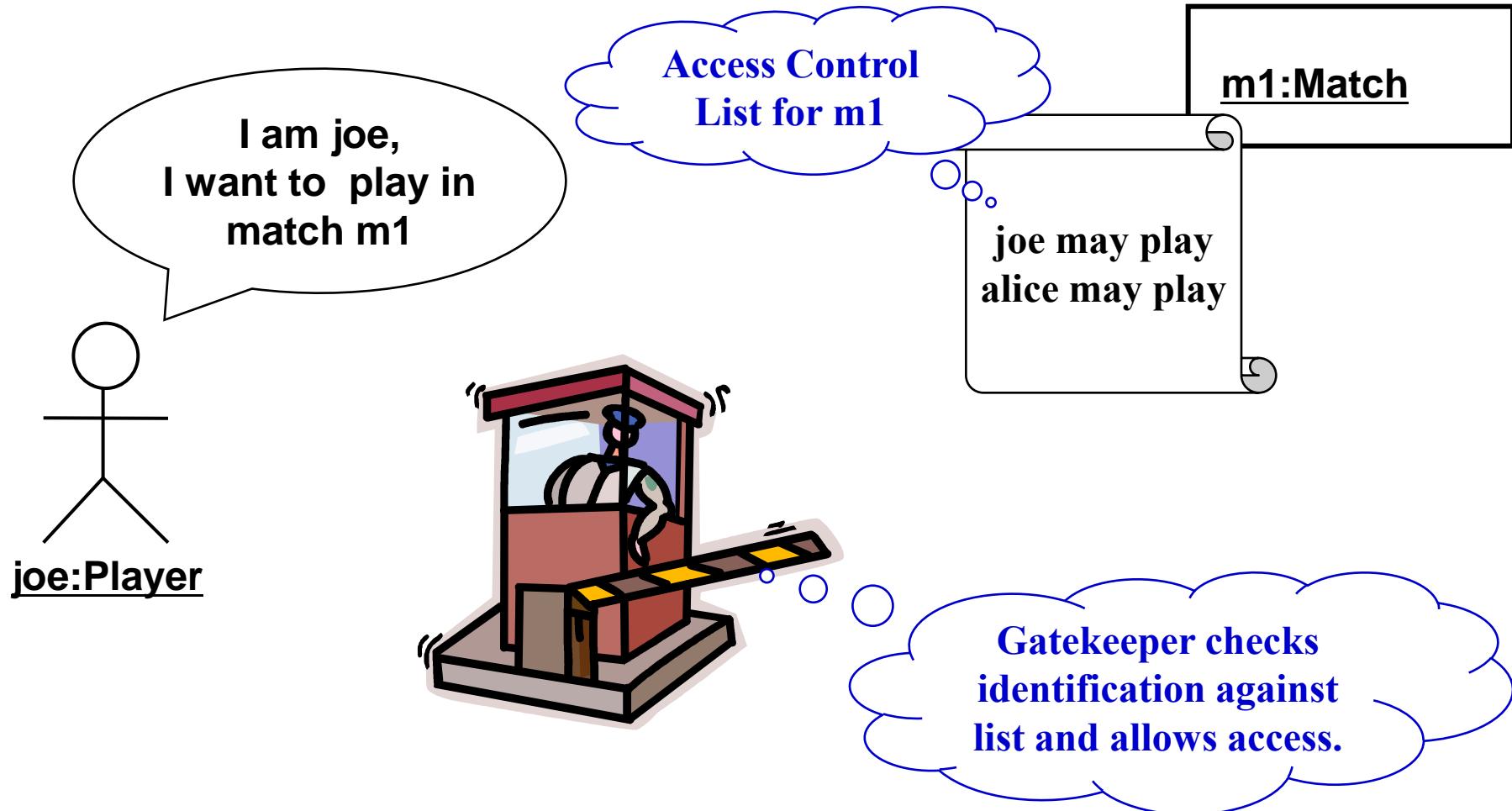


**Player**

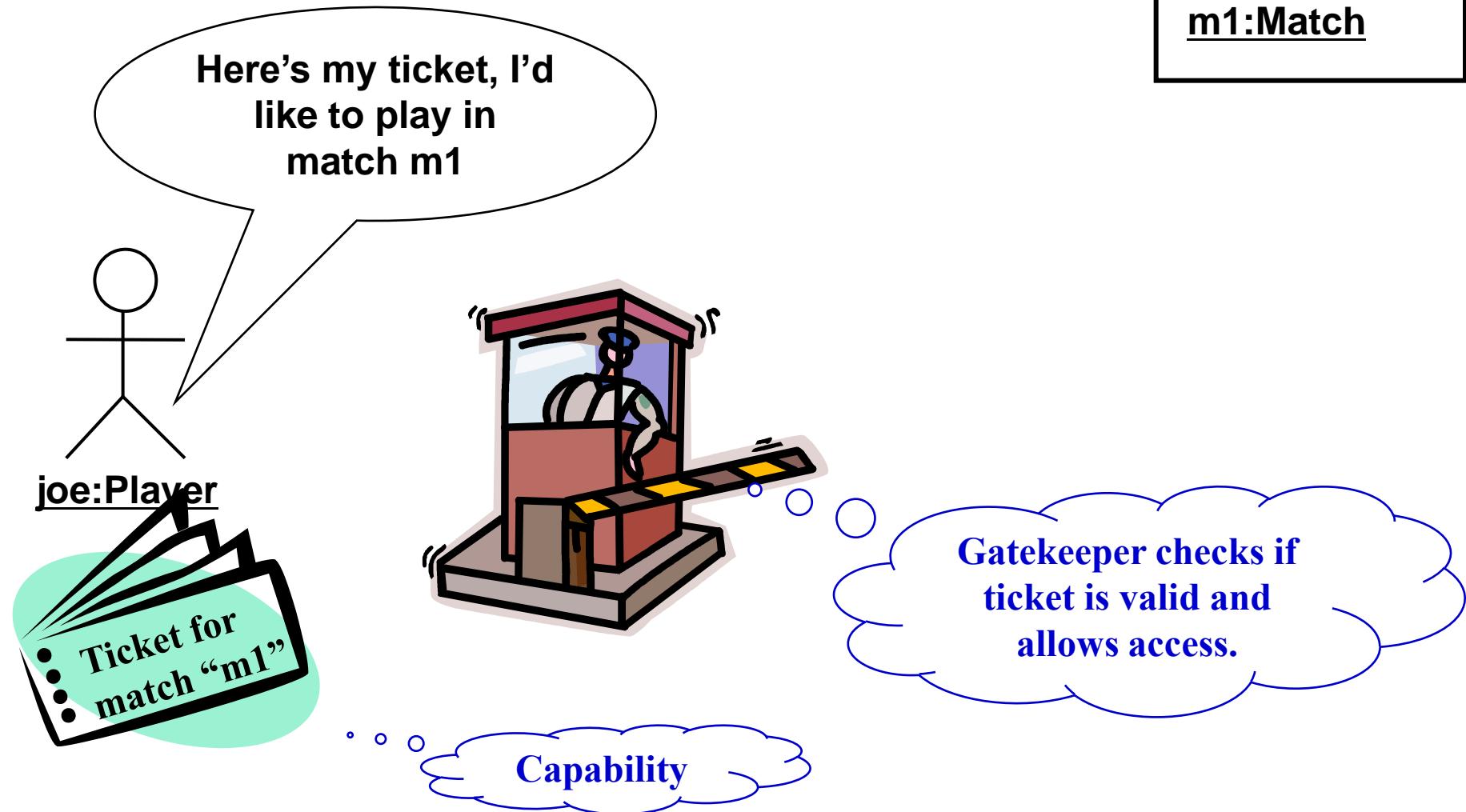
**Match**

`play()`  
`forfeit()`

# Access Control List Realization



# Capability Realization



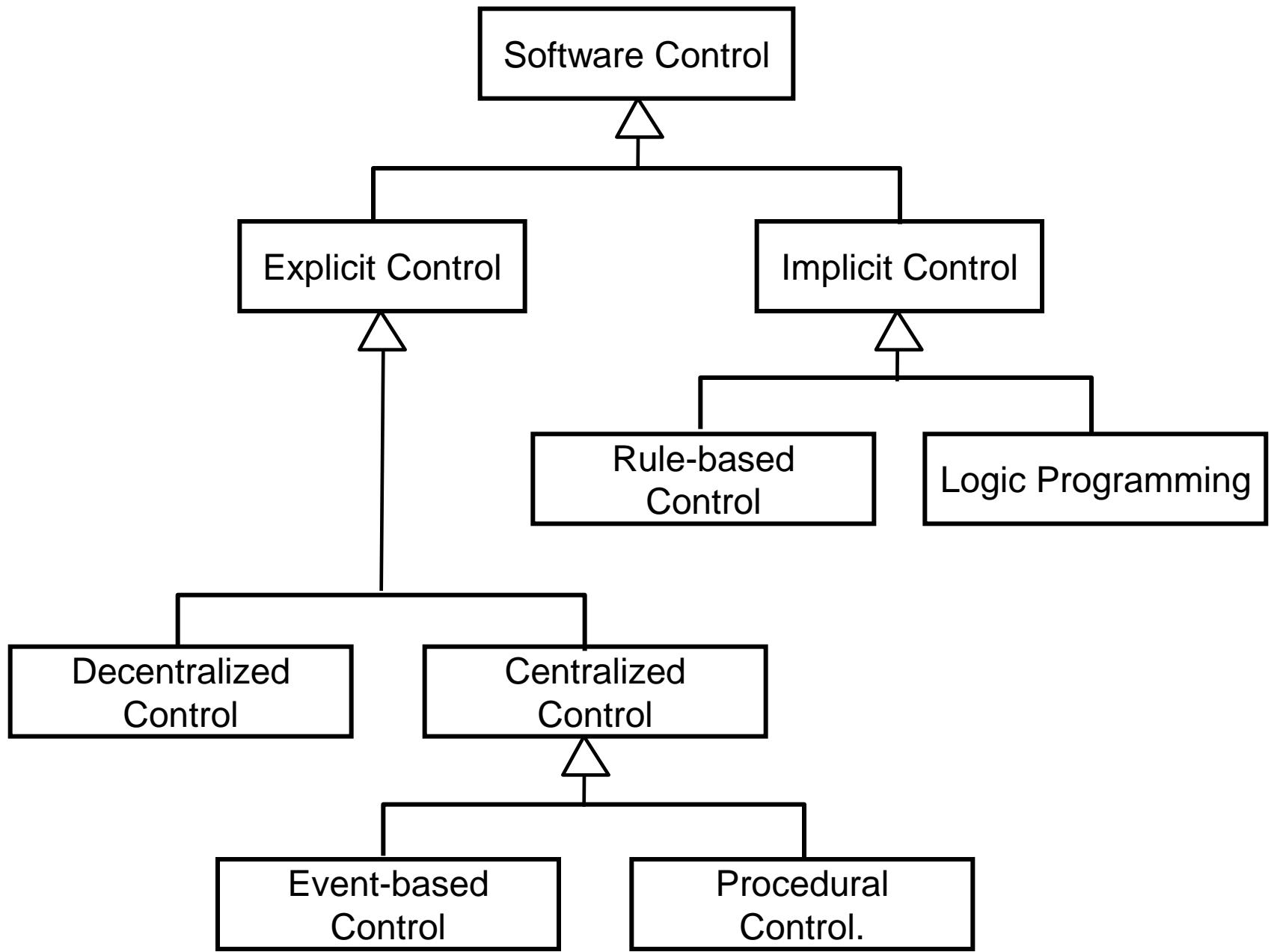
# Global Resource Questions

- *Does the system need authentication?*
- *If yes, what is the authentication scheme?*
  - User name and password? Access control list
  - Tickets? Capability-based
- *What is the user interface for authentication?*
- *Does the system need a network-wide name server?*
- *How is a service known to the rest of the system?*
  - At runtime? At compile time?
  - By Port?
  - By Name?

# 7. Decide on Software Control

*Two major design choices:*

1. *Choose implicit control*
2. *Choose explicit control*
  - Centralized or decentralized
  - **Centralized control:**
    - *Procedure-driven:* Control resides within program code.
    - *Event-driven:* Control resides within a dispatcher calling functions via callbacks.
  - **Decentralized control**
    - Control resides in several independent objects.
      - Examples: Message based system, RMI
    - Possible speedup by mapping the objects on different processors, increased communication overhead.



# Centralized vs. Decentralized Designs

- *Centralized Design*
  - *One control object or subsystem ("spider") controls everything*
    - *Pro: Change in the control structure is very easy*
    - *Con: The single control object is a possible performance bottleneck*
- *Decentralized Design*
  - *Not a single object is in control, control is distributed; That means, there is more than one control object*
    - *Con: The responsibility is spread out*
    - *Pro: Fits nicely into object-oriented development*

# Centralized vs. Decentralized Designs (2)

- *Should you use a centralized or decentralized design?*
- *Take the sequence diagrams and control objects from the analysis model*
- *Check the participation of the control objects in the sequence diagrams*
  - *If the sequence diagram looks like a fork => Centralized design*
  - *If the sequence diagram looks like a stair => Decentralized design.*

# 8. Boundary Conditions

- *Initialization*
  - *The system is brought from a non-initialized state to steady-state*
- *Termination*
  - *Resources are cleaned up and other systems are notified upon termination*
- *Failure*
  - *Possible failures: Bugs, errors, external problems*
  - *Good system design foresees fatal failures and provides mechanisms to deal with them.*

# Boundary Condition Questions

- *Initialization*
  - *What data need to be accessed at startup time?*
  - *What services have to registered?*
  - *What does the user interface do at start up time?*
- *Termination*
  - *Are single subsystems allowed to terminate?*
  - *Are subsystems notified if a single subsystem terminates?*
  - *How are updates communicated to the database?*
- *Failure*
  - *How does the system behave when a node or communication link fails?*
  - *How does the system recover from failure?.*

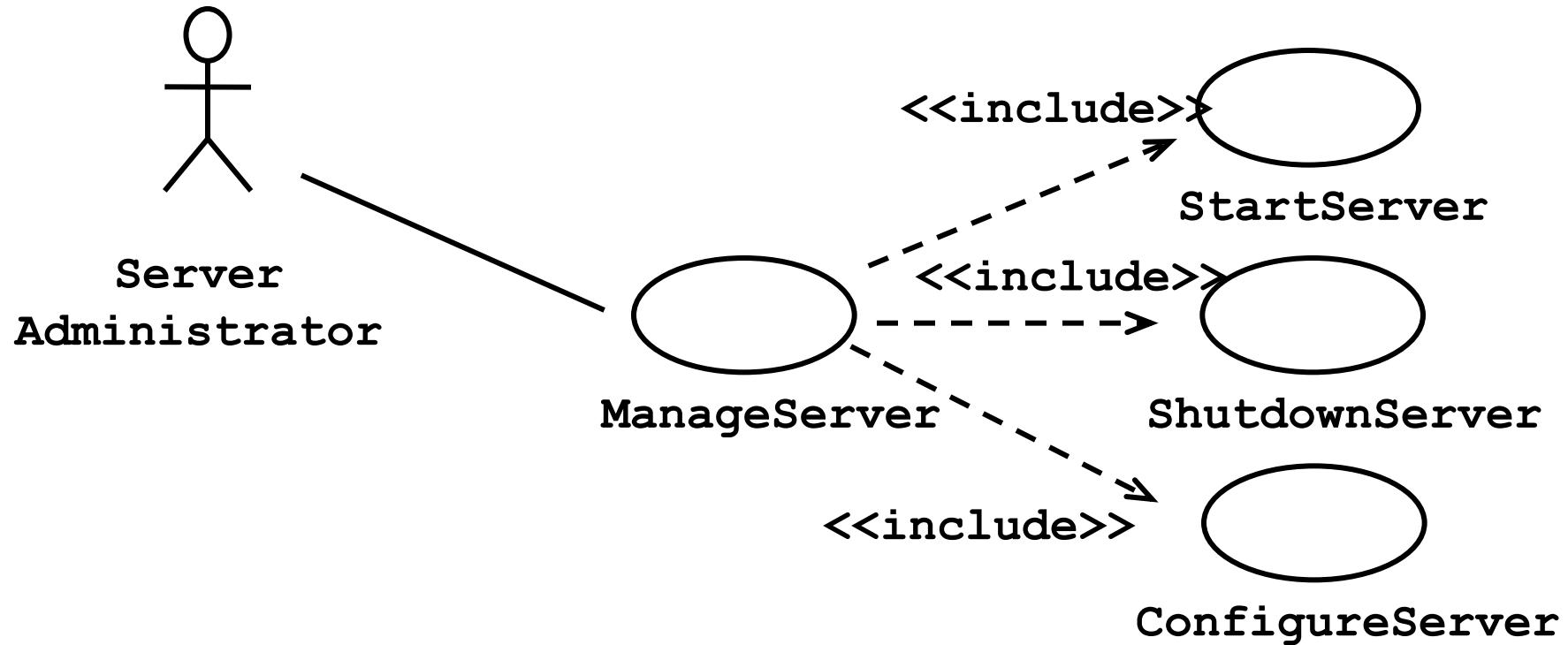
# Modeling Boundary Conditions

- *Boundary conditions are best modeled as use cases with actors and objects*
- We call them *boundary use cases or administrative use cases*
- *Actor: often the system administrator*
- *Interesting use cases:*
  - *Start up of a subsystem*
  - *Start up of the full system*
  - *Termination of a subsystem*
  - *Error in a subsystem or component, failure of a subsystem or component.*

# Example: Boundary Use Case for ARENA

- *Let us assume, we identified the subsystem AdvertisementServer during system design*
- *This server takes a big load during the holiday season*
- *During hardware software mapping we decide to dedicate a special node for this server*
- *For this node we define a new boundary use case ManageServer*
- *ManageServer includes all the functions necessary to start up and shutdown the AdvertisementServer.*

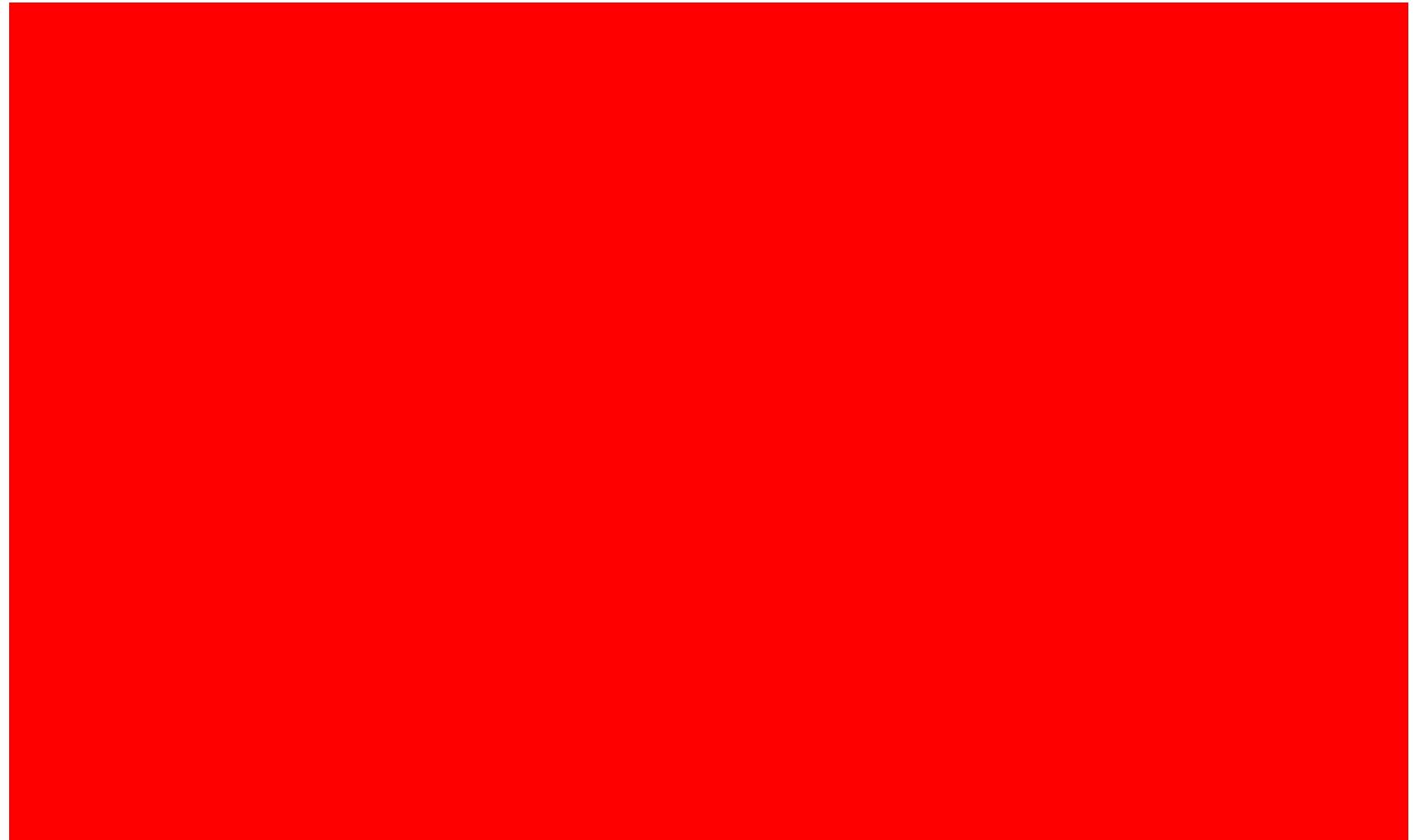
# ManageServer Boundary Use Case



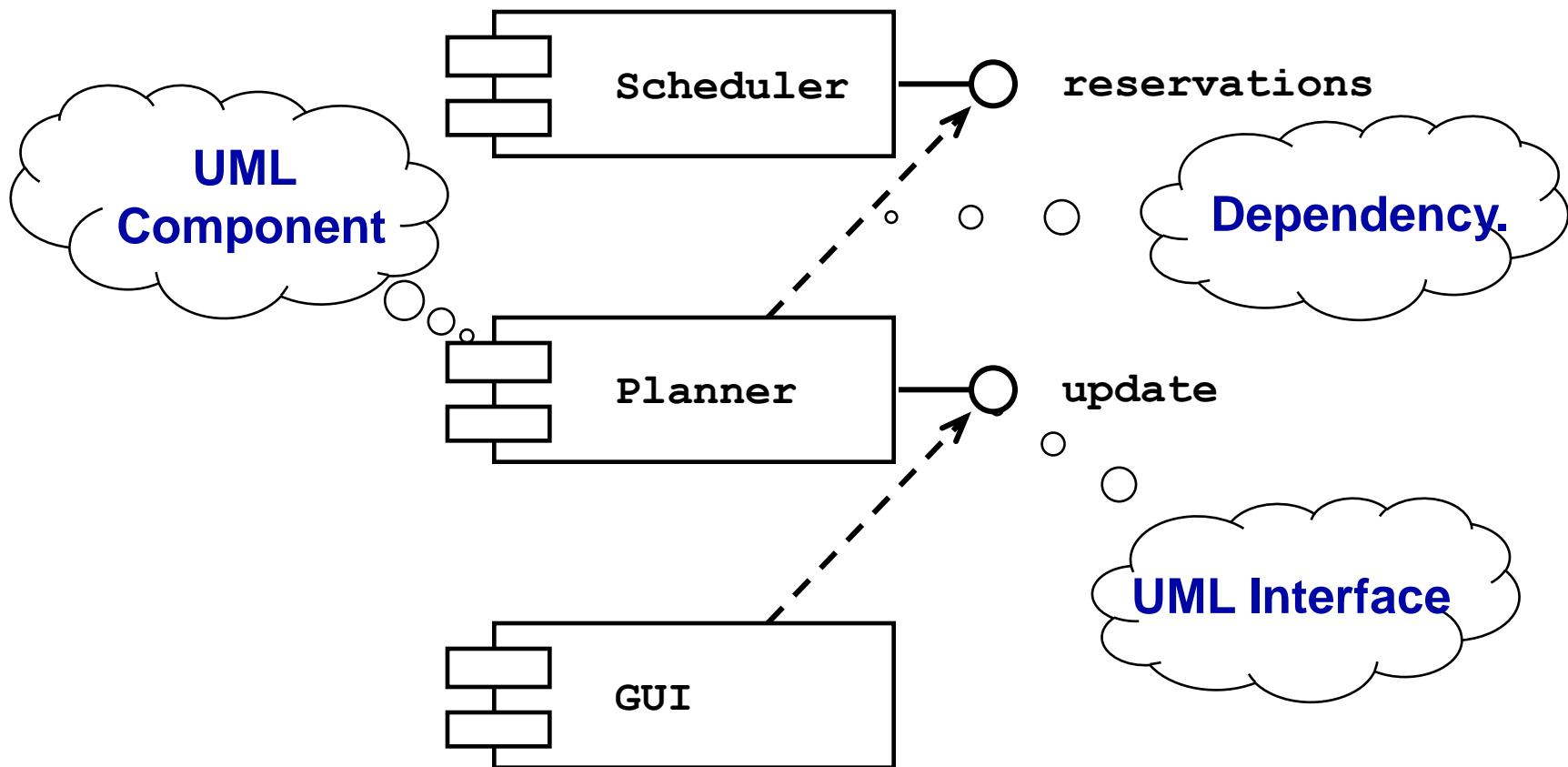
# Summary

- *System design activities:*
  - *Concurrency identification*
  - *Hardware/Software mapping*
  - *Persistent data management*
  - *Global resource handling*
  - *Software control selection*
  - *Boundary conditions*
- *Each of these activities may affect the subsystem decomposition*
- *Two new UML Notations*
  - *UML Component Diagram: Showing compile time and runtime dependencies between subsystems*
  - *UML Deployment Diagram: Drawing the runtime configuration of the system.*

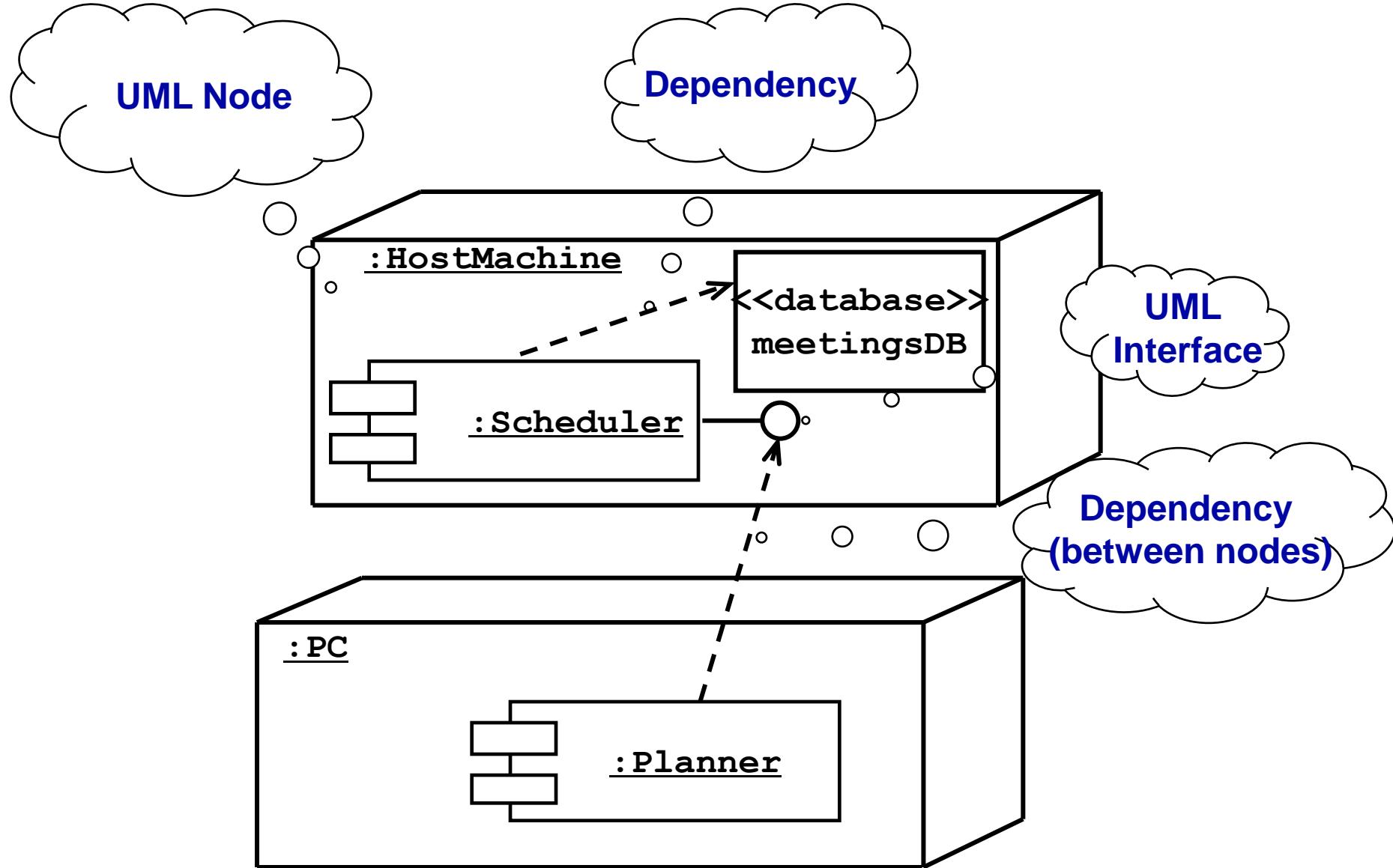
# Backup Slides



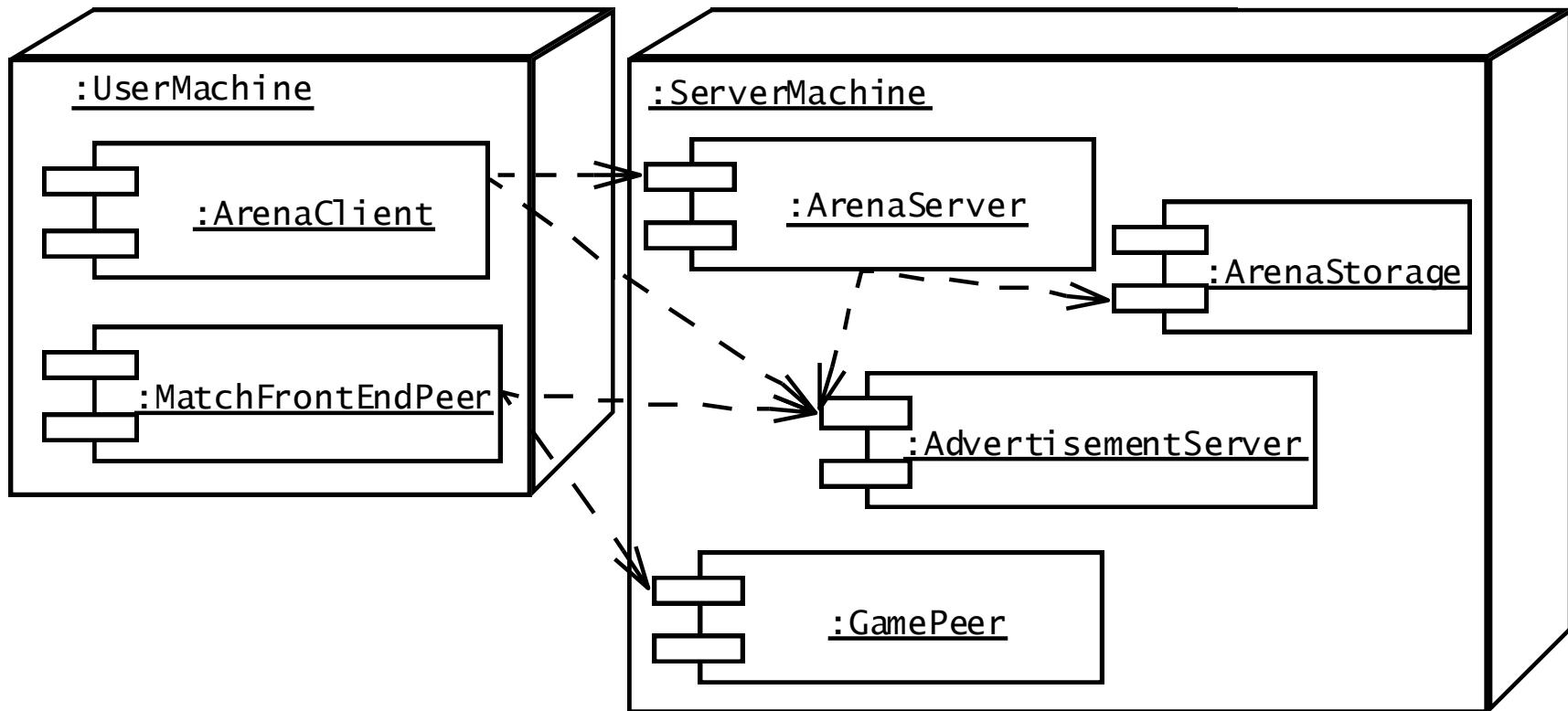
# Component Diagram (UML 1.0 notation)



# Deployment Diagram Example (UML 1.0)



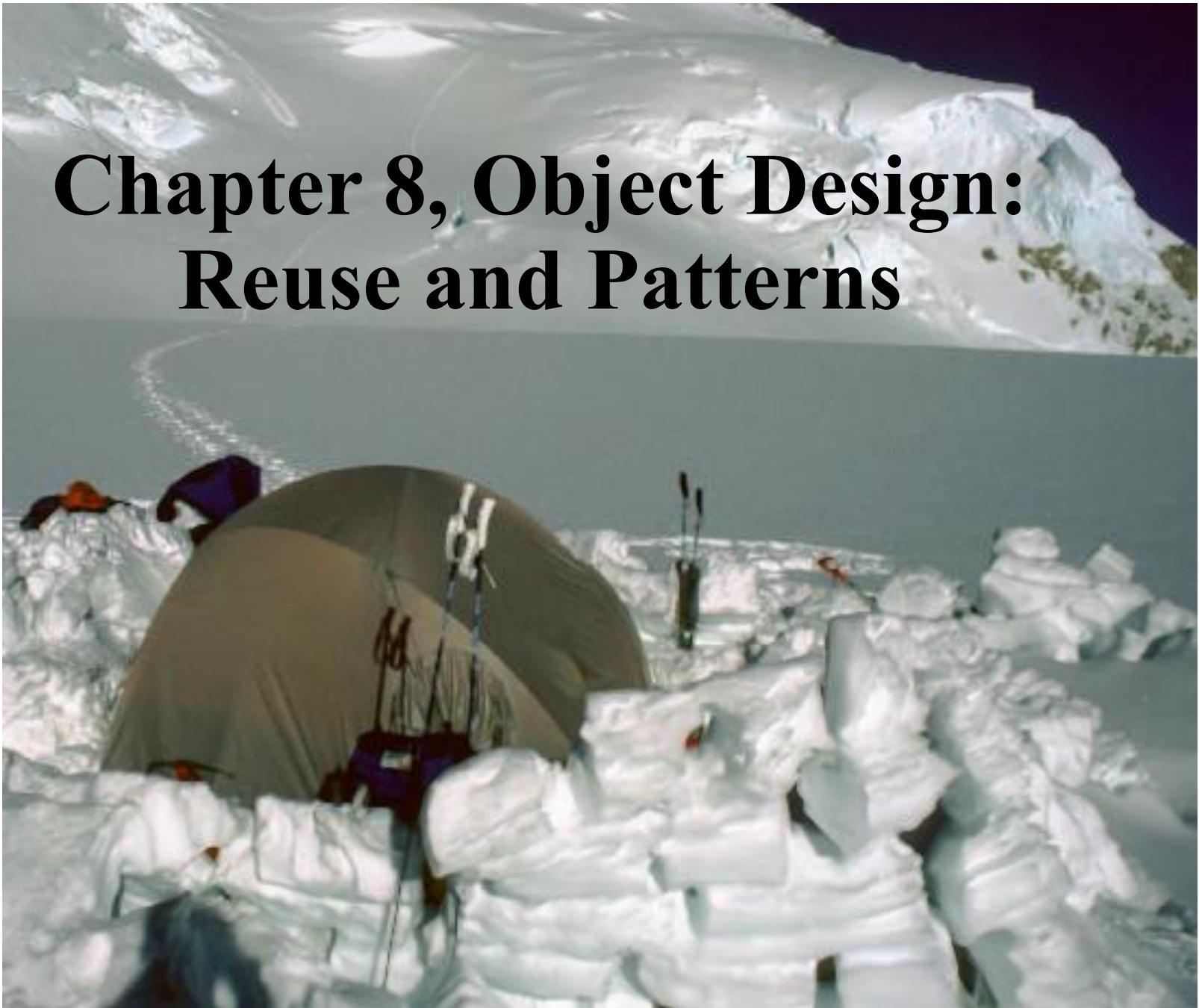
# ARENA Deployment Diagram (UML 1.0)



# Object-Oriented Software Engineering

Using UML, Patterns, and Java

## Chapter 8, Object Design: Reuse and Patterns



# Where are we? What comes next?

- We have covered:
  - *Introduction to Software Engineering (Chapter 1)*
  - *Modeling with UML (Chapter 2)*
  - *Requirements Elicitation (Chapter 4)*
  - *Analysis (Chapter 5)*
  - *Design Patterns (Chapter 8 and Appendix A)*
- Today:
  - *Object Design (Chapter 8)*
- Next week
  - *System Design (Chapter 6)*
- Saturday:
  - *Mid-Term.*

# Details for the Mid-Term:

- *Coverage:*
  - *Lecture 1 - lecture 10 (this lecture)*
  - *Textbook: Chapter 1 - 8 (Chapter 6 - 7 are not covered)*
- *Closed book exam*
  - *13:00 to 14:30 am: 90 min*
  - *Format: Paper-based, handwritten notes*
  - *Questions about definitions and modeling activities*
  - *Dictionaries are allowed*
- *For additional information, check the lecture portal*

# Outline of Today

- *Definition and Terminology: Object Design vs Detailed Design*
- *System Design vs Object Design*
- *Object Design Activities*
- *Reuse examples*
  - *Whitebox and Blackbox Reuse*
- *Object design leads also to new classes*
- *Implementation vs Specification Inheritance*
- *Inheritance vs Delegation*
- *Class Libraries and Frameworks*
- *Exercises: Documenting the Object Design*
  - *JavaDoc, Doxygen*

# Object Design

- *Purpose of object design:*
  - *Prepare for the implementation of the system model based on design decisions*
  - *Transform the system model (optimize it)*
- *Investigate alternative ways to implement the system model*
  - *Use design goals: minimize execution time, memory and other measures of cost.*
- *Object design serves as the basis of implementation.*

# Terminology: Naming of Design Activities

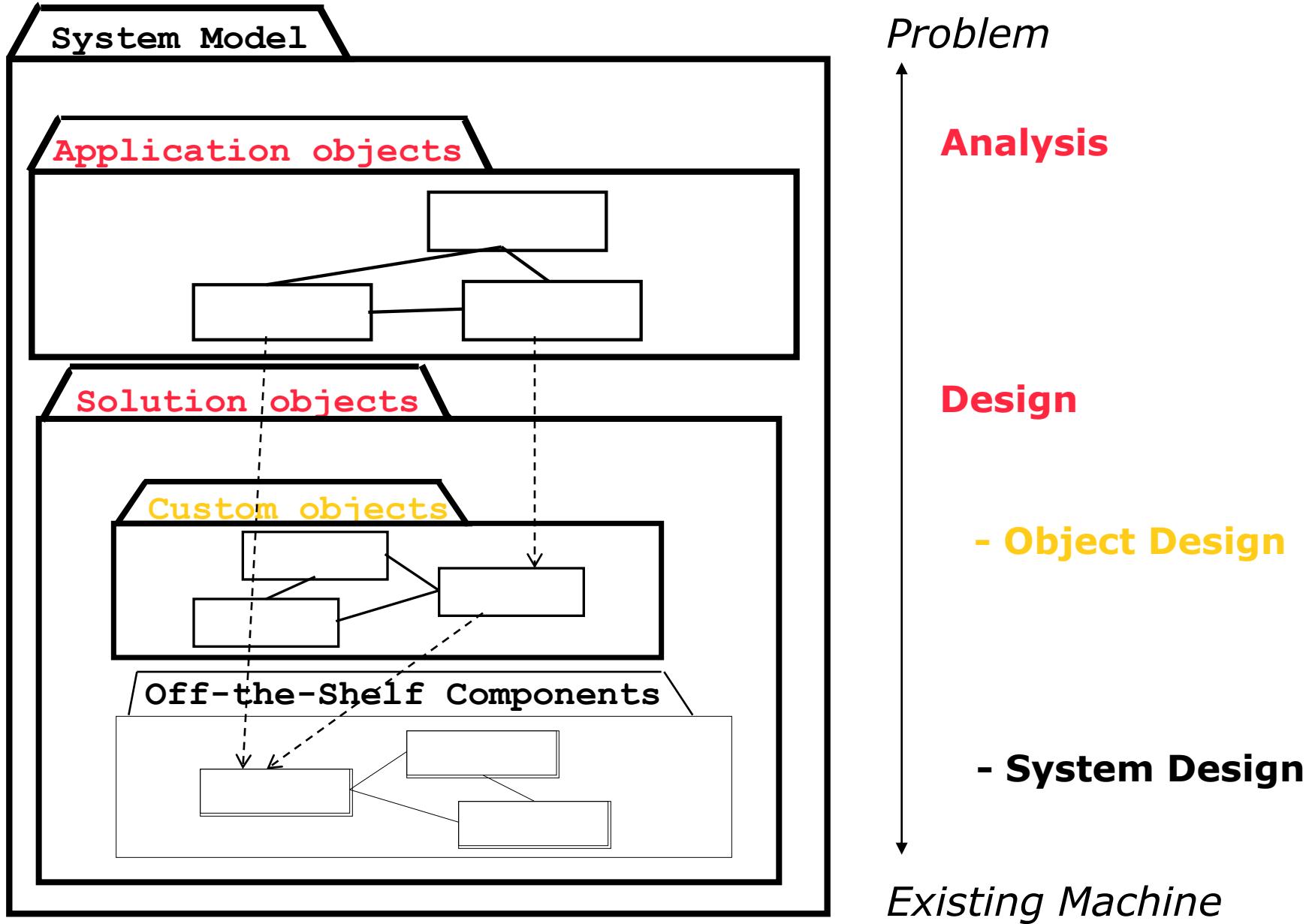
## Methodology: Object-oriented software engineering (OOSE)

- *System Design*
  - *Decomposition into subsystems, etc*
- *Object Design*
  - *Data structures and algorithms chosen*
- *Implementation*
  - *Implementation language is chosen*

## Methodology: Structured analysis/structured design (SA/SD)

- *Preliminary Design*
  - *Decomposition into subsystems, etc*
  - *Data structures are chosen*
- *Detailed Design*
  - *Algorithms are chosen*
  - *Data structures are refined*
  - *Implementation language is chosen.*

# System Development as a Set of Activities



# Design means “Closing the Gap”



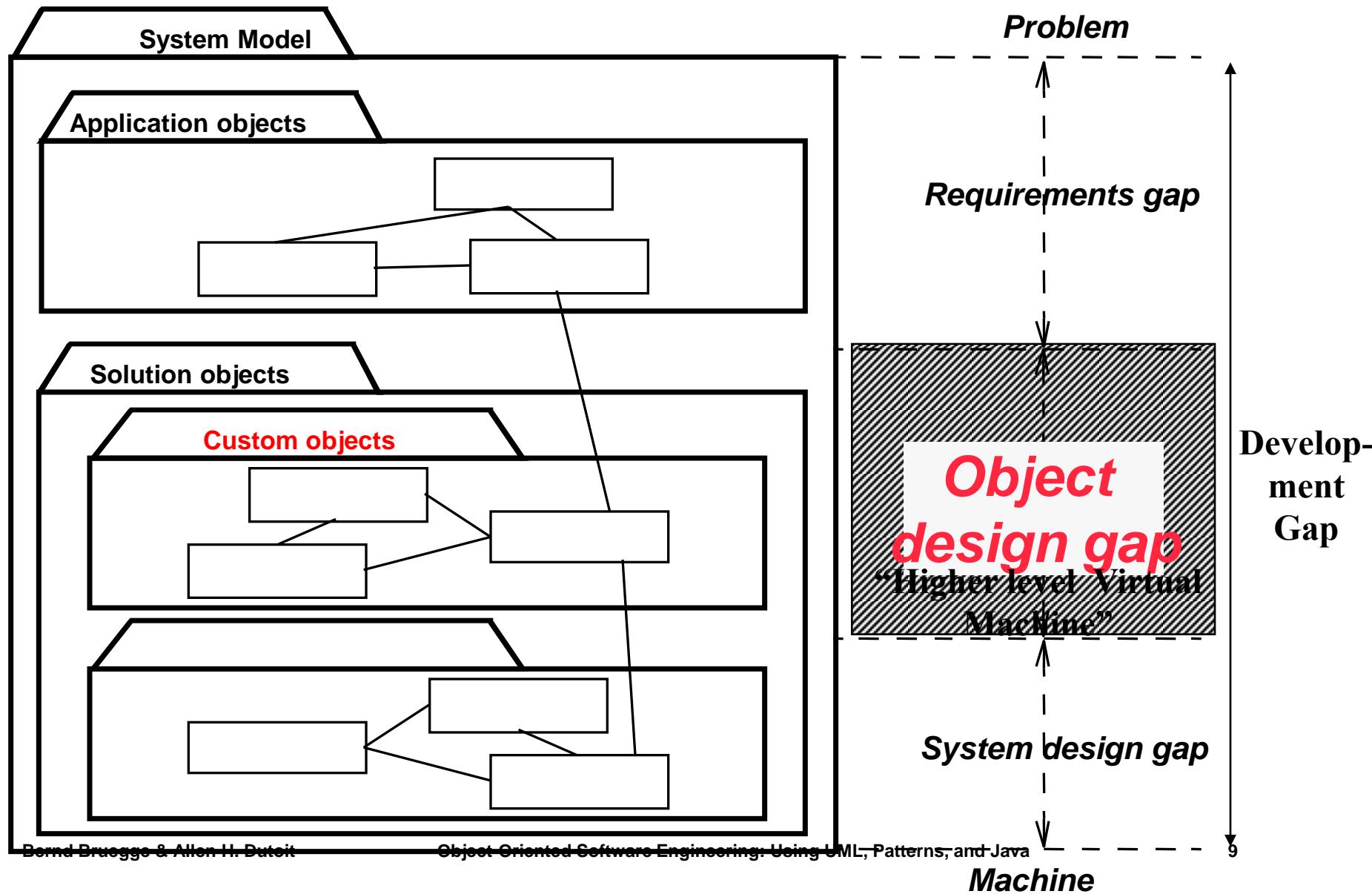
“Subsystem 1”: Rock material from the Southern Sierra Nevada mountains (moving north)

Example of a Gap:  
**San Andreas Fault**

“Subsystem 3” closes the Gap:  
San Andreas Lake

“Subsystem 2”: San Francisco Bay Area

# Design means “Closing the Gap”



# Object Design consists of 4 Activities

## 1. *Reuse: Identification of existing solutions*

- *Use of inheritance*
- *Off-the-shelf components and additional solution objects*
- *Design patterns*

## 2. *Interface specification*

- *Describes precisely each class interface*

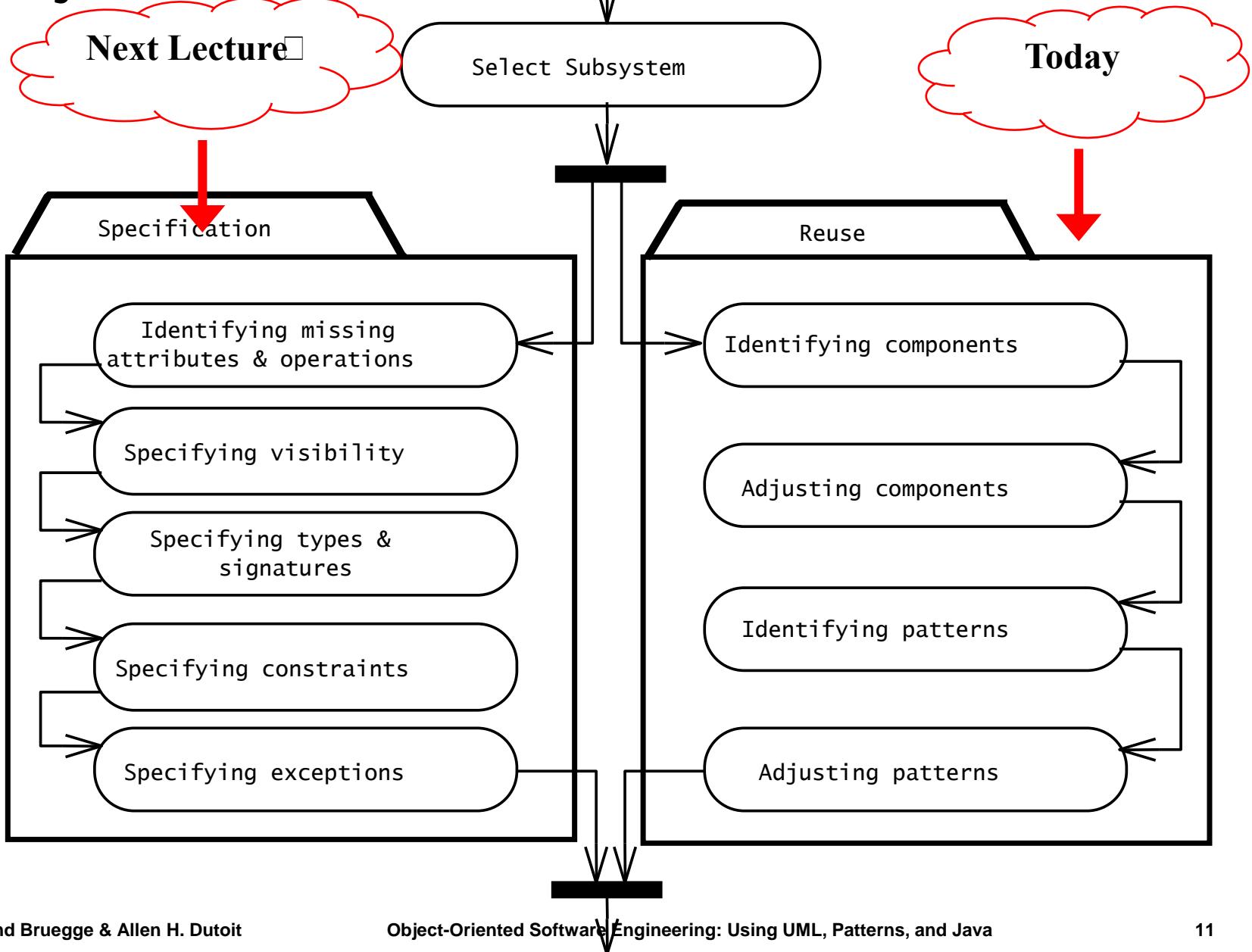
## 3. *Object model restructuring*

- *Transforms the object design model to improve its understandability and extensibility*

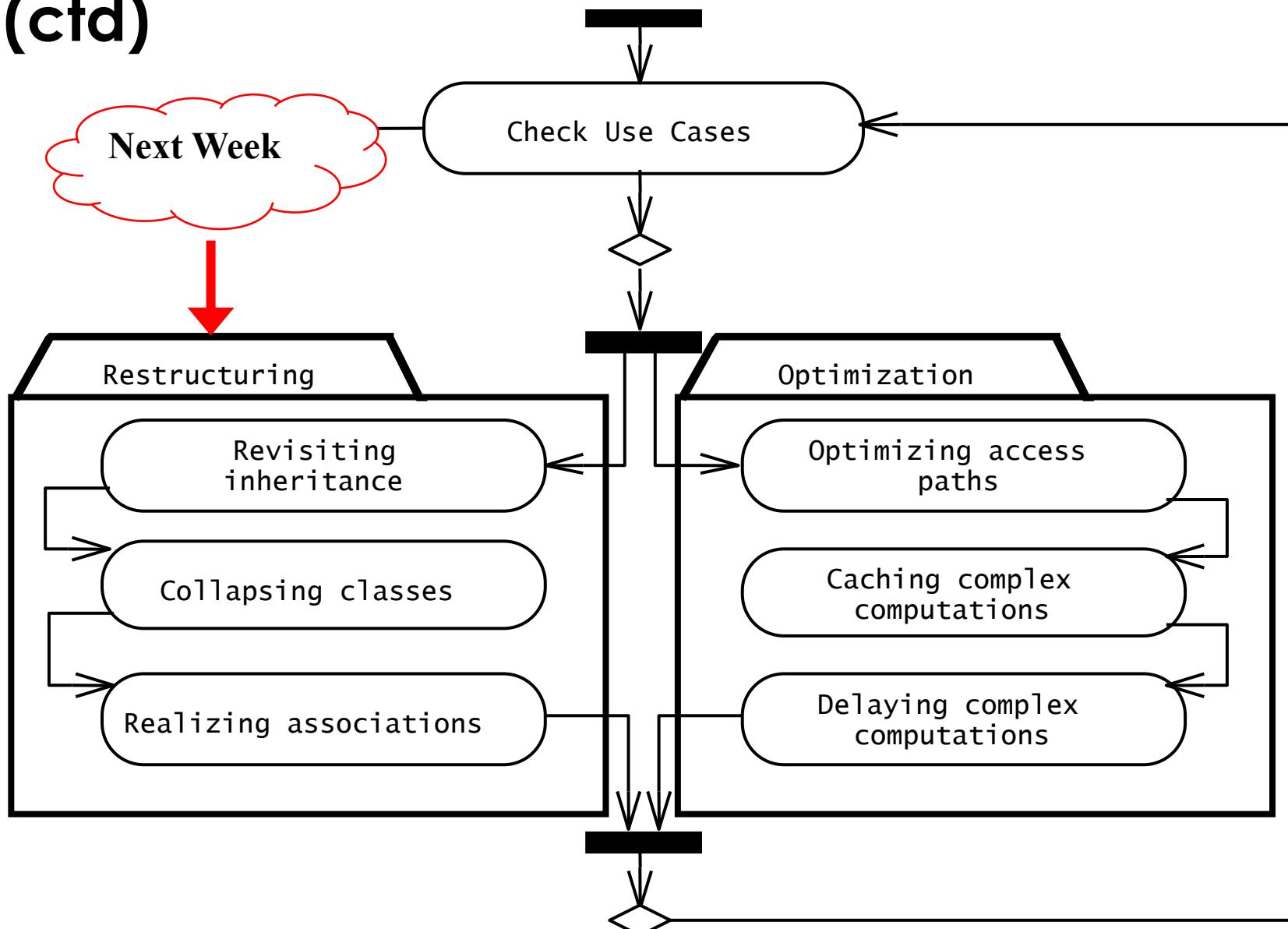
## 4. *Object model optimization*

- *Transforms the object design model to address performance criteria such as response time or memory utilization.*

# Object Design Activities



# Detailed View of Object Design Activities (ctd)



# One Way to do Object Design

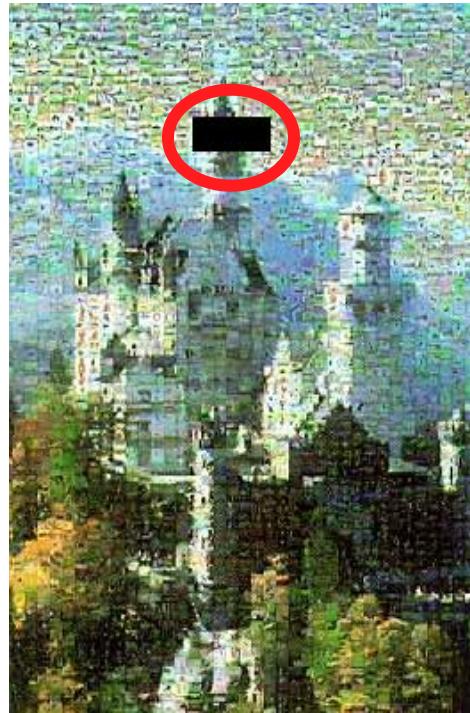
1. *Identify the missing components in the design gap*
2. *Make a build or buy decision to obtain the missing component*

=> *Component-Based Software Engineering:*

*The design gap is filled with available components ("0 % coding").*

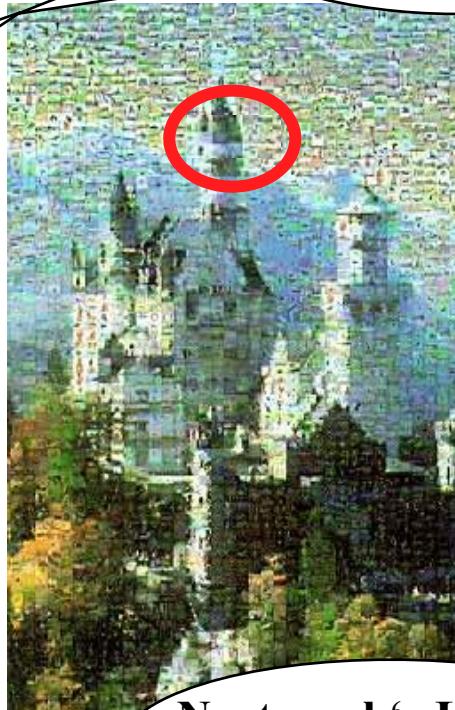
- *Special Case: COTS-Development*
    - *COTS: Commercial-off-the-Shelf*
    - *The design gap is completely filled with commercial-off-the-shelf-components.*
- => *Design with standard components.*

# Design with Standard Components is solving a Jigsaw Puzzle



What do we do  
if that is not true?“

Puzzle Piece  
("component")



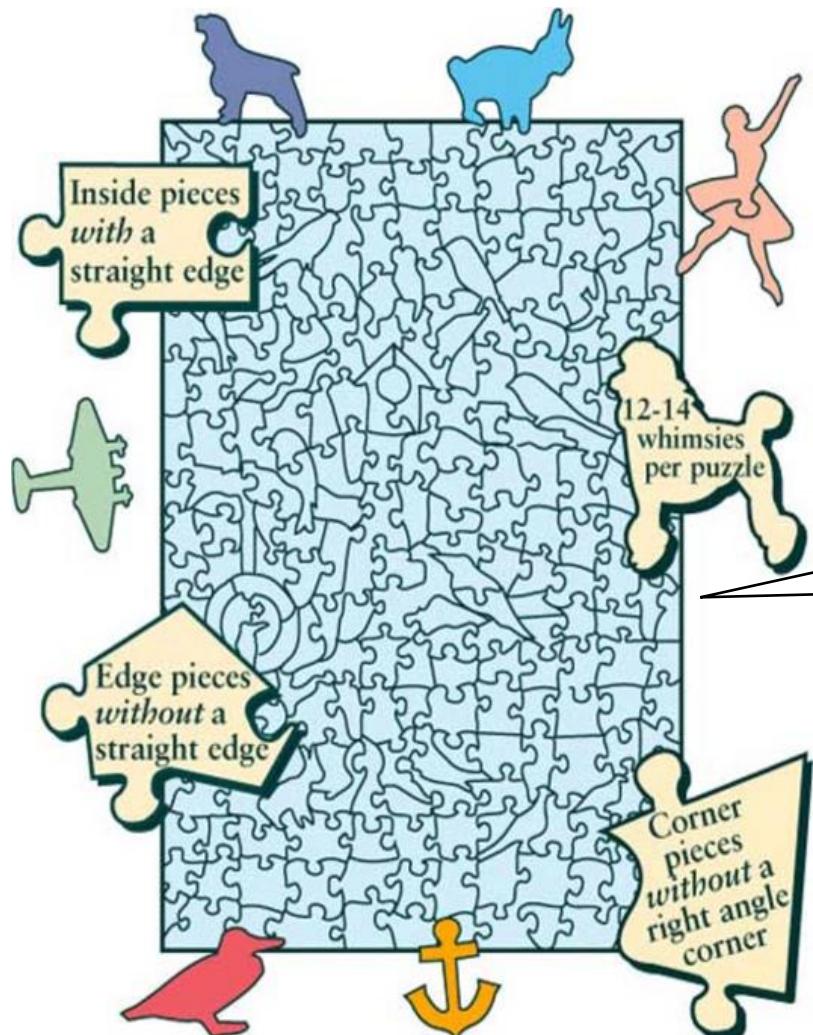
Standard Puzzles:  
„Corner pieces have  
two straight edges“

## Design Activities:

1. Start with the architecture (*subsystem decomposition*)
2. Identify the missing component
3. Make a build or buy decision for the component
4. Add the component to the system (*finalizing the design*).

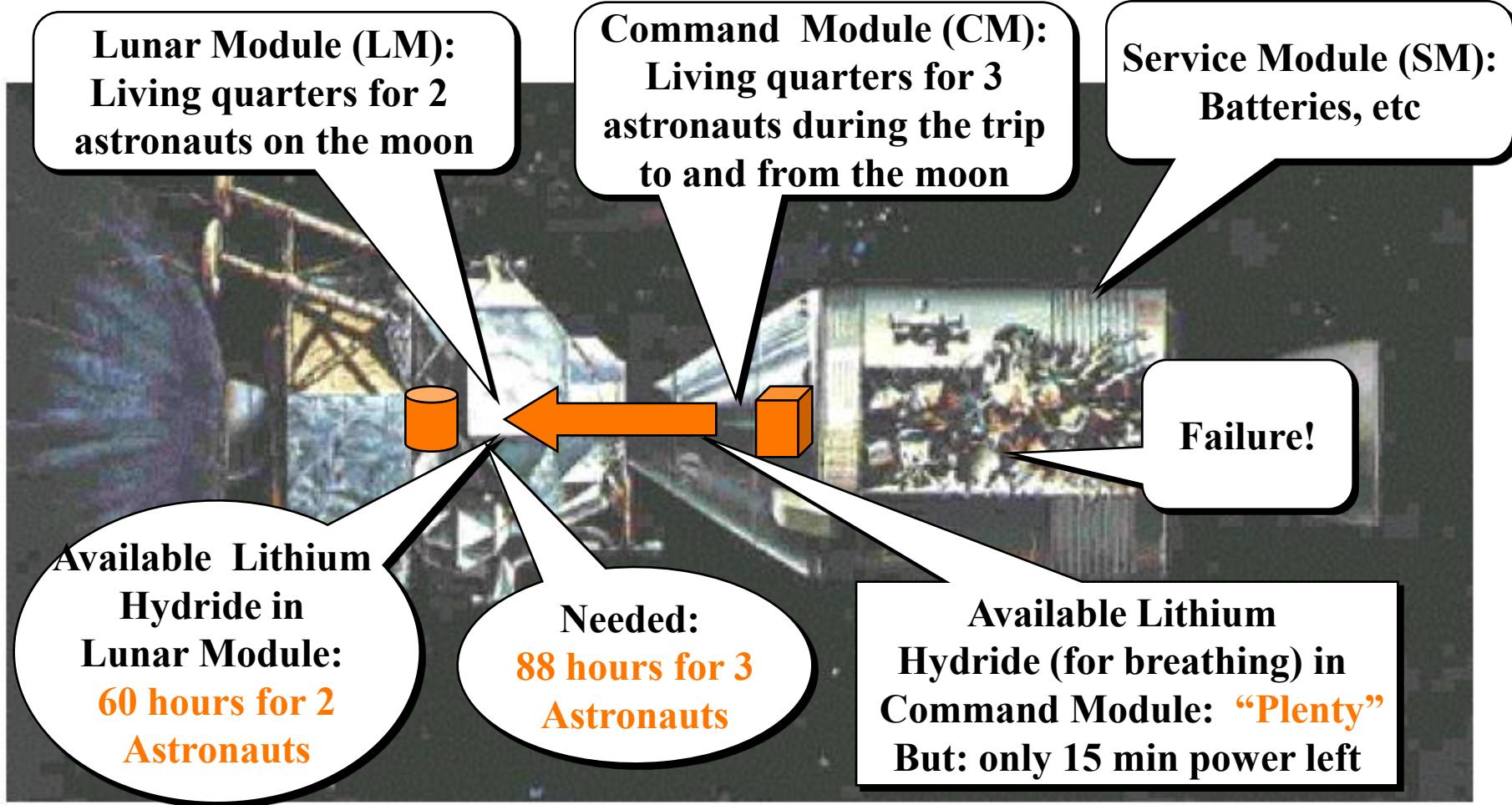
Next week's Lecture  
(Chapter 6)

# What do we do if we have non-Standard Components?



**Advanced  
Jigsaw Puzzles**

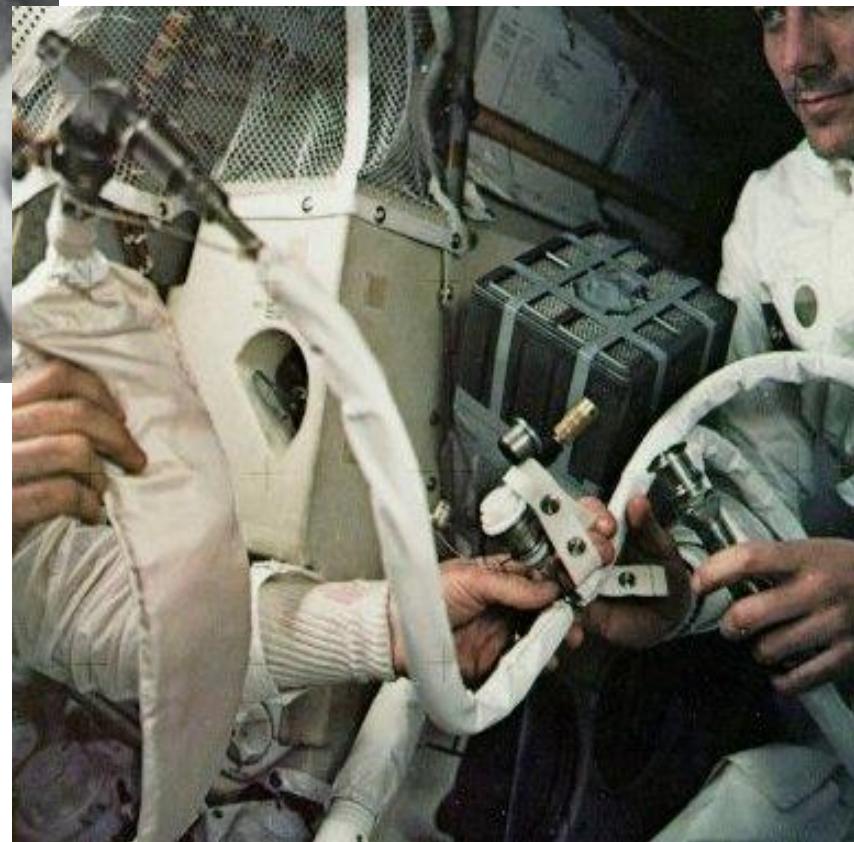
# Apollo 13: “Houston, we’ve had a Problem!”



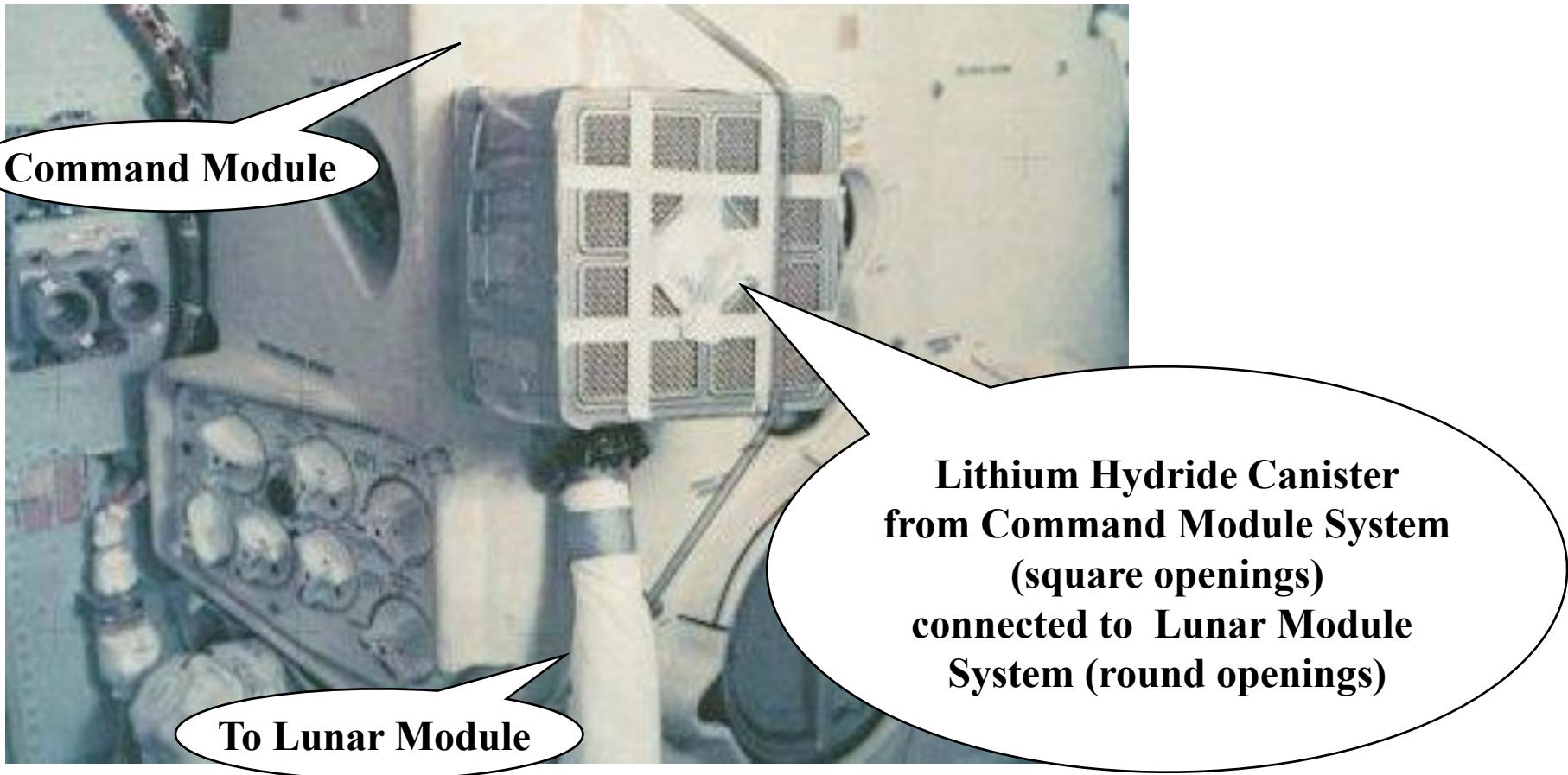
The LM was **designed** for 60 hours for 2 astronauts staying 2 days on the moon

**Redesign challenge:** Can the LM be used for 12 man-days (2 1/2 days until reentry into Earth)?

# Apollo 13: “Fitting a square peg in a round hole”



# A Typical Object Design Challenge: Connecting Incompatible Components

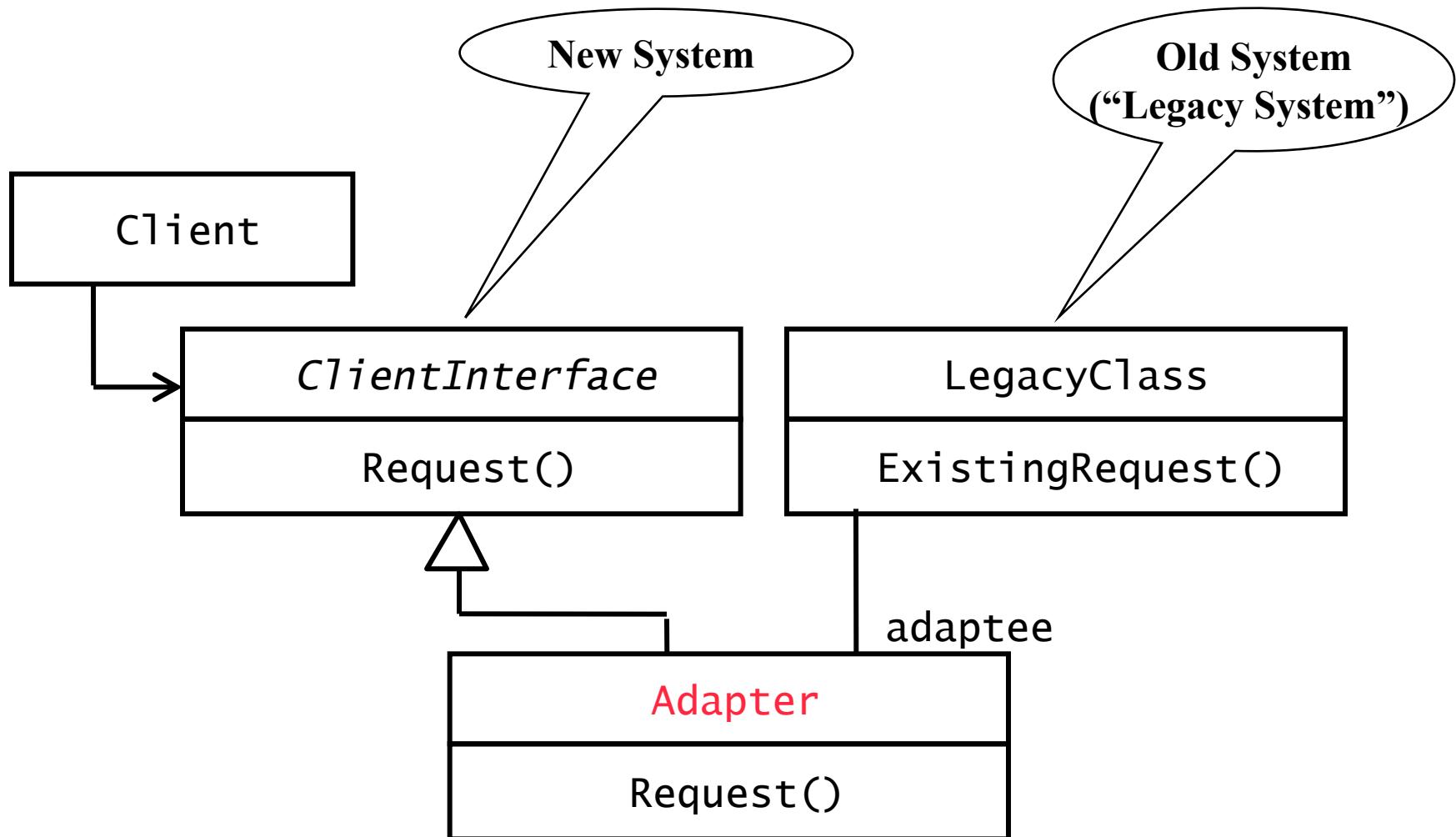


Source: <http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html>

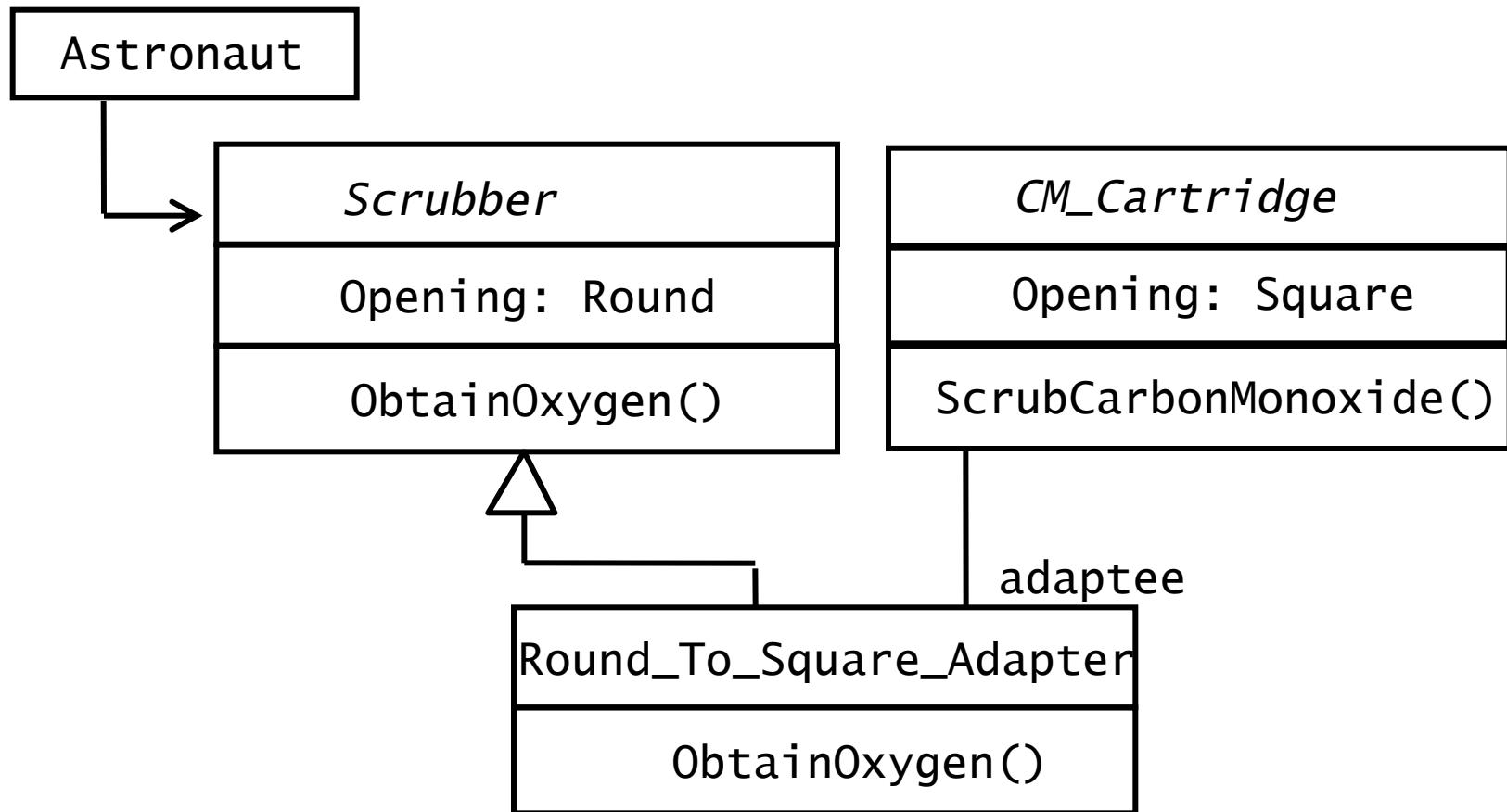
# Adapter Pattern

- *Adapter Pattern: Connects incompatible components.*
  - *It converts the interface of one component into another interface expected by the other (calling) component*
  - *Used to provide a new interface to existing legacy components (Interface engineering, reengineering)*
- *Also known as a wrapper.*

# Adapter Pattern



# Adapter for Scrubber in Lunar Module



- *Using a carbon monoxide scrubber (round opening) in the lunar module with square cartridges from the command module (square opening)*

# Modeling of the Real World

- *Modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.*
- *There is a need for reusable and flexible designs*
- *Design knowledge such as the adapter pattern complements application domain knowledge and solution domain knowledge.*

# Outline of Today

- *Object Design vs Detailed Design*
- *System Design vs Object Design*
- *Object Design Activities*

## → *Reuse examples*

- *Reuse of code, interfaces and existing classes*
- *White box and black box reuse*
- *The use of inheritance*
- *Implementation vs. specification inheritance*
- *Delegation vs. Inheritance*
- *Abstract classes and abstract methods*
- *Contraction: Bad example of inheritance*
- *Meta model for inheritance*

# Reuse of Code

- *I have a list, but my customer would like to have a stack*
  - *The list offers the operations Insert(), Find(), Delete()*
  - *The stack needs the operations Push(), Pop() and Top()*
  - *Can I reuse the existing list?*
- *I am an employee in a company that builds cars with expensive car stereo systems*
  - *Can I reuse the existing car software in a home stereo system?*

# Reuse of interfaces

- *I am an off-shore programmer in Hawaii. I have a contract to implement an electronic parts catalog for DaimlerChrysler*
  - *How can I and my contractor be sure that I implement it correctly?*
- *I would like to develop a window system for Linux that behaves the same way as in Vista*
  - *How can I make sure that I follow the conventions for Vista windows and not those of MacOS X?*
- *I have to develop a new service for cars, that automatically call a help center when the car is used the wrong way.*
  - *Can I reuse the help desk software that I developed for a company in the telecommuniction industry?*

# Reuse of existing classes

- *I have an implementation for a list of elements of Typ int*
  - *Can I reuse this list to build*
    - *a list of customers*
    - *a spare parts catalog*
    - *a flight reservation schedule?*
- *I have developed a class “Addressbook” in another project*
  - *Can I add it as a subsystem to my e-mail program which I purchased from a vendor (replacing the vendor-supplied addressbook)?*
  - *Can I reuse this class in the billing software of my dealer management system?*

# Customization: Build Custom Objects

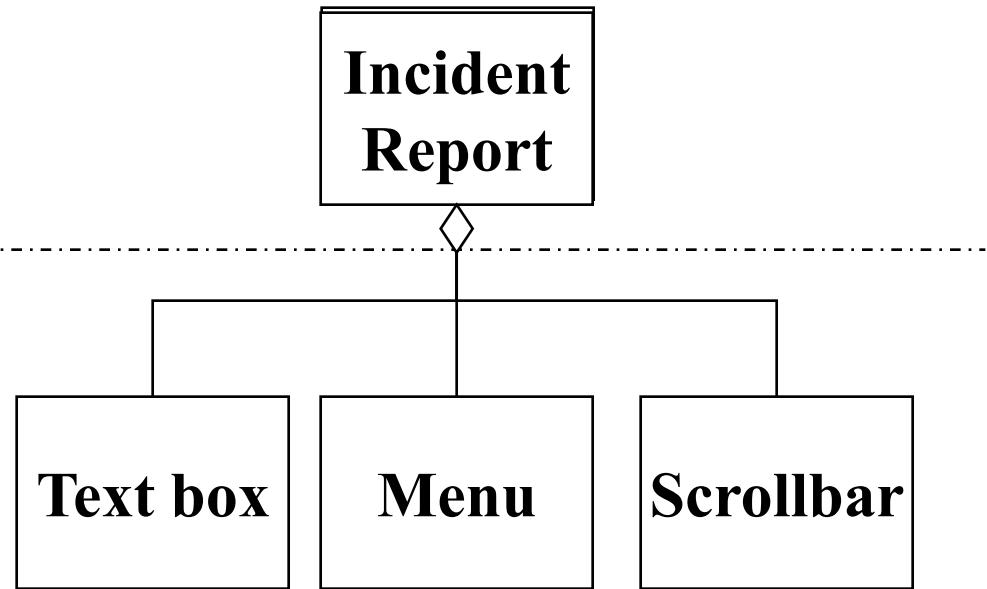
- *Problem: Close the object design gap*
  - *Develop new functionality*
- *Main goal:*
  - *Reuse knowledge from previous experience*
  - *Reuse functionality already available*
- *Composition* (also called *Black Box Reuse*)
  - *New functionality is obtained by aggregation*
  - *The new object with more functionality is an aggregation of existing objects*
- *Inheritance* (also called *White-box Reuse*)
  - *New functionality is obtained by inheritance*

# White Box and Black Box Reuse

- *White box reuse*
  - Access to the development products (*models, system design, object design, source code*) must be available
- *Black box reuse*
  - Access to *models and designs* is not available, or *models do not exist*
    - *Worst case: Only executables (binary code) are available*
    - *Better case: A specification of the system interface is available.*

# Identification of new Objects during Object Design

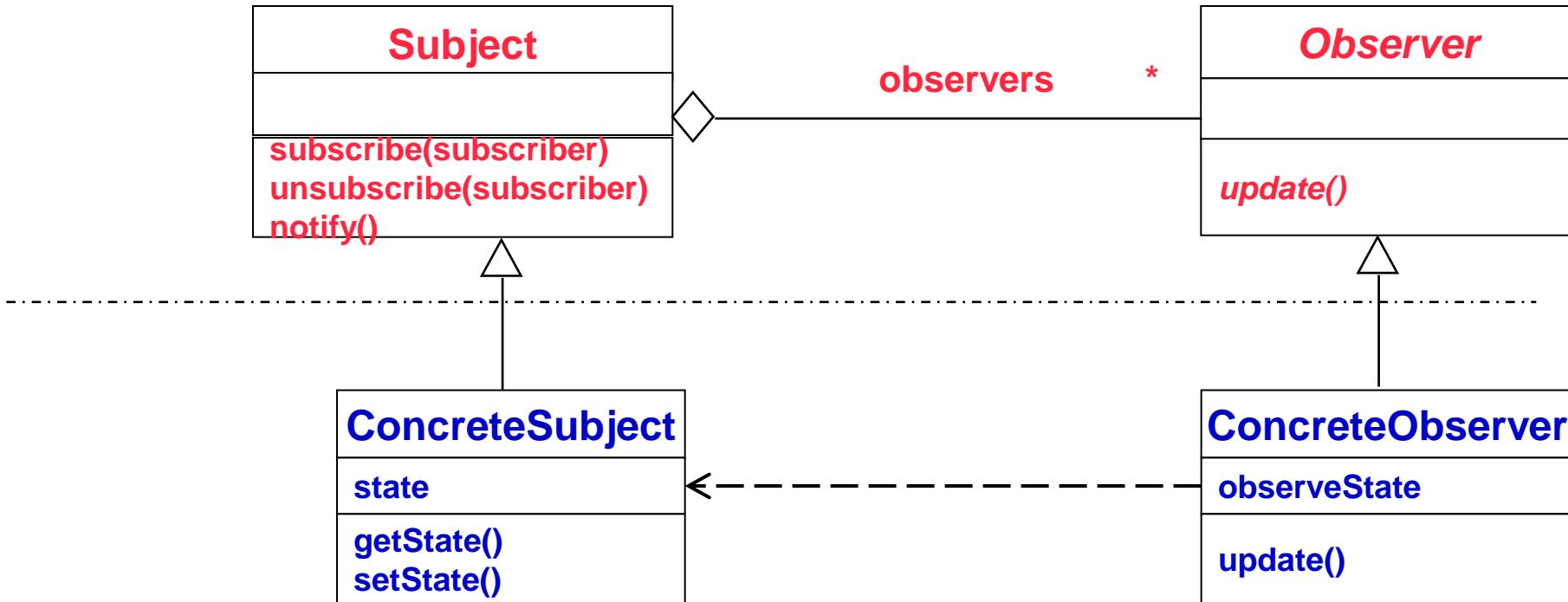
*Requirements Analysis  
(Language of Application Domain)*



*Object Design  
(Language of Solution Domain)*

# Application Domain vs Solution Domain Objects

*Requirements Analysis (Language of Application Domain)*



*Object Design (Language of Solution Domain)*

# Other Reasons for new Objects

- *The implementation of algorithms may necessitate objects to hold values*
- *New low-level operations may be needed during the decomposition of high-level operations*
- *Example: EraseArea () in a drawing program*
  - *Conceptually very simple*
  - *Implementation is complicated:*
    - *Area represented by pixels*
    - *We need a Repair () operation to clean up objects partially covered by the erased area*
    - *We need a Redraw () operation to draw objects uncovered by the erasure*
    - *We need a Draw () operation to erase pixels in background color not covered by other objects.*

# Types of Whitebox Reuse

1. *Implementation inheritance*
    - *Reuse of Implementations*
  2. *Specification Inheritance*
    - *Reuse of Interfaces*
- 
- *Programming concepts to achieve reuse*
    - *Inheritance*
      - *Delegation*
      - *Abstract classes and Method Overriding*
      - *Interfaces*

# Why Inheritance?

## 1. Organization (during analysis):

- *Inheritance helps us with the construction of taxonomies to deal with the application domain*
  - *when talking to customer and application domain experts we usually find already existing taxonomies*

## 2. Reuse (during object design):

- *Inheritance helps us to reuse models and code to deal with the solution domain*
  - *when talking to developers*

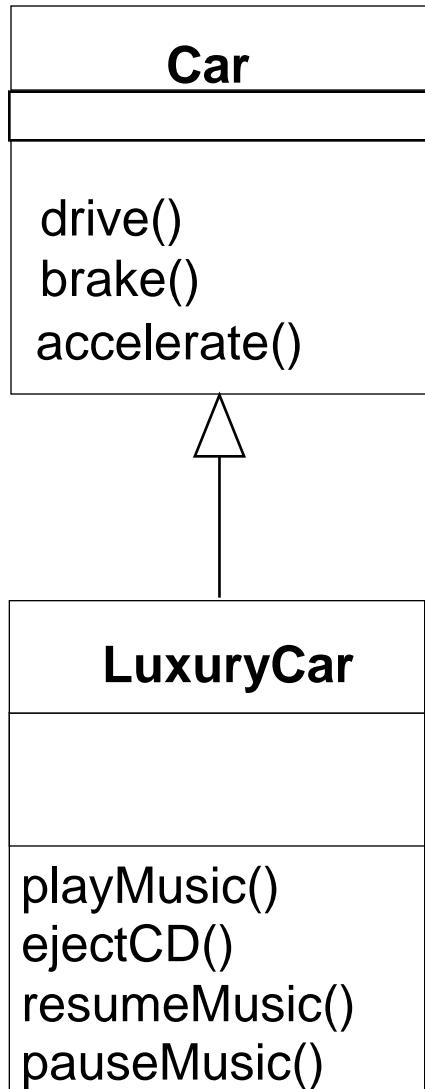
# The use of Inheritance

- *Inheritance is used to achieve two different goals*
  - *Description of Taxonomies*
  - *Interface Specification*
- *Description of Taxonomies*
  - *Used during requirements analysis*
  - *Activity: identify application domain objects that are hierarchically related*
  - *Goal: make the analysis model more understandable*
- *Interface Specification*
  - *Used during object design*
  - *Activity: identify the signatures of all identified objects*
  - *Goal: increase reusability, enhance modifiability and extensibility*

# Inheritance can be used during Modeling as well as during Implementation

- *Starting Point is always the requirements analysis phase:*
  - We start with use cases
  - We identify existing objects ("class identification")
  - We investigate the relationship between these objects; "Identification of associations":
    - general associations
    - aggregations
    - inheritance associations.

# Example of Inheritance



## Superclass:

```
public class Car {  
    public void drive() {...}  
    public void brake() {...}  
    public void accelerate() {...}  
}
```

## Subclass:

```
public class LuxuryCar extends Car {  
    public void playMusic() {...}  
    public void ejectCD() {...}  
    public void resumeMusic() {...}  
    public void pauseMusic() {...}  
}
```

# Inheritance comes in many Flavors

*Inheritance is used in four ways:*

- *Specialization*
- *Generalization*
- *Specification Inheritance*
- *Implementation Inheritance.*

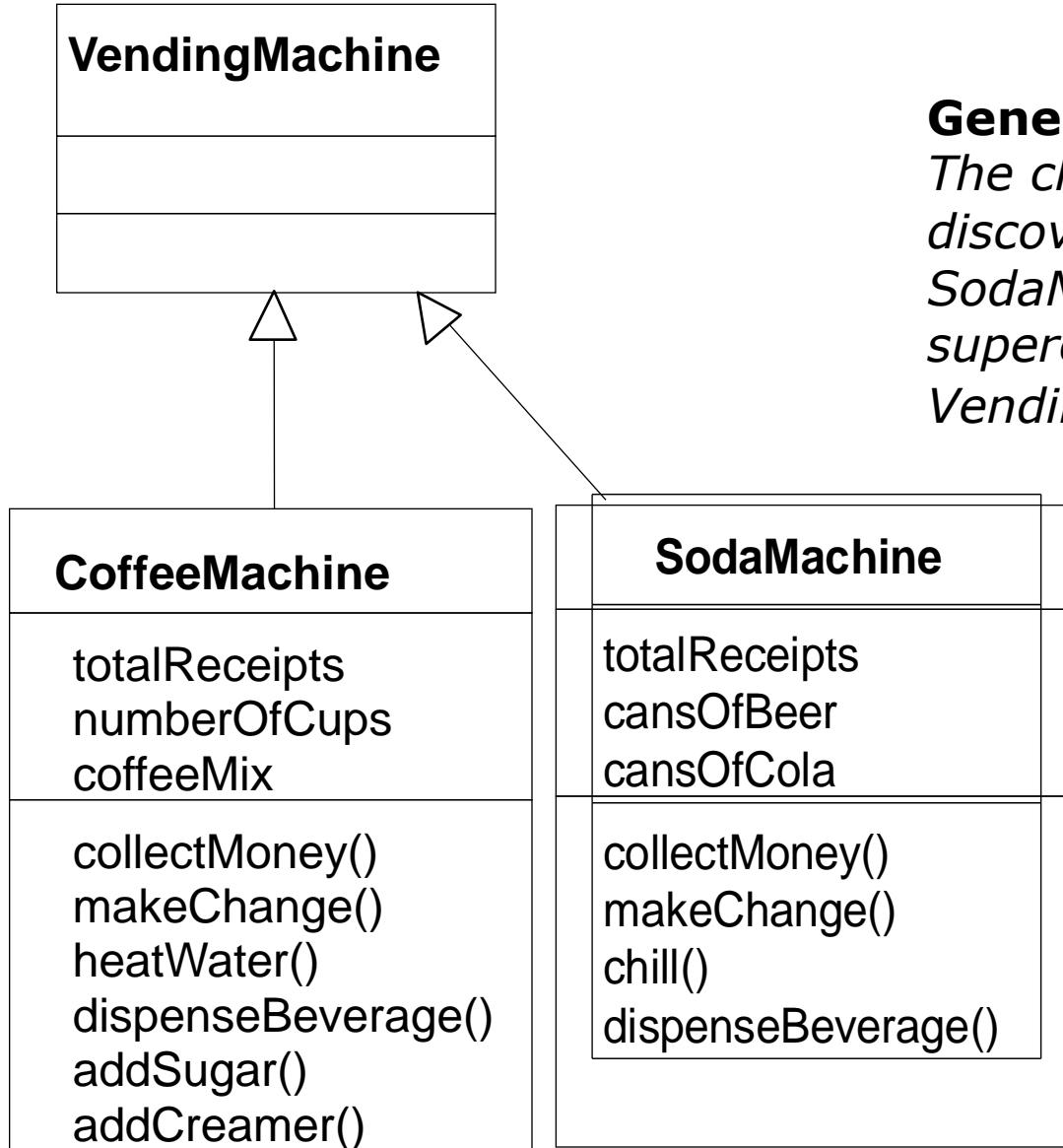
# Discovering Inheritance

- *To "discover" inheritance associations, we can proceed in two ways, which we call specialization and generalization*
- **Generalization:** *the discovery of an inheritance relationship between two classes, where the sub class is discovered first.*
- **Specialization:** *the discovery of an inheritance relationship between two classes, where the super class is discovered first.*

# Generalization

- *First we find the subclass, then the super class*
- *This type of discovery occurs often in science and engineering:*
  - **Biology:** *First we find individual animals (Elefant, Lion, Tiger), then we discover that these animals have common properties (mammals).*
  - **Engineering:** *What are the common properties of cars and airplanes?*

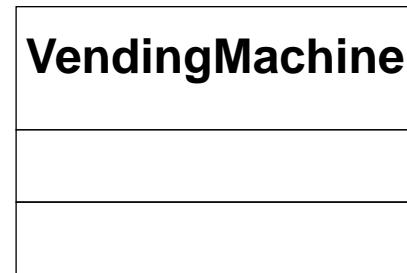
# Generalization Example: Modeling a Coffee Machine



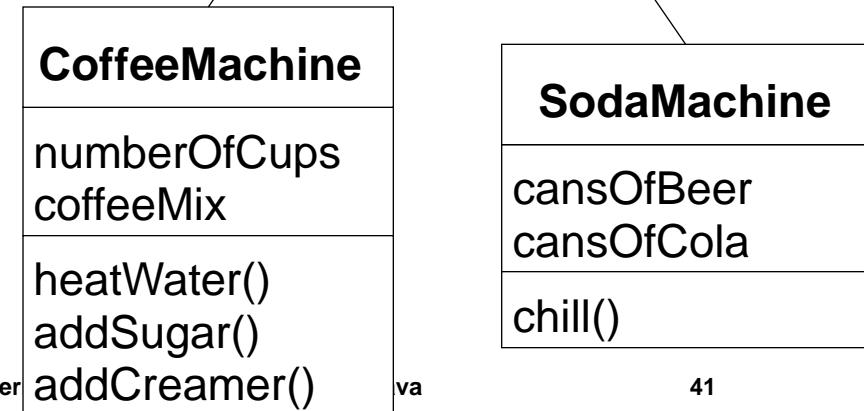
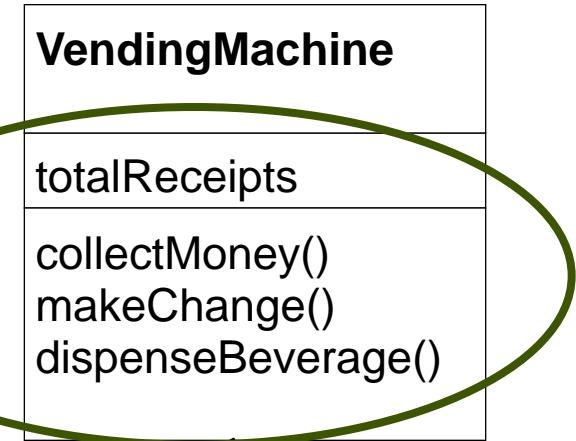
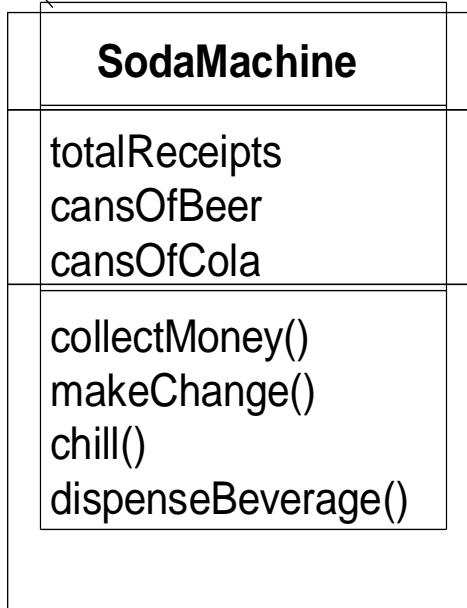
## Generalization:

*The class `CoffeeMachine` is discovered first, then the class `SodaMachine`, then the superclass `VendingMachine`*

# Restructuring of Attributes and Operations is often a Consequence of Generalization



Called **Remodeling** if done on  
the model level;  
called **Refactoring** if done on  
the source code level.

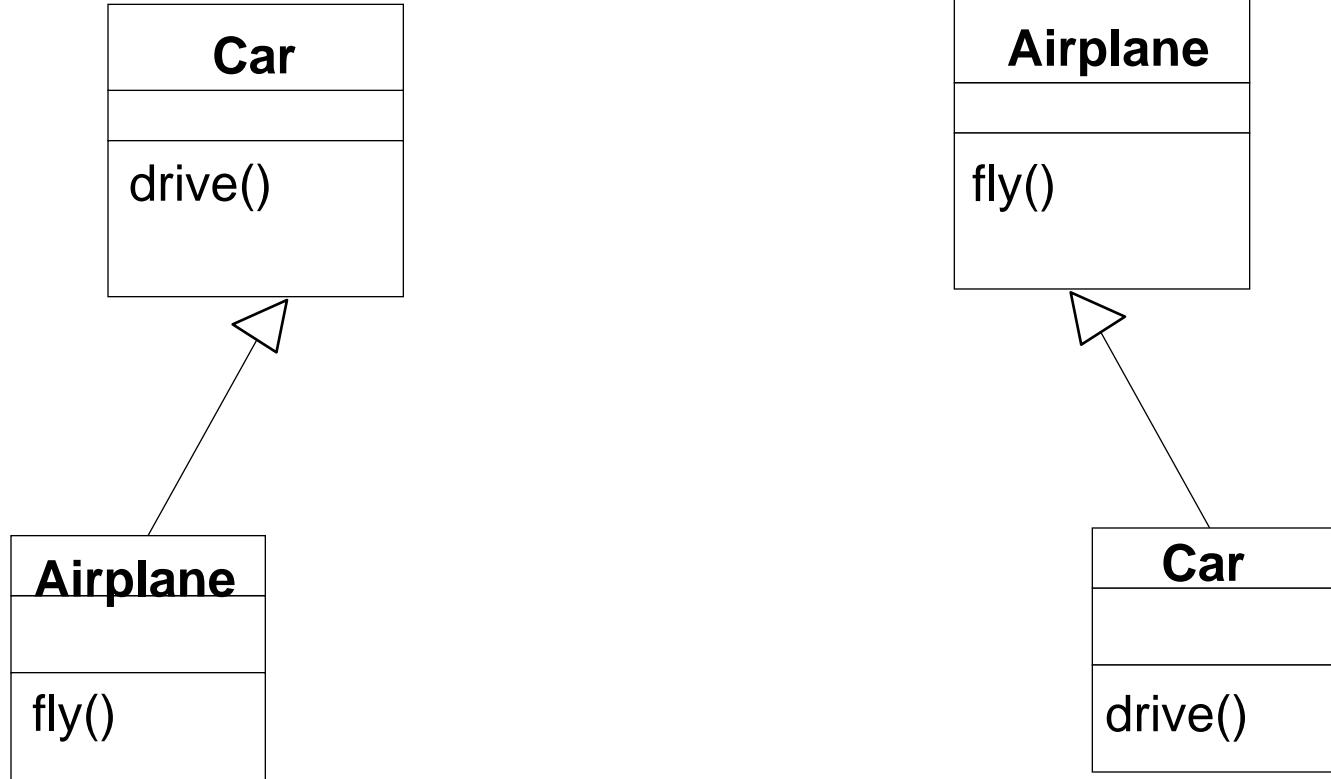


# Specialization

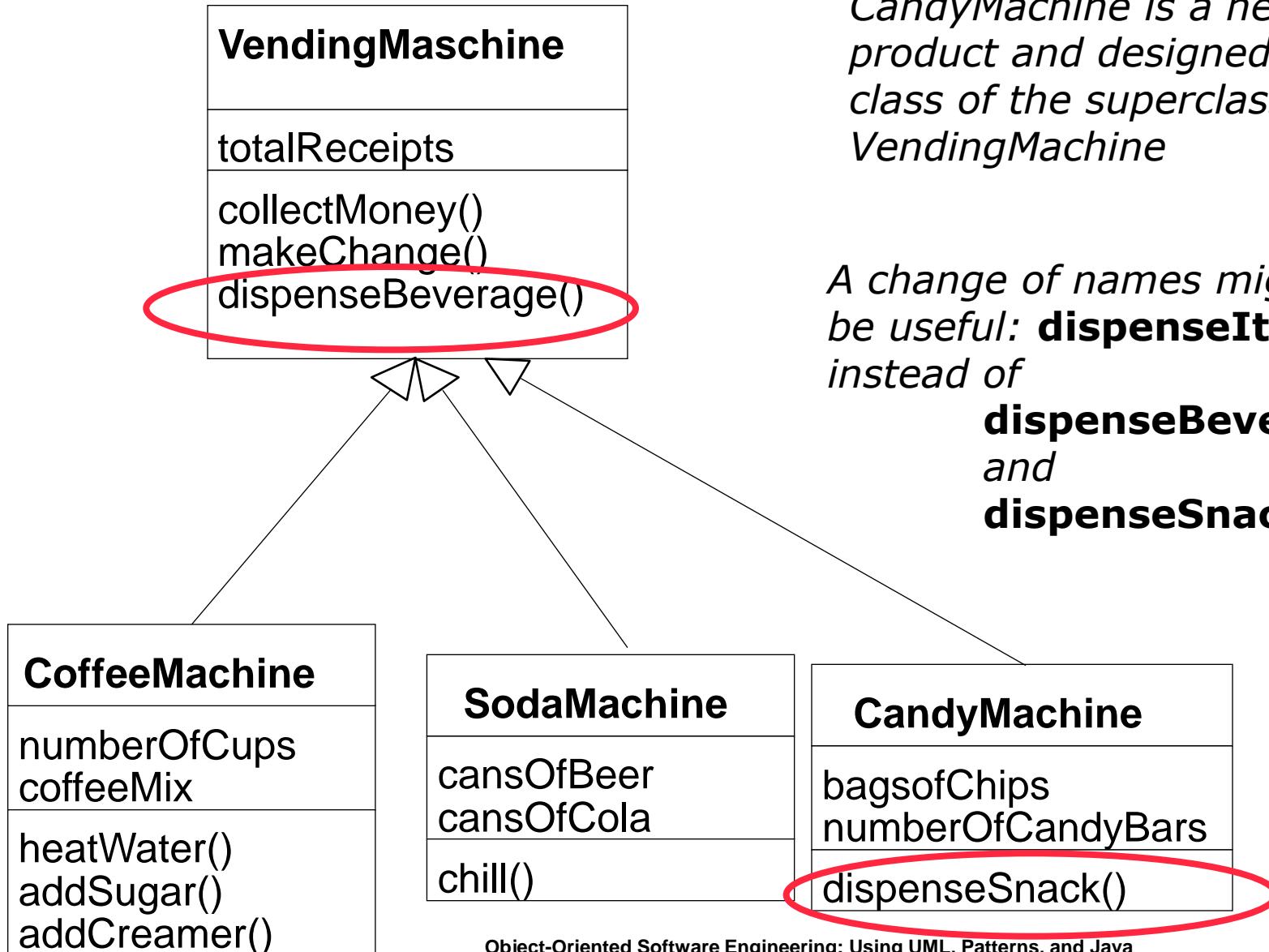
- *Specialization occurs, when we find a subclass that is very similar to an existing class.*
  - *Example: A theory postulates certain particles and events which we have to find.*
- *Specialization can also occur unintentionally:*



# Which Taxonomy is correct for the Example in the previous Slide?



# Another Example of a Specialization

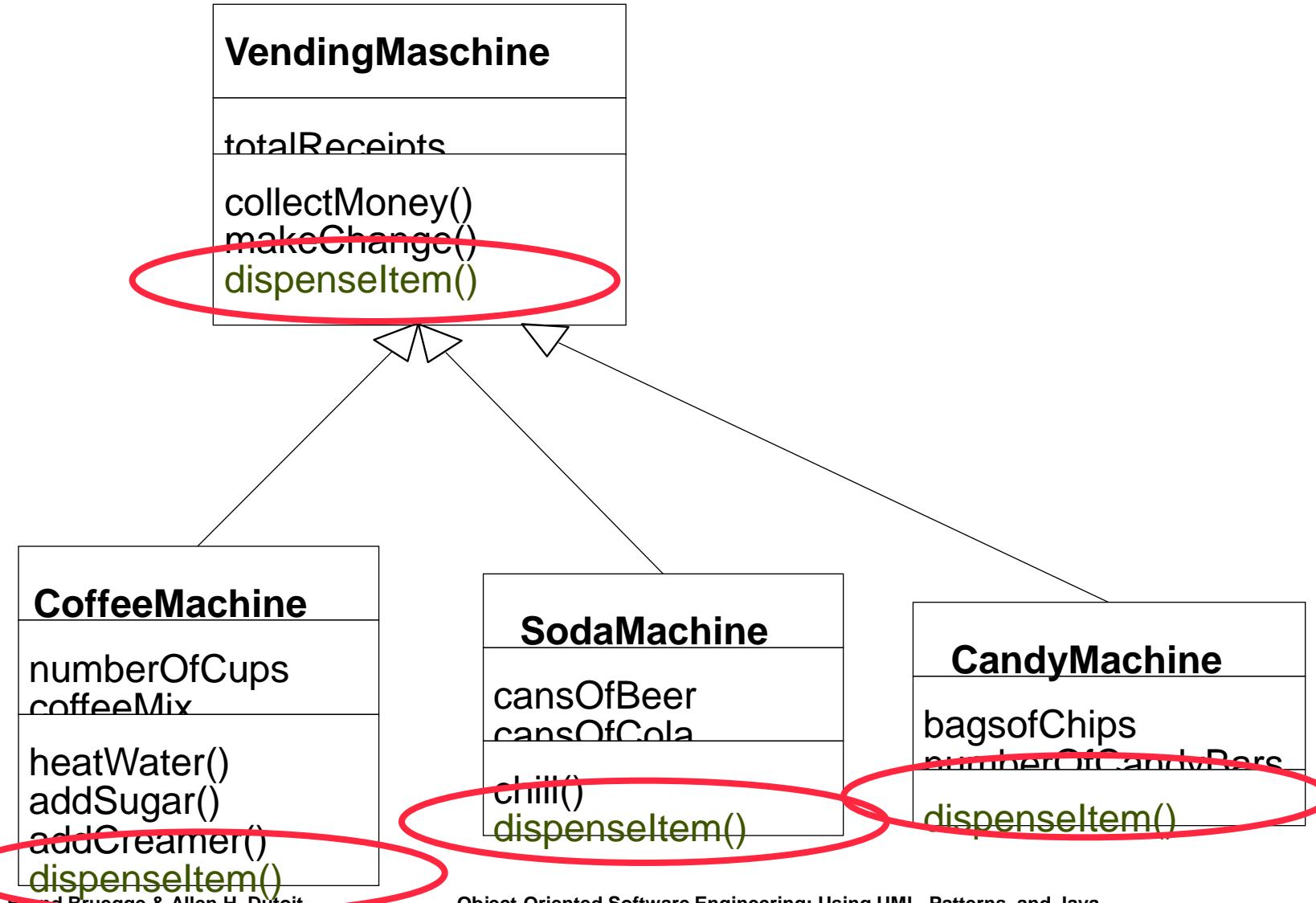


*CandyMachine is a new product and designed as a subclass of the superclass VendingMachine*

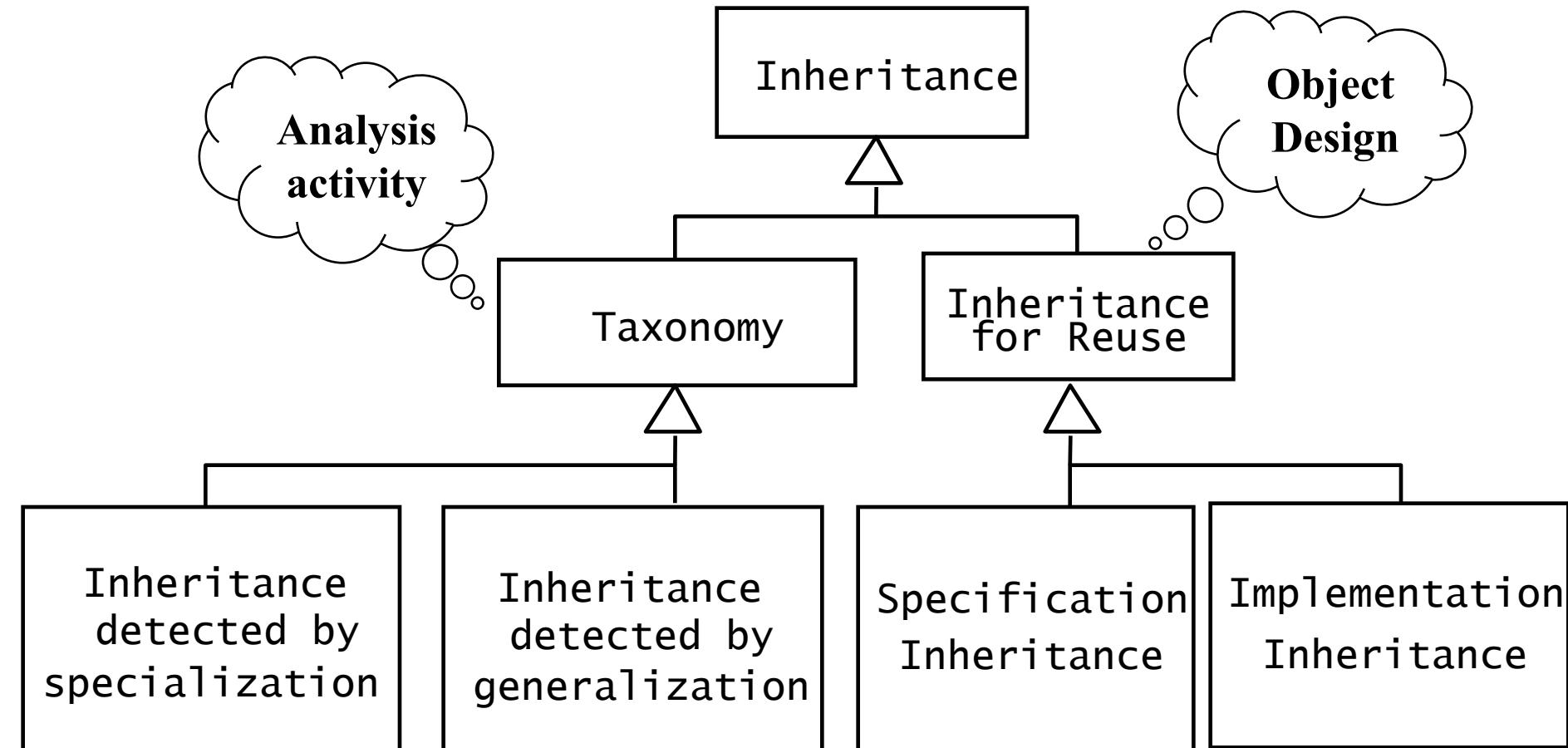
*A change of names might now be useful: **dispenseItem()** instead of*

**dispenseBeverage()**  
and  
**dispenseSnack()**

# Example of a Specialization (2)



# Meta-Model for Inheritance



# Implementation Inheritance and Specification Inheritance

- *Implementation inheritance*
  - *Also called class inheritance*
  - *Goal:*
    - *Extend an applications' functionality by reusing functionality from the super class*
    - *Inherit from an existing class with some or all operations already implemented*
- *Specification Inheritance*
  - *Also called subtyping*
  - *Goal:*
    - *Inherit from a specification*
    - *The specification is an abstract class with all operations specified, but not yet implemented.*

# Implementation Inheritance vs. Specification Inheritance

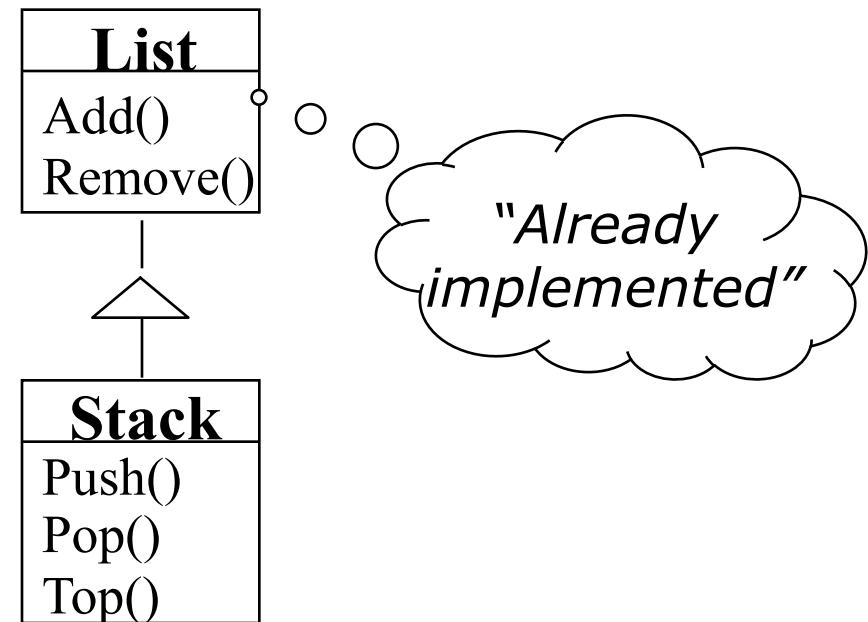
- *Implementation Inheritance: The combination of inheritance and implementation*
  - *The Interface of the superclass is completely inherited*
  - *Implementations of methods in the superclass ("Reference implementations") are inherited by any subclass*
- *Specification Inheritance: The combination of inheritance and specification*
  - *The Interface of the superclass is completely inherited*
  - *Implementations of the superclass (if there are any) are not inherited.*

# Example for Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation

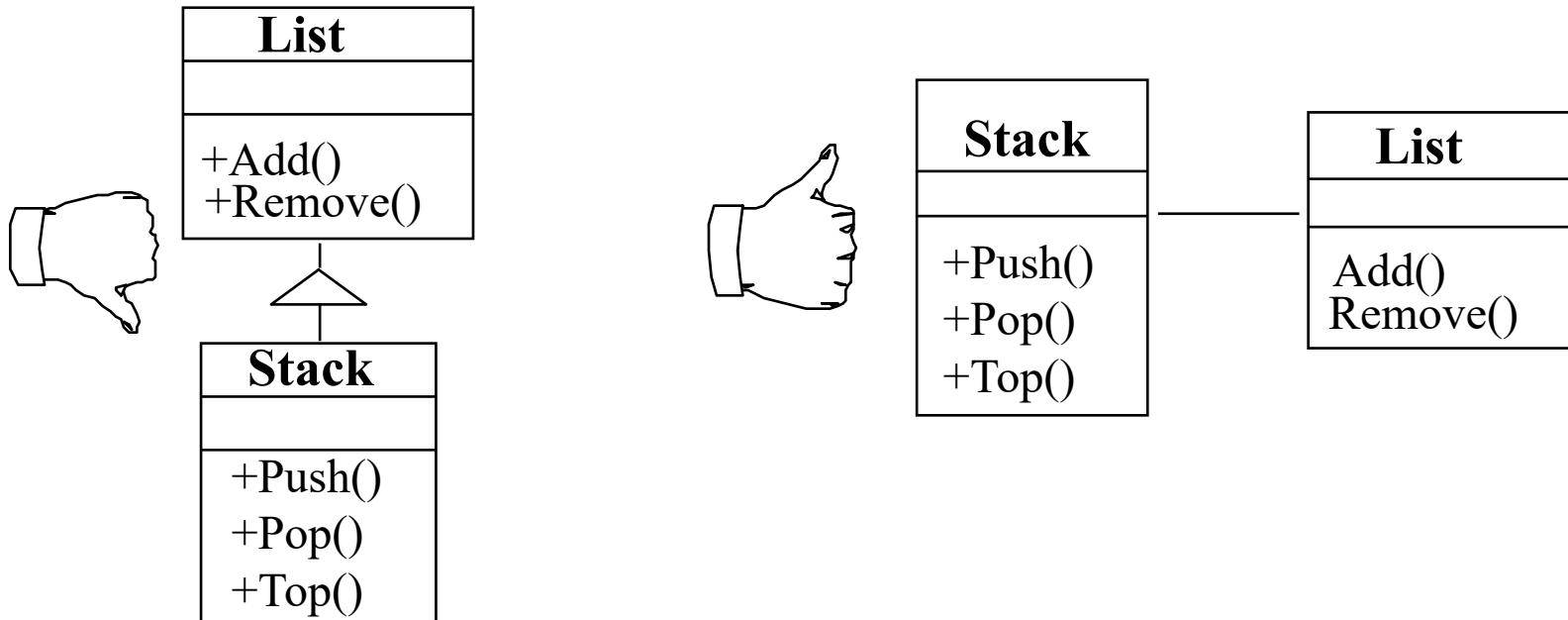
Example:

- I have a **List** class, I need a **Stack** class
  - How about subclassing the **Stack** class from the **List** class and implementing **Push()**, **Pop()**, **Top()** with **Add()** and **Remove()**?
- ❖ Problem with implementation inheritance:
- The inherited operations might exhibit unwanted behavior.
  - Example: What happens if the Stack user calls **Remove()** instead of **Pop()**?



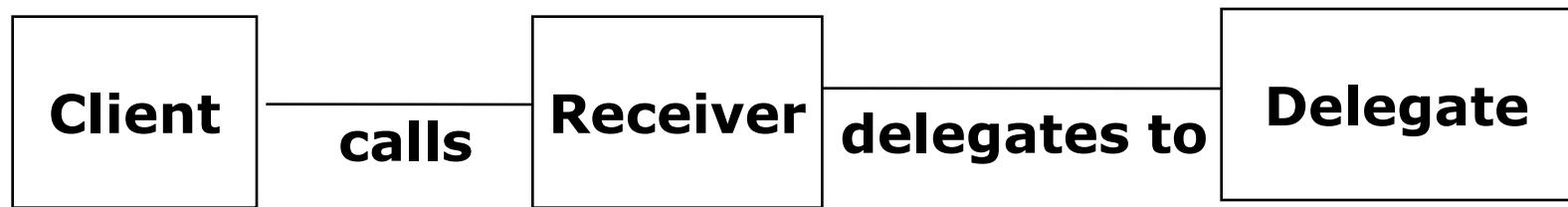
# Delegation instead of Implementation Inheritance

- *Inheritance: Extending a Base class by a new operation or overwriting an operation.*
- *Delegation: Catching an operation and sending it to another object.*
- *Which of the following models is better?*



# Delegation

- *Delegation is a way of making composition as powerful for reuse as inheritance*
- *In delegation two objects are involved in handling a request from a Client*
  - *The Receiver object delegates operations to the Delegate object*
  - *The Receiver object makes sure, that the Client does not misuse the Delegate object.*



# Comparison: Delegation vs Implementation Inheritance

- *Delegation*
  - ☺ *Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)*
  - ☹ *Inefficiency: Objects are encapsulated.*
- *Inheritance*
  - ☺ *Straightforward to use*
  - ☺ *Supported by many programming languages*
  - ☺ *Easy to implement new functionality*
  - ☹ *Inheritance exposes a subclass to the details of its parent class*
  - ☹ *Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)*

# Comparison: Delegation v. Inheritance

- *Code-Reuse can be done by delegation as well as inheritance*
- *Delegation*
  - *Flexibility: Any object can be replaced at run time by another one*
  - *Inefficiency: Objects are encapsulated*
- *Inheritance*
  - *Straightforward to use*
  - *Supported by many programming languages*
  - *Easy to implement new functionality*
  - *Exposes a subclass to details of its super class*
  - *Change in the parent class requires recompilation of the subclass.*

# Recall: Implementation Inheritance v. Specification-Inheritance

- *Implementation Inheritance: The combination of inheritance and implementation*
  - *The Interface of the super class is completely inherited*
  - *Implementations of methods in the super class ("Reference implementations") are inherited by any subclass*
- *Specification Inheritance: The combination of inheritance and specification*
  - *The super class is an abstract class*
    - *Implementations of the super class (if there are any) are not inherited*
  - *The Interface of the super class is completely inherited*

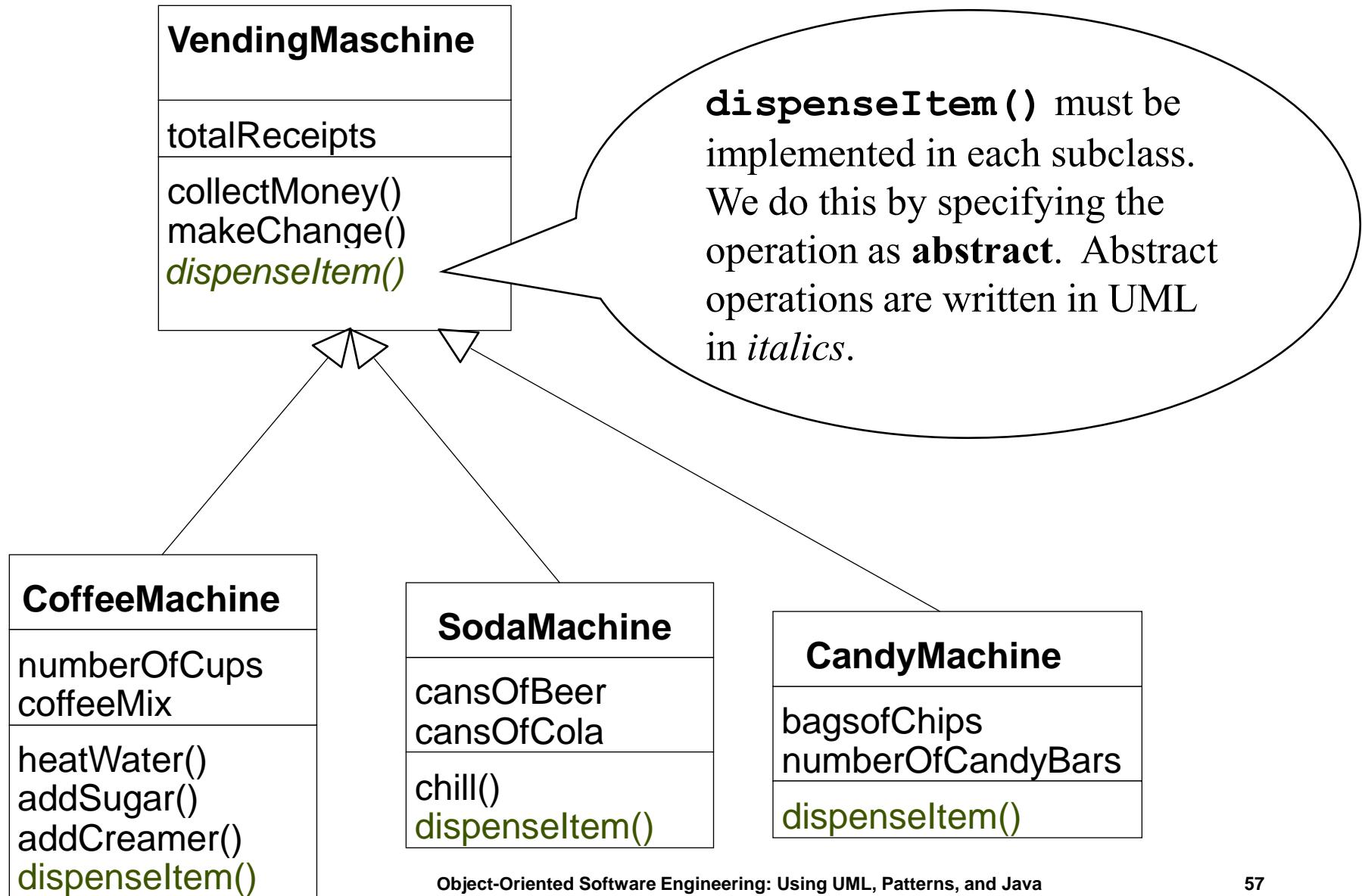
# Outline of Today

- ✓ *Reuse examples*
  - ✓ *Reuse of code, interfaces and existing classes*
- ✓ *White box and black box reuse*
- ✓ *Object design leads to new classes*
- ✓ *The use of inheritance*
- ✓ *Implementation vs. specification inheritance*
- ✓ *Delegation vs. Inheritance*
- ➡ *Abstract classes and abstract methods*
  - *Overwriting methods*
  - *Contraction: Bad example of inheritance*
  - *Meta model for inheritance*
  - *Frameworks and components*
  - *Documenting the object design.*

# Abstract Methods and Abstract Classes

- *Abstract method:*
  - A method with a signature but without an implementation (also called abstract operation)
- *Abstract class:*
  - A class which contains at least one abstract method is called abstract class
- *Interface:* An abstract class which has only abstract methods
  - An interface is primarily used for the specification of a system or subsystem. The implementation is provided by a subclass or by other mechanisms.

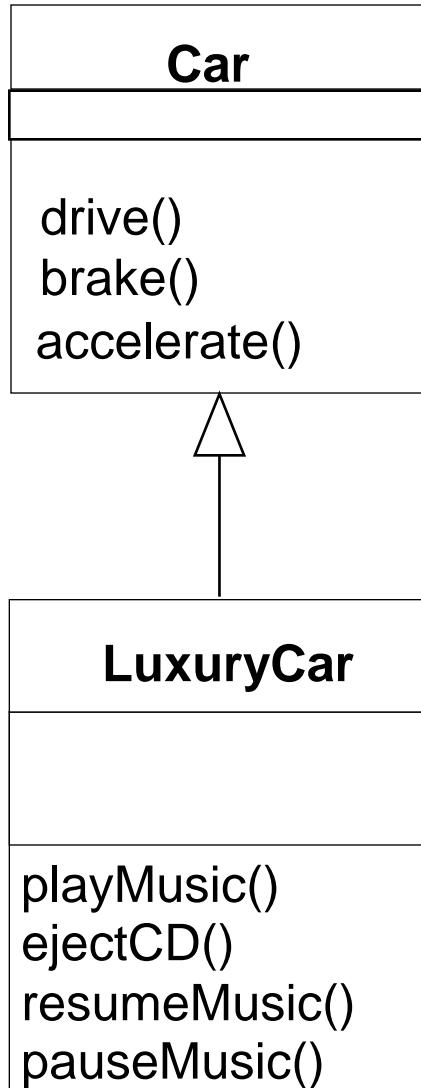
# Example of an Abstract Method



# Rewriteable Methods and Strict Inheritance

- *Rewriteable Method: A method which allow a reimplementation.*
  - *In Java methods are rewriteable by default, i.e. there is no special keyword.*
- *Strict inheritance*
  - *The subclass can only add new methods to the superclass, it cannot over write them*
  - *If a method cannot be overwritten in a Java program, it must be prefixed with the keyword final.*

# Strict Inheritance



## Superclass:

```
public class Car {  
    public final void drive() {...}  
    public final void brake() {...}  
    public final void accelerate()  
    {...}  
}
```

## Subclass:

```
public class LuxuryCar extends Car {  
    public void playMusic() {...}  
    public void ejectCD() {...}  
    public void resumeMusic() {...}  
    public void pauseMusic() {...}  
}
```

# Example: Strict Inheritance and Rewriteable Methods

## Original Java-Code:

```
class Device {  
    int serialnr;  
    public final void help() {....}  
    public void setSerialNr(int n) {  
        serialnr = n;  
    }  
}  
  
class Valve extends Device {  
    Position s;  
    public void on() {  
        ....  
    }  
}
```

help() not  
overwritable

setSerialNr()  
overwritable

# Example: Overwriting a Method

## Original Java-Code:

```
class Device {  
    int serialnr;  
    public final void help() {....}  
    public void setSerialNr(int n) {  
        serialnr = n;  
    }  
}  
  
class Valve extends Device {  
    Position s;  
    public void on() {  
    } ....  
}
```

## New Java-Code :

```
class Device {  
    int serialnr;  
    public final void help() {....}  
    public void setSerialNr(int n) {  
        serialnr = n;  
    }  
}
```

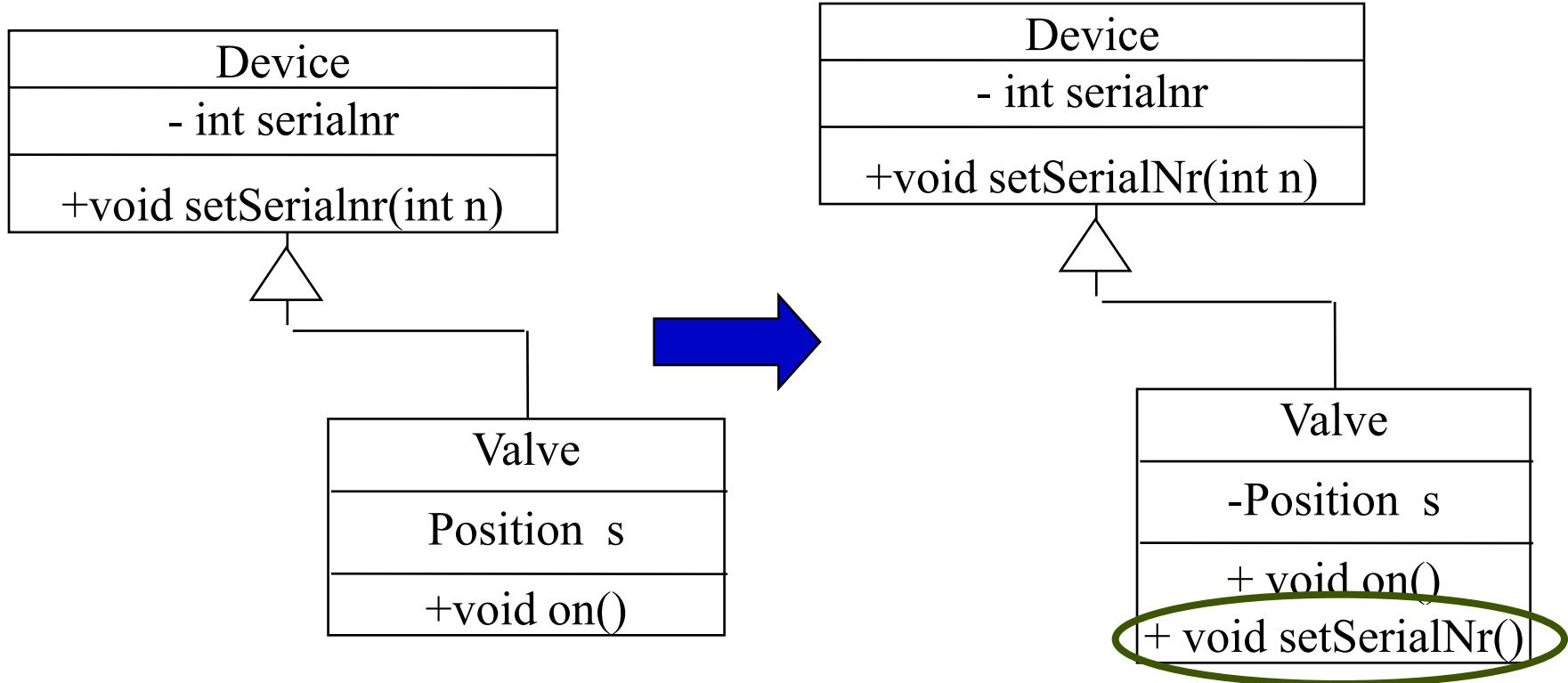
## class Valve extends Device {

```
Position s;  
public void on() {  
    ...  
}
```

```
public void setSerialNr(int n) {  
    serialnr = n + s.serialnr;  
}
```

```
} // class Valve
```

# UML Class Diagram



# Rewriteable Methods: Usually implemented with Empty Body

```
class Device {  
    int serialnr;  
    public void setSerialNr(int n) {}  
}  
  
class Valve extends Device {  
    Position s;  
    public void on() {  
        ....  
    }  
    public void setSerialNr(int n) {  
        seriennr = n + s.serialnr;  
    }  
} // class Valve
```

I expect, that the method  
setSerialNr () will be  
overwritten. I only write an  
empty body

Overwriting of the method  
setSerialNr () of Class  
Device

# Bad Use of Overwriting Methods

*One can overwrite the operations of a superclass with completely new meanings.*

*Example:*

```
Public class SuperClass {  
    public int add (int a, int b) { return a+b; }  
    public int subtract (int a, int b) { return a-b; }  
}  
  
Public class SubClass extends SuperClass {  
    public int add (int a, int b) { return a-b; }  
    public int subtract (int a, int b) { return a+b; }  
}
```

- *We have redefined addition as subtraction and subtraction as addition!!*

# Bad Use of Implementation Inheritance

- We have delivered a car with software that allows to operate an on-board stereo system
  - A customer wants to have software for a cheap stereo system to be sold by a discount store chain
- Dialog between project manager and developer:
  - Project Manager:
    - „Reuse the existing car software. Don't change this software, make sure there are no hidden surprises. There is no additional budget, deliver tomorrow!“
  - Developer:
    - „OK, we can easily create a subclass BoomBox inheriting the operations from the existing Car software“
    - „And we overwrite all method implementations from Car that have nothing to do with playing music with empty bodies!“

# What we have and what we want

## Auto

engine  
windows  
musicSystem

brake()  
accelerate()  
playMusic()  
ejectCD()  
resumeMusic()  
pauseMusic()

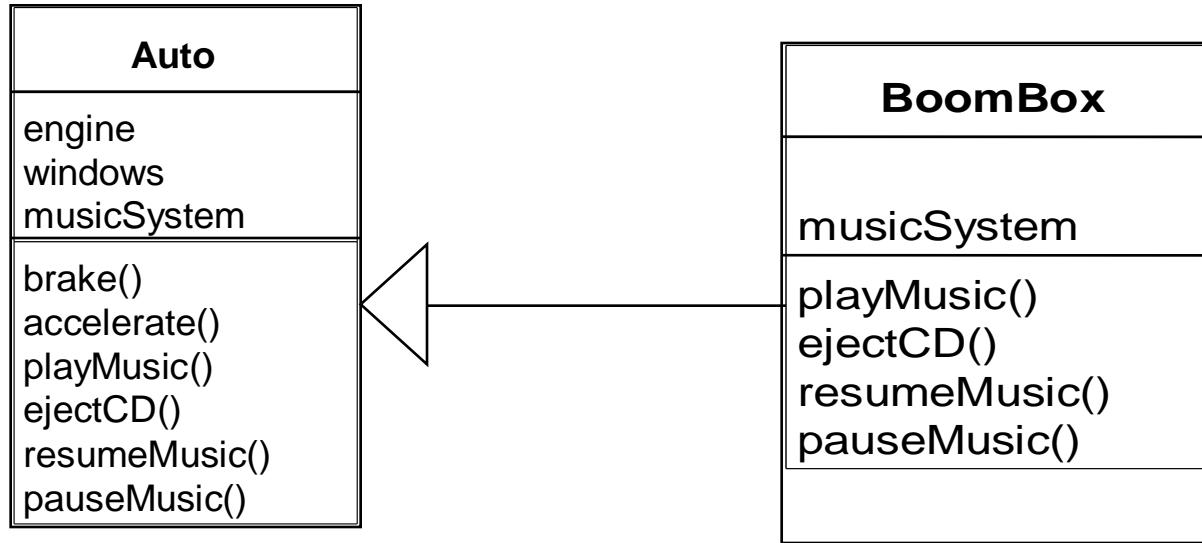
## BoomBox

musicSystem

playMusic()  
ejectCD()  
resumeMusic()  
pauseMusic()

**New Abstraction!**

# What we do to save money and time



## Existing Class:

```
public class Auto {  
    public void drive() {...}  
    public void brake() {...}  
    public void accelerate() {...}  
    public void playMusic() {...}  
    public void ejectCD() {...}  
    public void resumeMusic() {...}  
    public void pauseMusic() {...}  
}
```

## Boombox:

```
public class Boombox  
extends Auto {  
    public void drive() {};  
    public void brake() {};  
    public void accelerate()  
{};  
}
```

# Contraction

- **Contraction:** *Implementations of methods in the super class are overwritten with empty bodies in the subclass to make the super class operations "invisible"*
- *Contraction is a special type of inheritance*
- *It should be avoided at all costs, but is used often.*

# Contraction must be avoided by all Means

*A contracted subclass delivers the desired functionality expected by the client, but:*

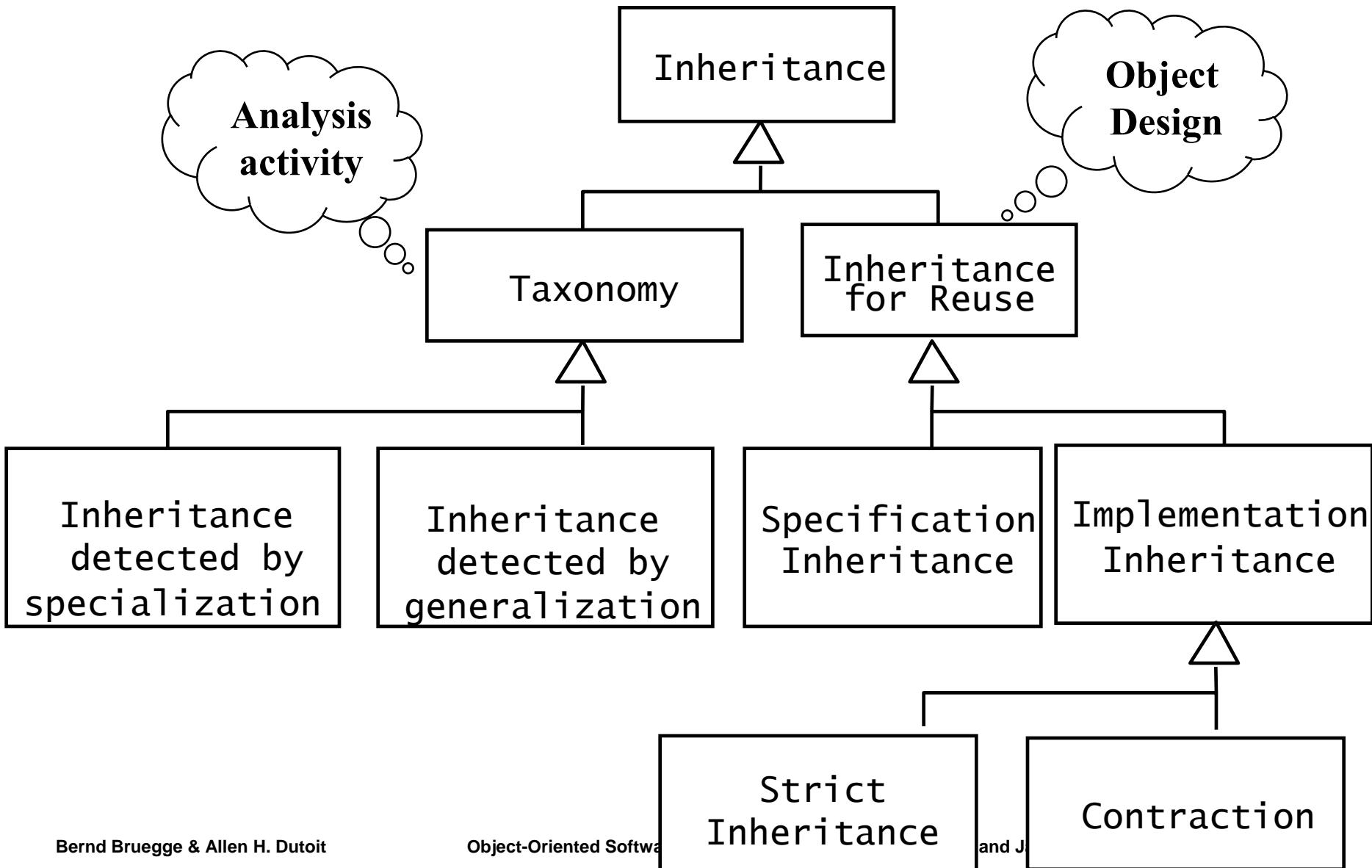
- *The interface contains operations that make no sense for this class*
- *What is the meaning of the operation brake() for a BoomBox?*

*The subclass does not fit into the taxonomy*

*A BoomBox is not a special form of Auto*

- *The subclass violates Liskov's Substitution Principle:*
  - *I cannot replace Auto with BoomBox to drive to work.*

# Revised Metamodel for Inheritance



# Frameworks

- A *framework* is a reusable partial application that can be specialized to produce custom applications.
- The key benefits of frameworks are reusability and extensibility:
  - *Reusability* leverages of the application domain knowledge and prior effort of experienced developers
  - *Extensibility* is provided by hook methods, which are overwritten by the application to extend the framework.

# Classification of Frameworks

- *Frameworks can be classified by their position in the software development process:*
  - *Infrastructure frameworks*
  - *Middleware frameworks*
- *Frameworks can also be classified by the techniques used to extend them:*
  - *Whitebox frameworks*
  - *Blackbox frameworks*

# Frameworks in the Development Process

- *Infrastructure frameworks* aim to simplify the software development process
  - Used internally, usually not delivered to a client.
- *Middleware frameworks* are used to integrate existing distributed applications
  - Examples: MFC, DCOM, Java RMI, WebObjects, WebSphere, WebLogic Enterprise Application [BEA].
- *Enterprise application frameworks* are application specific and focus on domains
  - Example of application domains: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities.

# White-box and Black-box Frameworks

- *White-box frameworks:*
  - Extensibility achieved through *inheritance* and *dynamic binding*.
  - Existing functionality is extended by subclassing framework base classes and overriding specific methods (so-called hook methods)
- *Black-box frameworks:*
  - Extensibility achieved by defining interfaces for components that can be plugged into the framework.
  - Existing functionality is reused by defining components that conform to a particular interface
  - These components are integrated with the framework via *delegation*.

# Class libraries vs. Frameworks

- *Class Library:*
  - *Provide a smaller scope of reuse*
  - *Less domain specific*
  - *Class libraries are passive; no constraint on the flow of control*
- *Framework:*
  - *Classes cooperate for a family of related applications.*
  - *Frameworks are active; they affect the flow of control.*

# Components vs. Frameworks

- *Components:*
  - *Self-contained instances of classes*
  - *Plugged together to form complete applications*
  - *Can even be reused on the binary code level*
    - *The advantage is that applications do not have to be recompiled when components change*
- *Framework:*
  - *Often used to develop components*
  - *Components are often plugged into blackbox frameworks.*

# Documenting the Object Design

- *Object design document (ODD)*
  - = *The Requirements Analysis Document (RAD) plus...*
    - ... additions to object, functional and dynamic models (from the solution domain)*
    - ... navigational map for object model*
    - ... Specification for all classes (use Javadoc)*

# Documenting Object Design: ODD Conventions

- *Each subsystem in a system provides a service*
  - *Describes the set of operations provided by the subsystem*
- *Specification of the service operations*
  - *Signature: Name of operation, fully typed parameter list and return type*
  - *Abstract: Describes the operation*
  - *Pre: Precondition for calling the operation*
  - *Post: Postcondition describing important state after the execution of the operation*
- *Use JavaDoc and Contracts for the specification of service operations*
  - *Contracts are covered in the next lecture.*

# Package it all up

- *Pack up design into discrete units that can be edited, compiled, linked, reused*
- *Construct physical modules*
  - *Ideally use one package for each subsystem*
  - *System decomposition might not be good for implementation.*
- *Two design principles for packaging*
  - *Minimize coupling:*
    - *Classes in client-supplier relationships are usually loosely coupled*
    - *Avoid large number of parameters in methods to avoid strong coupling (should be less than 4-5)*
    - *Avoid global data*
  - *Maximize cohesion: Put classes connected by associations into one package.*

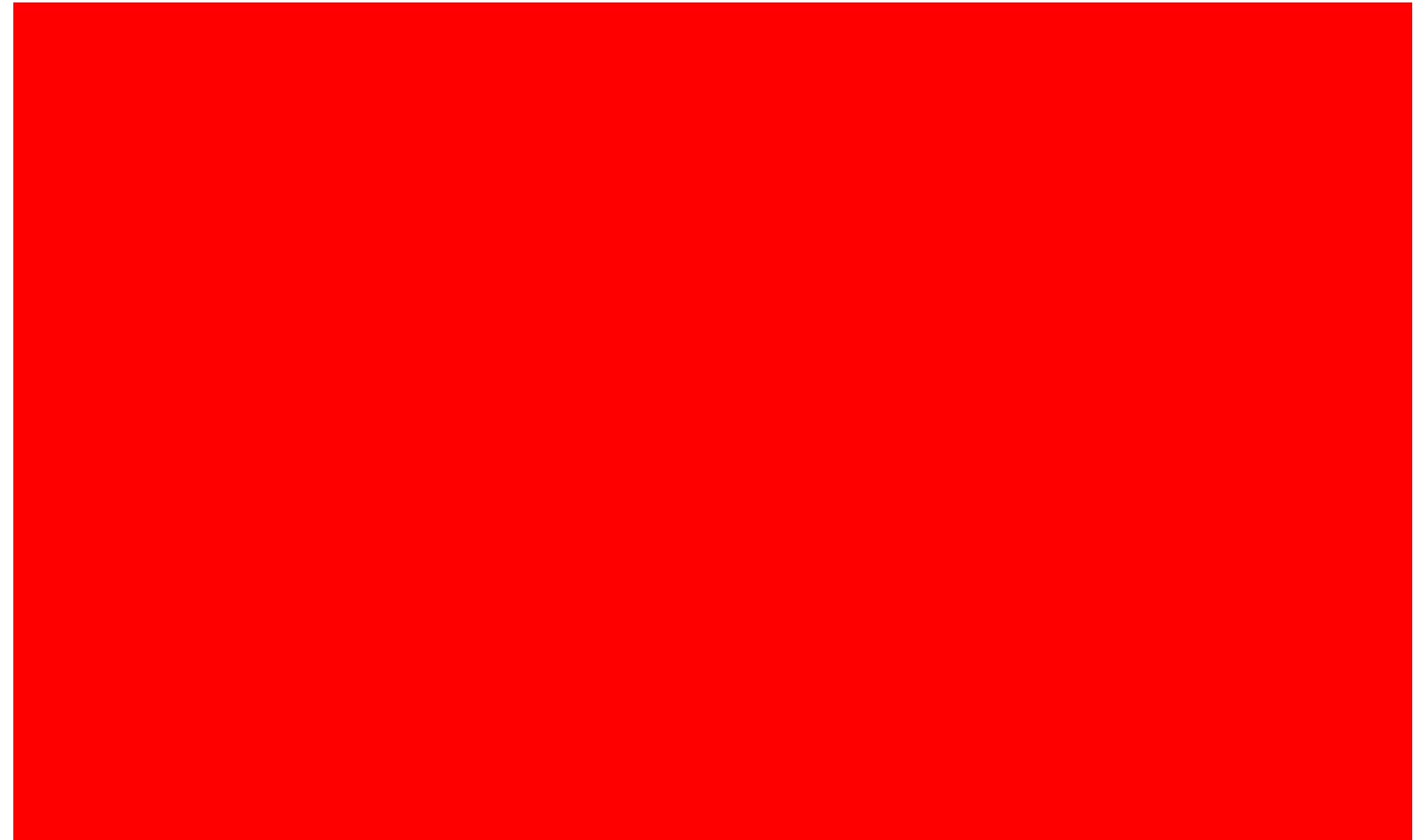
# Packaging Heuristics

- *Each subsystem service is made available by one or more interface objects within the package*
- *Start with one interface object for each subsystem service*
  - Try to limit the number of interface operations (7+-2)
- *If an interface object has too many operations, reconsider the number of interface objects*
- *If you have too many interface objects, reconsider the number of subsystems*
- *Interface objects vs Java interface:*
  - **Interface object:** Used during requirements analysis, system design, object design. Denotes a service or API
  - **Java interface:** Used during implementation in Java (May or may not implement an interface object).

# Summary

- *Object design closes the gap between the requirements and the machine*
- *Object design adds details to the requirements analysis and makes implementation decisions*
- *Object design activities include:*
  - ✓ *Identification of Reuse*
  - ✓ *Identification of Inheritance and Delegation opportunities*
  - ✓ *Component selection*
    - *Interface specification (Next lecture)*
    - *Object model restructuring*
    - *Object model optimization*
- *Object design is documented in the Object Design Document (ODD).*

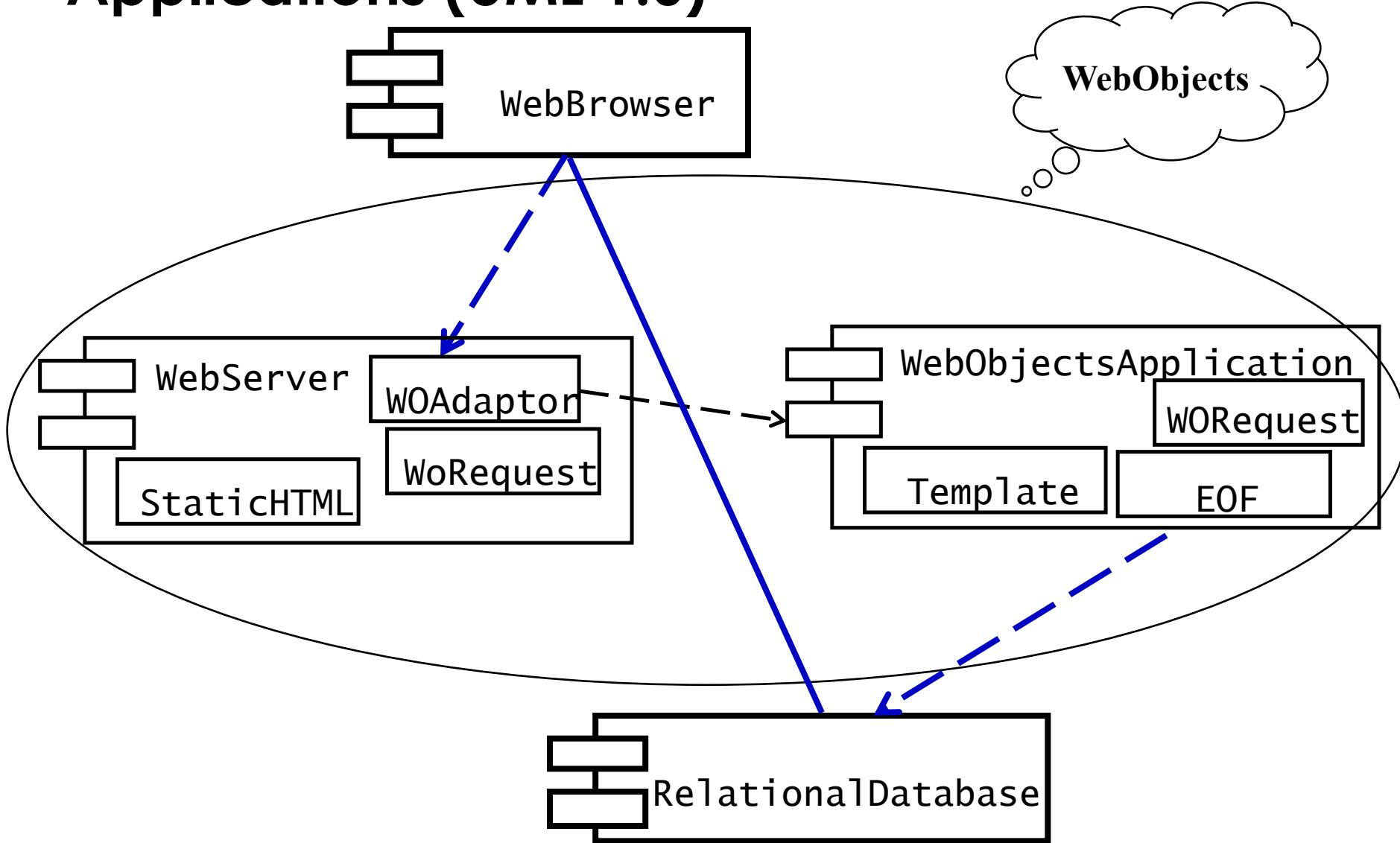
# Backup Slides



# Reuse

- *Main goal:*
  - *Reuse knowledge from previous experience to current problem*
  - *Reuse functionality already available*
- *Composition (also called Black Box Reuse)*
  - *New functionality is obtained by aggregation*
  - *The new object with more functionality is an aggregation of existing components*
- *Inheritance (also called White-box Reuse)*
  - *New functionality is obtained by inheritance.*
- *Three ways to get new functionality:*
  - *Implementation inheritance*
  - *Interface inheritance*
  - *Delegation*

# Example: Framework for Building Web Applications (UML 1.0)



# Customization Projects are like Advanced Jigsaw Puzzles



**Design Patterns!**

<http://www.puzzlehouse.com/>

# Object Design Activities

## 1. Reuse: Identification of existing solutions

- Use of inheritance
- Off-the-shelf components and additional solution objects
- Design patterns

## 2. Interface specification

- Describes precisely each class interface

## 3. Object model restructuring

- Transforms the object design model to improve its understandability and extensibility

## 4. Object model optimization

- Transforms the object design model to address performance criteria such as response time or memory utilization.

**Object  
Design**

**Mapping  
Models to  
Code**

# Chapter 8, Object Design: Object Constraint Language



# Outline of the Lecture

- *OCL*
- *Simple predicates*
- *Preconditions*
- *Postconditions*
- *Contracts*
- *Sets, Bags, and Sequences*

# OCL: Object Constraint Language

- *Formal language for expressing constraints over a set of objects and their attributes*
- *Part of the UML standard*
- *Used to write constraints that cannot otherwise be expressed in a diagram*
- *Declarative*
  - *No side effects*
  - *No control flow*
- *Based on Sets and Multi Sets*

# OCL Basic Concepts

- *OCL expressions*
  - *Return **True** or **False***
  - *Are evaluated in a specified context, either a class or an operation*
  - *All constraints apply to all instances*

# OCL Simple Predicates

*Example:*

**context** Tournament **inv**:

self.getMaxNumPlayers() > 0

*In English:*

*"The maximum number of players in any tournament should be a positive number."*

**Notes:**

- “self” denotes all instances of “Tournament”
- OCL uses the same dot notation as Java.

# OCL Preconditions

*Example:*

```
context Tournament::acceptPlayer(p) pre:  
    not self.isPlayerAccepted(p)
```

*In English:*

*"The acceptPlayer(p) operation can only be invoked if player p has not yet been accepted in the tournament."*

*Notes:*

- *The context of a precondition is an operation*
- *isPlayerAccepted(p) is an operation defined by the class Tournament*

# OCL Postconditions

*Example:*

**context** Tournament::acceptPlayer(p) **post**:

```
self.getNumPlayers() =  
    self@pre.getNumPlayers() + 1
```

*In English:*

*"The number of accepted player in a tournament increases by one after the completion of acceptPlayer()"*

**Notes:**

- *self@pre denotes the state of the tournament before the invocation of the operation.*
- *Self denotes the state of the tournament after the completion of the operation.*

# OCL Contract for acceptPlayer() in Tournament

```
context Tournament::acceptPlayer(p) pre:  
    not isPlayerAccepted(p)
```

```
context Tournament::acceptPlayer(p) pre:  
    getNumPlayers() < getMaxNumPlayers()
```

```
context Tournament::acceptPlayer(p) post:  
    isPlayerAccepted(p)
```

```
context Tournament::acceptPlayer(p) post:  
    getNumPlayers() = @pre.getNumPlayers() + 1
```

# OCL Contract for removePlayer() in Tournament

```
context Tournament::removePlayer(p) pre:  
    isPlayerAccepted(p)
```

```
context Tournament::removePlayer(p) post:  
    not isPlayerAccepted(p)
```

```
context Tournament::removePlayer(p) post:  
    getNumPlayers() = @pre.getNumPlayers() - 1
```

# Java Implementation of Tournament class (Contract as a set of JavaDoc comments)

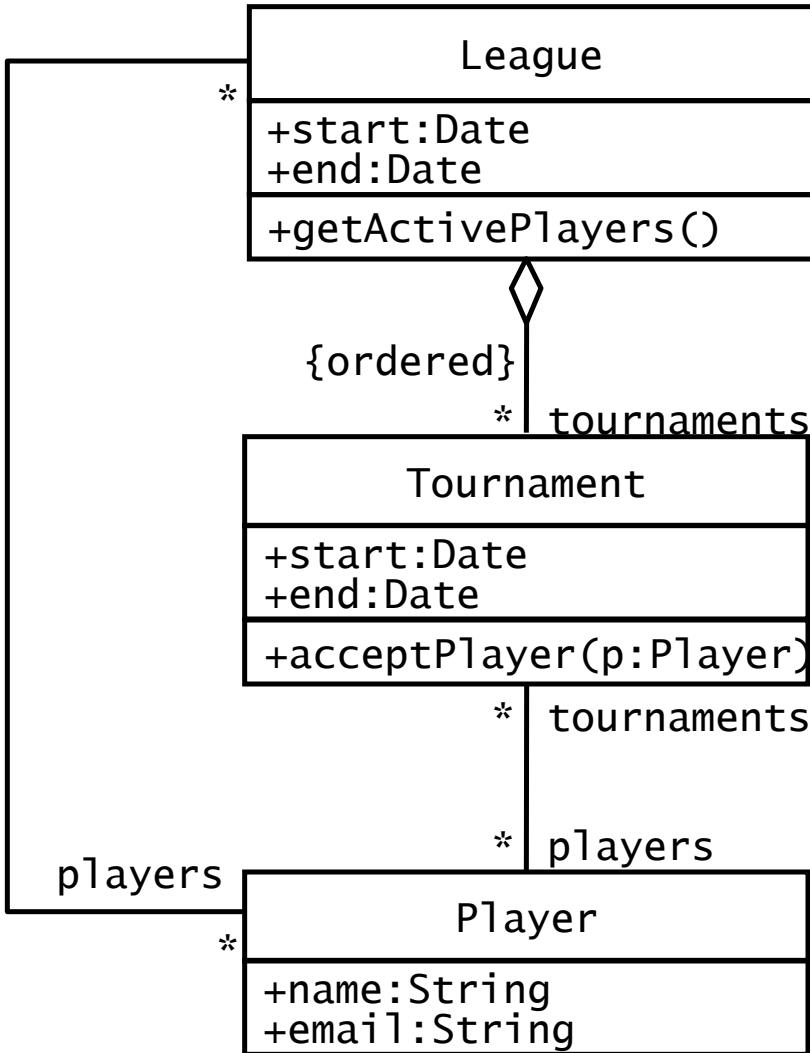
```
public class Tournament {  
    /** The maximum number of players  
     * is positive at all times.  
     * @invariant maxNumPlayers > 0  
     */  
    private int maxNumPlayers;  
  
    /** The players List contains  
     * references to Players who are  
     * are registered with the  
     * Tournament. */  
    private List<Player> players;  
  
    /** Returns the current number of  
     * players in the tournament. */  
    public int getNumPlayers() {...}  
  
    /** Returns the maximum number of  
     * players in the tournament. */  
    public int getMaxNumPlayers() {...}  
  
    /** The acceptPlayer() operation  
     * assumes that the specified  
     * player has not been accepted  
     * in the Tournament yet.  
     * @pre !isPlayerAccepted(p)  
     * @pre getNumPlayers()<maxNumPlayers  
     * @post isPlayerAccepted(p)  
     * @post getNumPlayers() =  
     *      @pre.getNumPlayers() + 1  
     */  
    public void acceptPlayer (Player p) {...}  
  
    /** The removePlayer() operation  
     * assumes that the specified player  
     * is currently in the Tournament.  
     * @pre isPlayerAccepted(p)  
     * @post !isPlayerAccepted(p)  
     * @post getNumPlayers() =  
     *      @pre.getNumPlayers() - 1  
     */  
    public void removePlayer(Player p) {...}  
}
```

# **Constraints can involve more than one class**

**How do we specify constraints on  
on a group of classes?**

Starting from a specific class in the UML class diagram, we navigate the associations in the class diagram to refer to the other classes and their properties (attributes and Operations).

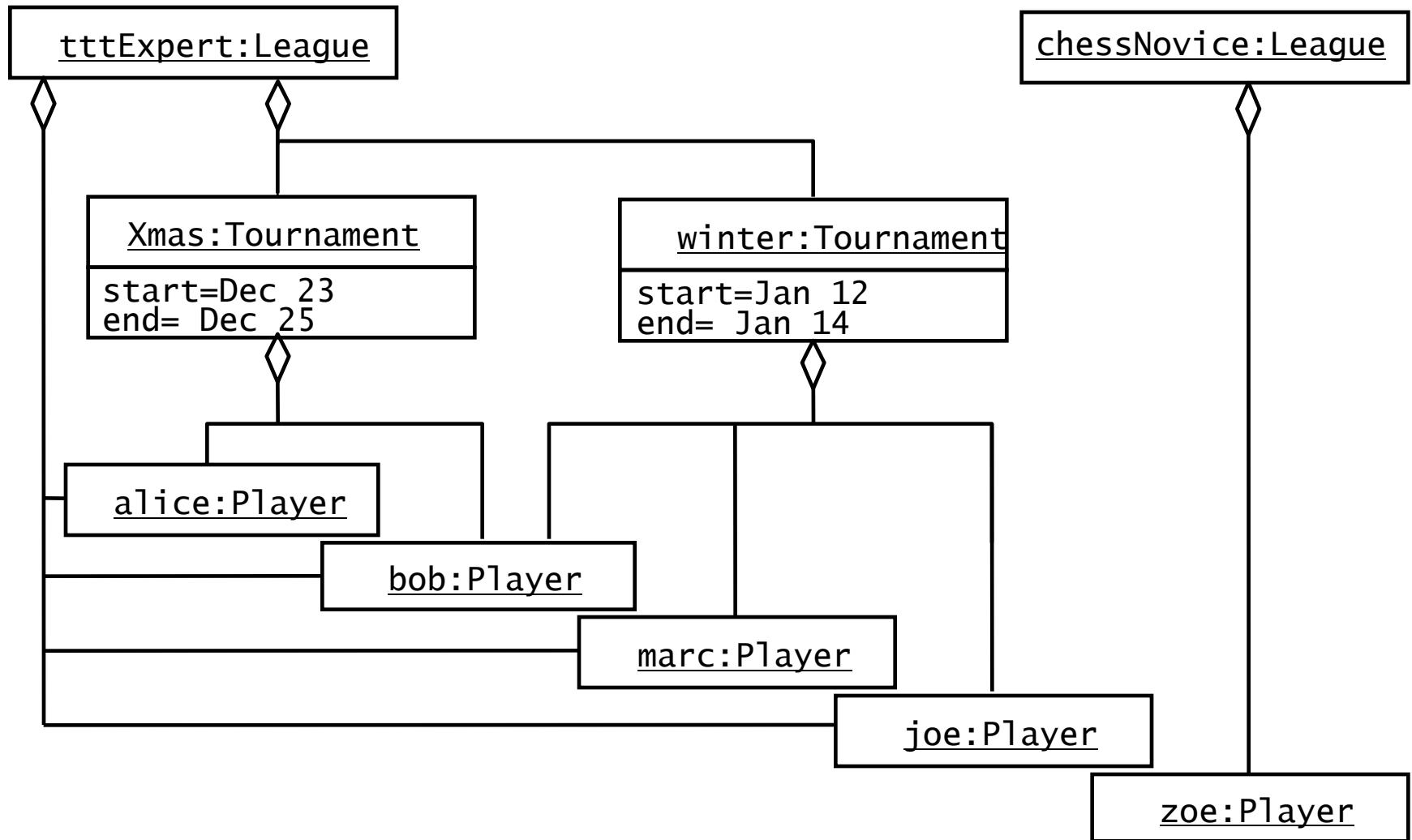
# Example from ARENA: League, Tournament and Player



*Constraints:*

1. *A Tournament's planned duration must be under one week.*
2. *Players can be accepted in a Tournament only if they are already registered with the corresponding League.*
3. *The number of active Players in a League are those that have taken part in at least one Tournament of the League.*

# Instance Diagram: 2 Leagues , 5 Players, 2 Tournaments

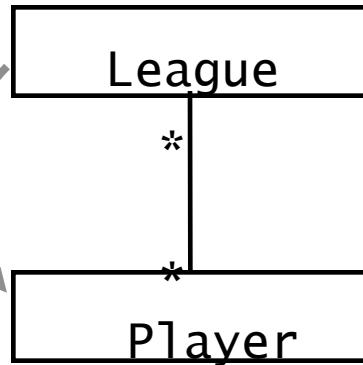


# 3 Types of Navigation through a Class Diagram

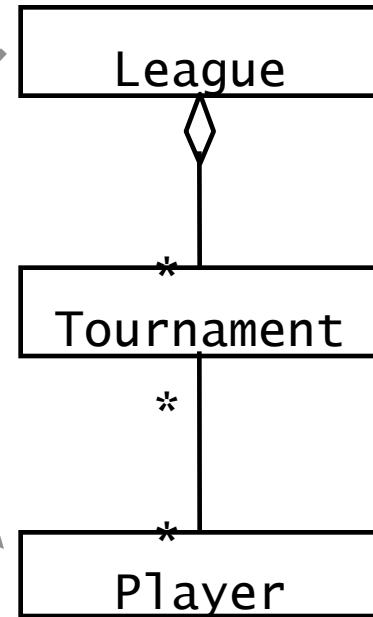
1. Local attribute



2. Directly related class



3. Indirectly related class



*Any constraint for an arbitrary UML class diagram can be specified using only a combination of these 3 navigation types!*

# Specifying the Model Constraints in OCL

*Local attribute navigation*

```
context Tournament inv:  
  end - start <= 7
```



*Directly related class navigation*

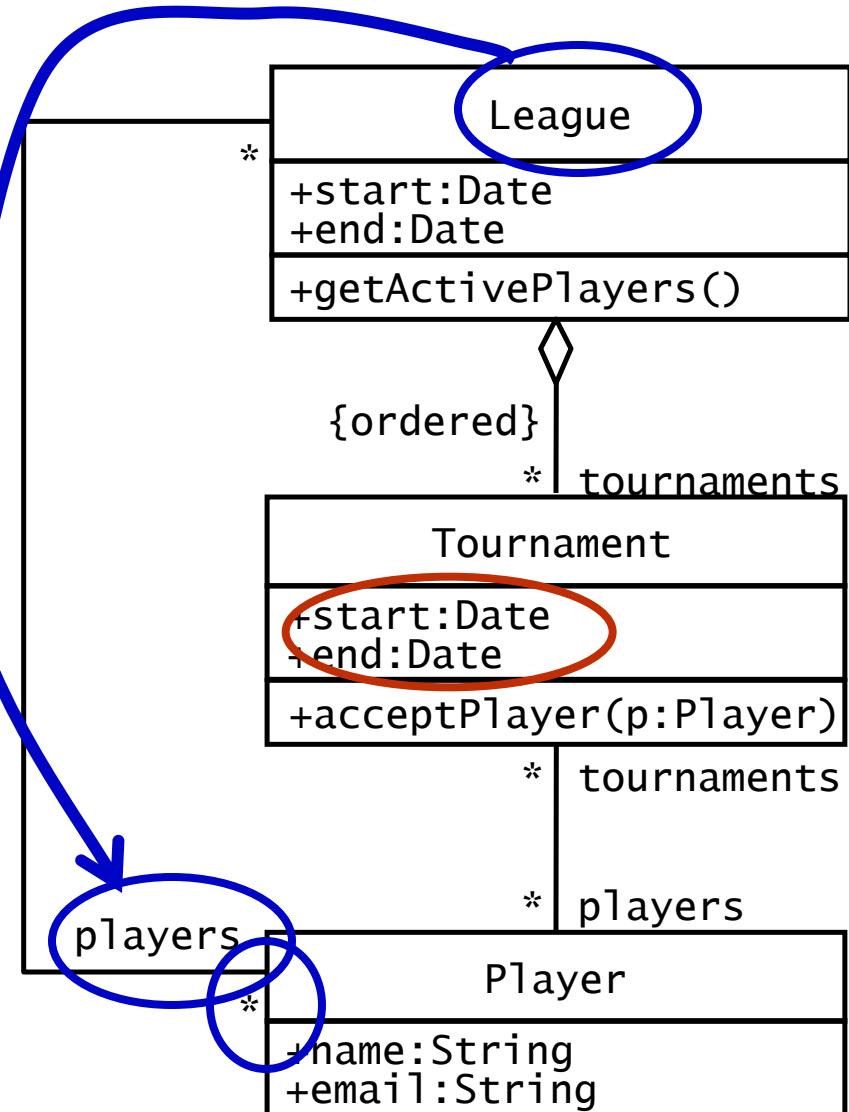


context

```
Tournament::acceptPlayer(p)
```

pre:

```
league.players->includes(p)
```



# OCL Sets, Bags and Sequences

- Sets, Bags and Sequences are predefined in OCL and subtypes of **Collection**. OCL offers a large number of predefined operations on collections. They are all of the form:

collection->operation(arguments)

# OCL-Collection

- *The OCL-Type Collection is the generic superclass of a collection of objects of Type T*
- *Subclasses of Collection are*
  - *Set: Set in the mathematical sense. Every element can appear only once*
  - *Bag: A collection, in which elements can appear more than once (also called multiset)*
  - *Sequence: A multiset, in which the elements are ordered*
- *Example for Collections:*
  - *Set(Integer): a set of integer numbers*
  - *Bag(Person): a multiset of persons*
  - *Sequence(Customer): a sequence of customers*

# OCL-Operations for OCL-Collections (1)

**size: Integer**

*Number of elements in the collection*

► **includes (o:OclAny) : Boolean**

*True, if the element o is in the collection*

**count (o:OclAny) : Integer**

*Counts how many times an element is contained in the collection*

**isEmpty: Boolean**

*True, if the collection is empty*

**notEmpty: Boolean**

*True, if the collection is not empty*

*The OCL-Type **OclAny** is the most general OCL-Type*

# OCL-Operations for OCL-Collections(2)

**union (c1 : Collection)**

*Union with collection c1*

**intersection (c2 : Collection)**

*Intersection with Collection c2 (contains only elements, which appear in the collection as well as in collection c2 auftreten)*

**including (o : OclAny)**

*Collection containing all elements of the Collection and element o*

**select (expr : OclExpression)**

*Subset of all elements of the collection, for which the OCL-expression **expr** is true*

# How do we get OCL-Collections?

- A *collection* can be generated by explicitly enumerating the elements
- A *collection* can be generated by navigating along one or more 1-N associations
  - Navigation along a single 1:n association yields a *Set*
  - Navigation along a couple of 1:n associations yields a *Bag* (*Multiset*)
  - Navigation along a single 1:n association labeled with the constraint {ordered } yields a *Sequence*

# Navigation through a 1:n-Association

*Example: A Customer should not have more than 4 cards*

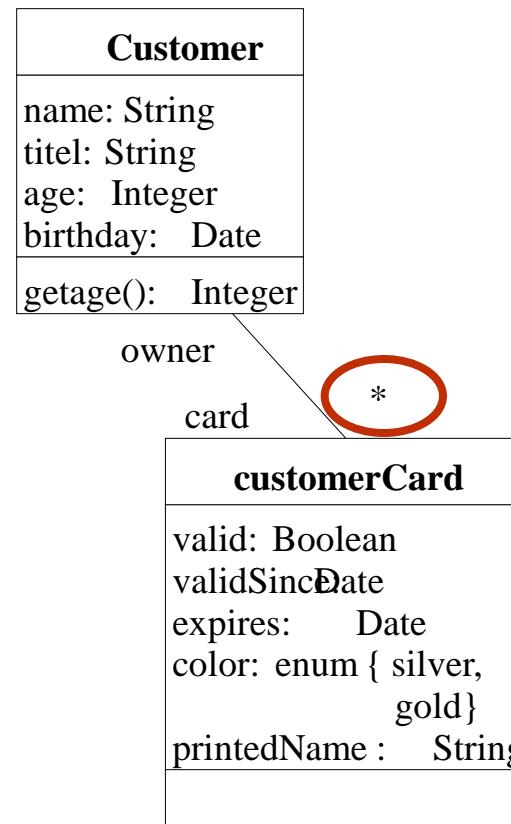
```
context Customer inv:  
    card->size <= 4
```

card denotes  
a set of  
**customercards**

Alternative writing style

Customer

```
card->size <= 4
```



# Navigation through several 1:n-Associations

Example:

programPartner

nrcustomer = bonusprogram.customer->size

**Customer** denotes a multiset of **customer**

**bonusprogram**  
denotes a set of  
**Bonusprograms**

**programPartner**

nrcustomer: Integer

1..\*

1..\*

Bonusprogram
register(k: Customer)

program

\*

**Customer**

name: String  
titel: String  
age: Integer  
birthday: Datum  
getage(): Integer

\*

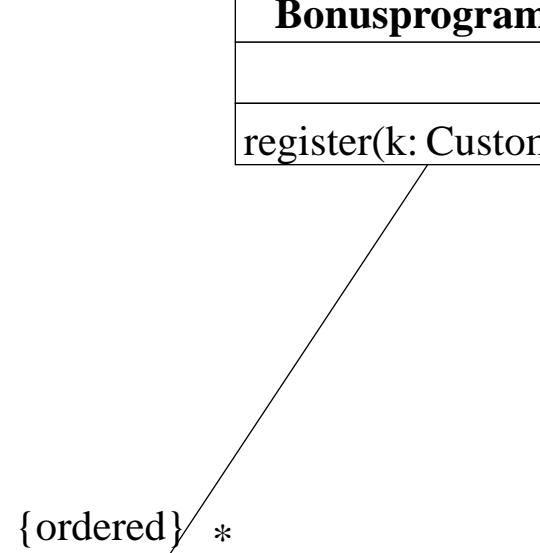
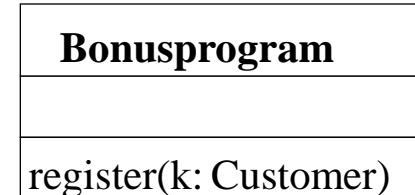
# Navigation through a constrained Association

- *Navigation through an association with the constraint {ordered} yields a sequence.*
- *Example:*

Bonusprogram

**level->size = 2**

**level** denotes a  
sequence von **levels**



# Conversion between OCL-Collections

- *OCL offers operations to convert OCL-Collections:*

*asSet*

*Transforms a multiset or sequence into a set*

*asBag*

*transforms a set or sequence into a multiset*

*asSequence*

*transforms a set or multiset into a sequence.*

# Example of a Conversion

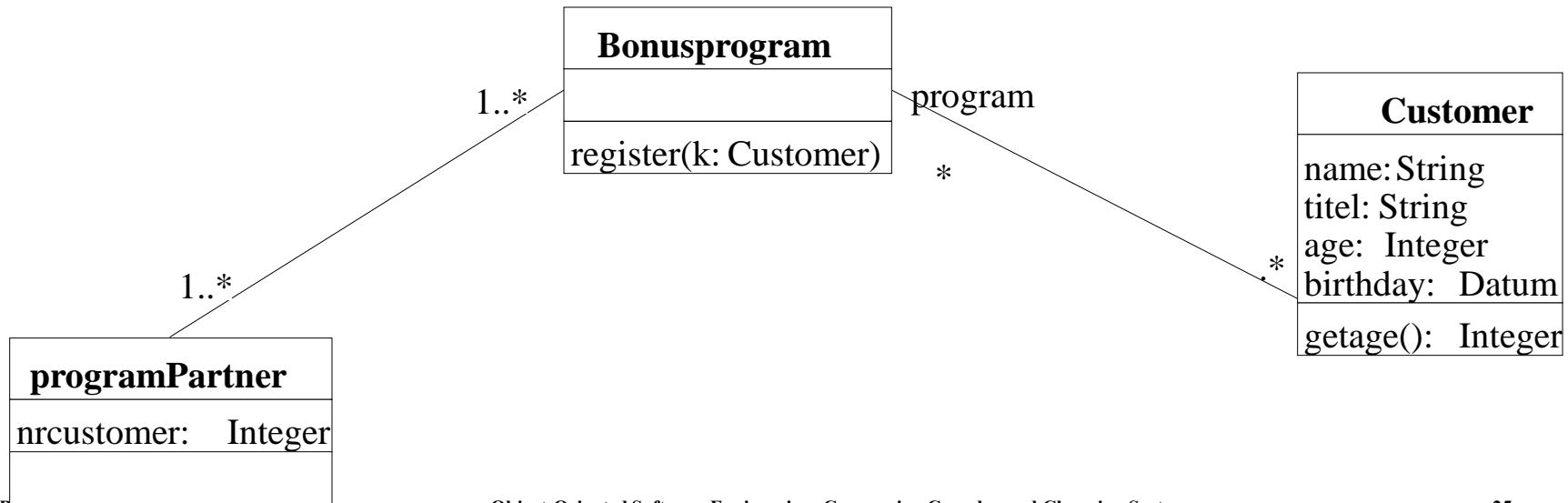
## programPartner

```
nrcustomer = bonusprogram.Customer->size
```

*This expression may contain customer multiple times, we can get the number of unique customers as follows:*

## programPartner

```
nrcustomer = bonusprogram.Customer->asset->size
```



# Specifying the Model Constraints: Using `asSet`

*Local attribute navigation*

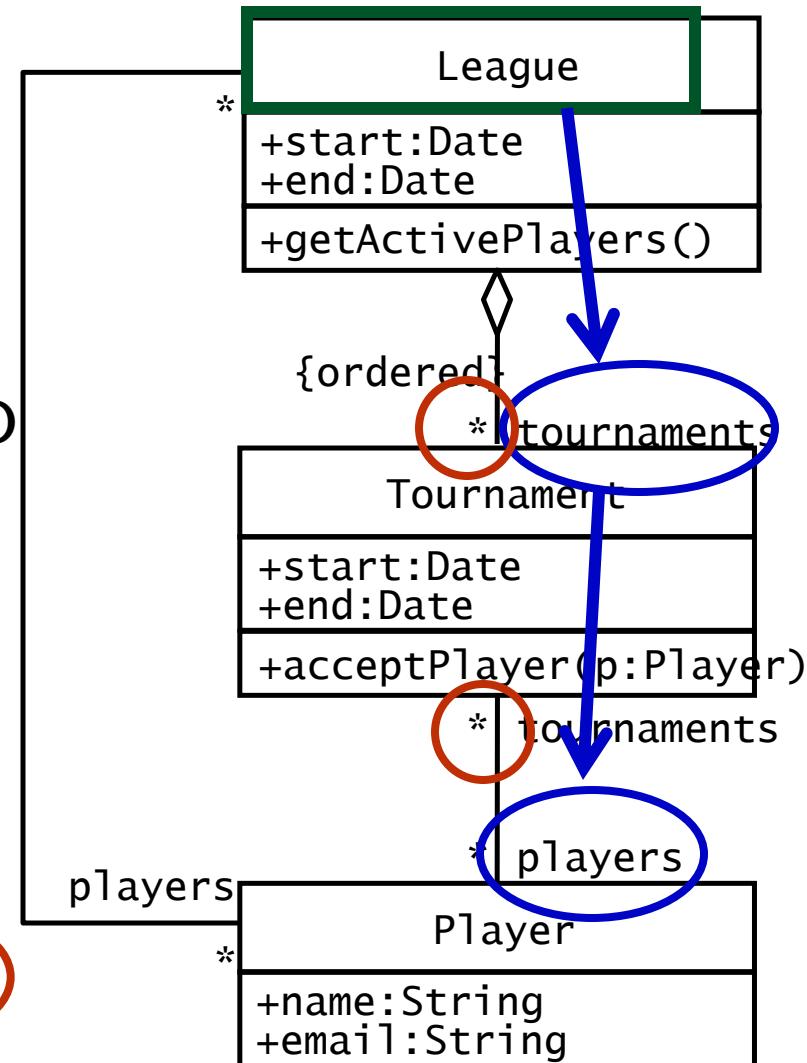
```
context Tournament inv:  
end - start <= Calendar.WEEK
```

*Directly related class navigation*

```
context Tournament::acceptPlayer(p)  
pre:  
league.players->includes(p)
```

*Indirectly related class navigation*

```
context League::getActivePlayers  
post:  
result=tournaments.players->asSet
```



# Evaluating OCL Expressions

*The value of an OCL expression is an object or a collection of objects.*

- *Multiplicity of the association-end is 1*
  - *The value of the OCL expression is a single object*
- *Multiplicity is 0..1*
  - *The result is an empty set if there is no object, otherwise a single object*
- *Multiplicity of the association-end is \**
  - *The result is a collection of objects*
    - *By default, the navigation result is a Set*
    - *When the association is {ordered}, the navigation results in a Sequence*
    - *Multiple "1-Many" associations result in a Bag*

# Additional Readings

- *J.B. Warmer, A.G. Kleppe*  
*The Object Constraint Language: Getting your Models ready for MDA*, Addison-Wesley, 2nd edition, 2003
- *B. Meyer*  
*Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.
- *B. Meyer*,  
*Design by Contract: The Lesson of Ariane*, Computer, IEEE, Vol. 30, No. 2, pp. 129-130, January 1997.  
<http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html>
- *C. A. R. Hoare*,  
*An axiomatic basis for computer programming*.  
*Communications of the ACM*, 12(10):576-585, October 1969. (Good starting point for Hoare logic:  
[http://en.wikipedia.org/wiki/Hoare\\_logic](http://en.wikipedia.org/wiki/Hoare_logic))

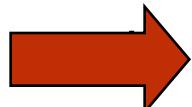
# Summary

- *Constraints are predicates (often boolean expressions) on UML model elements*
- *Contracts are constraints on a class that enable class users, implementors and extenders to share the same assumption about the class ("Design by contract")*
- *OCL is the example of a formal language that allows us to express constraints on UML models*
- *Complicated constraints involving more than one class, attribute or operation can be expressed with 3 basic navigation types.*

# Backup and Additional Slides



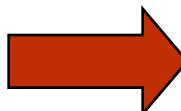
# Additional Constraints on this Model

 A Tournament's planned duration must be under one week.

2. Players can be accepted in a Tournament only if they are already registered with the corresponding League.
3. The number of active Players in a League are those that have taken part in at least one Tournament of the League.

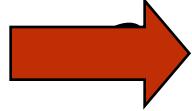
# Additional Constraints on this Model

1. *A Tournament's planned duration must be under one week.*

 *Players can be accepted in a Tournament only if they are already registered with the corresponding League.*

3. *The number of active Players in a League are those that have taken part in at least one Tournament of the League.*

# Additional Constraints on this Model

1. *A Tournament's planned duration must be under one week.*
  2. *Players can be accepted in a Tournament only if they are already registered with the corresponding League.*
-  *The number of active Players in a League are those that have taken part in at least one Tournament of the League.*

# OCL supports Quantification

*OCL forall quantifier*

```
/* All Matches in a Tournament occur within the  
Tournament's time frame */
```

**context** Tournament **inv**:

```
matches->forAll(m:Match |  
    m.start.after(t.start) and m.end.before(t.end))
```

*OCL exists quantifier*

```
/* Each Tournament conducts at least one Match on the  
first day of the Tournament */
```

**context** Tournament **inv**:

```
matches->exists(m:Match | m.start.equals(start))
```

# Pre- and post- conditions for ordering operations on TournamentControl

TournamentControl

```
+selectSponsors(advertisers):List
+advertizeTournament()
+acceptPlayer(p)
+announceTournament()
+isPlayerOverbooked():boolean
```

```
context TournamentControl::selectSponsors(advertisers) pre:
    interestedSponsors->notEmpty and
        tournament.sponsors->isEmpty
context TournamentControl::selectSponsors(advertisers) post:
    tournament.sponsors.equals(advertisers)
context TournamentControl::advertiseTournament() pre:
    tournament.sponsors->isEmpty and
        not tournament.advertised
context TournamentControl::advertiseTournament() post:
    tournament.advertised
context TournamentControl::acceptPlayer(p) pre:
    tournament.advertised and
        interestedPlayers->includes(p) and
            not isPlayerOverbooked(p)
context TournamentControl::acceptPlayer(p) post:
    tournament.players->includes(p)
```

# Specifying invariants on Tournament and Tournament Control

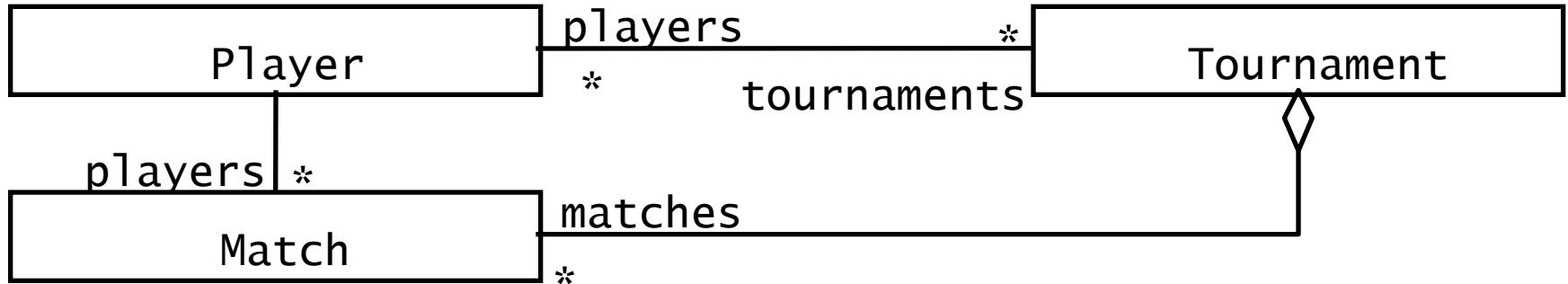
*English: "All Matches of in a Tournament must occur within the time frame of the Tournament"*

```
context Tournament inv:  
  matches->forAll(m|  
    m.start.after(start) and m.start.before(end))
```

*English: "No Player can take part in two or more Tournaments that overlap"*

```
context TournamentControl inv:  
  tournament.players->forAll(p|  
    p.tournaments->forAll(t|  
      t <> tournament implies  
        not t.overlap(tournament)))
```

# Specifying invariants on Match



*English: "A match can only involve players who are accepted in the tournament"*

```
context Match inv:  
    players->forAll(p |  
        p.tournaments->exists(t |  
            t.matches->includes(self)))
```

```
context Match inv:  
    players.tournaments.matches.includes(self)
```

# Chapter 1

---

## ■ The Nature of Software

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
by Roger S. Pressman and Bruce R. Maxim

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# What is Software?

---

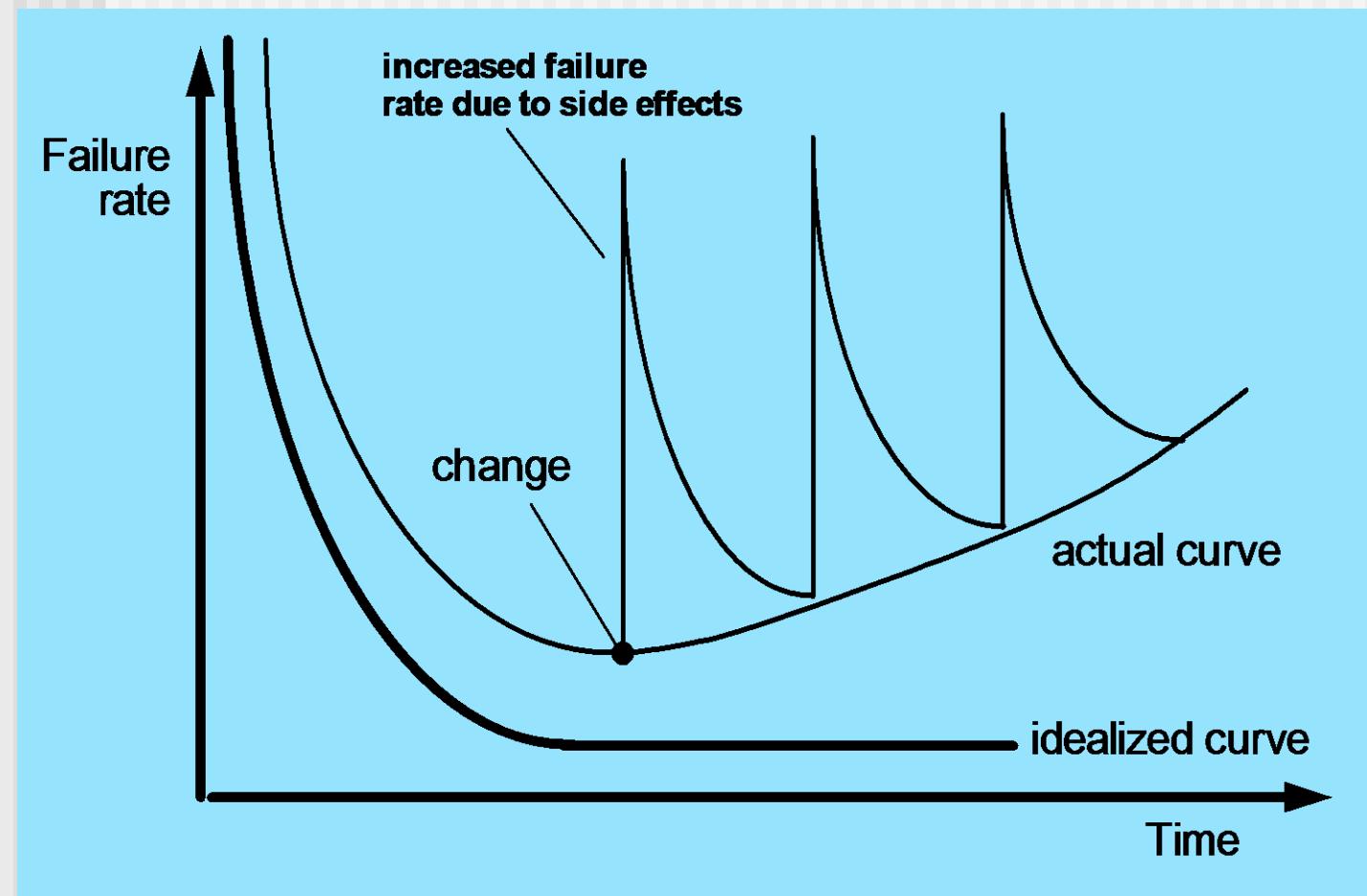
*Software is: (1) **instructions** (computer programs) that when executed provide desired features, function, and performance; (2) **data structures** that enable the programs to adequately manipulate information and (3) **documentation** that describes the operation and use of the programs.*

# What is Software?

---

- ***Software is developed or engineered, it is not manufactured in the classical sense.***
- ***Software doesn't "wear out."***
- ***Although the industry is moving toward component-based construction, most software continues to be custom-built.***

# Wear vs. Deterioration



# Software Applications

---

- System software
- Application software
- Engineering/Scientific software
- Embedded software
- Product-line software
- Web/Mobile applications)
- AI software (robotics, neural nets, game playing)

# Legacy Software

---

## *Why must it change?*

- software must be **adapted** to meet the needs of new computing environments or technology.
- software must be **enhanced** to implement new business requirements.
- software must be **extended to make it interoperable** with other more modern systems or databases.
- software must be **re-architected** to make it viable within a network environment.

# WebApps

---

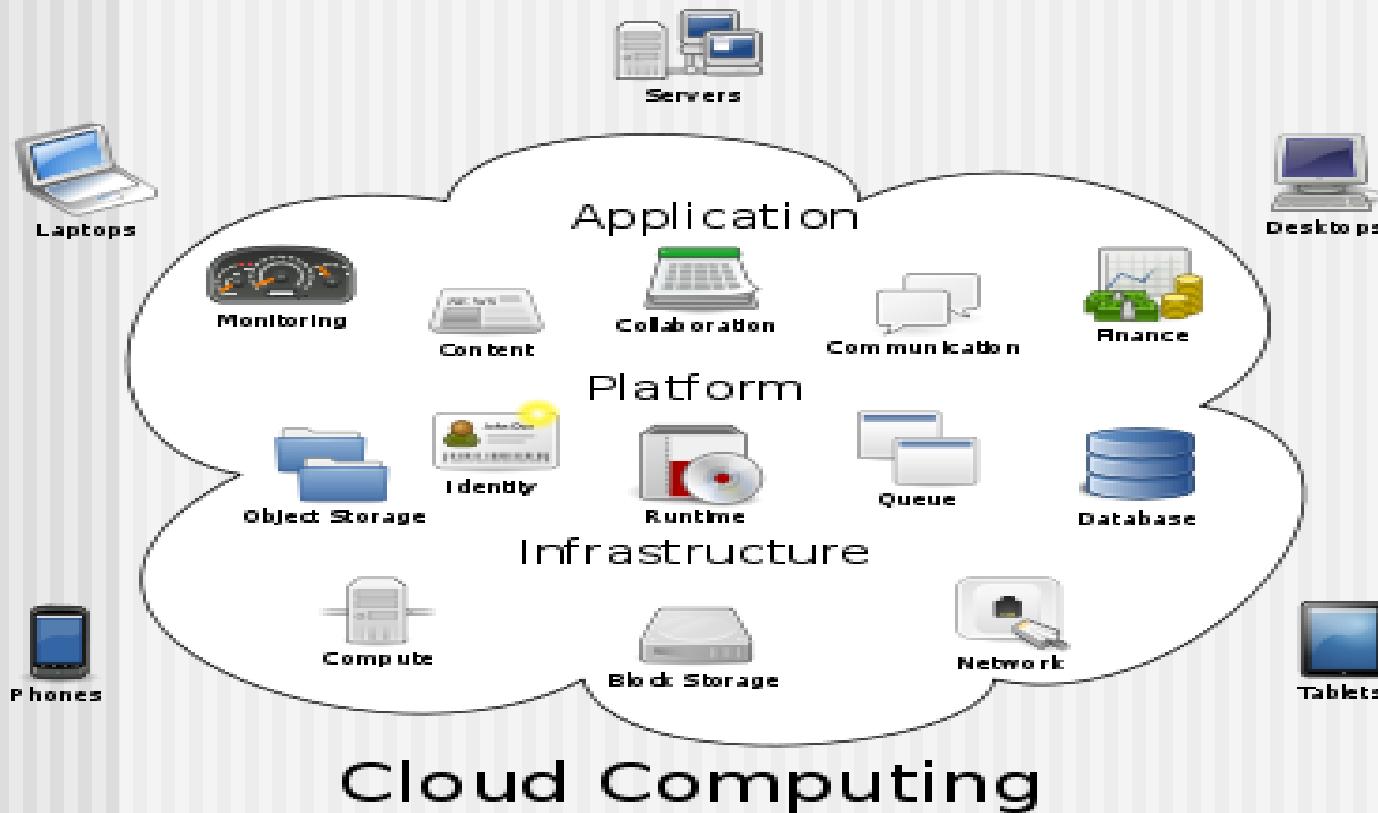
- Modern WebApps are much more than hypertext files with a few pictures
- WebApps are augmented with tools like XML and Java to allow Web engineers including interactive computing capability
- WebApps may standalone capability to end users or may be integrated with corporate databases and business applications
- Semantic web technologies (Web 3.0) have evolved into sophisticated corporate and consumer applications that encompass semantic databases that require web linking, flexible data representation, and application programmer interfaces (API's) for access
- The aesthetic nature of the content remains an important determinant of the quality of a WebApp.

# Mobile Apps

---

- Reside on mobile platforms such as cell phones or tablets
- Contain user interfaces that take both device characteristics and location attributes
- Often provide access to a combination of web-based resources and local device processing and storage capabilities
- Provide persistent storage capabilities within the platform
- A *mobile web application* allows a mobile device to access to web-based content using a browser designed to accommodate the strengths and weaknesses of the mobile platform
- A *mobile app* can gain direct access to the hardware found on the device to provide local processing and storage capabilities
- As time passes these differences will become blurred

# Cloud Computing



# Cloud Computing

---

- *Cloud computing* provides distributed data storage and processing resources to networked computing devices
- Computing resources reside outside the cloud and have access to a variety of resources inside the cloud
- Cloud computing requires developing an architecture containing both frontend and backend services
- Frontend services include the client devices and application software to allow access
- Backend services include servers, data storage, and server-resident applications
- Cloud architectures can be segmented to restrict access to private data

# Product Line Software

---

- *Product line software* is a set of software-intensive systems that share a common set of features and satisfy the needs of a particular market
- These software products are developed using the same application and data architectures using a common core of reusable software components
- A software product line shares a set of assets that include *requirements, architecture, design patterns, reusable components, test cases*, and other work products
- A software product line allows in the development of many products that are engineered by capitalizing on the commonality among all products within the product line

# Characteristics of WebApps - II

---

- **Data driven.** The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end-user.
- **Content sensitive.** The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.
- **Continuous evolution.** Unlike conventional application software that evolves over a series of planned, chronologically-spaced releases, Web applications evolve continuously.
- **Immediacy.** Although *immediacy*—the compelling need to get software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time to market that can be a matter of a few days or weeks.
- **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end-users who may access the application.
- **Aesthetics.** An undeniable part of the appeal of a WebApp is its look and feel.

# Chapter 2

---

## ■ Software Engineering

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach, 8/e*  
**by Roger S. Pressman and Bruce R. Maxim**

Slides copyright © 1996, 2001, 2005, 2009, 2014 by Roger S. Pressman

***For non-profit educational use only***

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 8/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

# Software Engineering

---

- Some realities:
  - *a concerted effort should be made to understand the problem before a software solution is developed*
  - *design becomes a pivotal activity*
  - *software should exhibit high quality*
  - *software should be maintainable*
- The seminal definition:
  - *[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

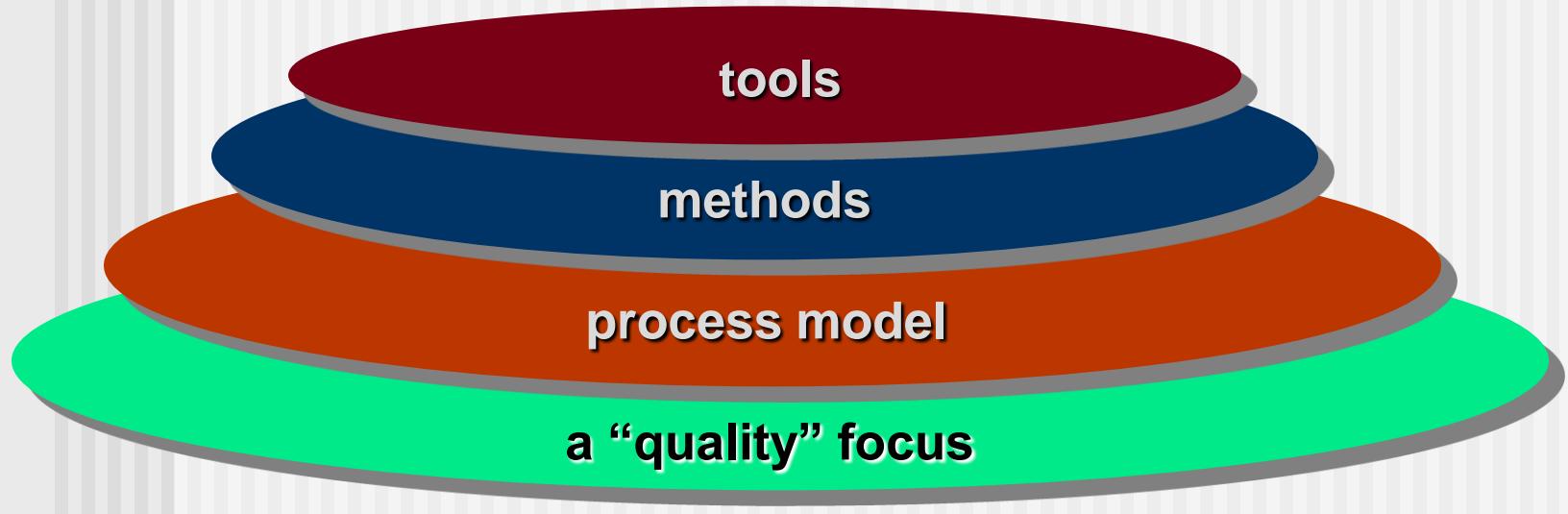
# Software Engineering

---

- The IEEE definition:
  - *Software Engineering:*
  - (1) *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*
  - (2) *The study of approaches as in (1).*

# A Layered Technology

---



***Software Engineering***

# A Process Framework

---

## Process framework

### Framework activities

work tasks

work products

milestones & deliverables

QA checkpoints

### Umbrella Activities

# Framework Activities

---

- Communication
- Planning
- Modeling
  - Analysis of requirements
  - Design
- Construction
  - Code generation
  - Testing
- Deployment

# Umbrella Activities

---

- Software project tracking and control
- Risk management
- Software quality assurance
- Technical reviews
- Measurement
- Software configuration management
- Reusability management
- Work product preparation and production

# Adapting a Process Model

---

- the overall flow of activities, actions, and tasks and the interdependencies among them
- the degree to which actions and tasks are defined within each framework activity
- the degree to which work products are identified and required
- the manner which quality assurance activities are applied
- the manner in which project tracking and control activities are applied
- the overall degree of detail and rigor with which the process is described
- the degree to which the customer and other stakeholders are involved with the project
- the level of autonomy given to the software team
- the degree to which team organization and roles are prescribed

# The Essence of Practice

---

- Polya suggests:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).

# Understand the Problem

---

- *Who has a stake in the solution to the problem?* That is, who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

# Plan the Solution

---

- *Have you seen similar problems before?* Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

# Carry Out the Plan

---

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

# Examine the Result

---

- *Is it possible to test each component part of the solution?* Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

# Hooker's General Principles

---

- 1: *The Reason It All Exists*
- 2: *KISS (Keep It Simple, Stupid!)*
- 3: *Maintain the Vision*
- 4: *What You Produce, Others Will Consume*
- 5: *Be Open to the Future*
- 6: *Plan Ahead for Reuse*
- 7: *Think!*

# Software Myths

---

- Affect managers, customers (and other non-technical stakeholders) and practitioners
- Are believable because they often have elements of truth,  
*but ...*
- Invariably lead to bad decisions,  
*therefore ...*
- Insist on reality as you navigate your way through software engineering

# How It all Starts

---

## ■ *SafeHome*:

- Every software project is precipitated by some business need—
  - the need to correct a defect in an existing application;
  - the need to adapt a ‘legacy system’ to a changing business environment;
  - the need to extend the functions and features of an existing application, or
  - the need to create a new product, service, or system.

# Software Development Life Cycle (SDLC)

“You’ve got to be very careful if you don’t know where you’re going, because you might not get there.”

Chandravadan Prajapati

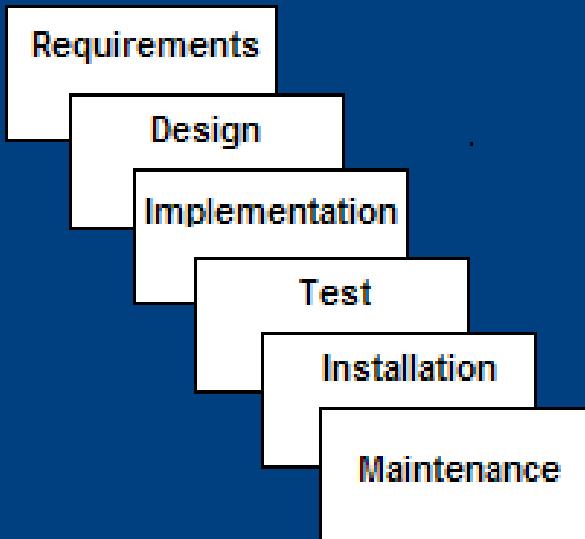


# SDLC Model

A framework that describes the activities performed at each stage of a software development project.



# Waterfall Model



- **Requirements** – defines needed information, function, behavior, performance and interfaces.
- **Design** – data structures, software architecture, interface representations, algorithmic details.
- **Implementation** – source code, database, user documentation, testing.

# Waterfall Strengths

- Easy to understand, easy to use
- Provides structure to inexperienced staff
- Milestones are well understood
- Sets requirements stability
- Good for management control (plan, staff, track)
- Works well when quality is more important than cost or schedule



# Waterfall Deficiencies

- All requirements must be known upfront
- Deliverables created for each phase are considered frozen – inhibits flexibility
- Can give a false impression of progress
- Does not reflect problem-solving nature of software development – iterations of phases
- Integration is one big bang at the end
- Little opportunity for customer to preview the system (until it may be too late)

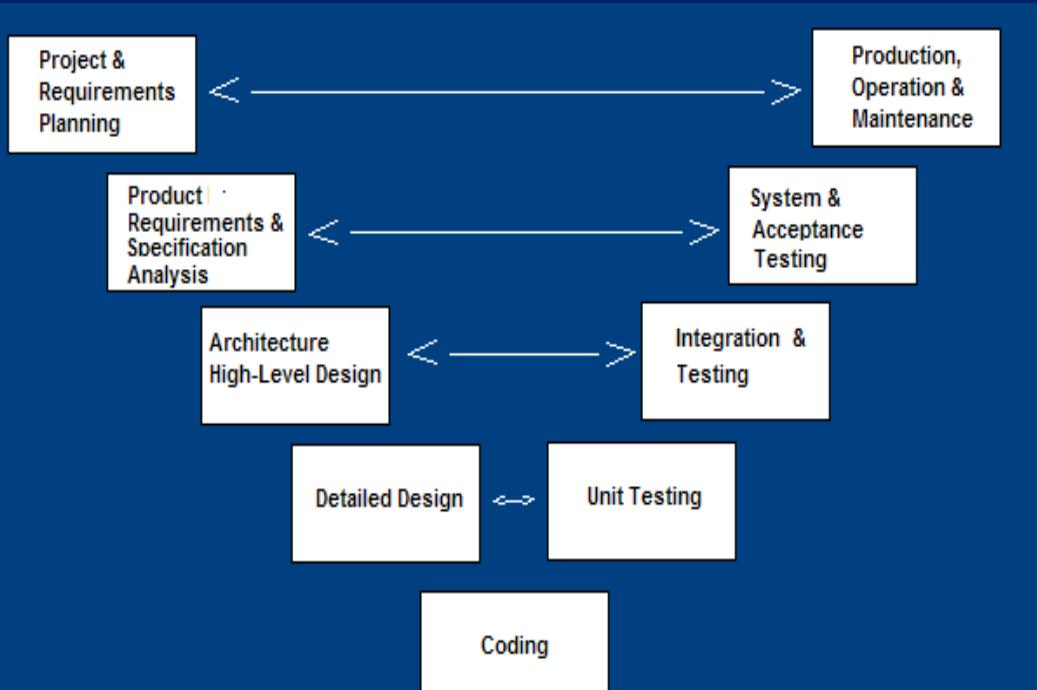


# When to use the Waterfall Model

- Requirements are very well known
- Product definition is stable
- Technology is understood
- New version of an existing product
- Porting an existing product to a new platform.



# V-Shaped SDLC Model



- A variant of the Waterfall that emphasizes the verification and validation of the product.
- Testing of the product is planned in parallel with a corresponding phase of development

# V-Shaped Steps

- Project and Requirements Planning – allocate resources
- Product Requirements and Specification Analysis – complete specification of the software system
- Architecture or High-Level Design – defines how software functions fulfill the design
- Detailed Design – develop algorithms for each architectural component
- Production, operation and maintenance – provide for enhancement and corrections
- System and acceptance testing – check the entire software system in its environment
- Integration and Testing – check that modules interconnect correctly
- Unit testing – check that each module acts as expected
- Coding – transform algorithms into software

# V-Shaped Strengths

- Emphasize planning for verification and validation of the product in early stages of product development
- Each deliverable must be testable
- Project management can track progress by milestones
- Easy to use



# V-Shaped Weaknesses

- Does not easily handle concurrent events
- Does not easily handle dynamic changes in requirements

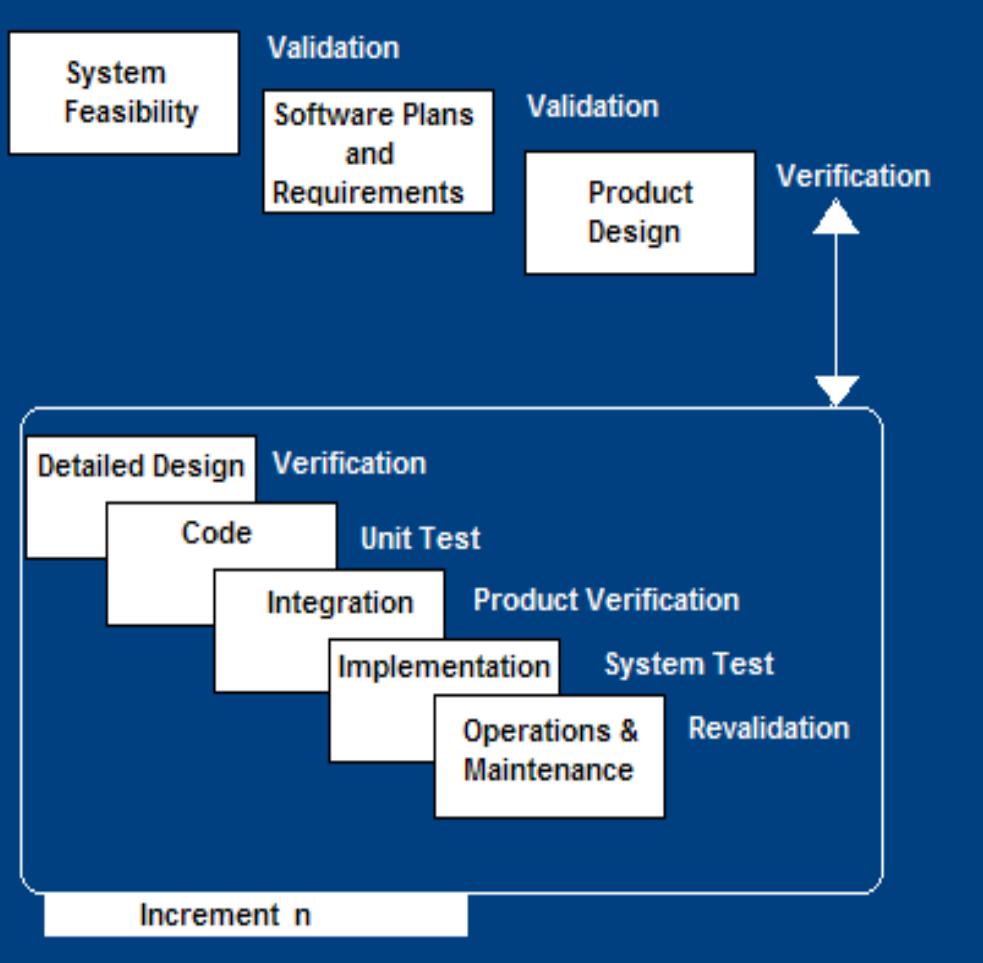


# When to use the V-Shaped Model

- Excellent choice for **systems requiring high reliability** – hospital patient control applications
- **All requirements are known up-front**
- When it can be modified to handle **changing requirements beyond analysis phase**
- **Solution and technology are known**



# Incremental SDLC Model



- Construct a partial implementation of a total system
- Then slowly add increased functionality
- The incremental model prioritizes requirements of the system and then implements them in groups.
- Each subsequent release of the system adds function to the previous release, until all designed functionality has been implemented.

# Incremental Model Strengths

- Develop high-risk or major functions first
- Each release delivers an operational product
- Customer can respond to each build
- Uses “divide and conquer” breakdown of tasks
- Lowers initial delivery cost
- Initial product delivery is faster
- Customers get important functionality early
- Risk of changing requirements is reduced



# Incremental Model Weaknesses

- Requires good planning and design
- Requires early definition of a complete and fully functional system to allow for the definition of increments
- Well-defined module interfaces are required (some will be developed long before others)
- Total cost of the complete system is not lower

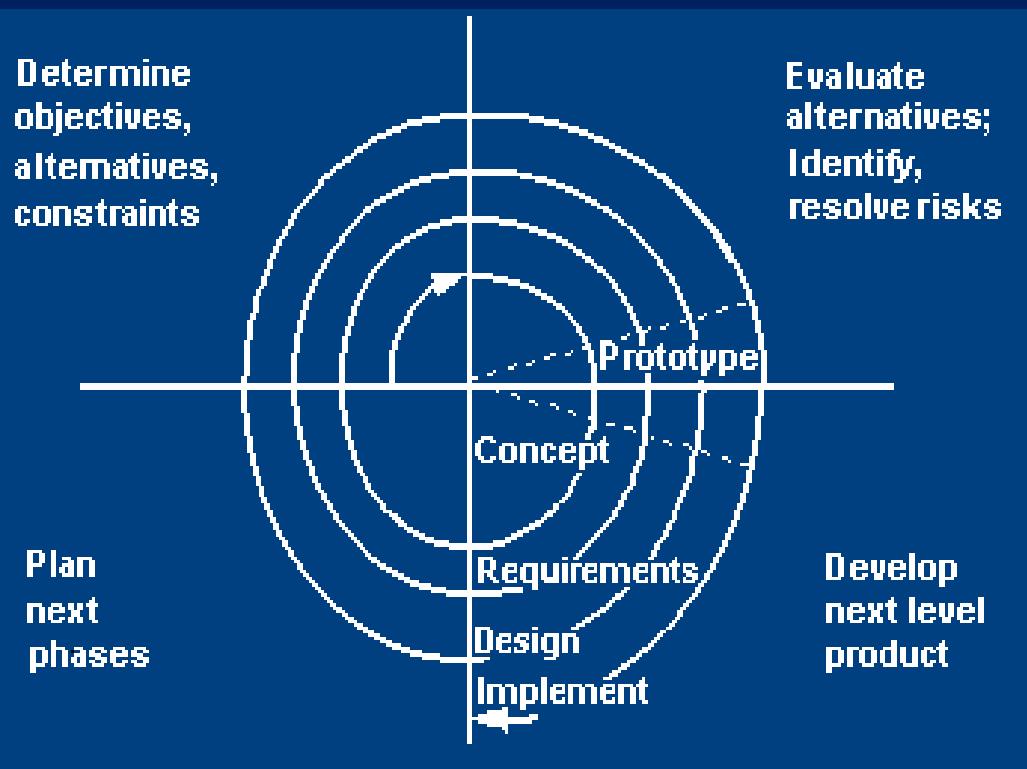


# When to use the Incremental Model

- Risk, funding, schedule, program complexity, or need for **early realization of benefits**.
- Most of the requirements are known up-front but are expected to **evolve over time**
- A need to **get basic functionality to the market early**
- On projects which have **lengthy development schedules**
- On a project with **new technology**



# Spiral SDLC Model



- Adds risk analysis, and 4gl RAD prototyping to the waterfall model
- Each cycle involves the same sequence of steps as the waterfall process model

# Spiral Quadrant

## Determine objectives, alternatives and constraints

- **Objectives:** functionality, performance, hardware/software interface, critical success factors, etc.
- **Alternatives:** build, reuse, buy, sub-contract, etc.
- **Constraints:** cost, schedule, interface, etc.



# Spiral Quadrant

## Evaluate alternatives, identify and resolve risks

- Study alternatives relative to objectives and constraints
- Identify risks (lack of experience, new technology, tight schedules, poor process, etc.)
- Resolve risks (evaluate if money could be lost by continuing system development)



# Spiral Quadrant

## Develop next-level product

- Typical activites:
  - Create a design
  - Review design
  - Develop code
  - Inspect code
  - Test product



# Spiral Quadrant

## Plan next phase

- Typical activities
  - Develop project plan
  - Develop configuration management plan
  - Develop a test plan
  - Develop an installation plan



# Spiral Model Strengths

- Provides early indication of insurmountable risks, without much cost
- Users see the system early because of rapid prototyping tools
- Critical high-risk functions are developed first
- The design does not have to be perfect
- Users can be closely tied to all lifecycle steps
- Early and frequent feedback from users
- Cumulative costs assessed frequently



# Spiral Model Weaknesses

- Time spent for evaluating risks too large for small or low-risk projects
- Time spent planning, resetting objectives, doing risk analysis and prototyping may be excessive
- The model is complex
- Risk assessment expertise is required
- Spiral may continue indefinitely
- Developers must be reassigned during non-development phase activities
- May be hard to define objective, verifiable milestones that indicate readiness to proceed through the next iteration



# When to use Spiral Model

- When creation of a prototype is appropriate
- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)



# Chapter 2 – Software Processes

# Topics covered

---

- ✧ Software process models
- ✧ Process activities
- ✧ Coping with change
- ✧ Process improvement

# The software process

---

- ✧ A structured set of activities required to develop a software system.
- ✧ Many different software processes but all involve:
  - Specification – defining what the system should do;
  - Design and implementation – defining the organization of the system and implementing the system;
  - Validation – checking that it does what the customer wants;
  - Evolution – changing the system in response to changing customer needs.
- ✧ A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

# Software process descriptions

---

- ✧ When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc. and the ordering of these activities.
- ✧ Process descriptions may also include:
  - Products, which are the outcomes of a process activity;
  - Roles, which reflect the responsibilities of the people involved in the process;
  - Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.

# Plan-driven and agile processes

---

- ✧ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ✧ In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- ✧ In practice, most practical processes include elements of both plan-driven and agile approaches.
- ✧ There are no right or wrong software processes.

# Software process models

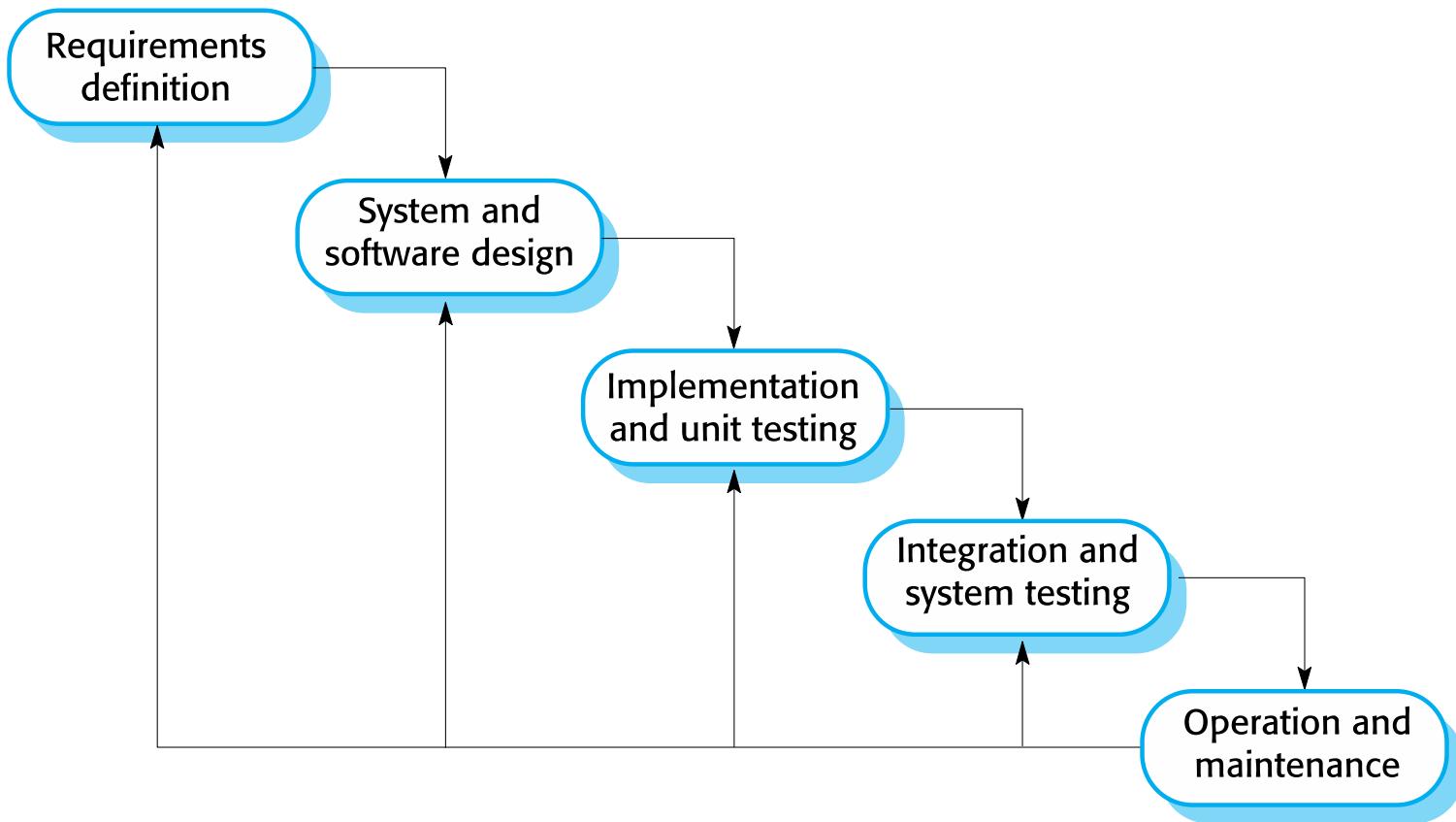
# Software process models

---

- ✧ The waterfall model
  - Plan-driven model. Separate and distinct phases of specification and development.
- ✧ Incremental development
  - Specification, development and validation are interleaved. May be plan-driven or agile.
- ✧ Integration and configuration
  - The system is assembled from existing configurable components. May be plan-driven or agile.
- ✧ In practice, most large systems are developed using a process that incorporates elements from all of these models.

# The waterfall model

---



# Waterfall model phases

---

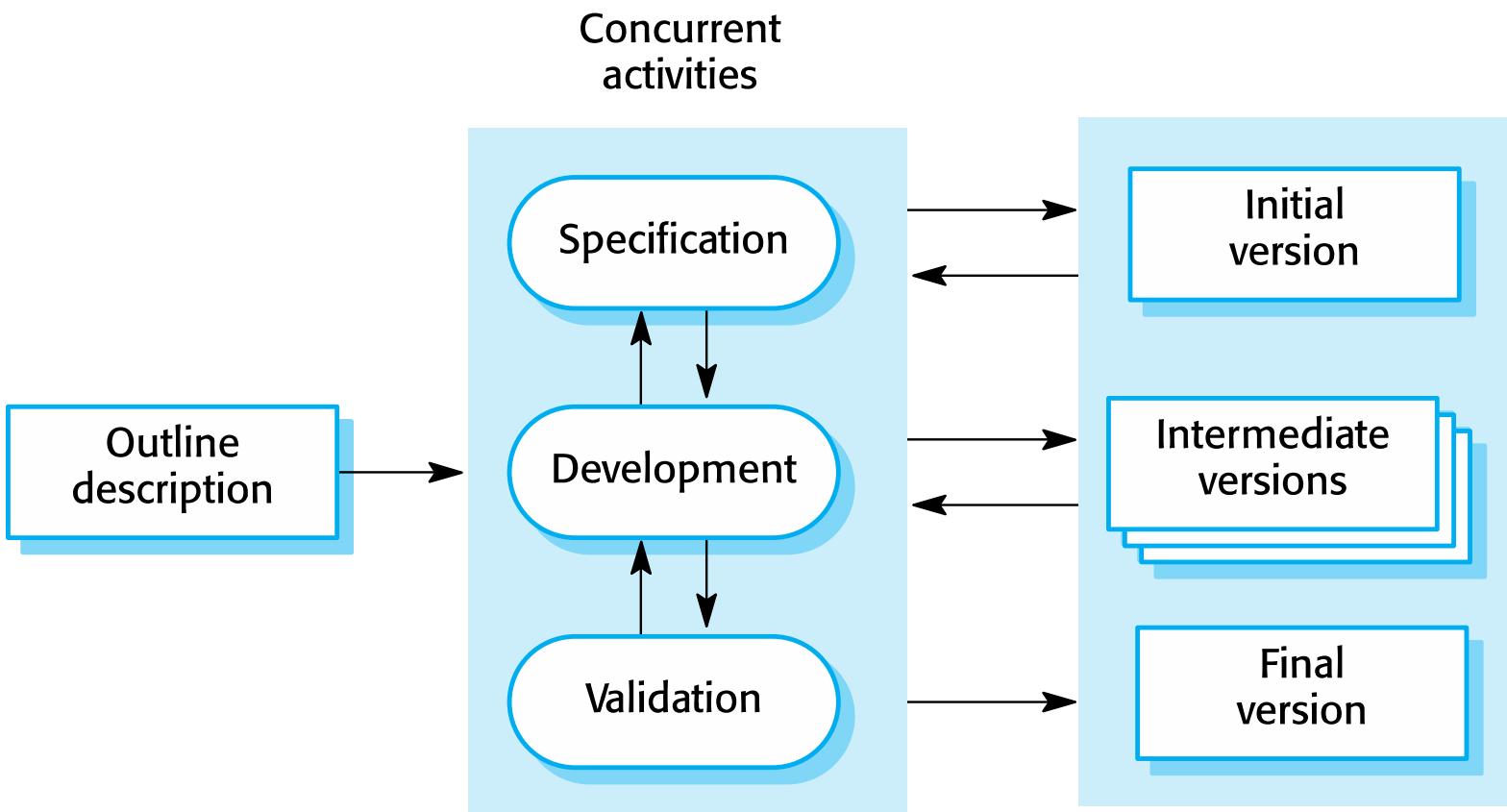
- ✧ There are separate identified phases in the waterfall model:
  - Requirements analysis and definition
  - System and software design
  - Implementation and unit testing
  - Integration and system testing
  - Operation and maintenance
- ✧ The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. In principle, a phase has to be complete before moving onto the next phase.

# Waterfall model problems

---

- ✧ Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
  - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
  - Few business systems have stable requirements.
- ✧ The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
  - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

# Incremental development



# Incremental development benefits

---

- ✧ The cost of accommodating changing customer requirements is reduced.
  - The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- ✧ It is easier to get customer feedback on the development work that has been done.
  - Customers can comment on demonstrations of the software and see how much has been implemented.
- ✧ More rapid delivery and deployment of useful software to the customer is possible.
  - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

# Incremental development problems

---

- ✧ The process is not visible.
  - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- ✧ System structure tends to degrade as new increments are added.
  - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

# Integration and configuration

---

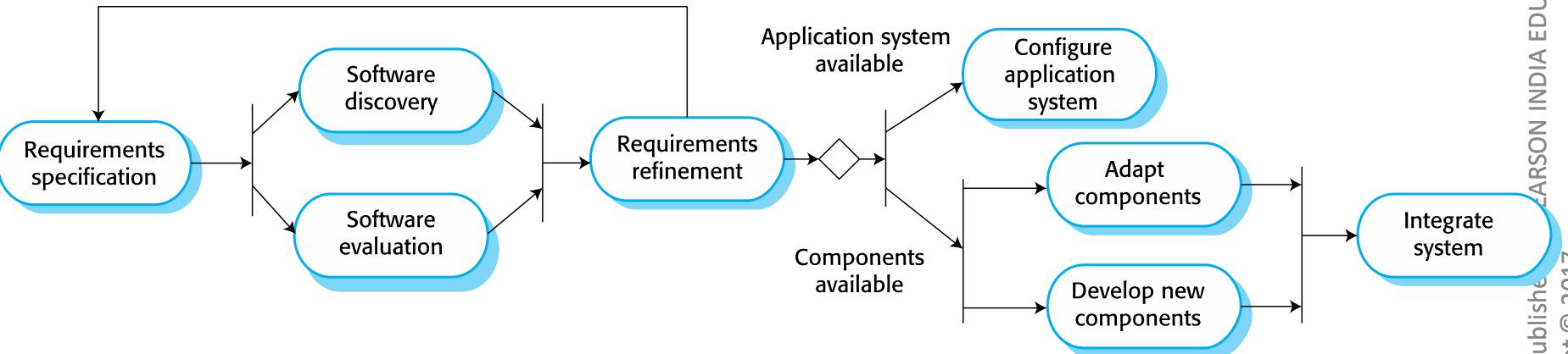
- ✧ Based on software reuse where systems are integrated from existing components or application systems (sometimes called COTS -Commercial-off-the-shelf) systems.
- ✧ Reused elements may be configured to adapt their behaviour and functionality to a user's requirements
- ✧ Reuse is now the standard approach for building many types of business system
  - Reuse covered in more depth in Chapter 15.

# Types of reusable software

---

- ✧ Stand-alone application systems (sometimes called COTS) that are configured for use in a particular environment.
- ✧ Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- ✧ Web services that are developed according to service standards and which are available for remote invocation.

# Reuse-oriented software engineering



# Key process stages

---

- ✧ Requirements specification
- ✧ Software discovery and evaluation
- ✧ Requirements refinement
- ✧ Application system configuration
- ✧ Component adaptation and integration

# Advantages and disadvantages

---

- ✧ Reduced costs and risks as less software is developed from scratch
- ✧ Faster delivery and deployment of system
- ✧ But requirements compromises are inevitable so system may not meet real needs of users
- ✧ Loss of control over evolution of reused system elements

---

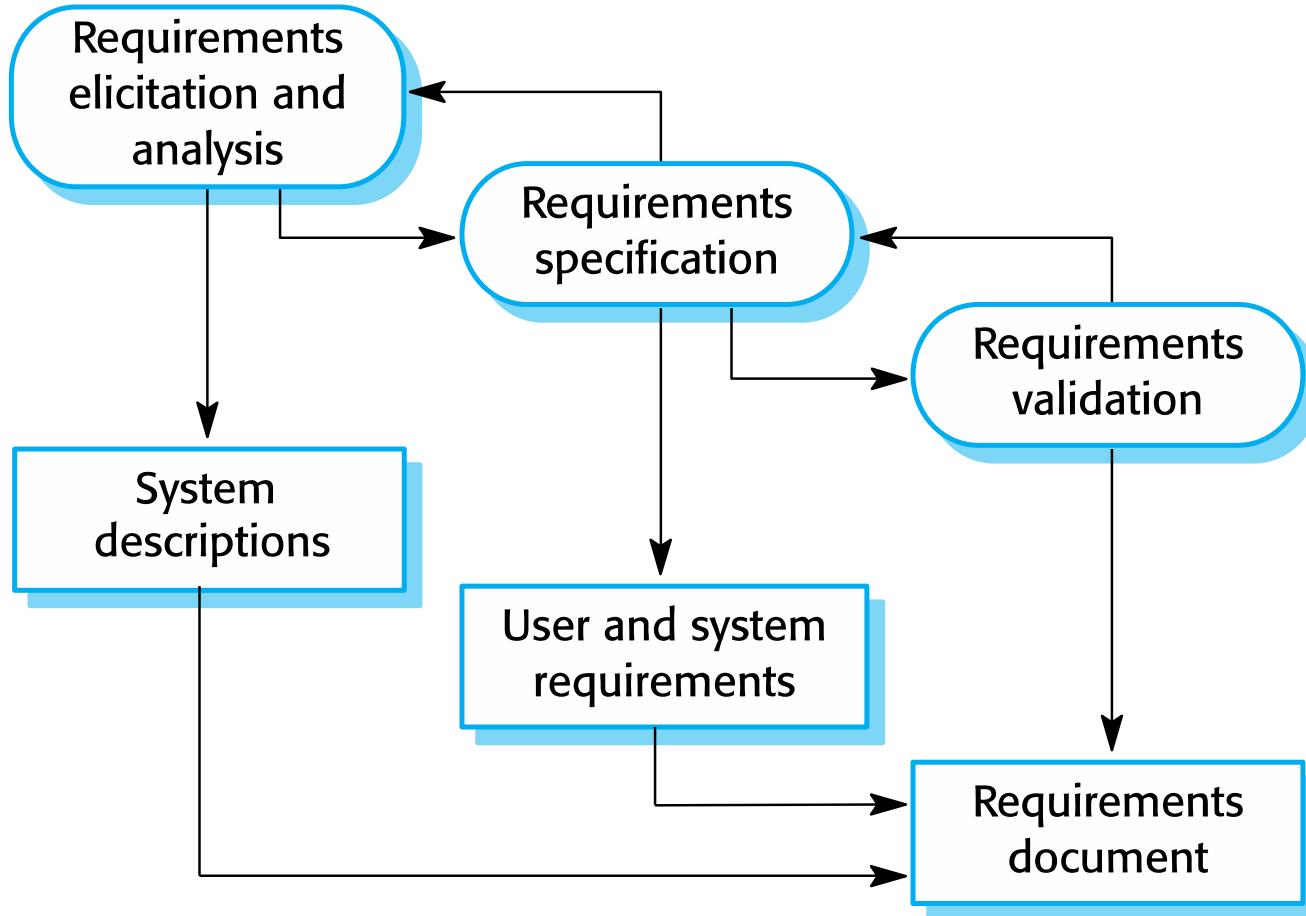
# **Process activities**

# Process activities

---

- ✧ Real software processes are inter-leaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- ✧ The four basic process activities of specification, development, validation and evolution are organized differently in different development processes.
- ✧ For example, in the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved.

# The requirements engineering process



# Software specification

---

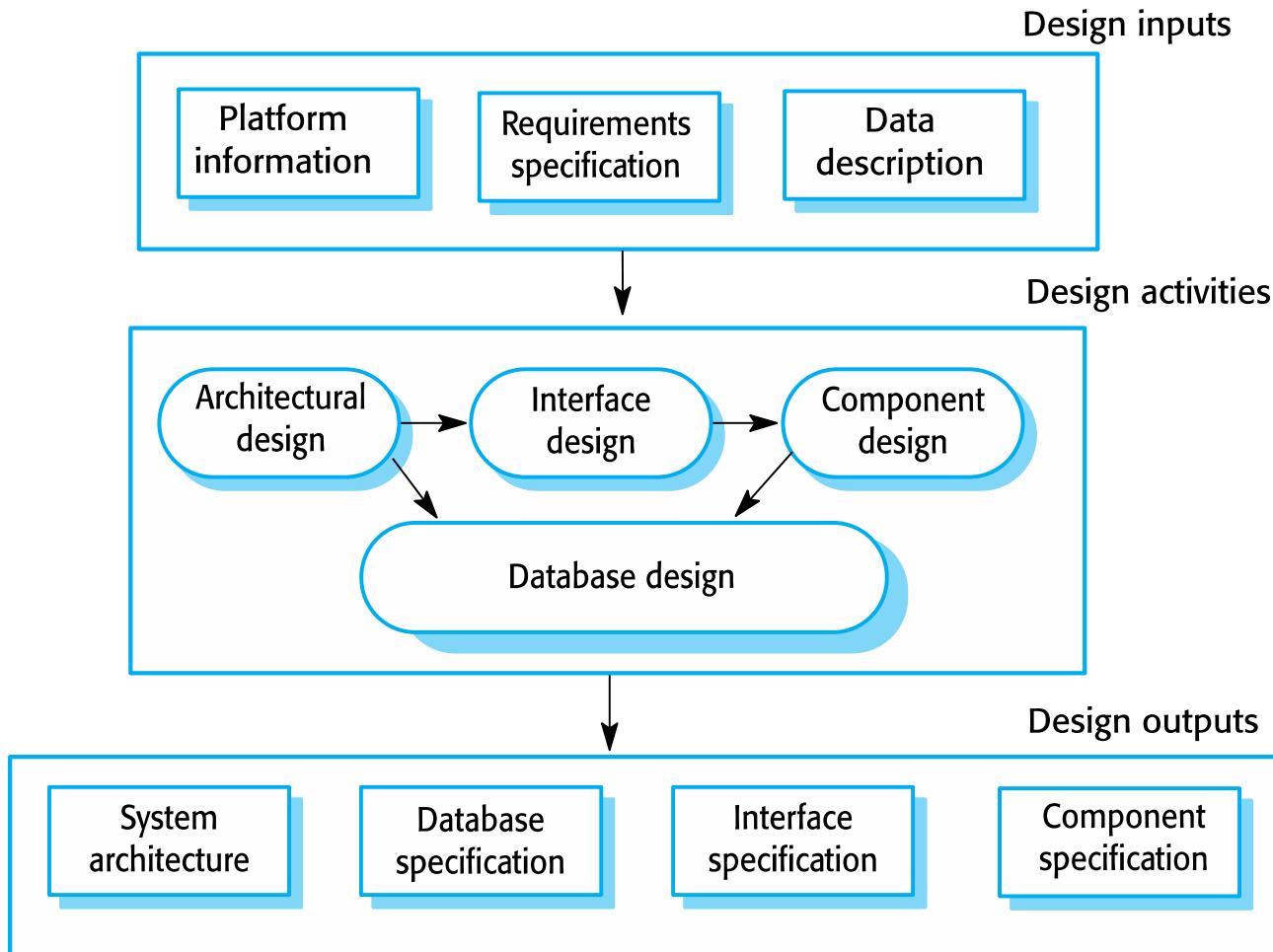
- ✧ The process of establishing what services are required and the constraints on the system's operation and development.
- ✧ Requirements engineering process
  - Requirements elicitation and analysis
    - What do the system stakeholders require or expect from the system?
  - Requirements specification
    - Defining the requirements in detail
  - Requirements validation
    - Checking the validity of the requirements

# Software design and implementation

---

- ✧ The process of converting the system specification into an executable system.
- ✧ Software design
  - Design a software structure that realises the specification;
- ✧ Implementation
  - Translate this structure into an executable program;
- ✧ The activities of design and implementation are closely related and may be inter-leaved.

# A general model of the design process



# Design activities

---

- ✧ *Architectural design*, where you identify the overall structure of the system, the principal components (subsystems or modules), their relationships and how they are distributed.
- ✧ *Database design*, where you design the system data structures and how these are to be represented in a database.
- ✧ *Interface design*, where you define the interfaces between system components.
- ✧ *Component selection and design*, where you search for reusable components. If unavailable, you design how it will operate.

# System implementation

---

- ✧ The software is implemented either by developing a program or programs or by configuring an application system.
- ✧ Design and implementation are interleaved activities for most types of software system.
- ✧ Programming is an individual activity with no standard process.
- ✧ Debugging is the activity of finding program faults and correcting these faults.

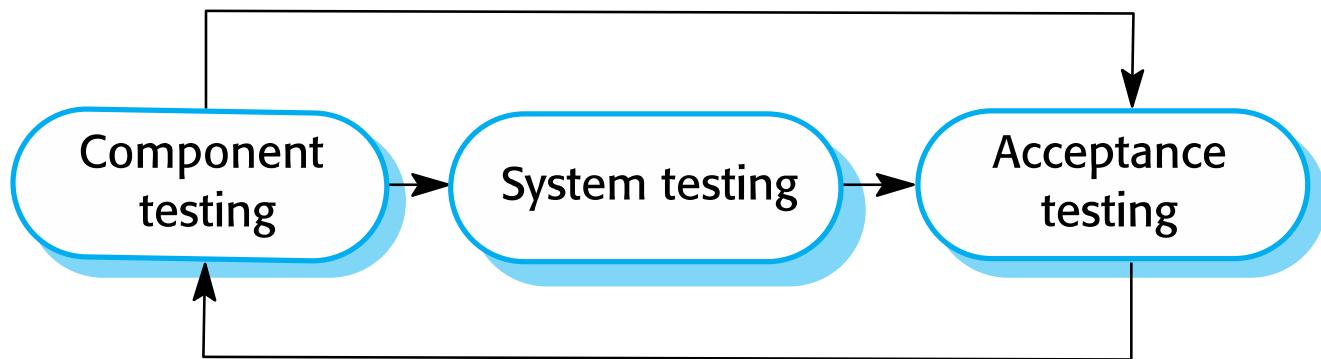
# Software validation

---

- ✧ Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- ✧ Involves checking and review processes and system testing.
- ✧ System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.
- ✧ Testing is the most commonly used V & V activity.

# Stages of testing

---



# Testing stages

---

## ✧ Component testing

- Individual components are tested independently;
- Components may be functions or objects or coherent groupings of these entities.

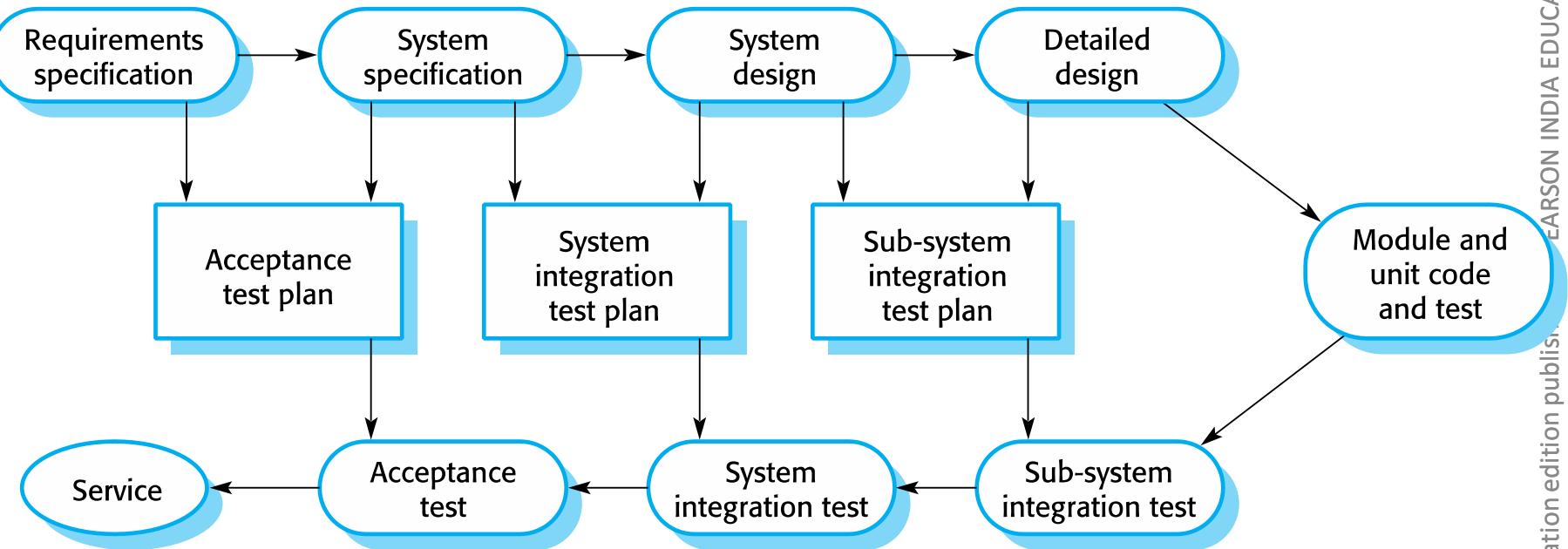
## ✧ System testing

- Testing of the system as a whole. Testing of emergent properties is particularly important.

## ✧ Customer testing

- Testing with customer data to check that the system meets the customer's needs.

# Testing phases in a plan-driven software process (V-model)

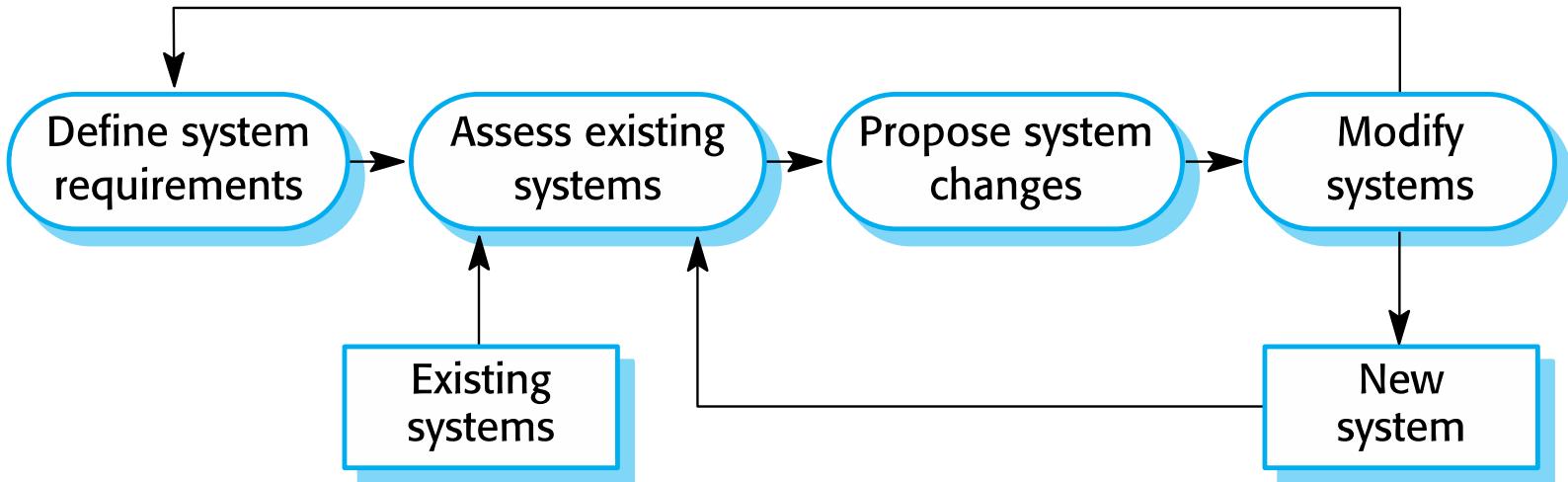


# Software evolution

---

- ✧ Software is inherently flexible and can change.
- ✧ As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- ✧ Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

# System evolution



# Coping with change

# Coping with change

---

- ✧ Change is inevitable in all large software projects.
  - Business changes lead to new and changed system requirements
  - New technologies open up new possibilities for improving implementations
  - Changing platforms require application changes
- ✧ Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality

# Reducing the costs of rework

---

- ✧ Change anticipation, where the software process includes activities that can anticipate possible changes before significant rework is required.
  - For example, a prototype system may be developed to show some key features of the system to customers.
- ✧ Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost.
  - This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have been altered to incorporate the change.

# Coping with changing requirements

---

- ✧ System prototyping, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This approach supports change anticipation.
- ✧ Incremental delivery, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance.

# Software prototyping

---

- ✧ A prototype is an initial version of a system used to demonstrate concepts and try out design options.
- ✧ A prototype can be used in:
  - The requirements engineering process to help with requirements elicitation and validation;
  - In design processes to explore options and develop a UI design;
  - In the testing process to run back-to-back tests.

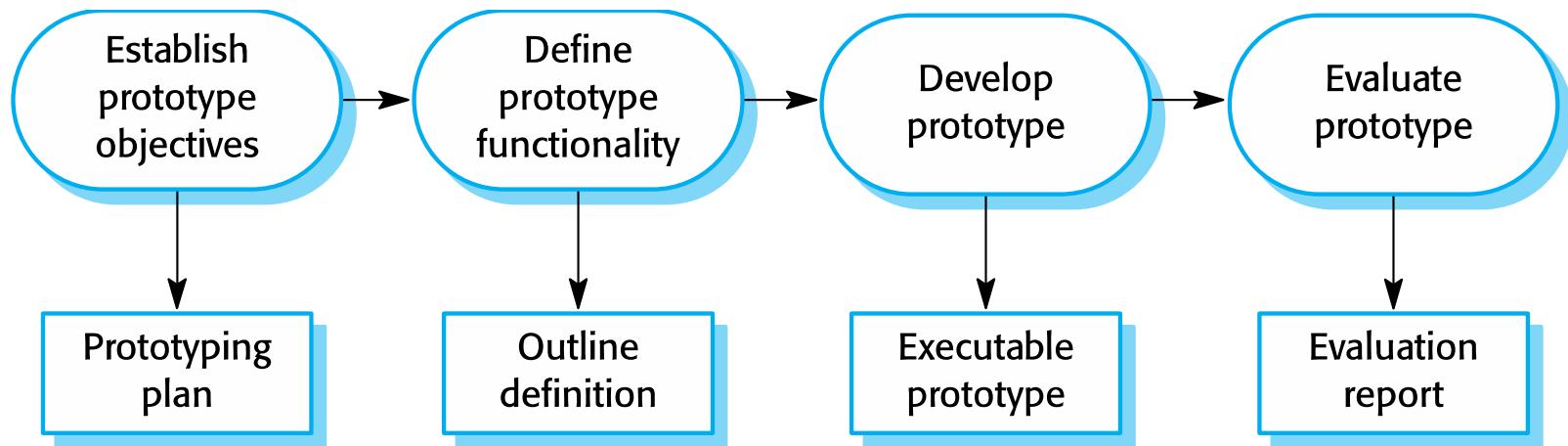
# Benefits of prototyping

---

- ✧ Improved system usability.
- ✧ A closer match to users' real needs.
- ✧ Improved design quality.
- ✧ Improved maintainability.
- ✧ Reduced development effort.

# The process of prototype development

---



# Prototype development

---

- ✧ May be based on rapid prototyping languages or tools
- ✧ May involve leaving out functionality
  - Prototype should focus on areas of the product that are not well-understood;
  - Error checking and recovery may not be included in the prototype;
  - Focus on functional rather than non-functional requirements such as reliability and security

# Throw-away prototypes

---

- ✧ Prototypes should be discarded after development as they are not a good basis for a production system:
  - It may be impossible to tune the system to meet non-functional requirements;
  - Prototypes are normally undocumented;
  - The prototype structure is usually degraded through rapid change;
  - The prototype probably will not meet normal organisational quality standards.

# Incremental delivery

---

- ✧ Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- ✧ User requirements are prioritised and the highest priority requirements are included in early increments.
- ✧ Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

# Incremental development and delivery

---

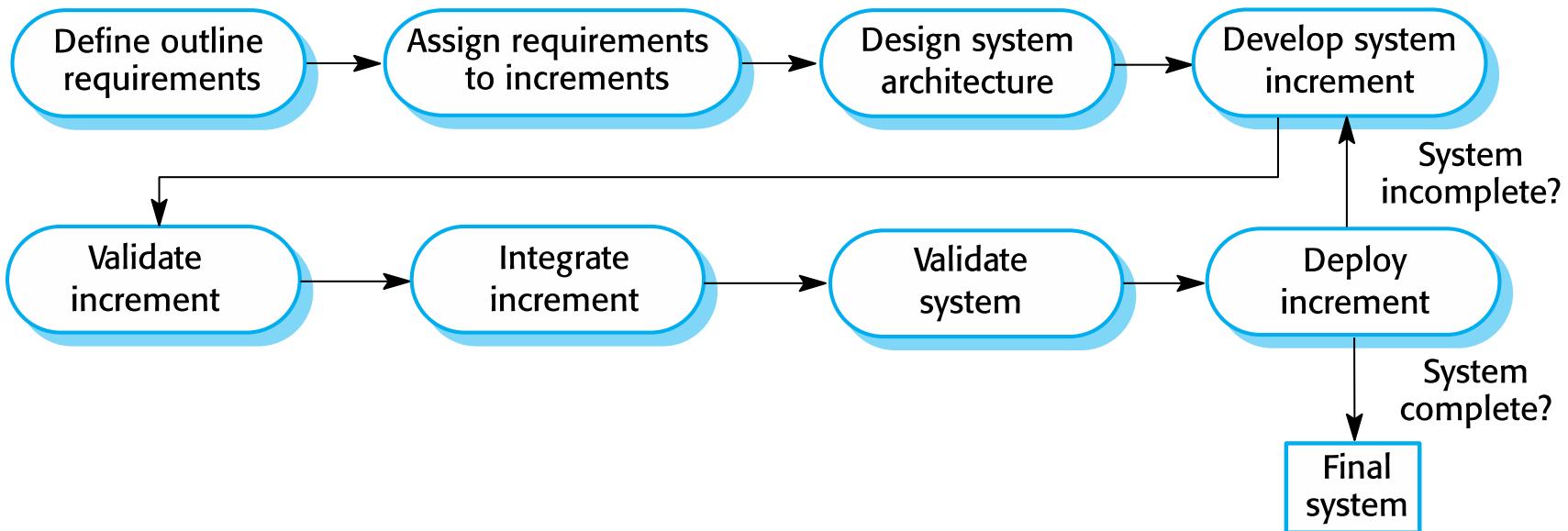
## ✧ Incremental development

- Develop the system in increments and evaluate each increment before proceeding to the development of the next increment;
- Normal approach used in agile methods;
- Evaluation done by user/customer proxy.

## ✧ Incremental delivery

- Deploy an increment for use by end-users;
- More realistic evaluation about practical use of software;
- Difficult to implement for replacement systems as increments have less functionality than the system being replaced.

# Incremental delivery



# Incremental delivery advantages

---

- ✧ Customer value can be delivered with each increment so system functionality is available earlier.
- ✧ Early increments act as a prototype to help elicit requirements for later increments.
- ✧ Lower risk of overall project failure.
- ✧ The highest priority system services tend to receive the most testing.

# Incremental delivery problems

---

- ✧ Most systems require a set of basic facilities that are used by different parts of the system.
  - As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
- ✧ The essence of iterative processes is that the specification is developed in conjunction with the software.
  - However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

# Process improvement

# Process improvement

---

- ✧ Many software companies have turned to software process improvement as a way of enhancing the quality of their software, reducing costs or accelerating their development processes.
- ✧ Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time.

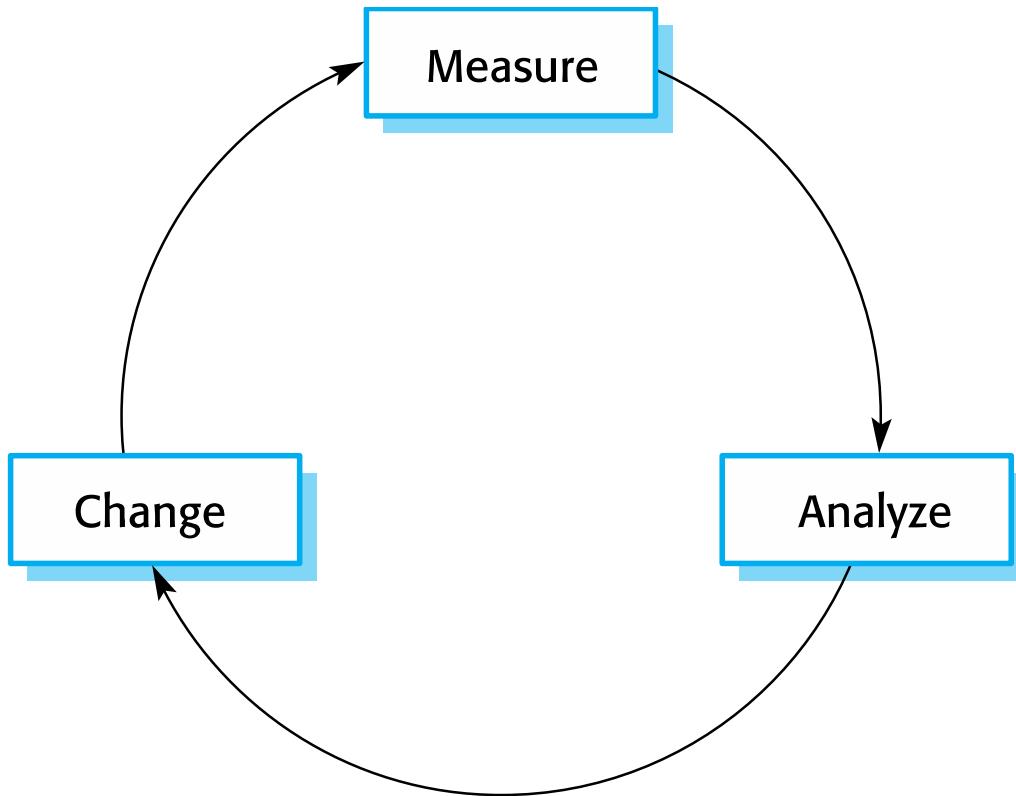
# Approaches to improvement

---

- ✧ The process maturity approach, which focuses on improving process and project management and introducing good software engineering practice.
  - The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes.
- ✧ The agile approach, which focuses on iterative development and the reduction of overheads in the software process.
  - The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements.

# The process improvement cycle

---



# Process improvement activities

---

## ✧ *Process measurement*

- You measure one or more attributes of the software process or product. These measurements forms a baseline that helps you decide if process improvements have been effective.

## ✧ *Process analysis*

- The current process is assessed, and process weaknesses and bottlenecks are identified. Process models (sometimes called process maps) that describe the process may be developed.

## ✧ *Process change*

- Process changes are proposed to address some of the identified process weaknesses. These are introduced and the cycle resumes to collect data about the effectiveness of the changes.

# Process measurement

---

- ✧ Wherever possible, quantitative process data should be collected
  - However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible.
- ✧ Process measurements should be used to assess process improvements
  - But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives.

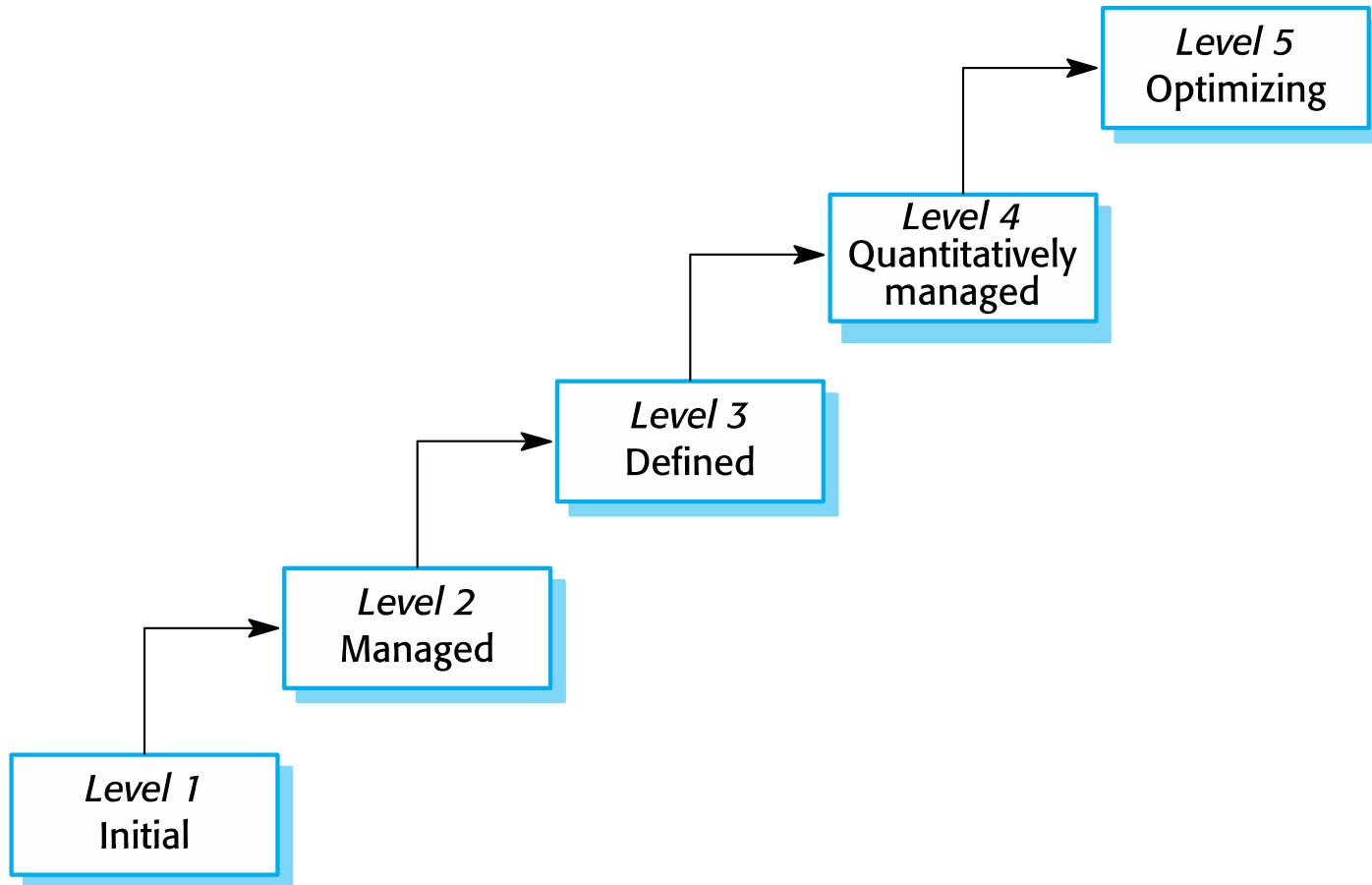
# Process metrics

---

- ✧ Time taken for process activities to be completed
  - E.g. Calendar time or effort to complete an activity or process.
- ✧ Resources required for processes or activities
  - E.g. Total effort in person-days.
- ✧ Number of occurrences of a particular event
  - E.g. Number of defects discovered.

# Capability maturity levels

---



# CMM Levels

---

## **Level 5 – Optimizing (< 1%)**

- process change management
- technology change management
- defect prevention

## **Level 1 – Initial (~ 70%)**

## **Level 4 – Managed (< 5%)**

- software quality management
- quantitative process management

## **Level 3 – Defined (< 10%)**

- peer reviews
- intergroup coordination
- software product engineering
- integrated software management
- training program
- organization process definition
- organization process focus

## **Level 2 – Repeatable (~ 15%)**

- software configuration management
- software quality assurance
- software project tracking and oversight
- software project planning
- requirements management

# The SEI capability maturity model

---

## ✧ Initial

- Essentially uncontrolled

## ✧ Repeatable

- Product management procedures defined and used

## ✧ Defined

- Process management procedures and strategies defined and used

## ✧ Managed

- Quality management strategies defined and used

## ✧ Optimising

- Process improvement strategies defined and used

# Key points

---

- ✧ Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- ✧ General process models describe the organization of software processes.
  - Examples of these general models include the ‘waterfall’ model, incremental development, and reuse-oriented development.
- ✧ Requirements engineering is the process of developing a software specification.

## Key points

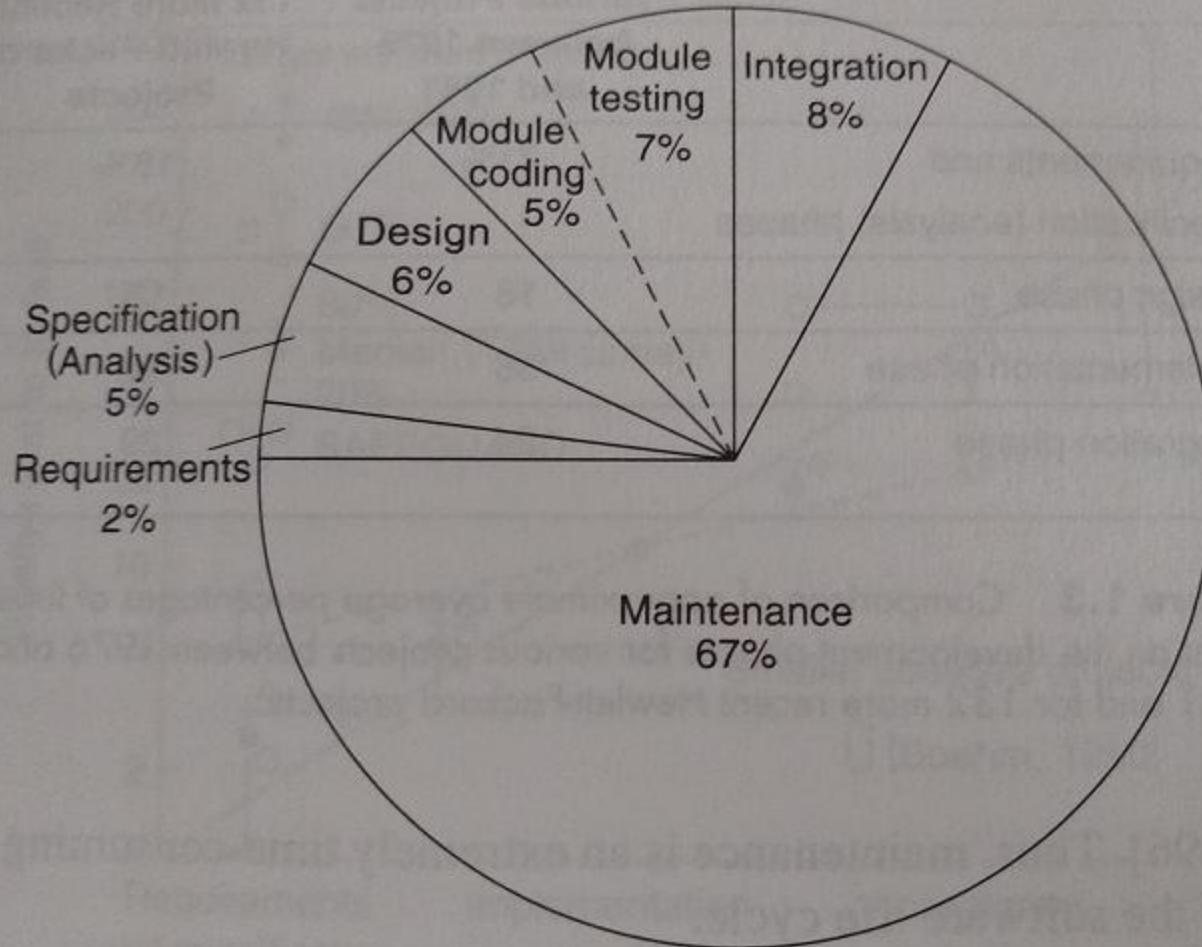
---

- ✧ Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- ✧ Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- ✧ Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.
- ✧ Processes should include activities such as prototyping and incremental delivery to cope with change.

## Key points

---

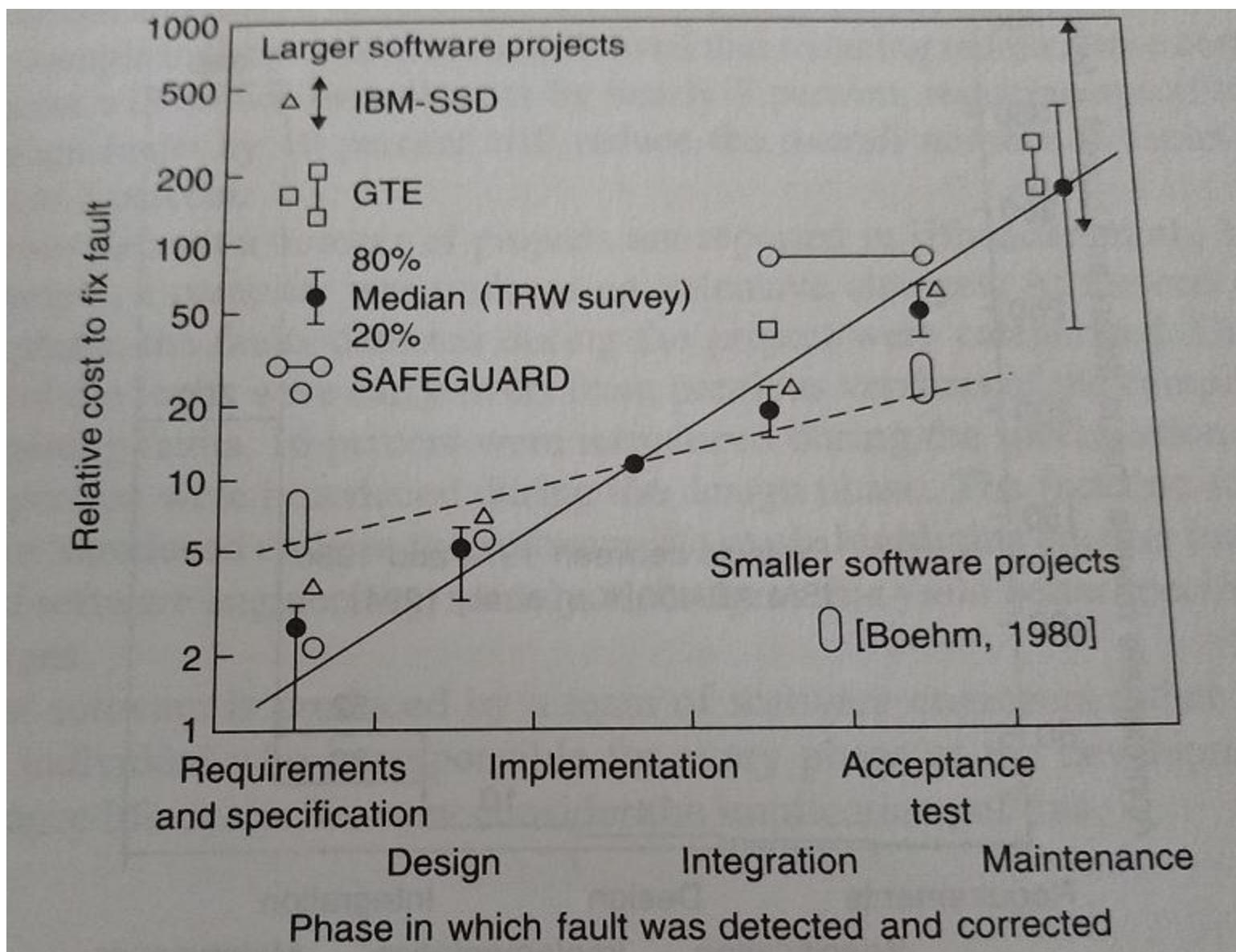
- ✧ Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- ✧ The principal approaches to process improvement are agile approaches, geared to reducing process overheads, and maturity-based approaches based on better process management and the use of good software engineering practice.
- ✧ The SEI process maturity framework identifies maturity levels that essentially correspond to the use of good software engineering practice.

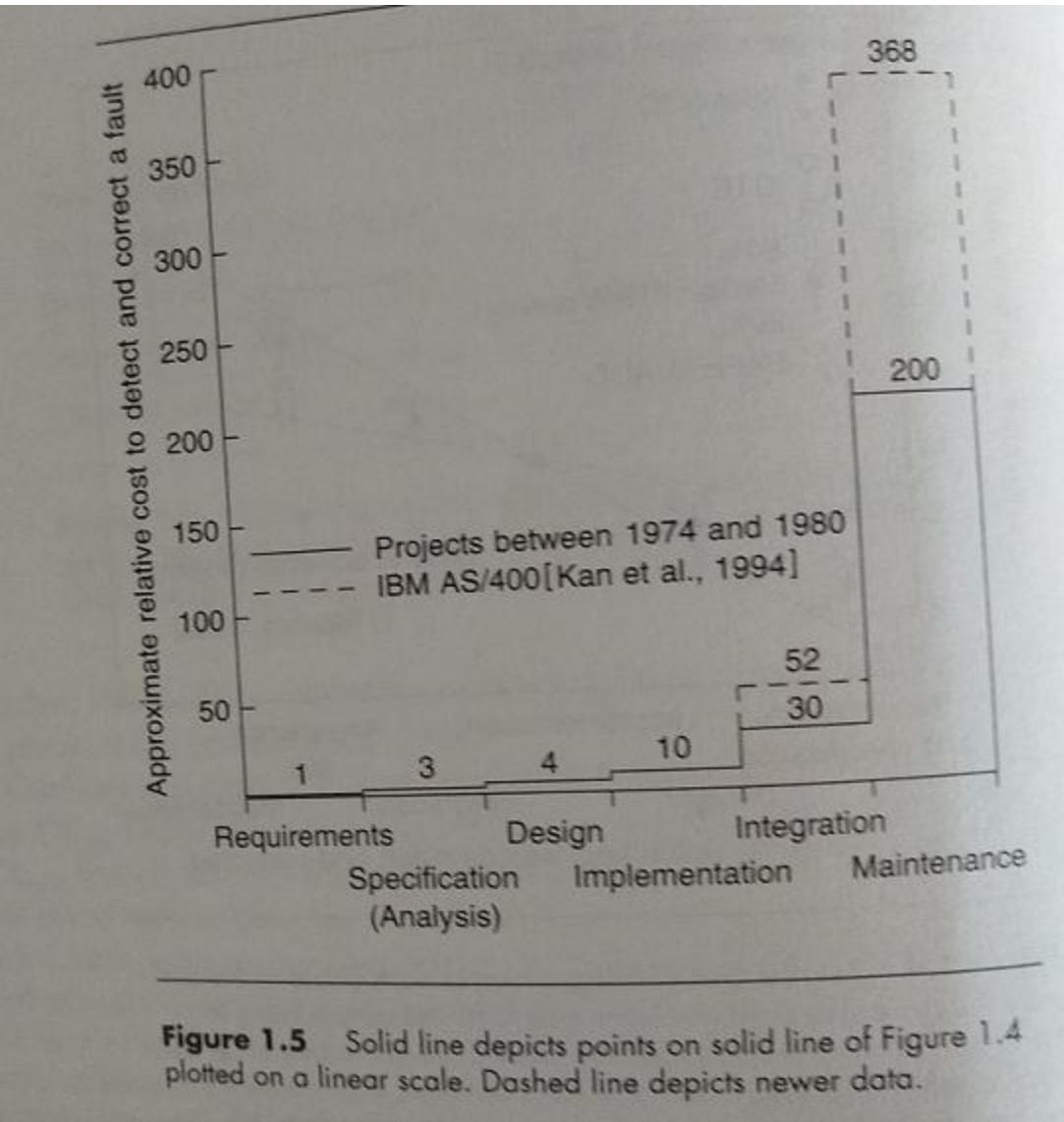


**Figure 1.2** Approximate relative costs of the phases of the soft-

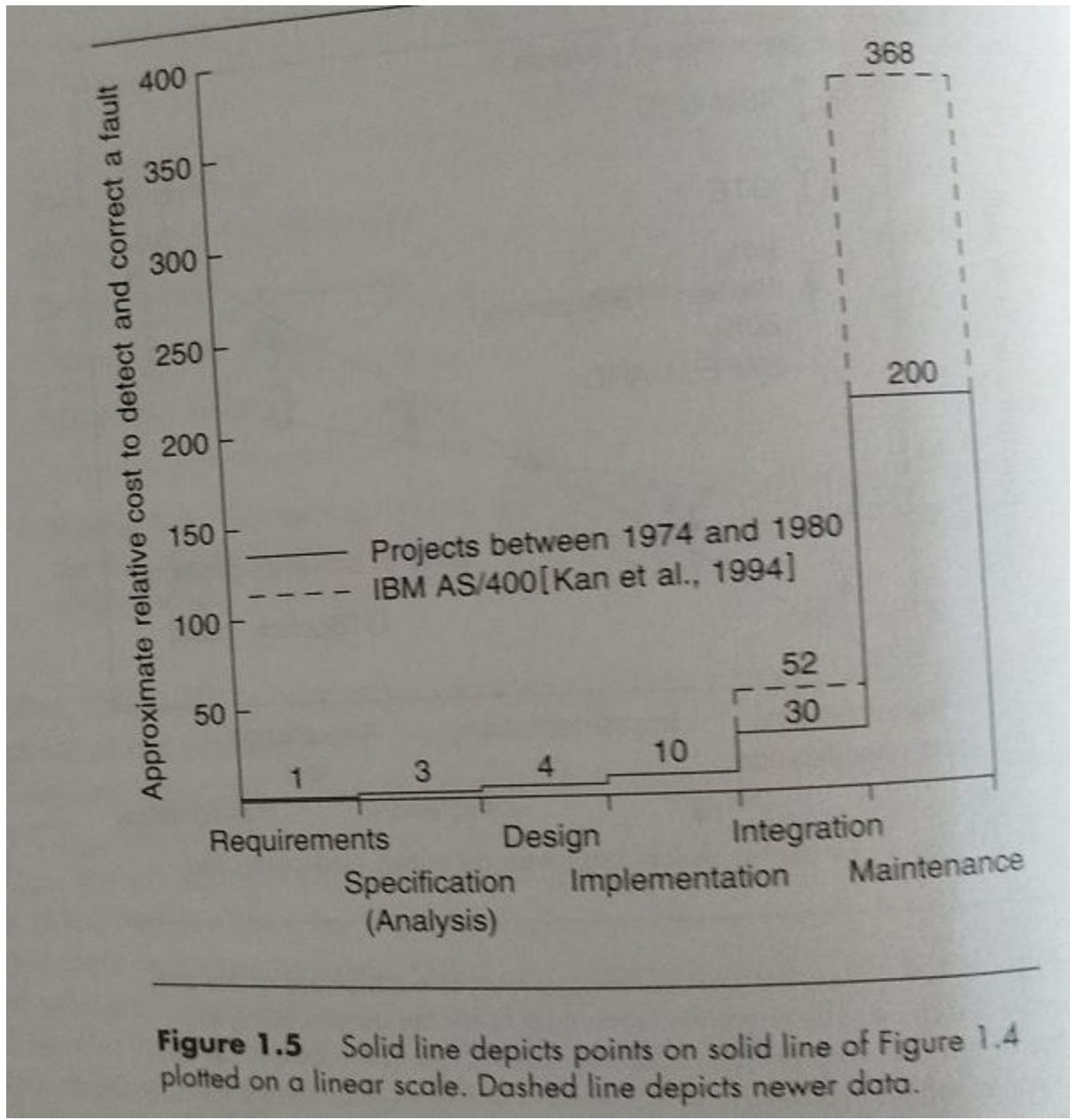
	Various Projects between 1976 and 1981	132 More Recent Hewlett-Packard Projects
Requirements and specification (analysis) phases	21%	18%
Design phase	18	19
Implementation phase	36	34
Integration phase	24	29

**Figure 1.3** Comparison of approximate average percentages of time spent on the development phases for various projects between 1976 and 1981 and for 132 more recent Hewlett-Packard projects.

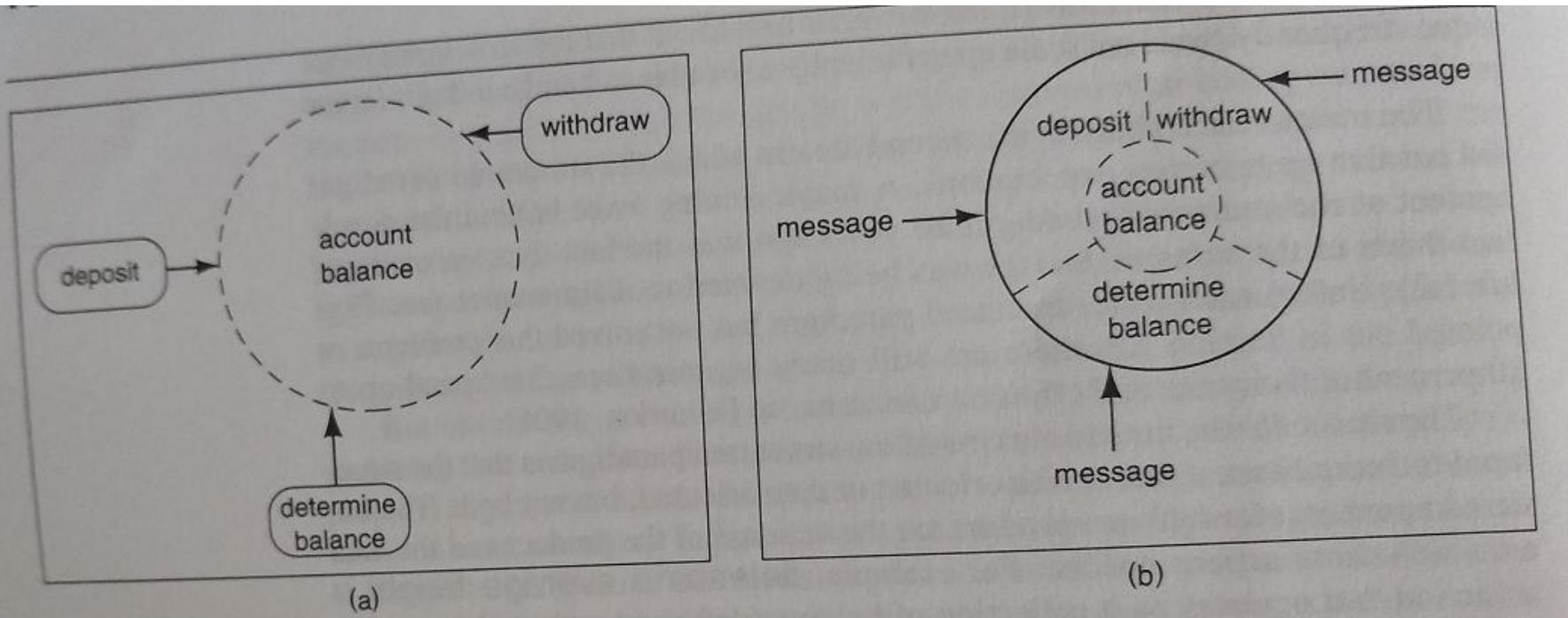




**Figure 1.5** Solid line depicts points on solid line of Figure 1.4 plotted on a linear scale. Dashed line depicts newer data.



**Figure 1.5** Solid line depicts points on solid line of Figure 1.4 plotted on a linear scale. Dashed line depicts newer data.



**Figure 1.6** Comparison of implementations of bank account using (a) structured paradigm and (b) object-oriented paradigm. Solid black line surrounding object denotes that details as to how account balance is implemented are not known outside object.

<b>Structured Paradigm</b>	<b>Object-Oriented Paradigm</b>
1. Requirements phase	1. Requirements phase
2. Specification (analysis) phase	2'. Object-oriented analysis phase
3. Design phase	3'. Object-oriented design phase
4. Implementation phase	4'. Object-oriented programming phase
5. Integration phase	5. Integration phase
6. Maintenance phase	6. Maintenance phase
7. Retirement	7. Retirement

**Figure 1.7** Comparison of life cycles of structured paradigm and object-oriented paradigm.

<b>Structured Paradigm</b>	<b>Object-Oriented Paradigm</b>
2. Specification (analysis) phase <ul style="list-style-type: none"> <li>• Determine what the product is to do</li> </ul>	2'. Object-oriented analysis phase <ul style="list-style-type: none"> <li>• Determine what the product is to do</li> <li>• Extract the objects</li> </ul>
3. Design phase <ul style="list-style-type: none"> <li>• Architectural design (extract the modules)</li> <li>• Detailed design</li> </ul>	3'. Object-oriented design phase <ul style="list-style-type: none"> <li>• Detailed design</li> </ul>
4. Implementation phase <ul style="list-style-type: none"> <li>• Implement in appropriate programming language</li> </ul>	4'. Object-oriented programming phase <ul style="list-style-type: none"> <li>• Implement in appropriate object-oriented programming language</li> </ul>

**Figure 1.8** Differences between structured paradigm and object-oriented paradigm.

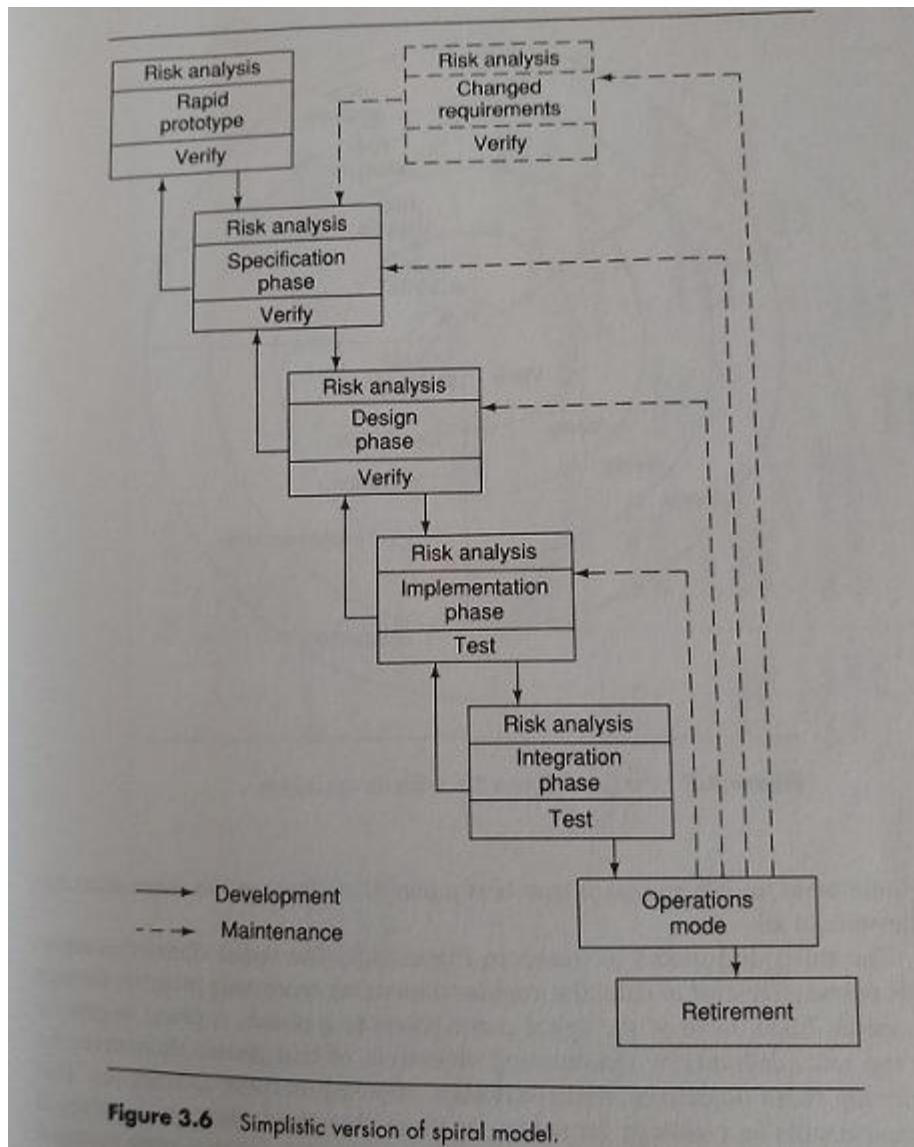


Figure 3.6 Simplistic version of spiral model.

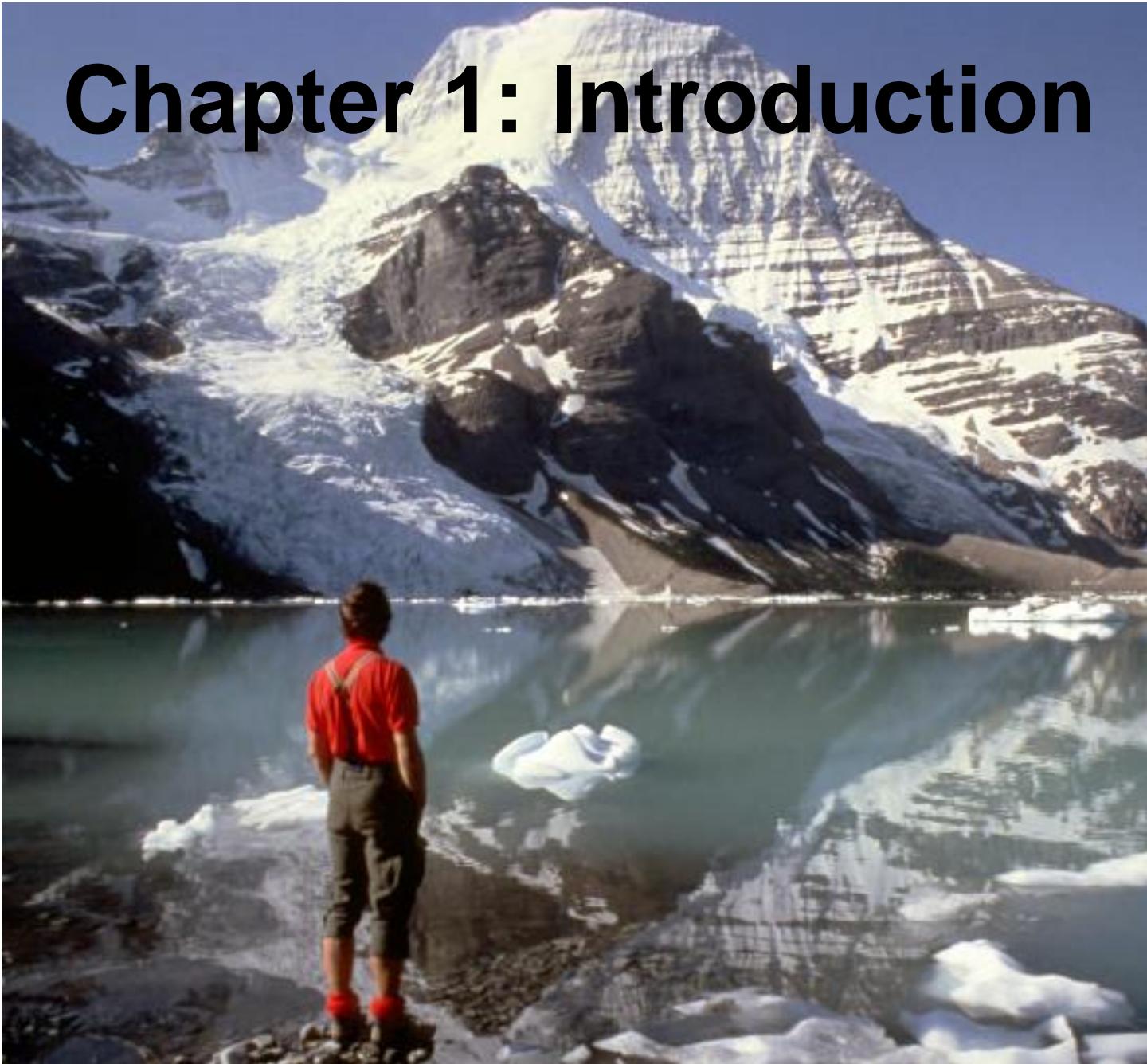
Life-Cycle Model	Strengths	Weaknesses
Build-and-fix model (Section 3.1)	Fine for short programs that will not require any maintenance	Totally unsatisfactory for nontrivial programs
Waterfall model (Section 3.2)	Disciplined approach Document-driven	Delivered product may not meet client's needs
Rapid prototyping model (Section 3.3)	Ensures that delivered product meets client's needs	See Chapter 9
Incremental model (Section 3.4)	Maximizes early return on investment Promotes maintainability	Requires open architecture May degenerate into build-and-fix
Synchronize-and-stabilize model (Section 3.5)	Future users' needs are met Ensures components can be successfully integrated	Has not been widely used other than at Microsoft
Spiral model (Section 3.6)	Incorporates features of all the above models	Can be used only for large-scale, in-house products Developers have to be competent in risk analysis and risk resolution
Object-oriented models (Section 3.7)	Supports iteration within phases, parallelism between phases	May degenerate into CABTAB

**Figure 3.10** Comparison of life-cycle models described in this chapter, including the section in which each is defined.

# Object-Oriented Software Engineering

Using UML, Patterns, and Java

# Chapter 1: Introduction



Title (German)	Software Engineering I: Softwaretechnik																							
Title	Software Engineering I: Software Technology																							
Short title	SWEng1																							
Type	Vorlesung																							
Hours per term	3V + 2Ü																							
ECTS Credits	6.0																							
Cycle	WS																							
Programs	<table border="1"> <thead> <tr> <th>Program</th> <th></th> <th>Comment</th> </tr> </thead> <tbody> <tr> <td>Informatics (Diploma)</td> <td>wa</td> <td></td> </tr> <tr> <td>Informatics (Bachelor)</td> <td>wa</td> <td></td> </tr> <tr> <td>Informatics (Master)</td> <td>wa</td> <td></td> </tr> <tr> <td>Applied Informatics (Master)</td> <td>wa</td> <td></td> </tr> <tr> <td>Maschinenwesen, Fachmodul Elektronik und Informatik</td> <td>vt</td> <td></td> </tr> <tr> <td>Information Systems (Master)</td> <td>pf</td> <td></td> </tr> </tbody> </table>			Program		Comment	Informatics (Diploma)	wa		Informatics (Bachelor)	wa		Informatics (Master)	wa		Applied Informatics (Master)	wa		Maschinenwesen, Fachmodul Elektronik und Informatik	vt		Information Systems (Master)	pf	
Program		Comment																						
Informatics (Diploma)	wa																							
Informatics (Bachelor)	wa																							
Informatics (Master)	wa																							
Applied Informatics (Master)	wa																							
Maschinenwesen, Fachmodul Elektronik und Informatik	vt																							
Information Systems (Master)	pf																							
Responsible for module	Bernd Bruegge																							
Content	<ul style="list-style-type: none"> <li>• Fundamentals about software engineering</li> <li>• Process models</li> <li>• Description and modelling techniques</li> <li>• System analysis - Requirements engineering</li> <li>• System design</li> <li>• Implementation</li> <li>• Principles of system development</li> </ul>																							
Learning goal																								
Prerequisites	Modul IN0006																							

# Intended audience

- *Informatics (Master, Bachelor)*
- *Information Systems (Master)*
- *Applied Informatics (Master)*
- *Diploma Students*
- *Mechanical Engineering: Module Elektronics and Informatics*

# Objectives of the Lectures

- *Appreciate the Fundamentals of Software Engineering:*
  - *Methodologies*
  - *Process models*
  - *Description and modeling techniques*
  - *System analysis - Requirements engineering*
  - *System design*
  - *Implementation: Principles of system development*

# Assumptions for this Class

- *Assumption:*
  - You have taken Module 006 EIST: Introduction to Software Engineering or a similar course
  - You have already experience in at least one analysis and design technique
- *Beneficial:*
  - You have had practical experience with a large software system
  - You have already participated in a large software project
  - You have experienced major problems.

# Times and Locations

- *Main lecture: HS 2*
  - *Tuesdays 16:00 – 18:00*
  - *Fridays 9:00 - 10:00*
- *Exercises: Thursday 8:00-10:00*
  - Registration starts today
  - Registration ends Thursday
  - Exercise sessions start on Thursday , Oct 16
- *Written Exams:*
  - *Mid-term*
  - *Final*

# Grading Criteria

*The final grade is the weighted average of the mid term (30%) and final grades (70%)*

- *To pass this course your final grade must be 4.0 or better*
- *Participation in the exercises is required (admission requirement for the final exam)*
- *Information about the exercises will be made available on the exercise portal*
- *Hours per week: 3 hours (lecture) + 2 hour (exercises)*
- *ECTS Credits: 6.0.*

# Focus: Acquire Technical Knowledge

- *Different methodologies ("philosophies") to model and develop software systems*
- *Different modeling notations*
- *Different modeling methods*
- *Different software lifecycle models (empirical control models, defined control models)*
- *Different testing techniques (eg. vertical testing, horizontal testing)*
- *Rationale Management*
- *Release and Configuration Management*

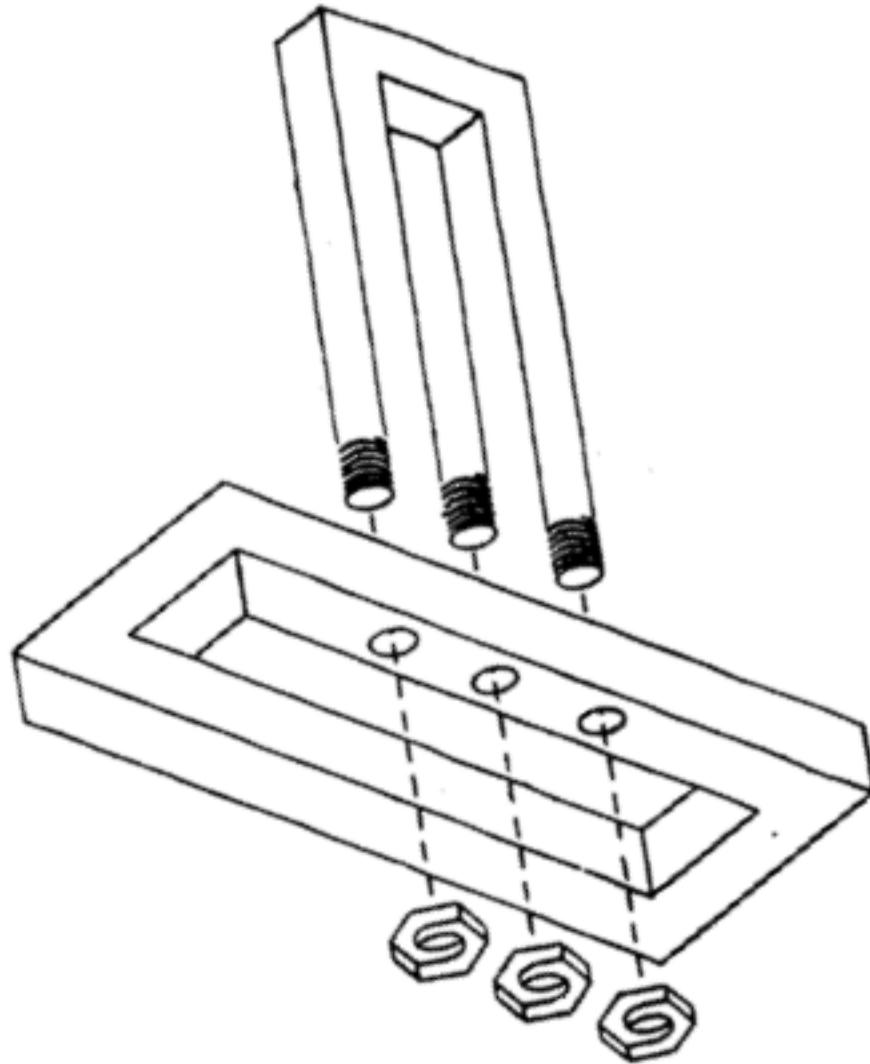
# Acquire Managerial Knowledge

- *Learn the basics of software project management*
- *Understand how to manage with a software lifecycle*
- *Be able to capture software development knowledge (Rationale Management)*
- *Manage change: Configuration Management*
- *Learn the basic methodologies*
  - *Traditional software development*
  - *Agile methods.*

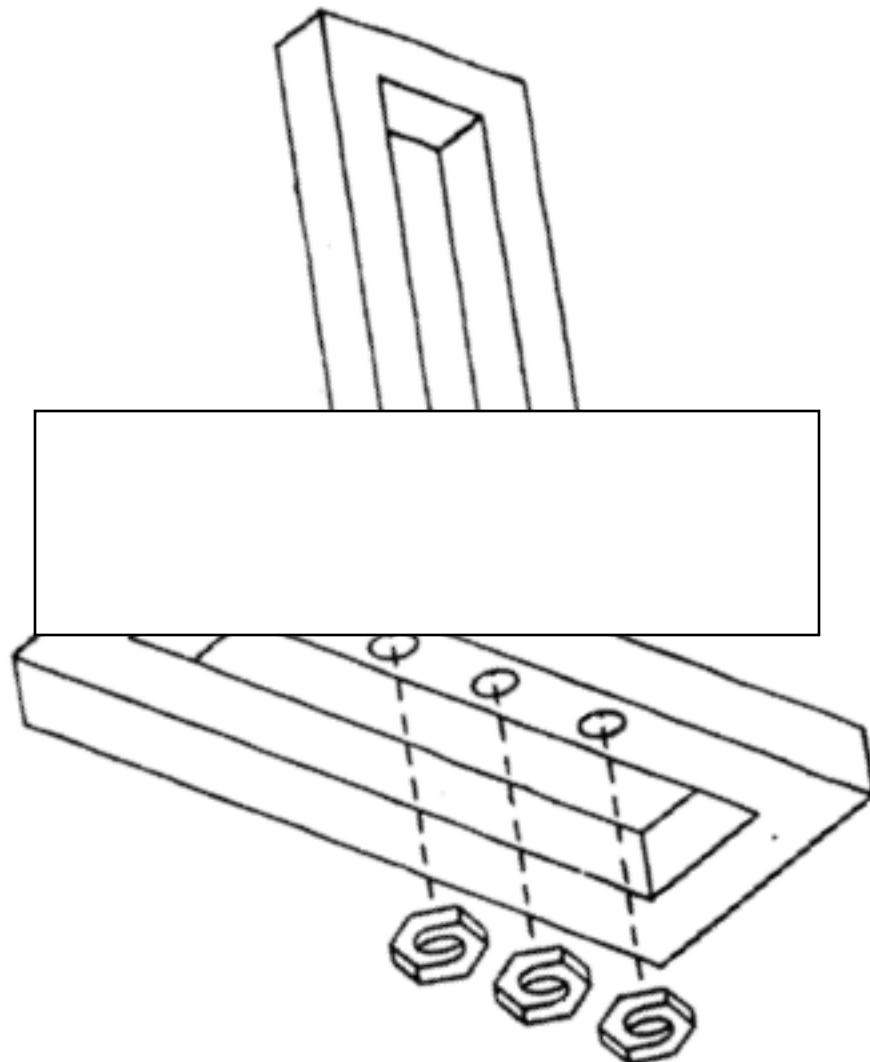
# Outline of Today's Lecture

- *The development challenge*
- *Dealing with change*
- *Concepts: Abstraction, Modeling, Hierarchy*
- *Methodologies*
- *Organizational issues*
  - *Lecture schedule*
  - *Exercise schedule*
  - *Associated Project*

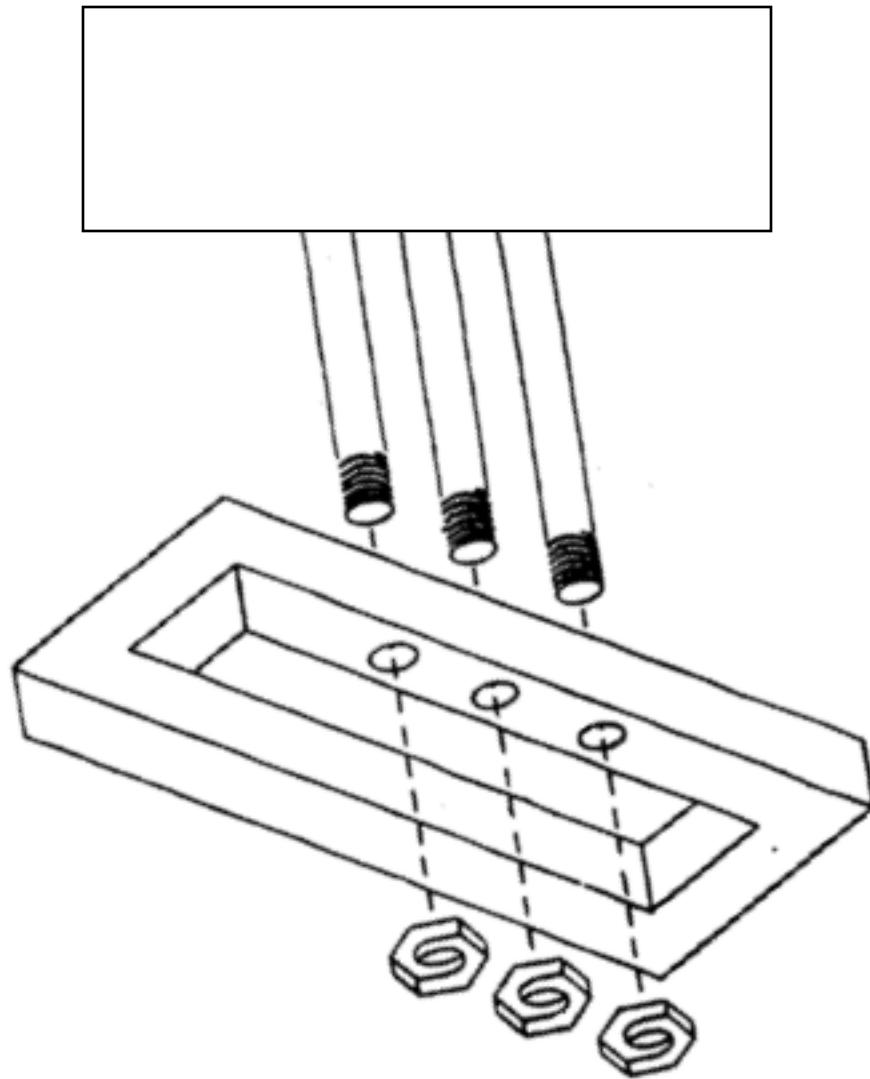
# Can you develop this system?



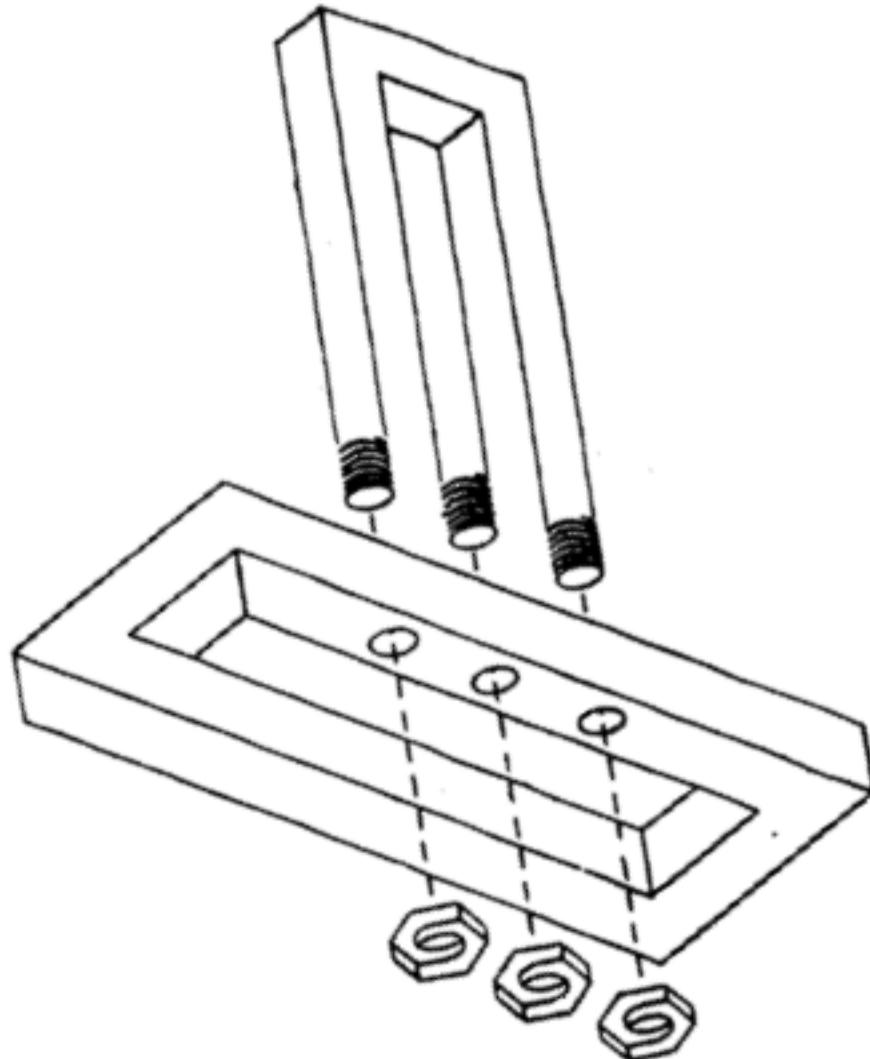
# Can you develop this system?



# Can you develop this system?



# Can you develop this system?



**The impossible  
Fork**

# Physical Model of the impossible Fork (Shigeo Fukuda)

See <http://illusionworks.com/mod/movies/fukuda/DisappearingColumn.mov>

# Physical Model of the impossible Fork (Shigeo Fukuda)

Additional material can be found on  
<http://illusionworks.com/mod/movies/fukuda/>  
Images may be subject to copyright

# Why is software development difficult?

- *The problem domain (also called application domain) is difficult*
- *The solution domain is difficult*
- *The development process is difficult to manage*
- *Software offers extreme flexibility*
- *Software is a discrete system*
  - *Continuous systems have no hidden surprises*
  - *Discrete systems can have hidden surprises! (Parnas)*

**David Lorge Parnas** is an early pioneer in software engineering who developed the concepts of modularity and information hiding in systems which are the foundation of object oriented methodologies.



# Software Engineering is more than writing Code

- *Problem solving*
  - *Creating a solution*
  - *Engineering a system based on the solution*
- *Modeling*
- *Knowledge acquisition*
- *Rationale management*

# Techniques, Methodologies and Tools

- **Techniques:**
  - *Formal procedures for producing results using some well-defined notation*
- **Methodologies:**
  - *Collection of techniques applied across software development and unified by a philosophical approach*
- **Tools:**
  - *Instruments or automated systems to accomplish a technique*
  - *CASE = Computer Aided Software Engineering*

# Computer Science vs. Engineering

- *Computer Scientist*
  - Assumes techniques and tools have to be developed.
  - Proves theorems about algorithms, designs languages, defines knowledge representation schemes
  - Has infinite time...
- *Engineer*
  - Develops a solution for a problem formulated by a client
  - Uses computers & languages, techniques and tools
- *Software Engineer*
  - Works in multiple application domains
  - Has only 3 months...
  - ...while changes occurs in the problem formulation (requirements) and also in the available technology.

# Software Engineering: A Working Definition

*Software Engineering is a collection of techniques, methodologies and tools that help with the production of*

*A high quality software system developed with a given budget before a given deadline while change occurs*

**Challenge: Dealing with complexity and change**

# Software Engineering: A Problem Solving Activity

- **Analysis:**
  - *Understand the nature of the problem and break the problem into pieces*
- **Synthesis:**
  - *Put the pieces together into a large structure*

*For problem solving we use techniques,  
methodologies and tools.*

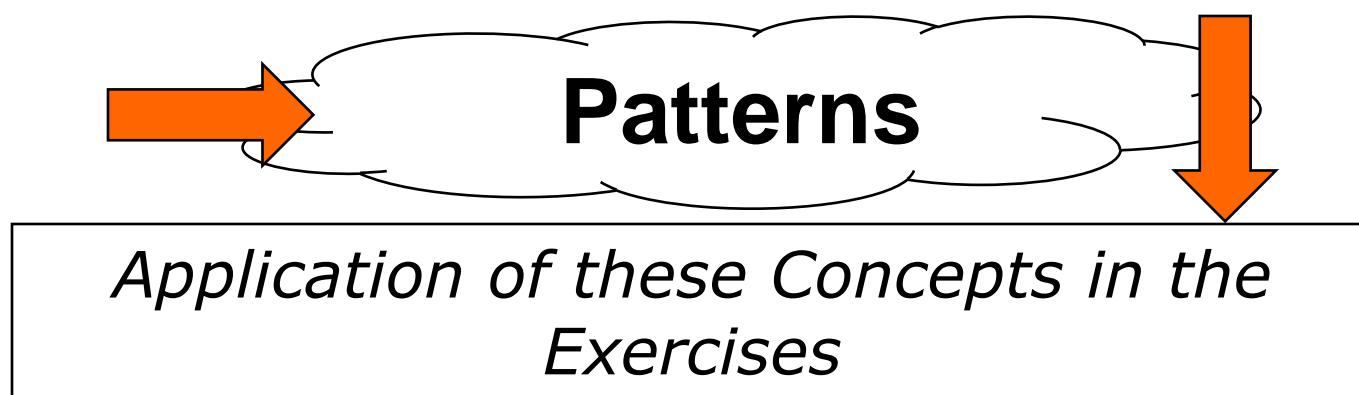
# Course Outline

## *Dealing with Complexity*

- *Notations (UML, OCL)*
- *Requirements Engineering, Analysis and Design*
  - *OOSE, SA/SD, scenario-based design, formal specifications*
- *Testing*
  - *Vertical and horizontal testing*

## *Dealing with Change*

- *Rationale Management*
  - *Knowledge Management*
- *Release Management*
  - *Big Bang vs Continuous Integration*
- *Software Life Cycle*
  - *Linear models*
  - *Iterative models*
  - *Activity-vs Entity-based views*



# Tentative Lecture Schedule

<b>Block 1 (Oct 14 – Oct 17)</b>	Introduction, Methodologies
<b>Block 2 (Oct 21 – Oct 24)</b>	The UML notation
<b>Block 3 (Oct 28 – Nov 14)</b>	Requirements Elicitation, Analysis/Design
<b>Block 4 (Nov 18 – Nov 21)</b>	Build and Release Management
<b>Block 5 (Nov 25 – Nov 28)</b>	Testing, Programming Contest
<b>Block 6 (Dec 9– Dec 12)</b>	Software Lifecycle Models
<b>Block 7 (Dec 16– Jan 9)</b>	Detailed Design/Implementation Concepts (Mapping models to code)

**Midterm Exam: Dec 18**

**Christmas Break until January 8**

## **Block 8 (January and February)**

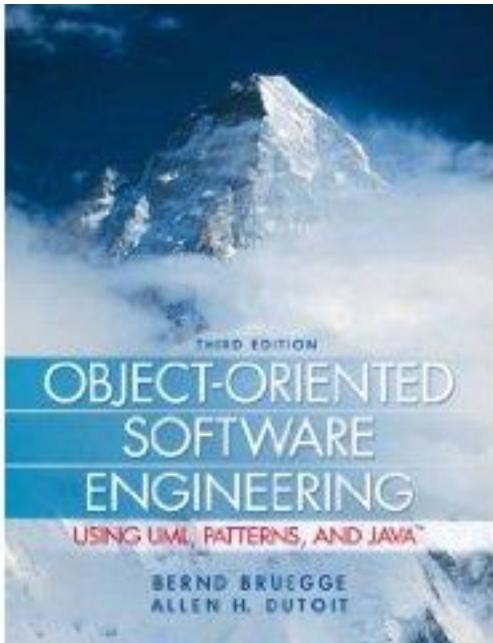
- Specific methodologies: XP, Scrum, Rugby, Royce
- Plus invited lectures from industry

**Final Exam: Feb 9**

# Exercises

- *The exercises are organized in two groups*
- *Each group has one exercise session (2 hour) per week: Thursday 8:00-10:00*
- *Registration, attendance in the exercise sessions and attempting/finishing the homeworks is mandatory.*

# Textbook



*Bernd Bruegge, Allen H. Dutoit*

Object-Oriented Software Engineering:  
Using UML, Patterns and Java, 3<sup>rd</sup>  
Edition

Publisher: **Prentice Hall**, Upper Saddle  
River, NJ, 2009;

*ISBN-10: 0136061257*

*ISBN-13: 978-0136061250*

- *Additional readings will be added during each lecture.*

# Questions?

- *Lecture Portal:*
  - *The lecture slides will be posted in PDF format after the lecture is given*
- *Exercise Portal:*
  - *Separate home page will be set up for the exercise materials*
- *What happens if I don't participate in the exercises?*

# What happens if I don't participate in the exercises?

Play the movie

[http://www.youtube.com/watch?v=\\_VFS8zRo0pc](http://www.youtube.com/watch?v=_VFS8zRo0pc)

# **UML**

## **The Unified Modeling Language**

# Introduction

- **Modeling:** drawing a flowchart listing the steps carried out by an application.
- **Why do we use modeling?**

Defining a model makes it easier to break up a complex application or a huge system into simple, discrete pieces that can be individually studied. We can focus more easily on the smaller parts of a system and then understand the "big picture."
- **The reasons behind modeling can be summed up in two words:**
  - Readability
  - Reusability

- **Readability:** brings clarity—ease of understanding. Understanding a system is the first step in either building or enhancing a system. This involves knowing what a system is made up of, how it behaves, and so forth. Depicting a system to make it readable involves capturing the structure of a system and the behavior of the system.
- **Reusability:** is the byproduct of making a system readable. After a system has been modeled to make it easy to understand, we tend to identify similarities or redundancy, be they in terms of functionality, features, or structure. UML provides the ability to capture the characteristics of a system by using notations. UML provides a wide array of simple notations for documenting systems based on the object-oriented design principles. These notations are called the nine diagrams of UML.

# What is UML?

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML is a very important part of developing object oriented software and the software development process. The UML uses graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

# Goals of UML

**The primary goals in the design of the UML were:**

- Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the OO tools market.

# Why use UML

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, and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. They recognize the need to solve the architectural problems, such as physical distribution, concurrency, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web.

→ The Unified Modeling Language (UML) was designed to respond to these needs.

# UML Diagrams

UML is made up of nine diagrams that can be used to model a system at different points of time in the software life cycle of a system. The nine UML diagrams are:

- **Use case diagram**
- **Class diagram**
- **Object diagram**
- **State diagram**
- **Activity diagram**
- **Sequence diagram**
- **Collaboration diagram**
- **Component diagram**
- **Deployment diagram**

# UML Diagram Classification

- A software system can be said to have three distinct characteristics: *static*, *dynamic*, and *implementation*.
- **Static:** the structural aspect of the system, define what parts the system is made up of.
- **Dynamic:** The behavioral features of a system; for example, the ways a system behaves in response to certain events or actions are the dynamic characteristics of a system.
- **Implementation:** The implementation characteristic of a system is an entirely new feature that describes the different elements required for deploying a system.

The UML diagrams that fall under each of these categories are:

- **Static**

- Use case diagram
- Class diagram

- **Dynamic**

- Object diagram
- State diagram
- Activity diagram
- Sequence diagram
- Collaboration diagram

- **Implementation**

- Component diagram
- Deployment diagram

# Use Case Diagram

The Use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." The Use case diagram shows which actors interact with each use case.

- UML Use Case Diagrams (**UCDs**) can be used to describe the functionality of a system, they capture the functional aspects and business process in the system.
- UCDs have only 4 major elements: The **actors** that the system you are describing interacts with, the **system** itself, the **use cases**, or services, that the system knows how to perform, and the lines that represent **relationships** between these elements.

- **Actors:** An actor portrays any entity (or entities) that performs certain roles in a given system. The most obvious candidates for actors are the humans in the system. If your system interacts with other systems (databases, servers maintained by other people, legacy systems) you will be best to treat these as actors.



- **Use case:** A use case in a use case diagram is a visual representation of a distinct business functionality in a system.

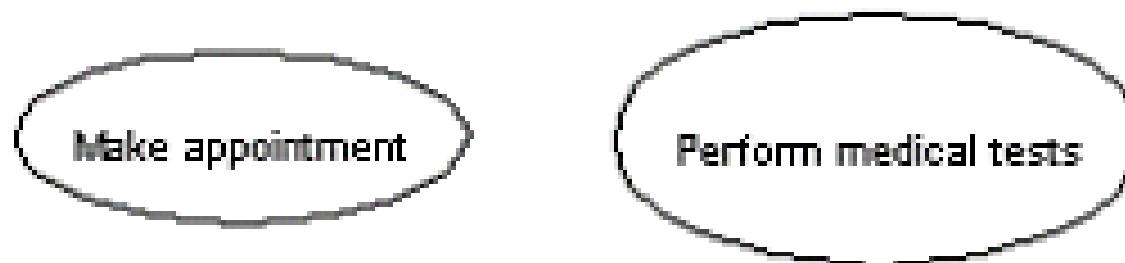
- **As-is scenarios** describe a current situation. During reengineering, for example, the current system is understood by observing users and describing their actions as scenarios. These scenarios can then be validated for correctness and accuracy with the users.
- **Visionary scenarios** describe a future system. Visionary scenarios are used both as a point in the modeling space by developers as they refine their ideas of the future system and as a communication medium to elicit requirements from users. Visionary scenarios can be viewed as an inexpensive prototype.
- **Evaluation scenarios** describe user tasks against which the system is to be evaluated. The collaborative development of evaluation scenarios by users and developers also improves the definition of the functionality tested by these scenarios.
- **Training scenarios** are tutorials used for introducing new users to the system. These are step-by-step instructions designed to hand-hold the user through common tasks.

### **Questions for identifying scenarios**

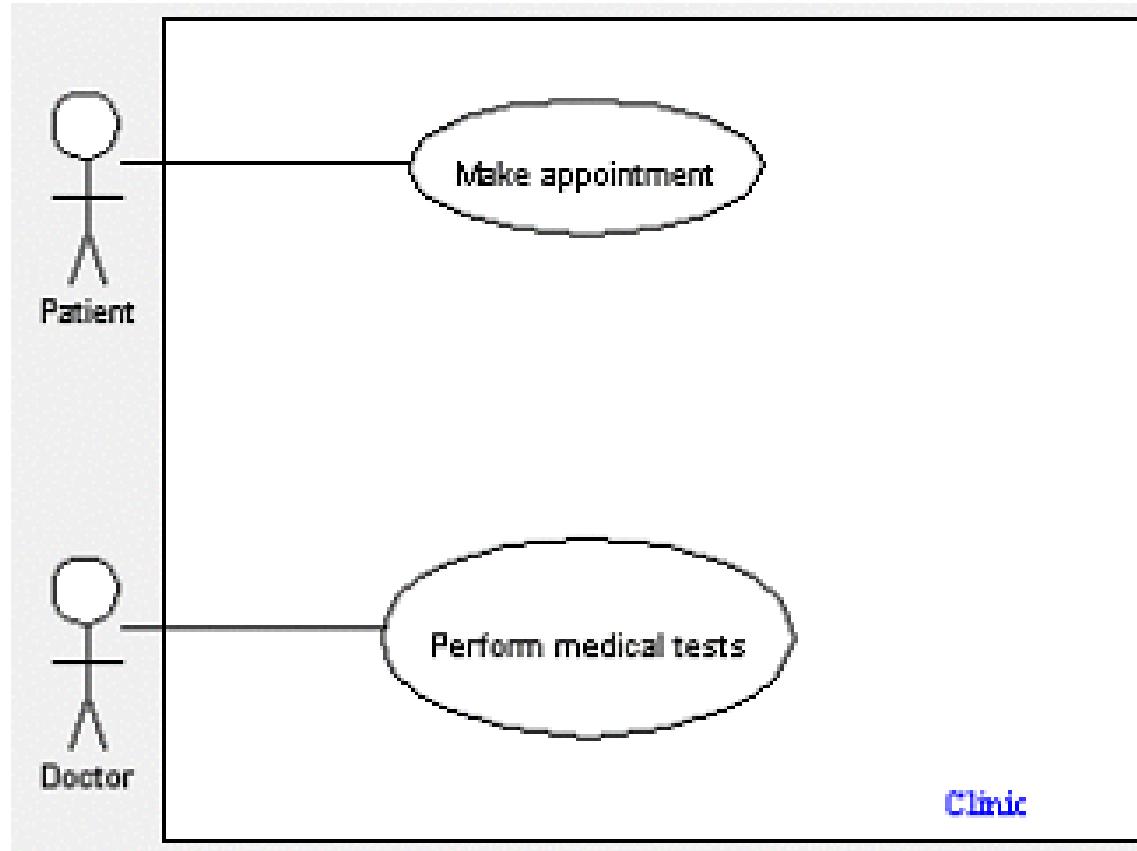
- What are the tasks that the actor wants the system to perform?
- What information does the actor access? Who creates that data? Can it be modified or removed? By whom?
- Which external changes does the actor need to inform the system about? How often? When?
- Which events does the system need to inform the actor about? With what latency?

As the first step in identifying use cases, you should list the discrete business functions in your problem statement. Each of these business functions can be classified as a potential use case.

→ A use case is an external view of the system that represents some action the user might perform in order to complete a task.



- **System boundary:** A system boundary defines the scope of what a system will be. A system boundary of a use case diagram defines the limits of the system.

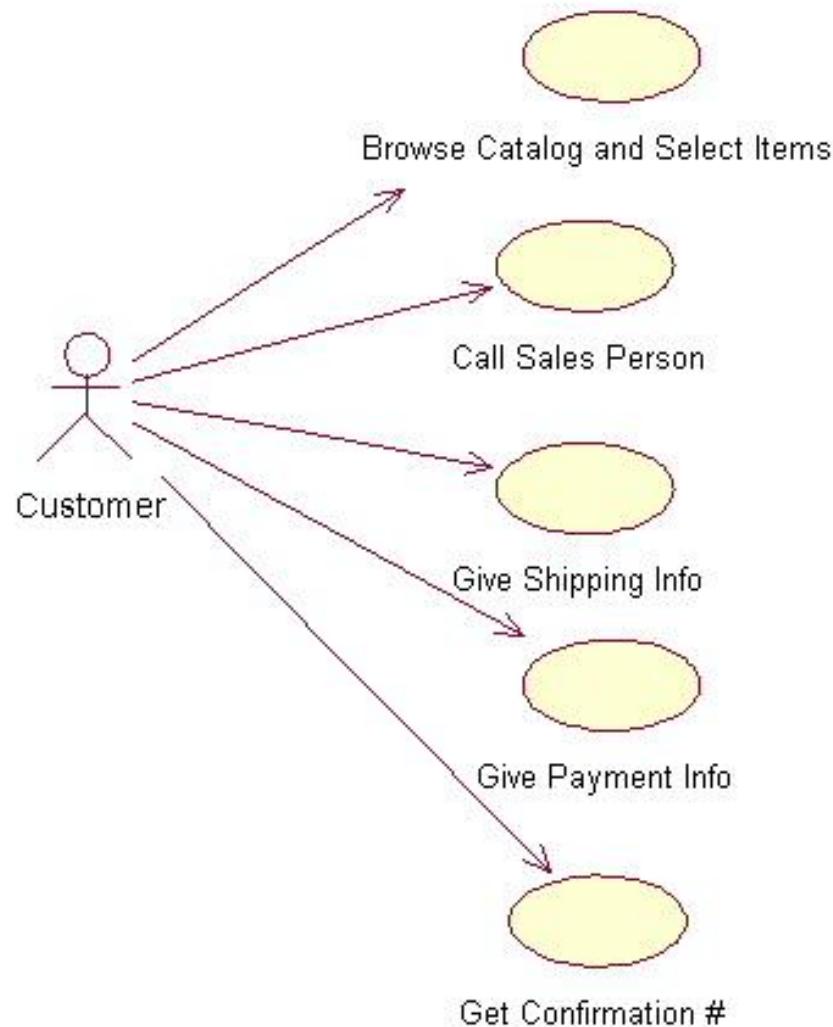


## Example 1

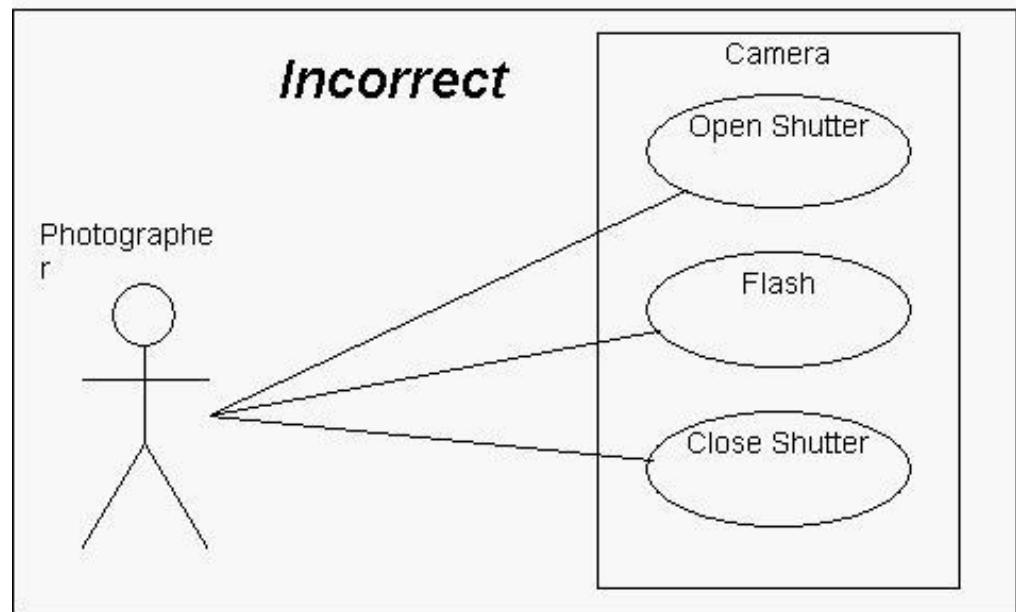
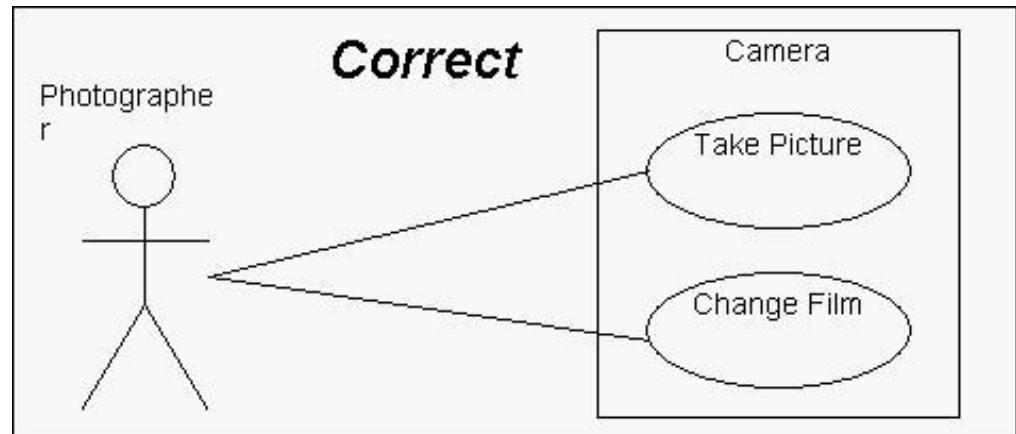
a user placing an order with a sales company might follow these steps.

- Browse catalog and select items.
- Call sales representative.
- Supply shipping information.
- Supply payment information.
- Receive conformation number from salesperson.

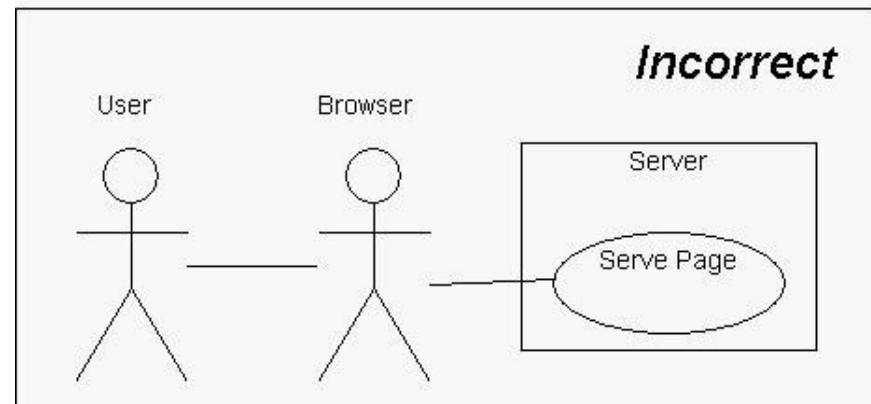
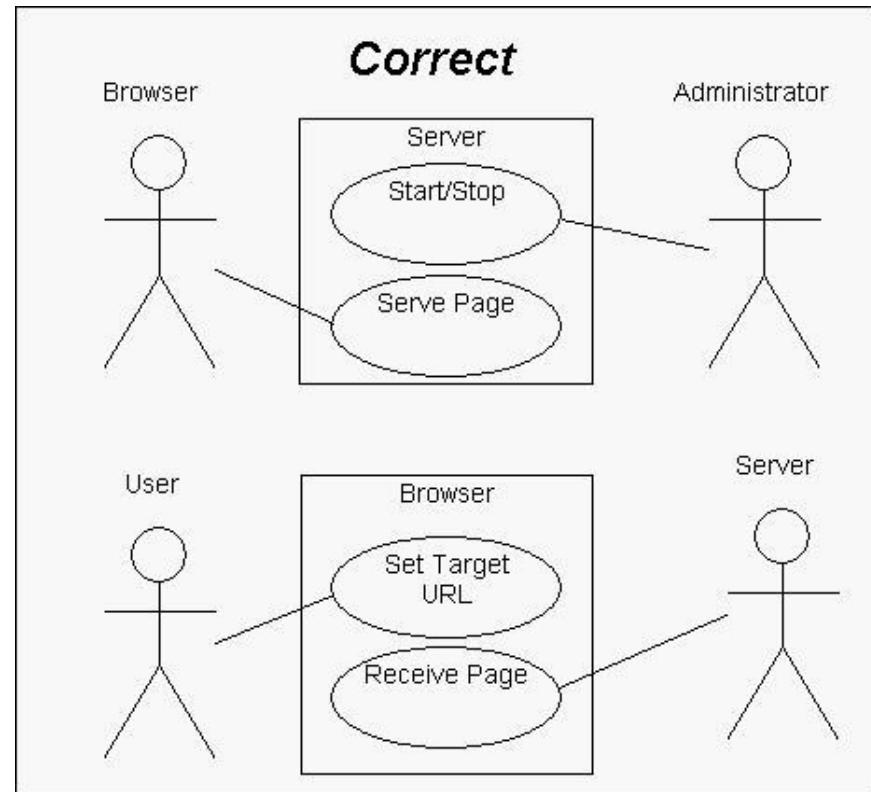
**The previous steps would generate this simple use case diagram:**



**Example2:** In the diagram below we would like to represent the use cases for a camera



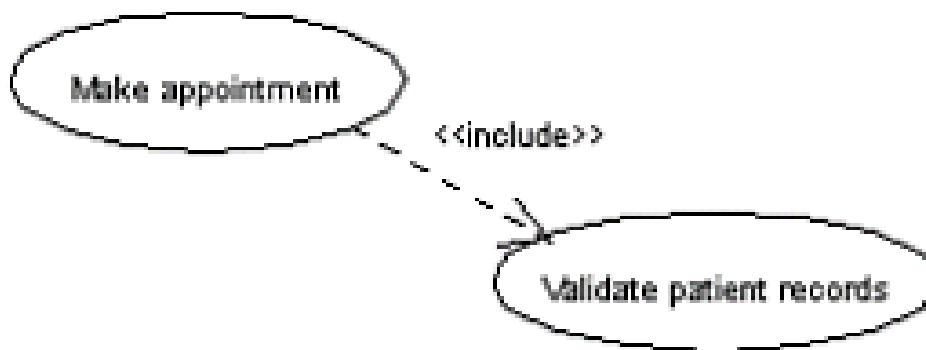
**Example3:** Suppose you wanted to diagram the interactions between a user, a web browser, and the server it contacts.



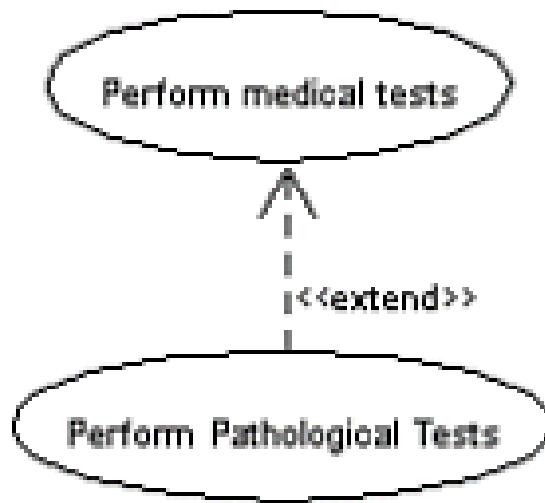
# Relationships in Use Cases

- A relationship between two use cases is basically a dependency between the two use cases. Defining a relationship between two use cases is the decision of the modeler of the use case diagram.
- Reuse of an existing use case using different types of relationships reduces the overall effort required in defining use cases in a system.
- Use case relationships can be one of the following:
  - **Include/uses**
  - **Extend**

- **Include/uses:** When a use case is depicted as using the functionality of another use case in a diagram, this relationship between the use cases is named as an *include* relationship.
  - Literally speaking, in an *include* relationship, a use case includes the functionality described in the another use case as a part of its business process flow.
  - An include relationship is depicted with a directed arrow having a dotted shaft. The tip of the arrowhead points to the parent use case and the child use case is connected at the base of the arrow.

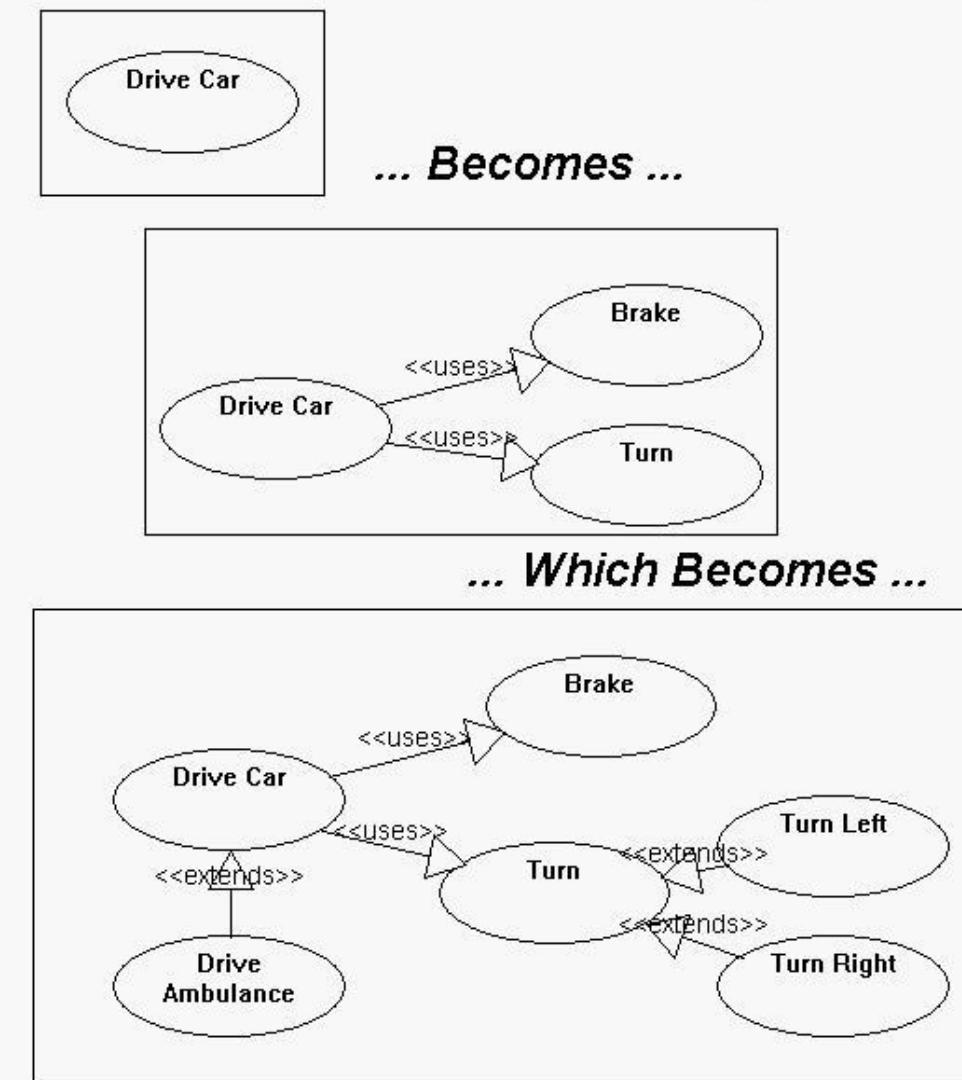


- **Extend:** In an *extend* relationship between two use cases, the child use case adds to the existing functionality and characteristics of the parent use case.
- The tip of the arrowhead points to the parent use case and the child use case is connected at the base of the arrow. The stereotype "<<extend>>" identifies the relationship as an extend relationship.



## *Evolution of a UML Use Case Diagram*

### Example 4



# **Assignment No. 1:** Airline Reservation system with focus on Checkin, Weigh Luggage and Assign seat.

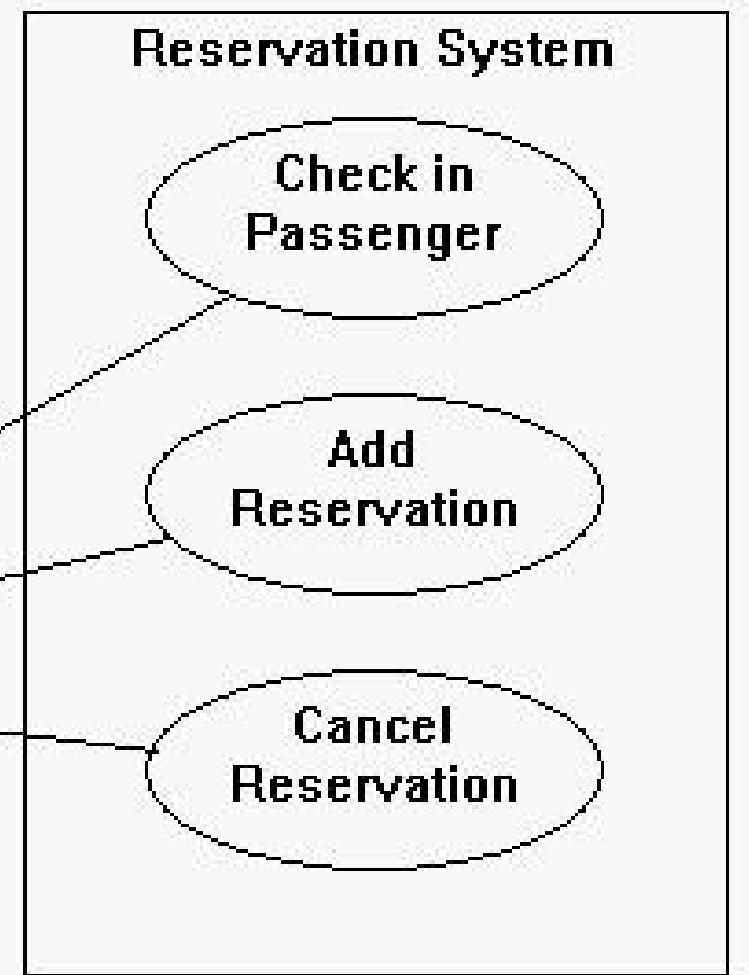
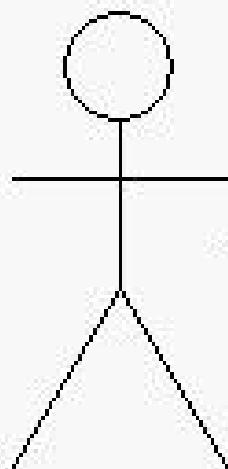
## **Example 5:**

Suppose you wanted to add detail to the diagram shown in the next slide, representing an airline reservation system. First, you would create a separate diagram for the top-level services, and then you would add new use cases that make up the top-level ones. There is a uses edge from "Check in Passenger" to "Weigh Luggage" and from "Check in Passenger" to "Assign Seat"; this indicates that *in order to Check in a Passenger, Luggage must be Weighed and a Seat must be Assigned*.

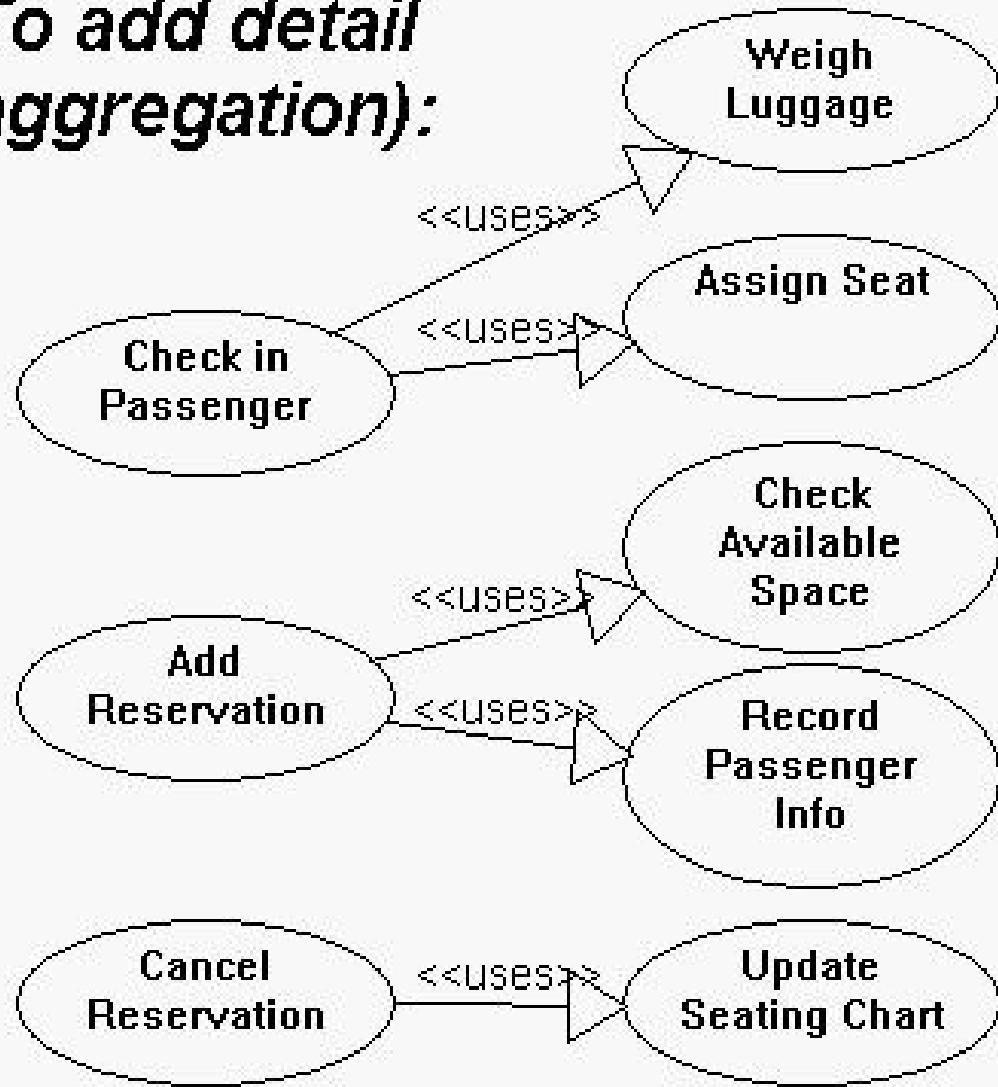
Similarly, the diagram indicates that in order to add a reservation to the system, the available space must be checked and the passenger's information must be recorded

# *Initial Design:*

Ticket Clerk



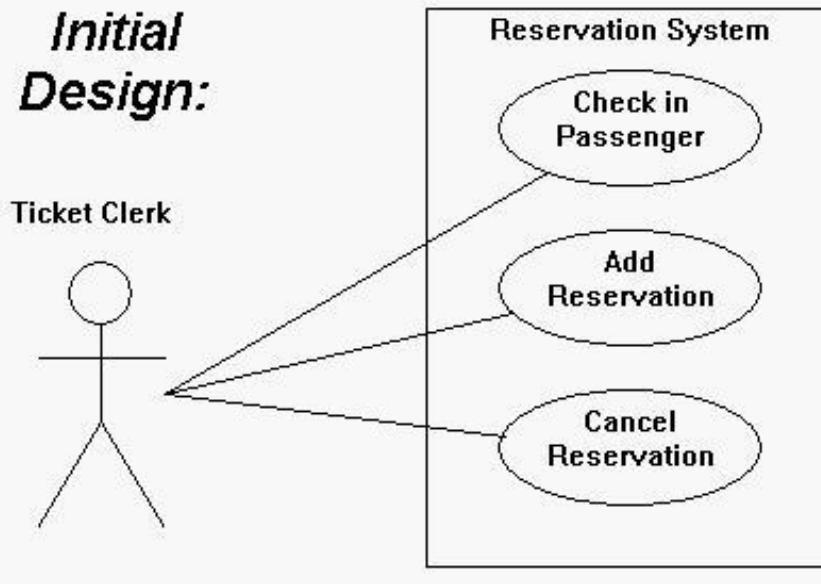
**To add detail  
(aggregation):**



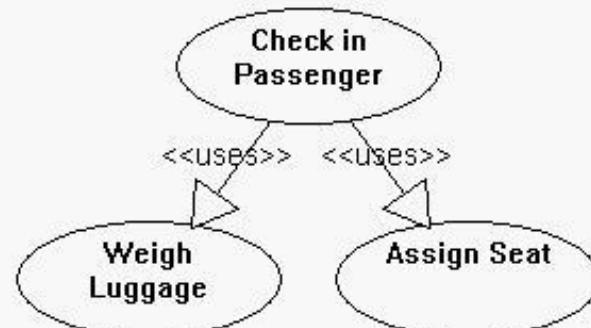
**More detailed  
design**

what you would like to show is that not all of the seats aboard the airplane are exactly alike (some window and some aisle seats), and sometimes passengers will express a preference for one of these types of seats but not the other. But of course, they cannot just be given their preference right away, because the seat they want might not be available. Therefore, the process of assigning a window seat involves checking for the availability of window seats, whereas the process of assigning an aisle seat involves checking for the availability of aisle seats

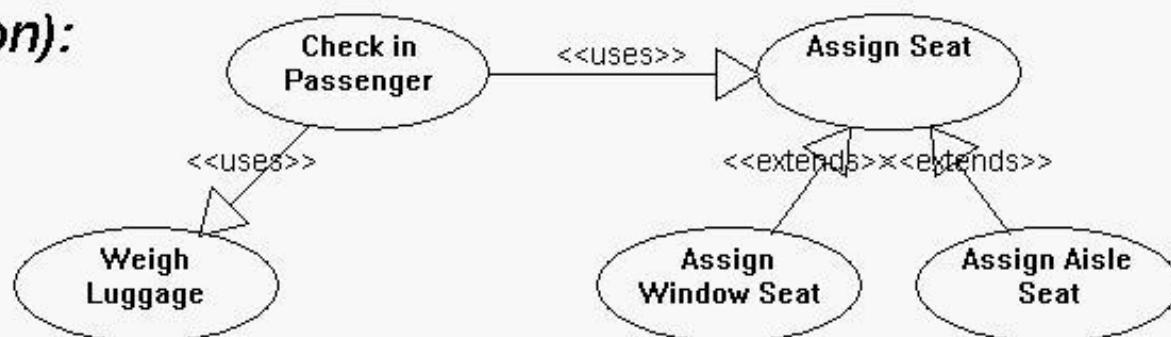
## *Initial Design:*



## *Sub-Diagram:*



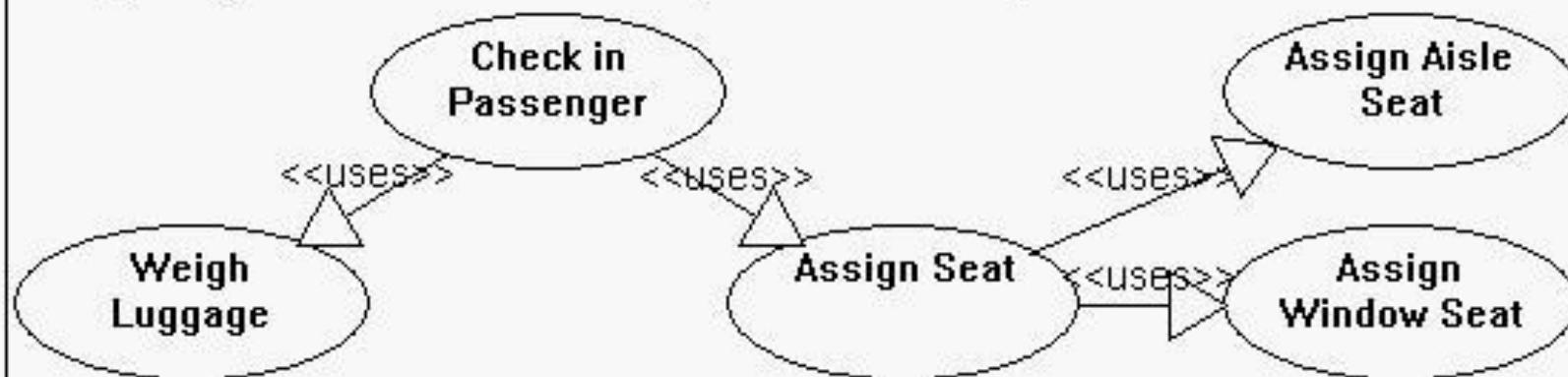
## *To add detail (extension):*



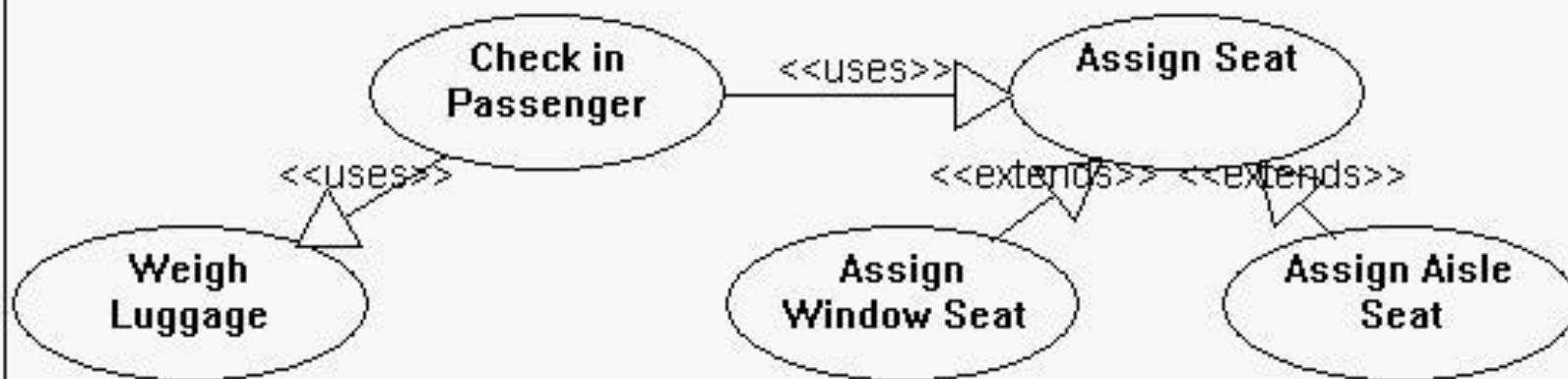
# What is the difference between extends and uses/include?

- "*X uses Y*" or "*X include Y*" indicates that the task "*X*" **has a** subtask "*Y*"; that is, in the process of completing task "*X*", task "*Y*" will be completed at least once
- "*X extends Y*" indicates that "*X*" **is a** task to the same type as "*Y*", but "*X*" is a special, more specific case of doing "*Y*". That is, doing *X* is a lot like doing *Y*, but *X* has a few extra processes to it that go above and beyond the things that must be done in order to complete *Y*.

*Trying to add detail (incorrect):*



*Trying to add detail (correct):*



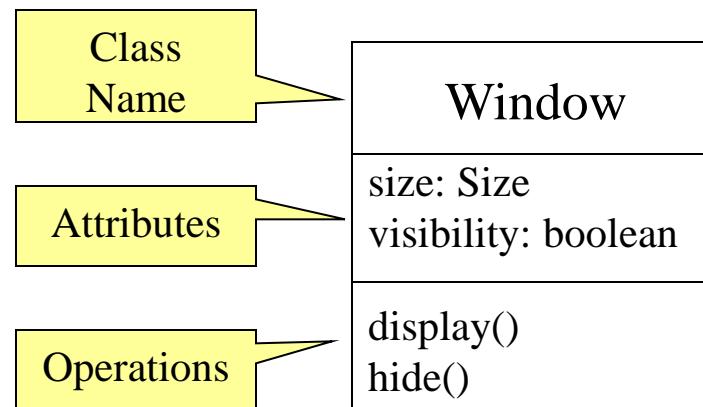
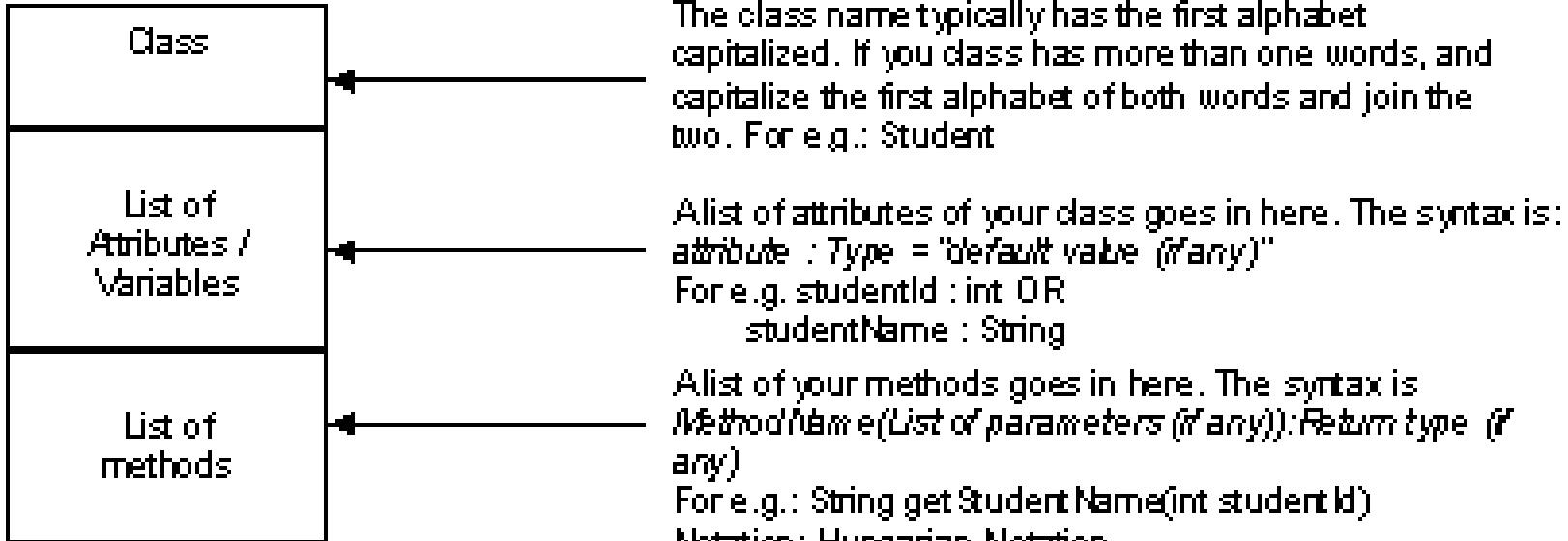
# Class Diagram

- **Definition:** A class diagram is a diagram showing a collection of classes and interfaces, along with the collaborations and relationships among classes and interfaces.
- When you designed the use cases, you must have realized that the use cases talk about "what are the requirements" of a system?
- The aim of designing classes is to convert this "what" to a "how" for each requirement. Each use case is further analyzed and broken up into atomic components that form the basis for the classes that need to be designed.

- A class diagram is a pictorial representation of the *detailed* system design.
- A thing to remember is that a class diagram is a static view of a system. The structure of a system is represented using class diagrams. Class diagrams are referenced time and it used by the developers while implementing the system.
- Class diagrams are used in nearly all Object Oriented software designs. Use them to describe the Classes of the system and their relationships to each other.
- Note that these diagrams describe the relationships between *classes*, not those between specific *objects* instantiated from those classes

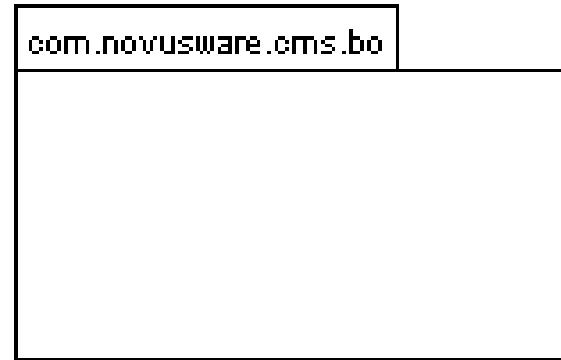
## Elements of a Class Diagram

- **Class:** A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as *methods*. A class also has properties that reflect unique features of a class. The properties of a class are called *attributes*.
- As an example, let us take a class named Student. A Student class represents student entities in a system. The Student class encapsulates student information such as student id #, student name, and so forth. Student id, student name, and so on are the attributes of the Student class. The Student class also exposes functionality to other classes by using methods such as `getStudentName()`, `getStudentId()`, and the like



- *Interface*: An interface is a variation of a class. As we saw from the previous point, a class provides an encapsulated implementation of certain business functionality of a system. An interface on the other hand provides only a definition of business functionality of a system. A separate class implements the actual business functionality.
- You can define an abstract class that declares business functionality as abstract methods. A child class can provide the actual implementation of the business functionality

- *Package*: A package provides the ability to group together classes and/or interfaces that are either similar in nature or related. Grouping these design elements in a package element provides for better readability of class diagrams, especially complex class diagrams.



# Relationships Between Classes

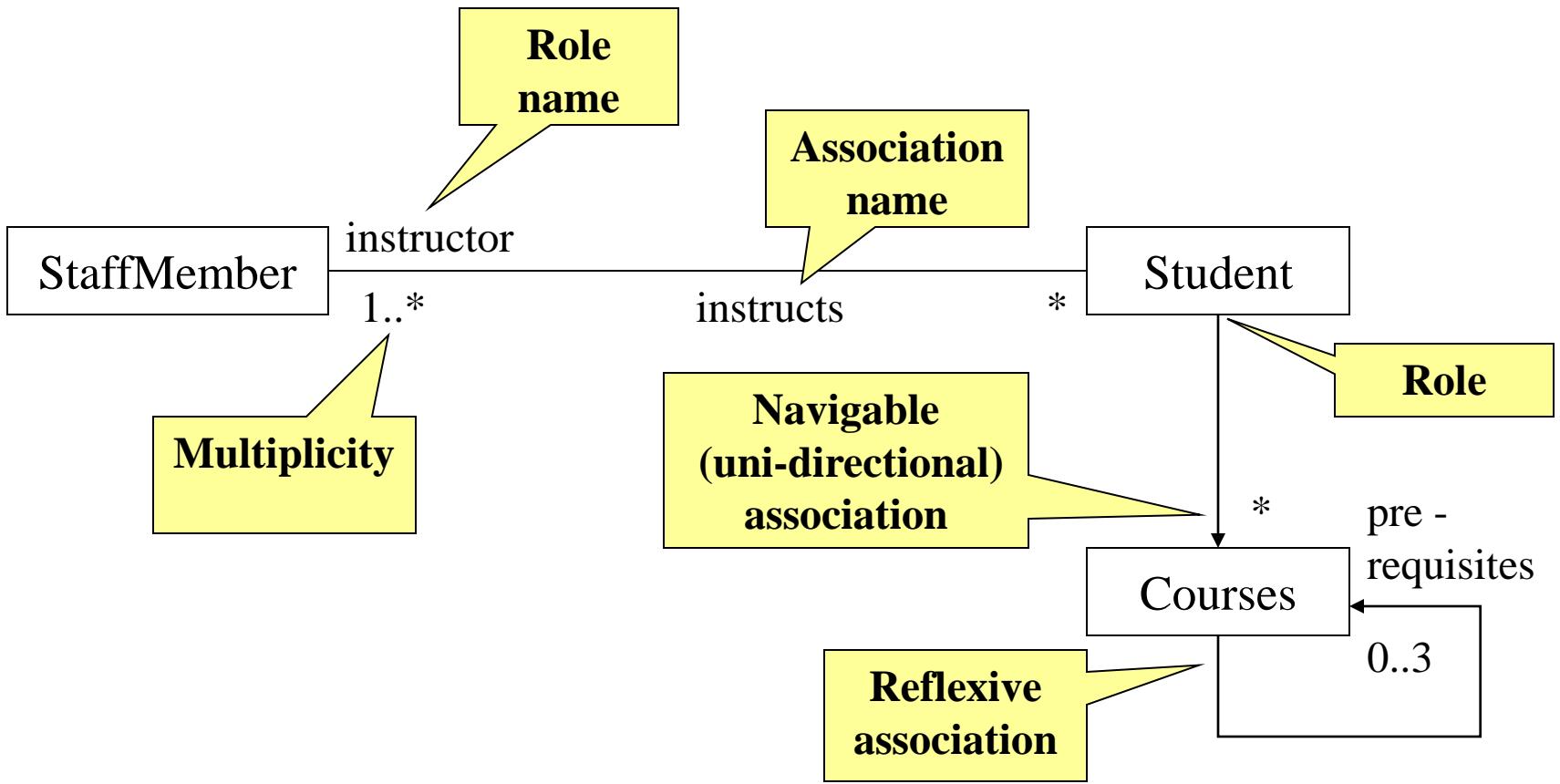
- *Lines* that model the relationships between classes and interfaces in the system.
- *Generalization*
  - *Inheritance*: a solid line with a solid arrowhead that points from a sub-class to a superclass or from a sub-interface to its super-interface.
  - *Implementation*: a dotted line with a solid arrowhead that points from a class to the interface that it implements
- *Association* -- a solid line with an open arrowhead that represents a "has a" relationship. The arrow points from the containing to the contained class

- A semantic relationship between two or more classes that specifies connections among their instances.
- A structural relationship, specifying that objects of one class are connected to objects of a second (possibly the same) class.
- Example: “An Employee works for a Company”



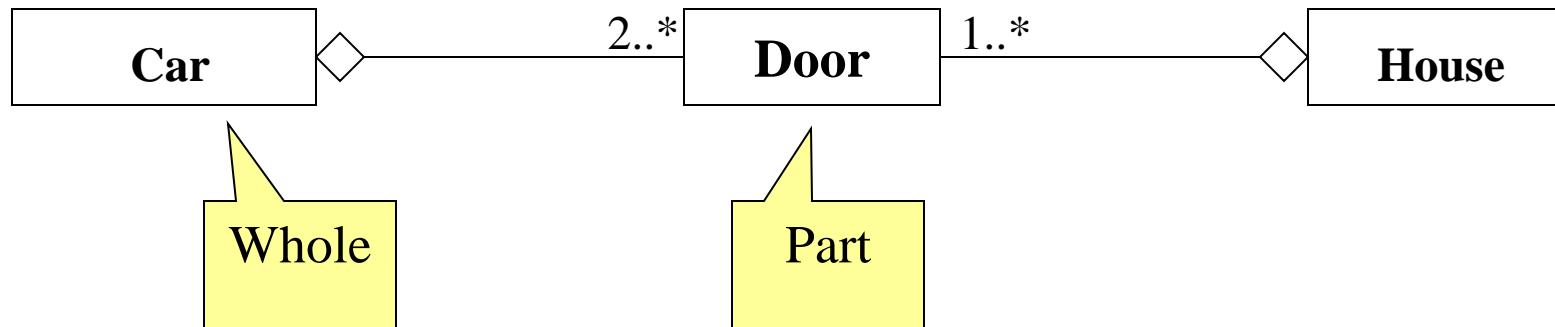
## **Associations can be one of the following two types:**

- *Composition*: Represented by an association line with a solid diamond at the tail end. A composition models the notion of one object "owning" another and thus being responsible for the creation and destruction of another object.
- *Aggregation*: Represented by an association line with a hollow diamond at the tail end. An aggregation models the notion that one object uses another object without "owning" it and thus is *not* responsible for its creation or destruction.



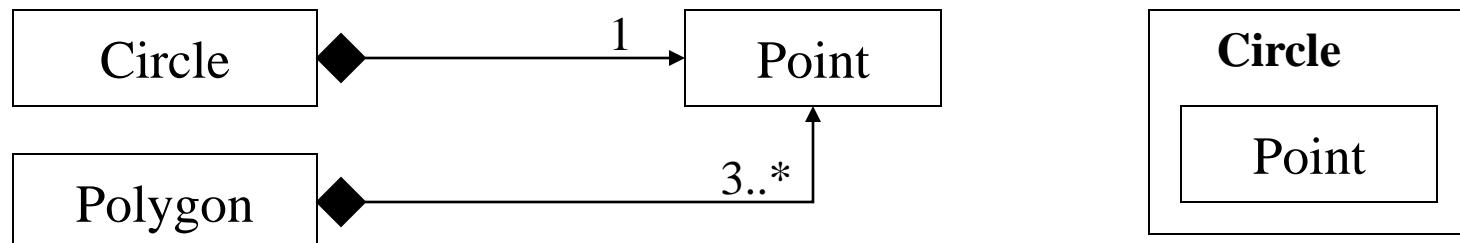
# Aggregation

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
  - Models a “is a part-of” relationship.



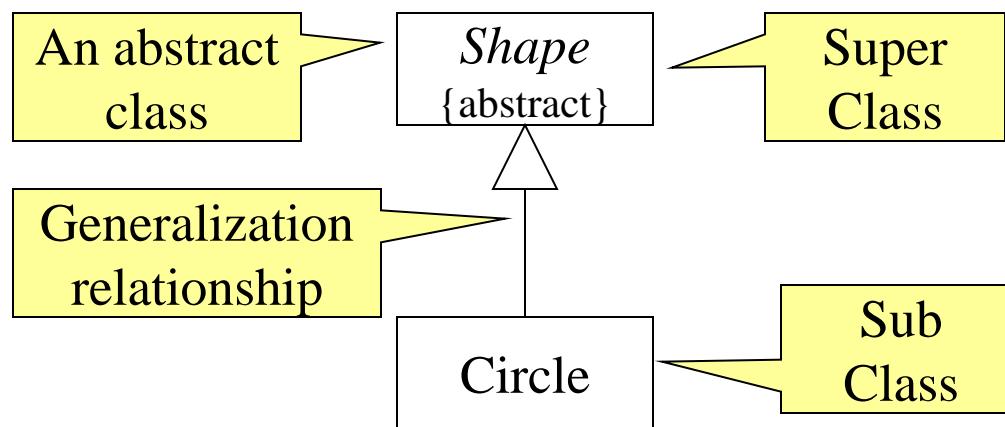
# Composition

- A strong form of aggregation
  - The whole is the sole owner of its part.
    - The part *object* may belong to only one whole
  - The life time of the part is dependent upon the whole.
    - The composite must manage the creation and destruction of its parts.



# Generalization

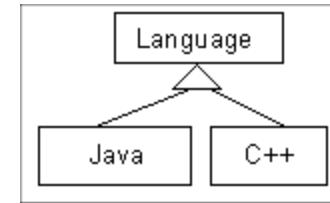
- Indicates that objects of the specialized class (subclass) are substitutable for objects of the generalized class (super-class).
  - “is a” relationship.



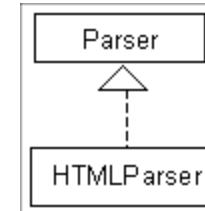
- A sub-class inherits from its super-class
  - Attributes
  - Operations
  - Relationships
- A sub-class may
  - Add attributes and operations
  - Add relationships
  - Refine (override) inherited operations

*Generalization*

Inheritance

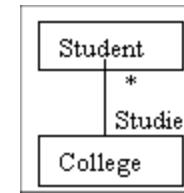


Implementation

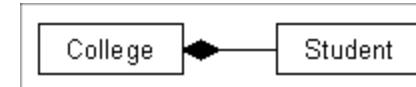


*Association*

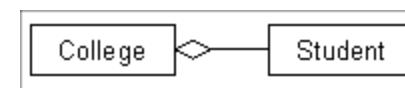
Multiplicity :  
many students  
belonging to same  
college.



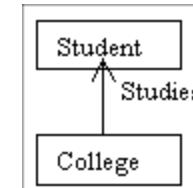
Composition



Aggregation



Directed Association



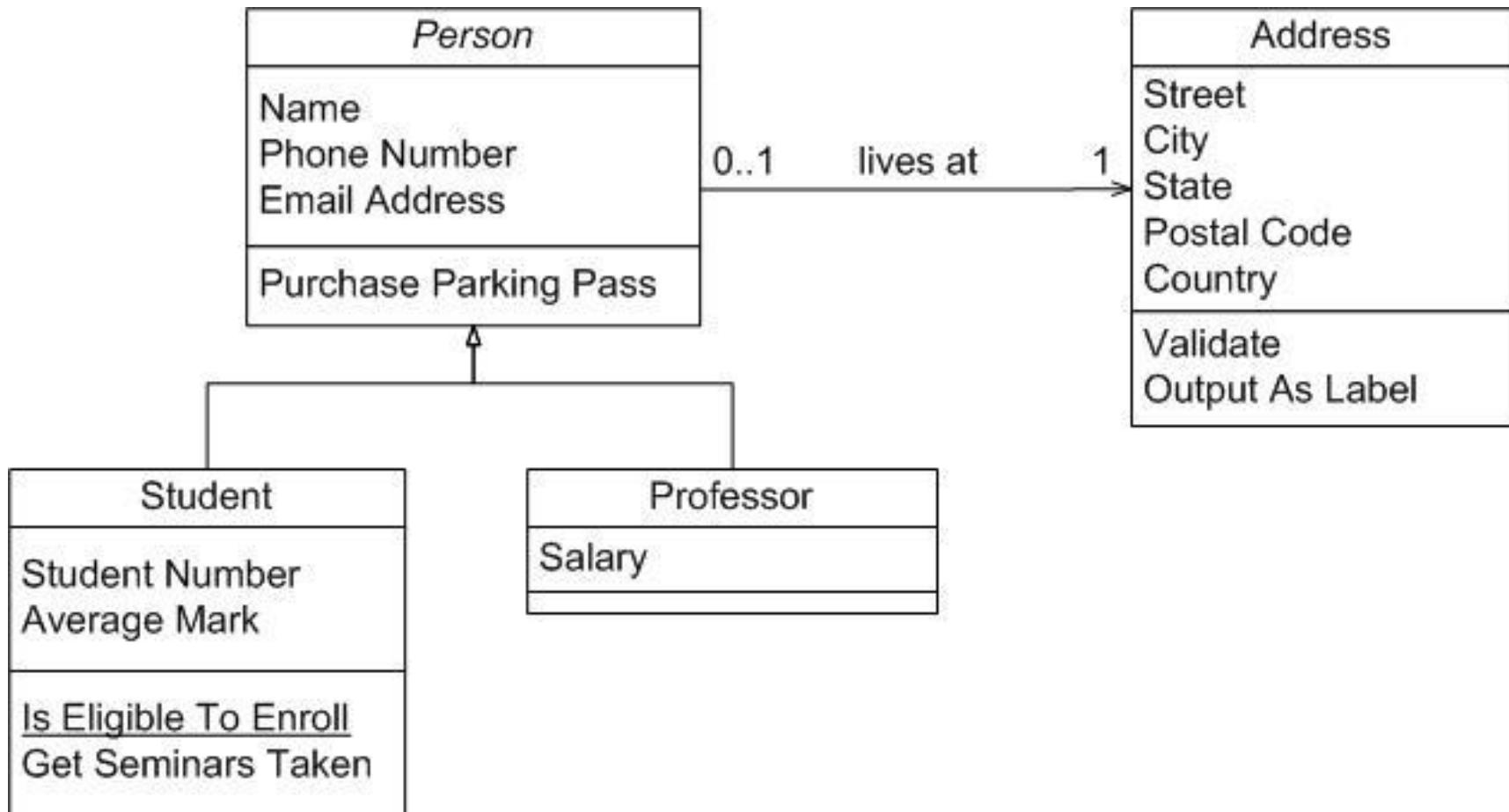
## A Few Terms

- *Responsibility of a class:* It is the statement defining what the class is expected to provide.
- *Stereotypes:* Classes created at the early phases of the system design.
- *Boundary class:* Users interact with the system through the boundary classes. (interface classes)
- *Control class:* A control class typically does not perform any business functions, but only redirects to the appropriate business function class depending on the function requested by the boundary class or the user.
- *Entity class:* An entity class consists of all the business logic and interactions with databases.

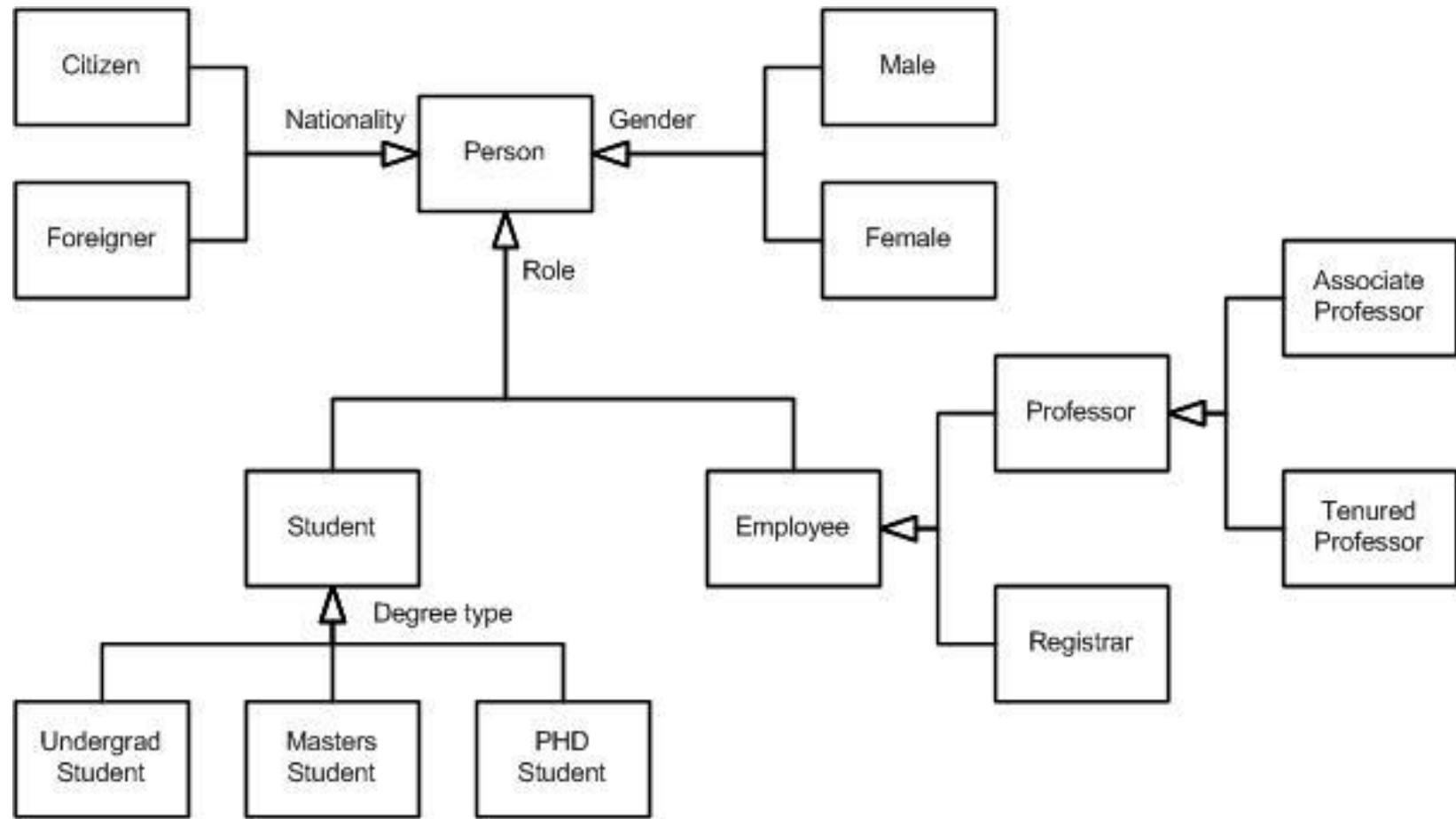
# Multiplicity Indicators.

Indicator	Meaning
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
n	Only $n$ (where $n > 1$ )
0..n	Zero to $n$ (where $n > 1$ )
1..n	One to $n$ (where $n > 1$ )

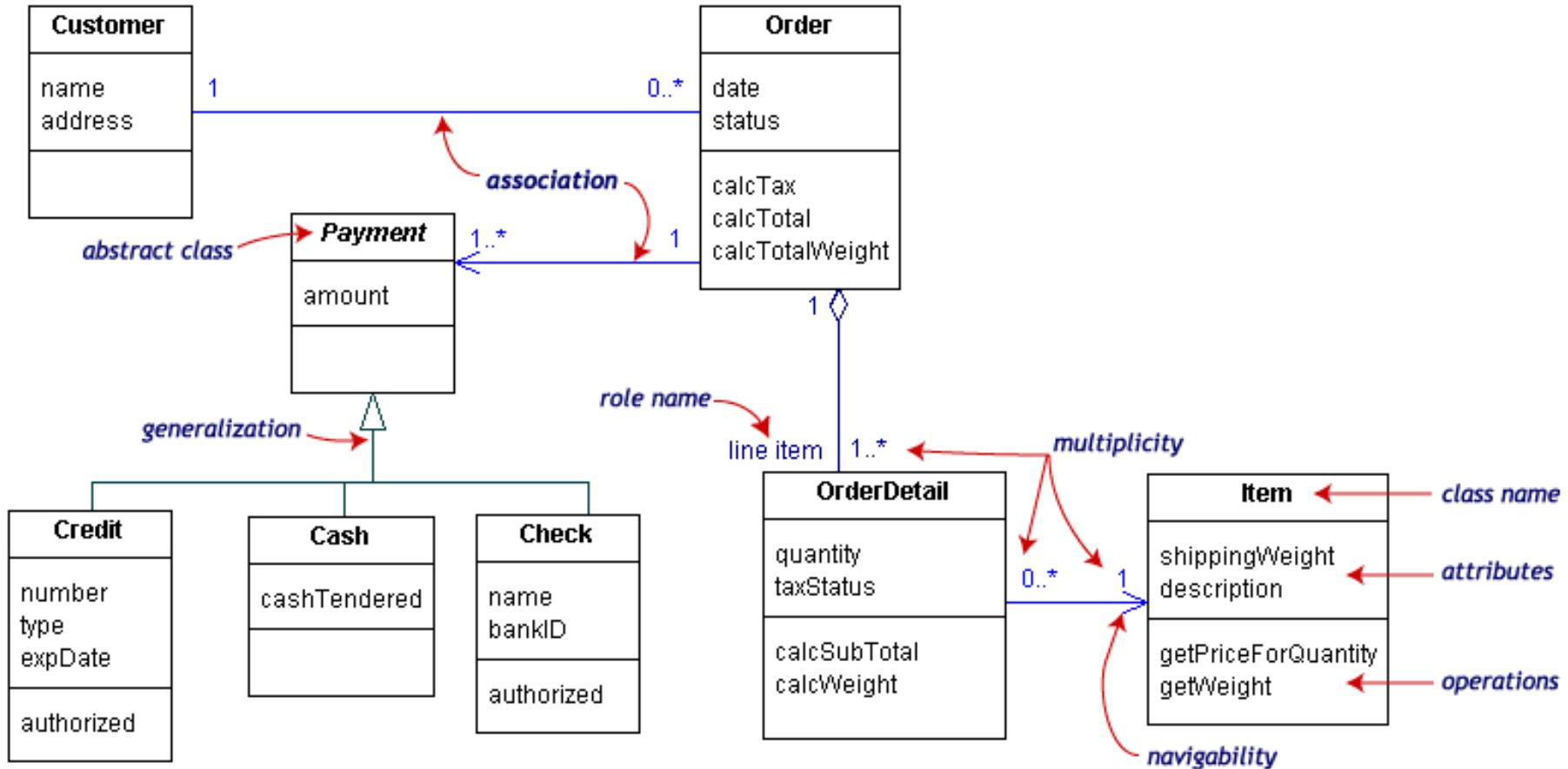
# Assignment No. 2: Relationship between Student, Professor and Employee of the company (Page 108, UML User Guide)

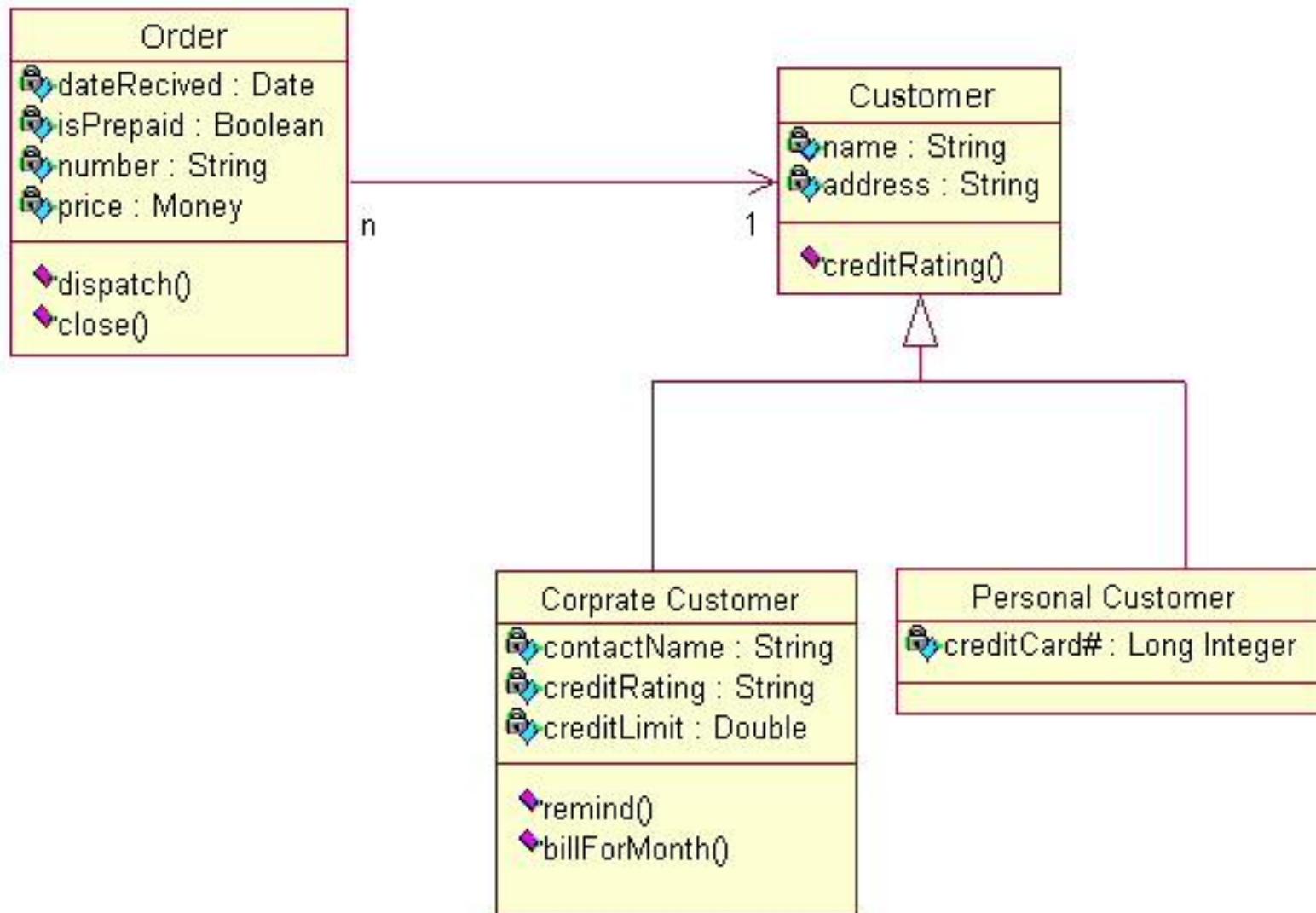


# Assignment No. 2: Relationship between Student, Professor and Employee of the company



# Customer Order & Payment



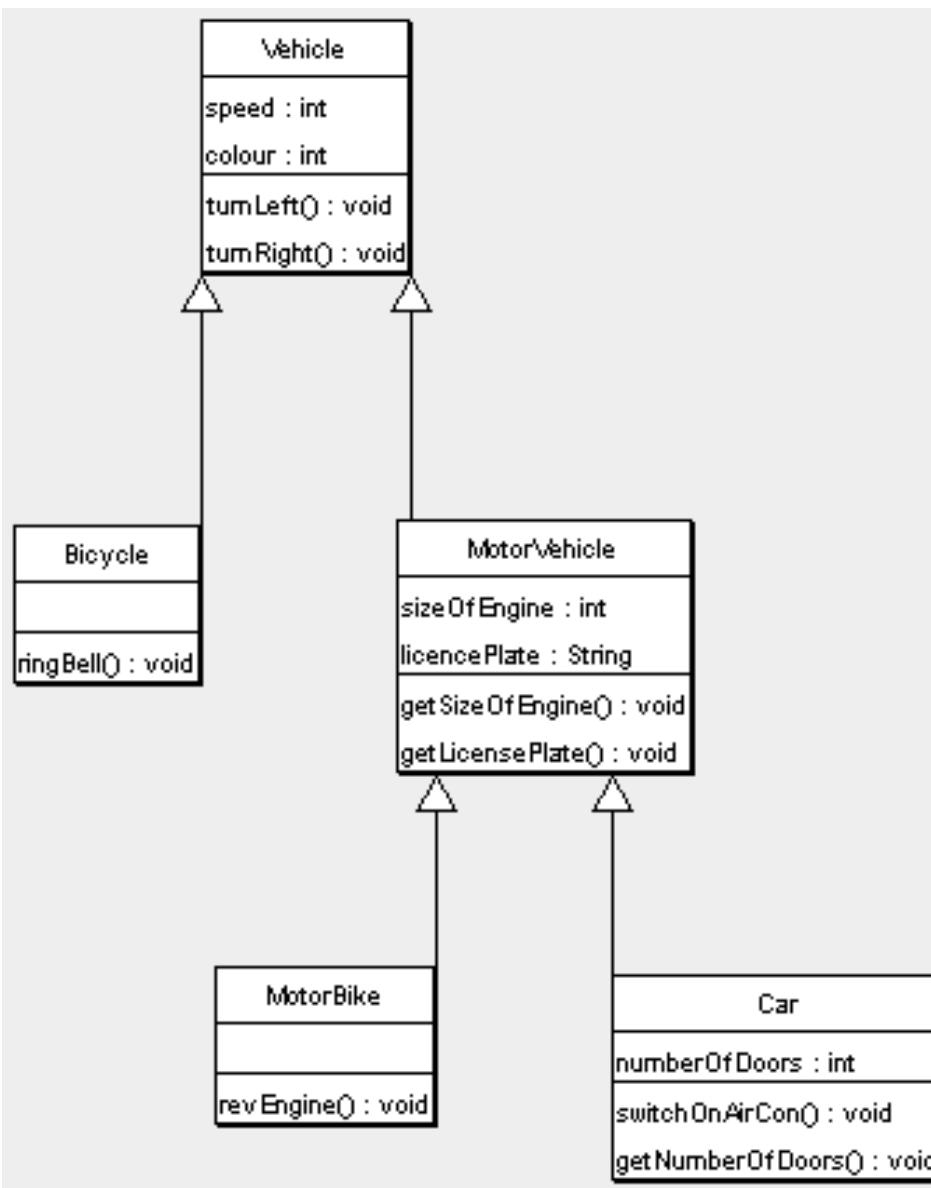


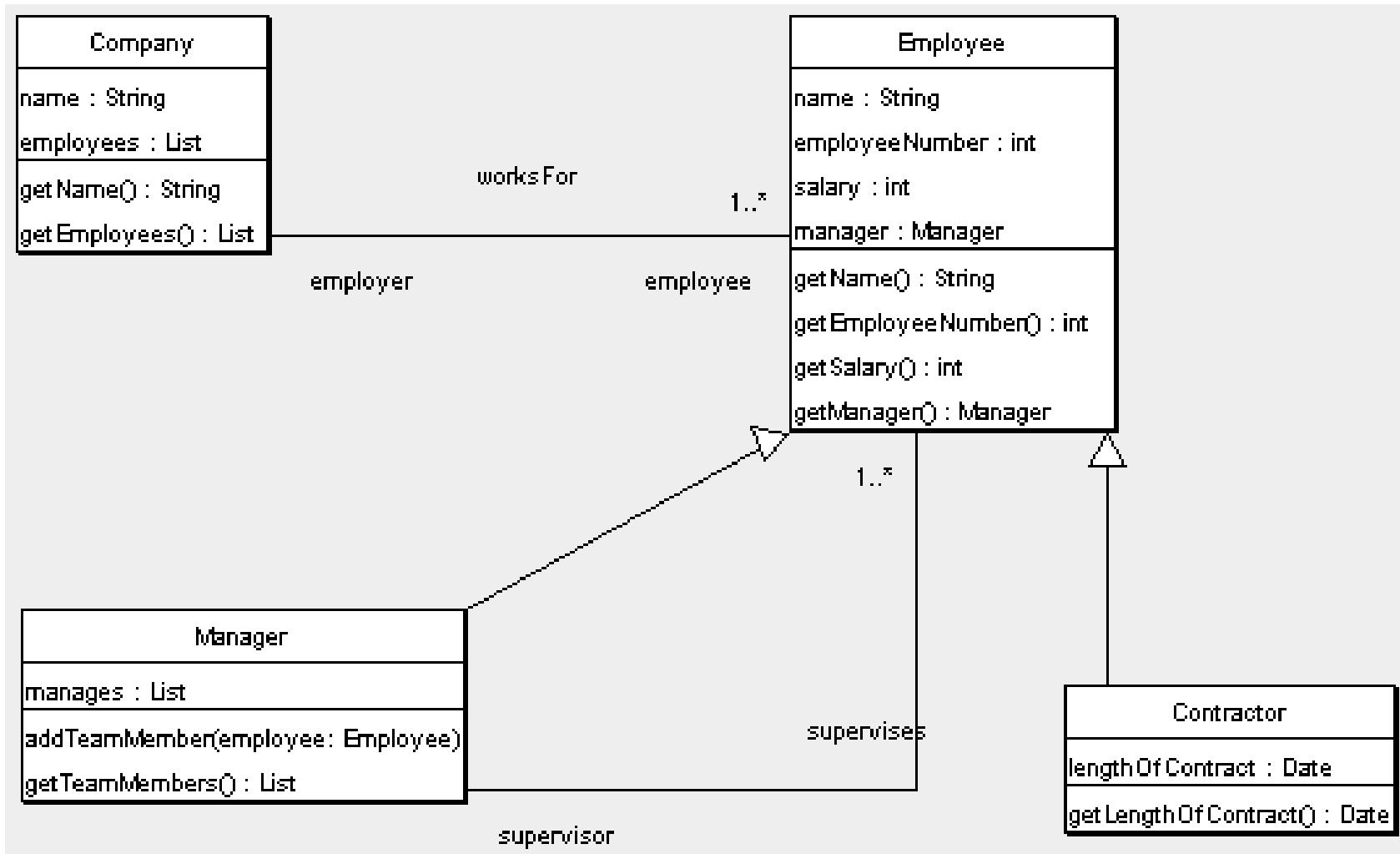
## Example Vehicles

We would like to model an application that shows different kinds of vehicles such as bicycles, motor bike and cars.

Notes:

- All Vehicles have some common attributes (speed and colour) and common behaviour (turnLeft, turnRight)
- Bicycle and MotorVehicle are both kinds of Vehicle and are therefore shown to inherit from Vehicle. To put this another way, Vehicle is the superclass of both Bicycle and MotorVehicle
- In our model MotorVehicles have engines and license plates. Attributes have been added accordingly, along with some behaviour that allows us to examine those attributes
- MotorVehicles is the base class of both MotorBike and Car, therefore these classes not only inherit the speed and colour properties from Vehicle, but also the additional attributes and behaviour from MotorVehicle
- Both MotorBike and Car have additional attributes and behaviour which are specific to those kinds of object.





Show a relationship between Rectangle, Circle and Polygon

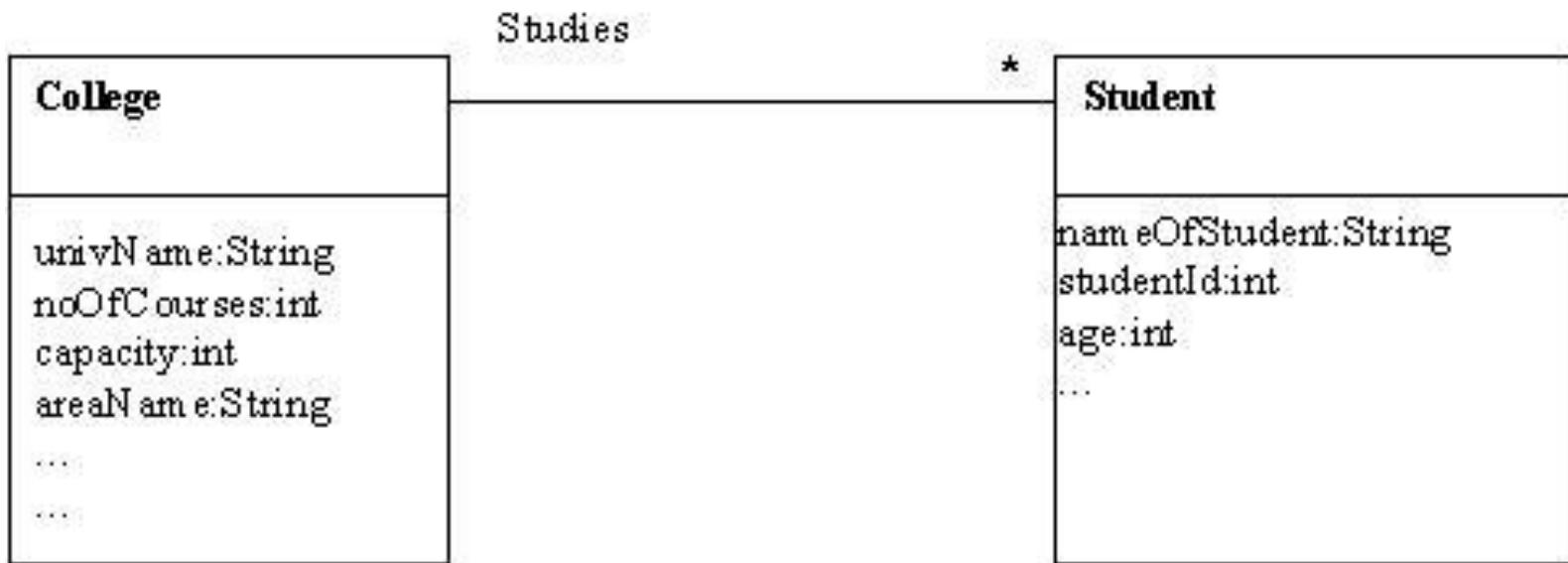
UML – User Guider Page No. 61

# Object Diagrams in UML

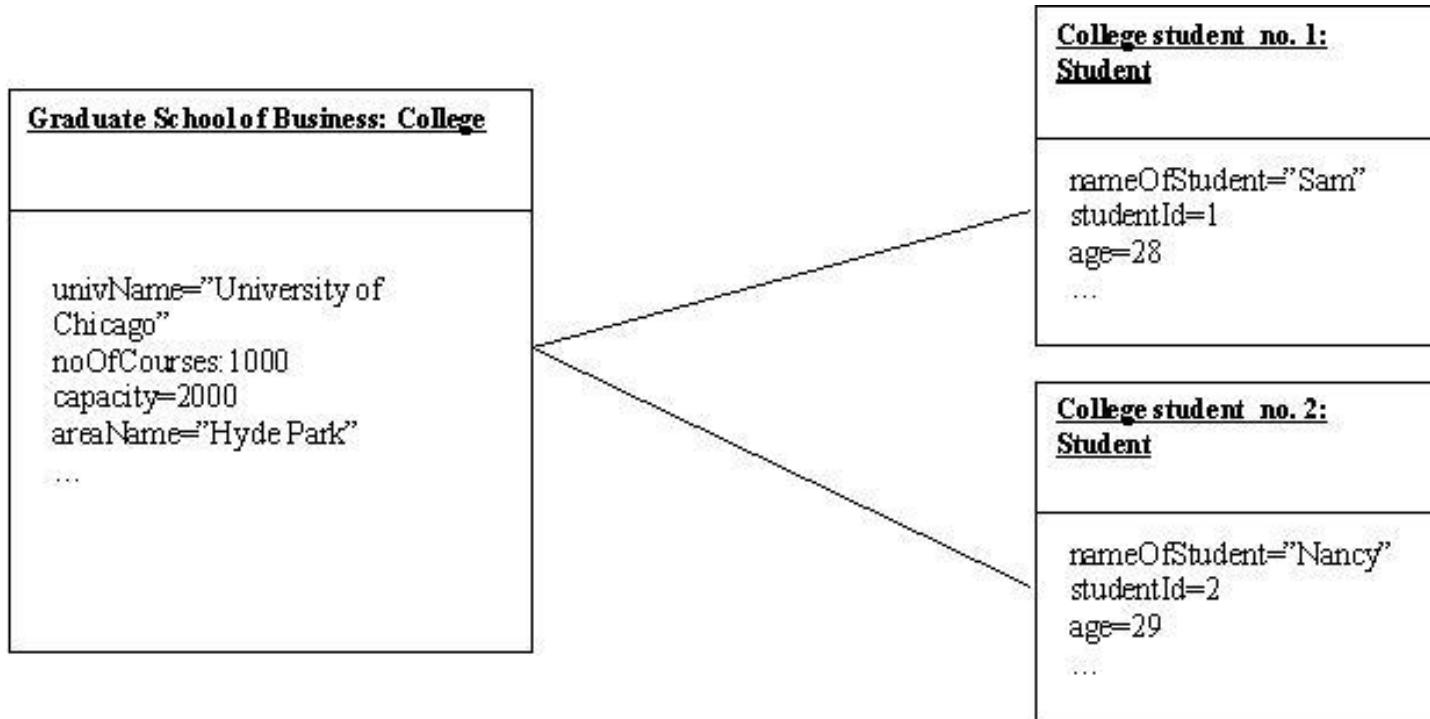
- In a live application classes are not directly used, but instances or objects of these classes are used. A pictorial representation of the relationships between these instantiated classes at any point of time (called objects) is called an "**Object diagram.**"
- It looks very similar to a class diagram, and uses the similar notations to denote relationships.
- It reflects the picture of how classes interact with each others at runtime. and in the actual system, how the objects created at runtime are related to the classes.
- shows this relation between the instantiated classes and the defined class, and the relation between these objects.

# Elements of an Object Diagram

The minor difference between class diagram and the object diagram is that, the class diagram shows a class with attributes and methods declared. However, in an object diagram, these attributes and method parameters are allocated values.



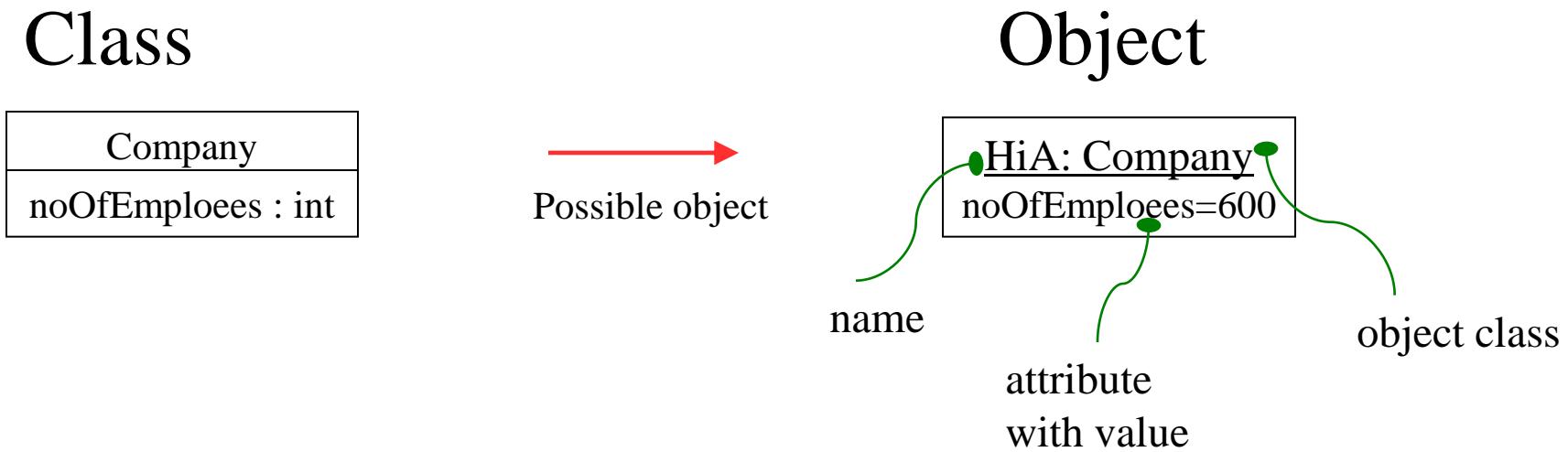
Now, when an application with the class diagram as shown above is run, instances of College and Student class will be created, with values of the attributes initialized. The object diagram for such a scenario will be represented as shown below:



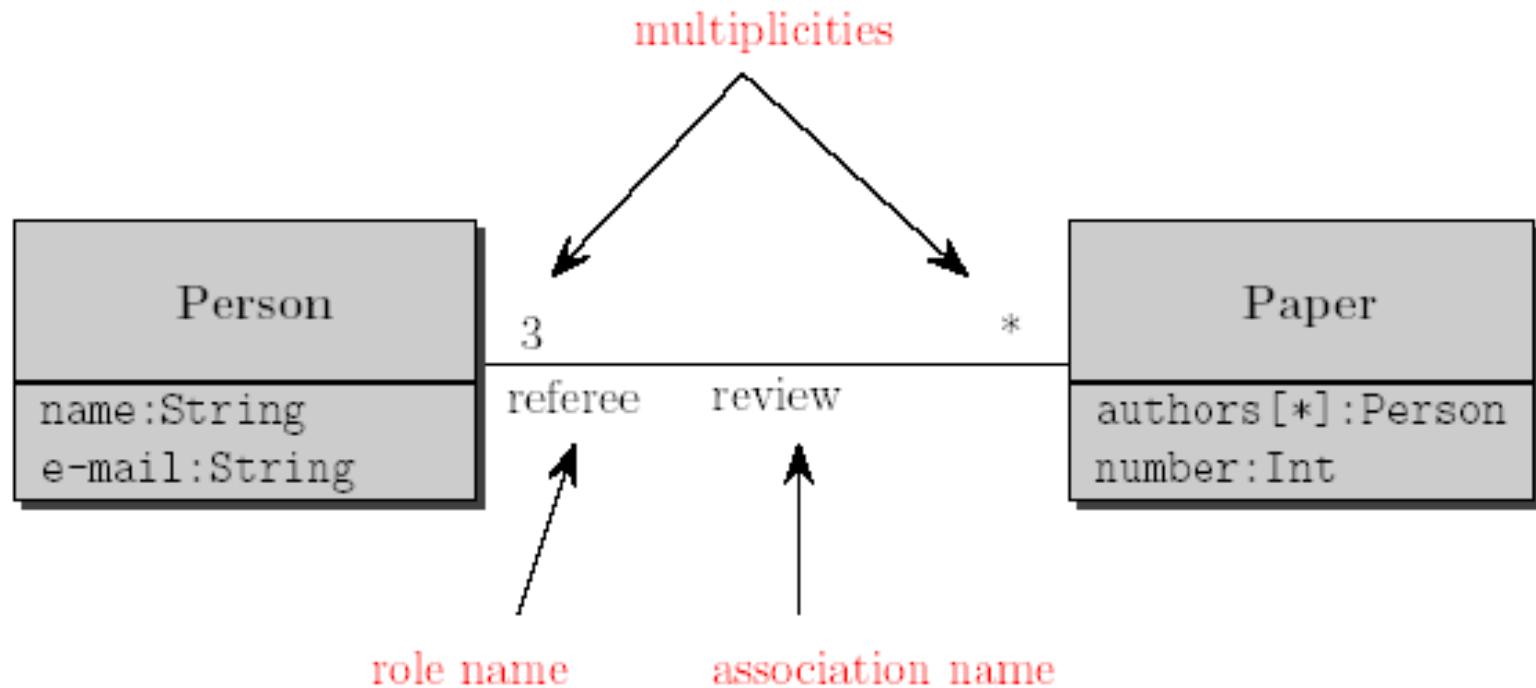
The object diagram shows the name of the instantiated object, separated from the class name by a ":" , and underlined, to show an instantiation.

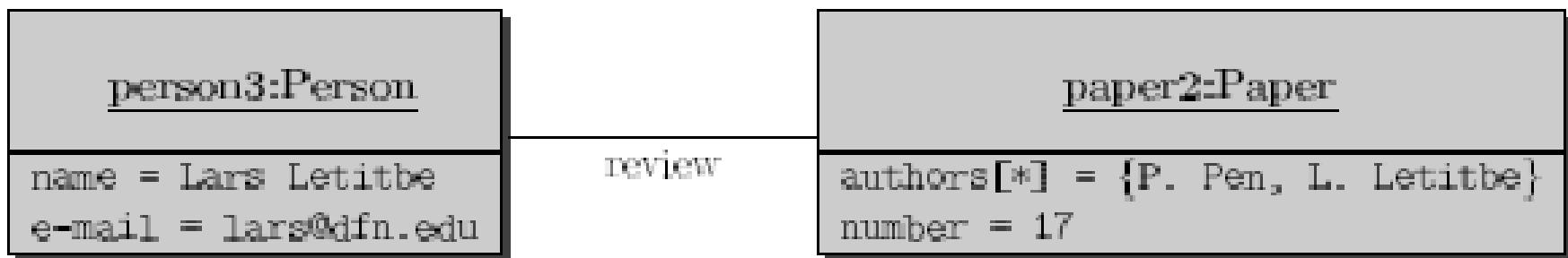
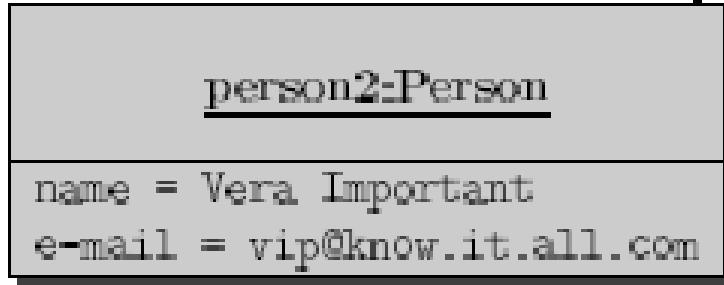
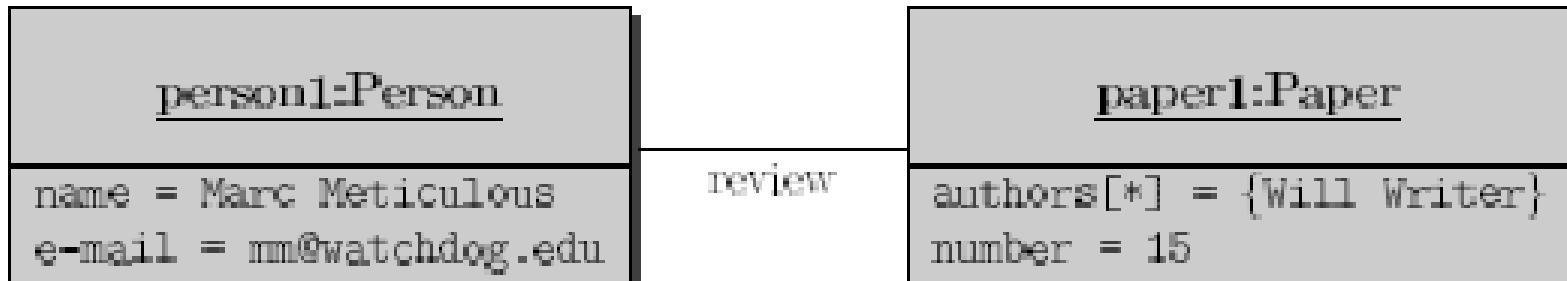
Eg. Graduate School of Business: College

Example2:



## Example 3





## **When to use object diagrams?**

- Use the object diagram as a means of debugging the functionality of your system.
- Check whether the system has been designed as per the requirements, and behaves how the business functionality needs the system to respond.

## **Be careful !!**

- Avoid representing all the objects of your system in an object diagram → complex → unreadable. Use object diagram to represent the state of objects in important or critical flows in your application.

# State Diagram

## Basics

- We are now taking a deeper look at system dynamics.
- Some of the dynamic behavior will be specified in terms of sequencing / timing
- Some of the dynamic behavior will be specified in terms of functions (transformations / computations)
- State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur.
- It is important to note that having a State diagram for your system is not a mandatory, but must be defined only on a need basis (to understand the behavior of the object through the entire system)

**Event:** Is Something that happens at a point of time . E.g.  
user presses a button.

One event may logically precede other or follow another.

State may occur between two events

Every event is a unique occurrence

Event may be external or Internal. External events are those that pass between the system and its actor.

Event conveys information from one object to another

**Signals:** A message is a named object that is sent asynchronously by one object and the received by another.

A signal is a classifier for messages; it is a message type. Signals have lot of common with plain classes.

**Call Events:** It represents the receipt by an object of a call request for operations on the object. A call event may trigger a state transition in a state machine or it may invoke a method on the target object.

**Time and Change Events:** A time event is an that represents the passage of time.

**States:** A states is an abstraction of attribute values and links of an object.

Set of values are grouped together into a state according to properties that affect the gross behavior of the object.

A state specifies the response of the object to the input events.

**Scenario:** I a sequence of events that occurs during one particular execution of a system.

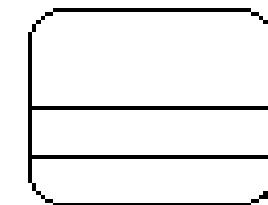
It may include all events in the system or may include only those events impinging or generated by certain objects in the system.

# Elements of a State diagram

**Initial State:** This shows the starting point or first activity of the flow



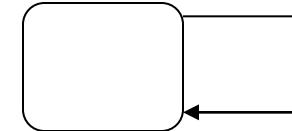
**State:** Represents the state of object at an instant of time. In a state diagram, there will be multiple of such symbols, one for each state of the Object



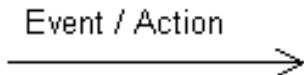
**Transition:** An arrow indicating the Object to transition from one state to the other. The actual trigger event and action causing the transition are written beside the arrow.



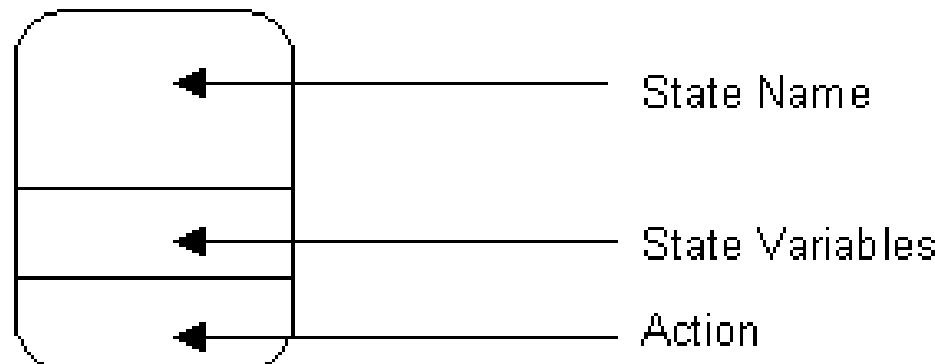
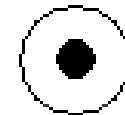
**Self Transitions:** Sometimes an object is required to perform some action when it recognizes an event, but it ends up in the same state it started in



**Event and Action:** A trigger that causes a transition to occur is called as an event or action.

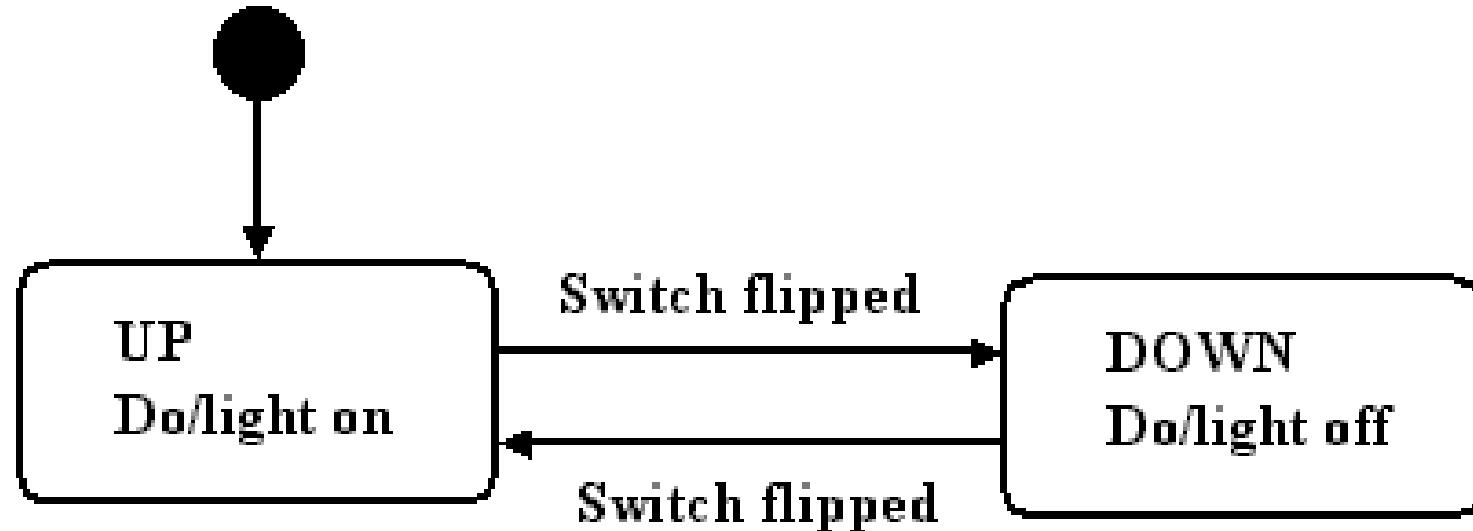


**Final State:** The end of the state diagram is shown by a bull's eye symbol, also called a final state.



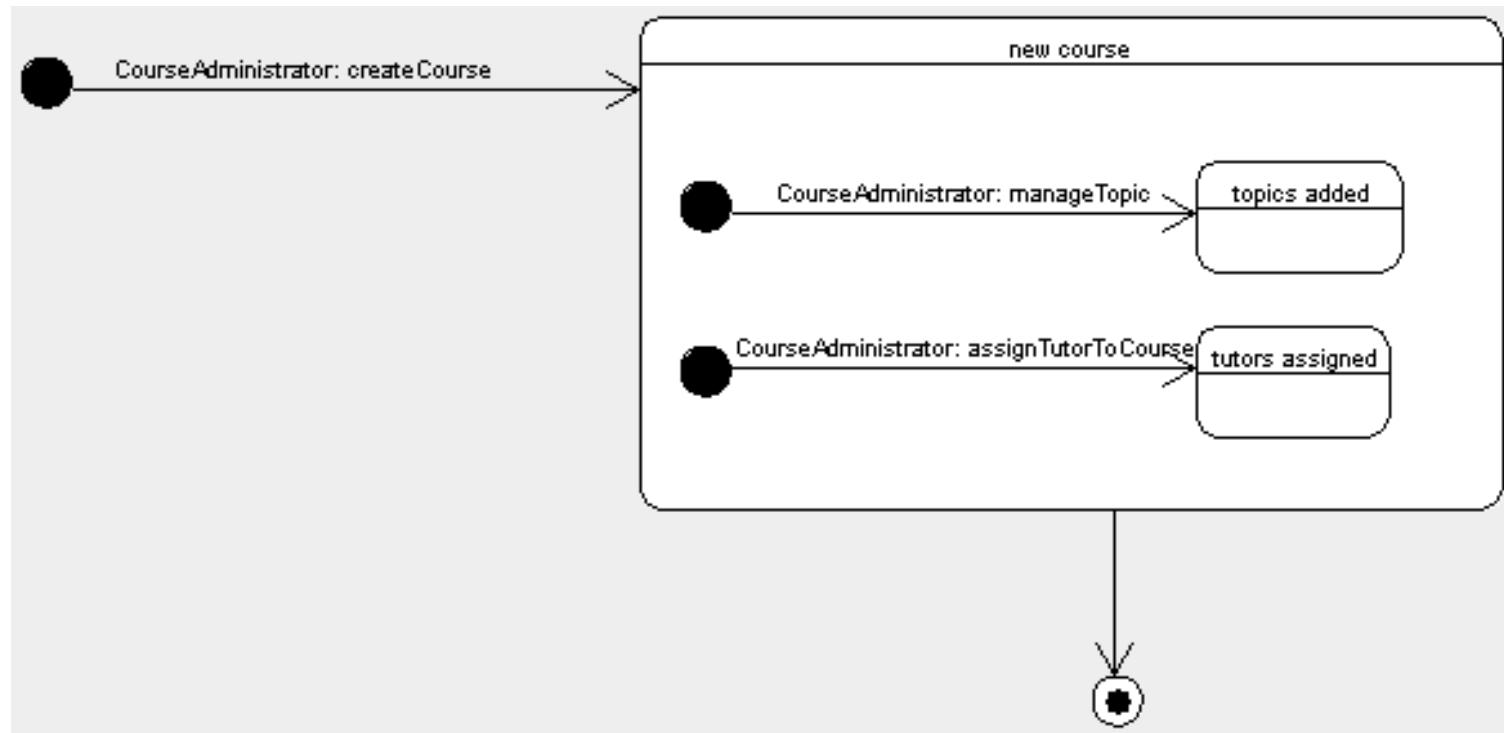
## Example: ordinary two-position light switch.

A light switch will have two states: up and down. (We could call them "on" and "off" if we liked.) In a UML state diagram, each state is represented by a rounded rectangle.



## Example: Identifying states and events of the Course object

- The events that occur in the lifecycle of the Course object are listed below:
  - Create new course—add information for the course
  - Add topics—add topics to the course
  - Assign tutors—assign the available tutors for the course
  - Close—finished adding or updating the course
- Assume that the admin. Is responsible for adding new course

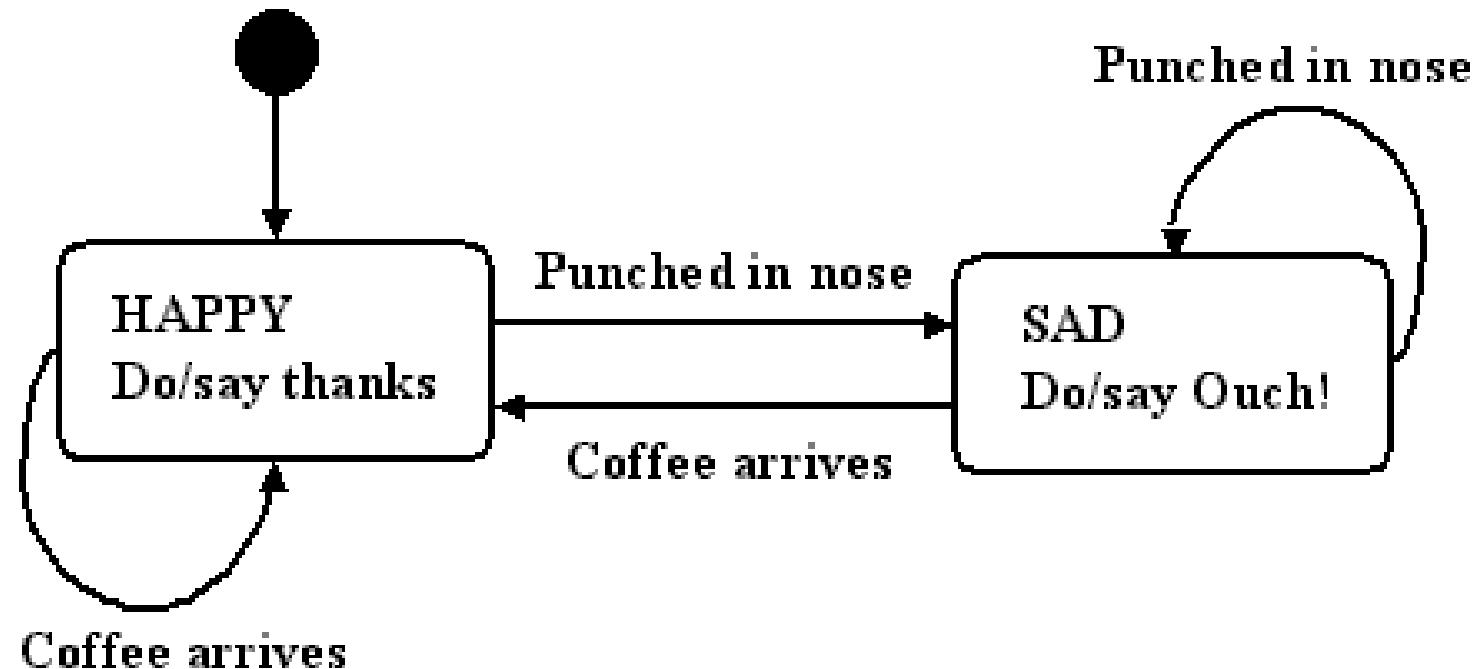


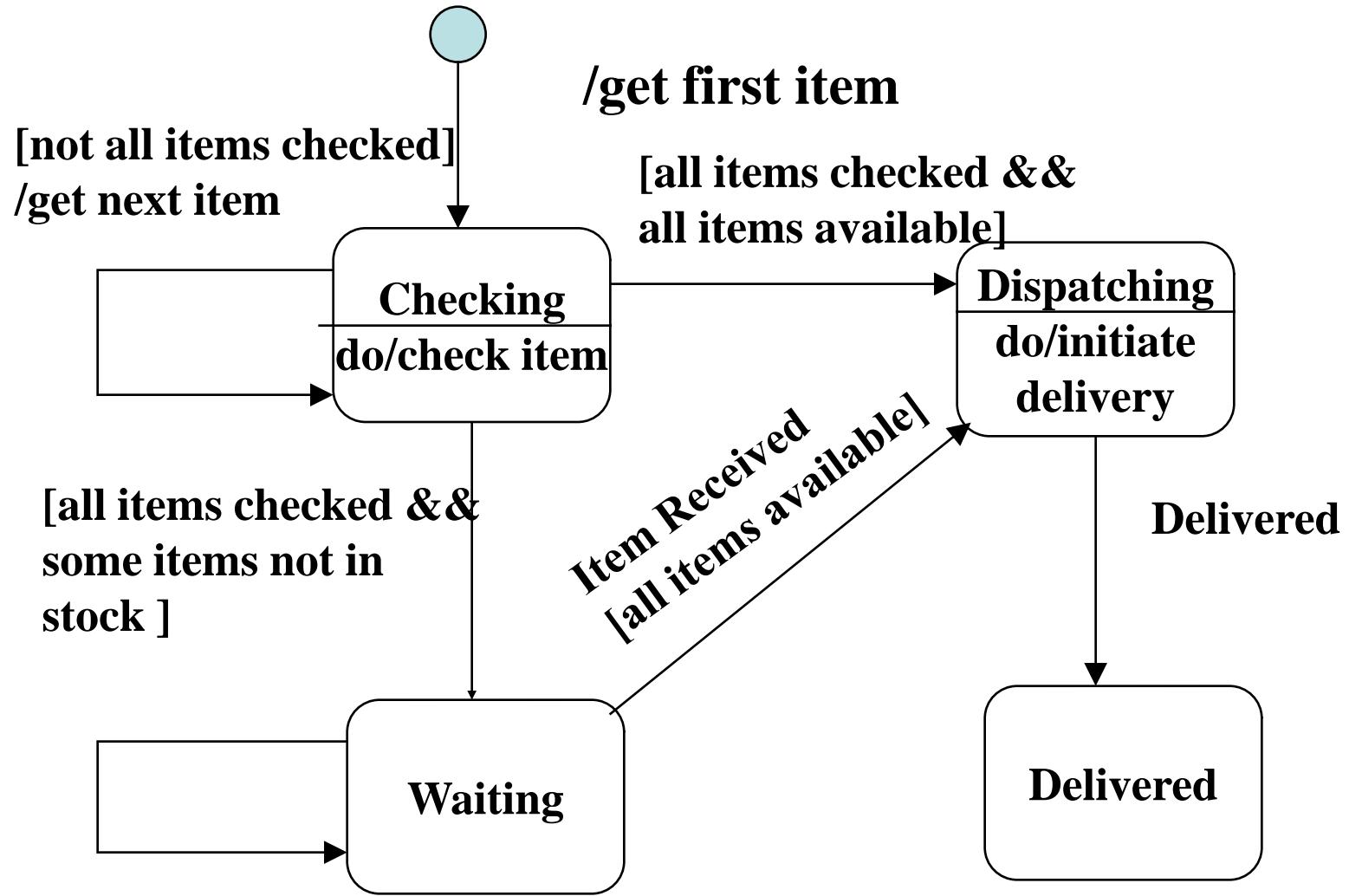
## Assignment No 3: Example: simplistic Teaching Assistant (TA)

TA only has two states:

- Happy when getting coffee.
- Sad when getting punched in the nose.

Suppose that TA is basically cheerful and starts out happy





**Item received**  
[some items not in stock]

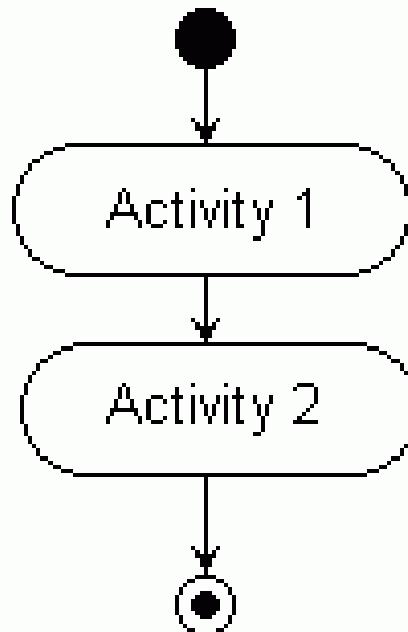
**Assignment No 4: Draw a Event State diagram for Landline phone**

**Assignment No 5: Draw a Event State diagram of a car using Ignition, Transmission, Brake and Accelerator**

# Activity Diagram

- The easiest way to visualize an Activity diagram is to think of a flowchart of a code.
- The flowchart is used to depict the business logic flow and the events that cause decisions and actions in the code to take place.
- An Activity diagram is a dynamic diagram that shows the activity and the event that causes the object to be in the particular state.
- The activity diagram is an extension of the state diagram. State diagrams highlight states and represent activities as arrows between states. Activity diagrams put the spotlight on the activities
- The Activity Diagrams are often used to model the paths through a use case. And to document the logic of a single use case.

- Each activity is represented by a rounded rectangle - narrower and more oval-shaped than the state icon
- The processing within an activity goes to completion and then an automatic transmission to the next activity occurs
- An arrow represents the transition from one activity to the next. Also an activity diagram has a starting point represented by filled-in circle, and endpoint represented by a bull's eye.



# Elements of an Activity diagram

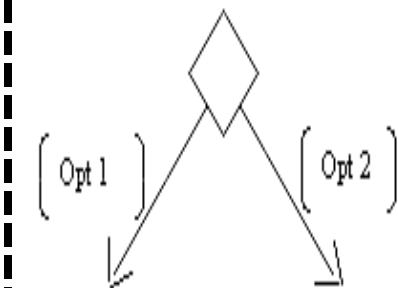
**Initial Activity:** This shows the starting point or first activity of the flow



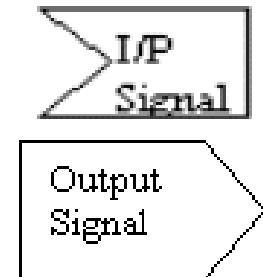
**Activity:** Represented by a rectangle with rounded (almost oval) edges.



**Decisions:** Similar to flowcharts, a logic where a decision is to be made is depicted by a diamond, with the options written on either sides of the arrows emerging from the diamond



**Signal:** When an activity sends or receives a message, that activity is called a signal. Signals are of two types: Input signal and Output signal

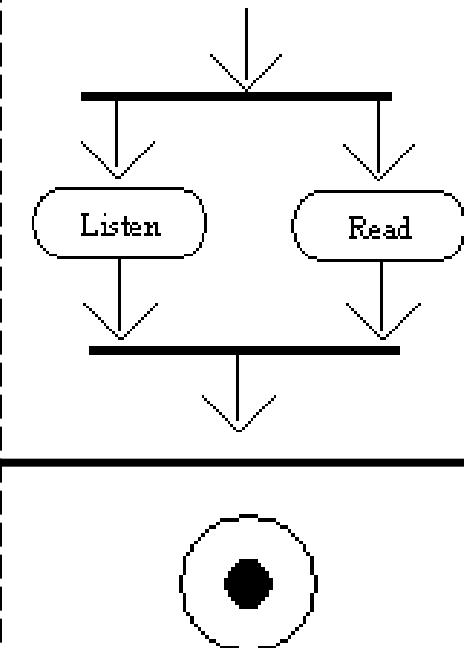


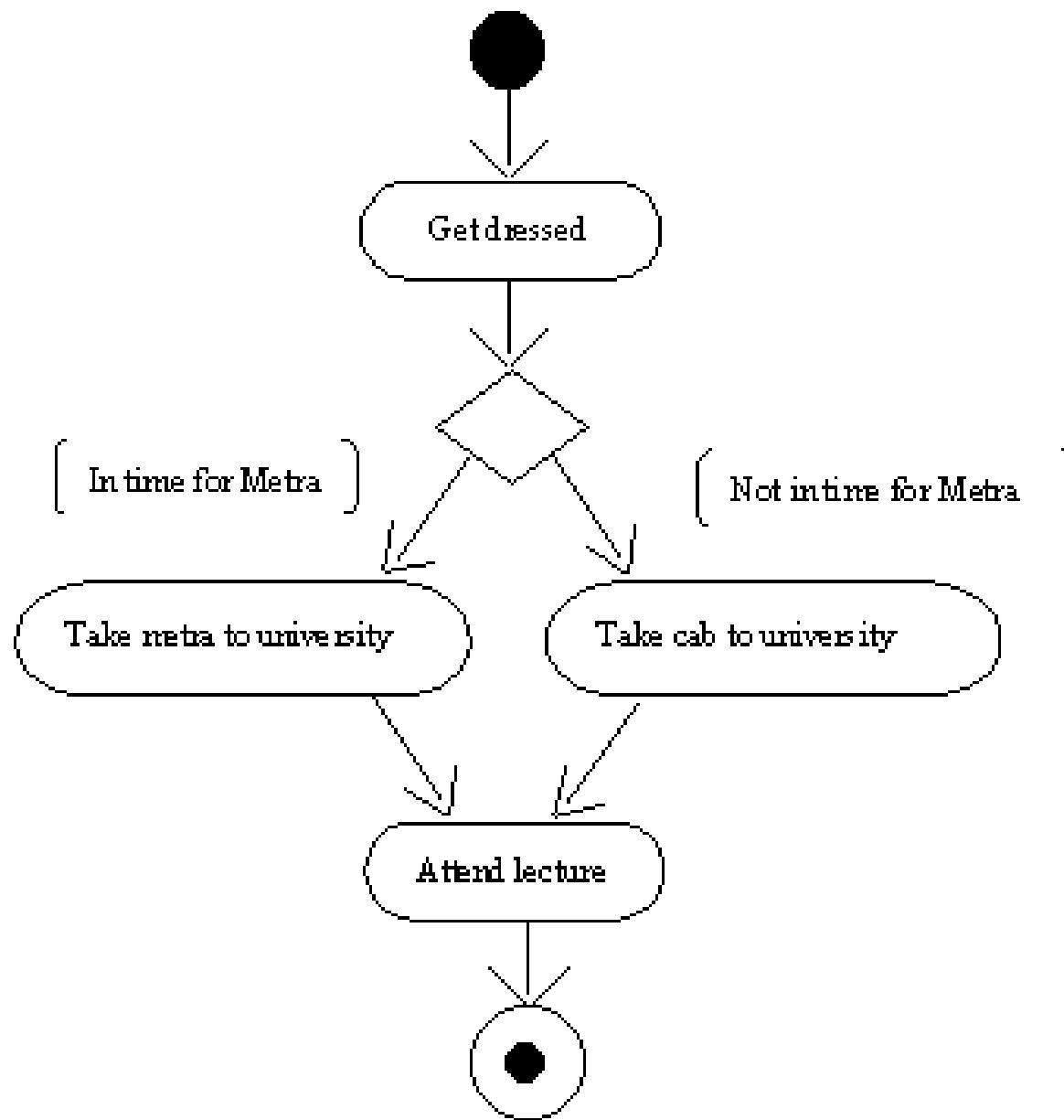
**Concurrent Activities:** Some activities occur simultaneously or in parallel. Such activities are called concurrent activities. For example, listening to the lecturer and looking at the blackboard is a parallel activity.

**Final Activity:** The end of the Activity diagram is shown by a bull's eye symbol, also called as a final activity.

- An activity diagram may have only one initial action state, but may have any number of final action states.

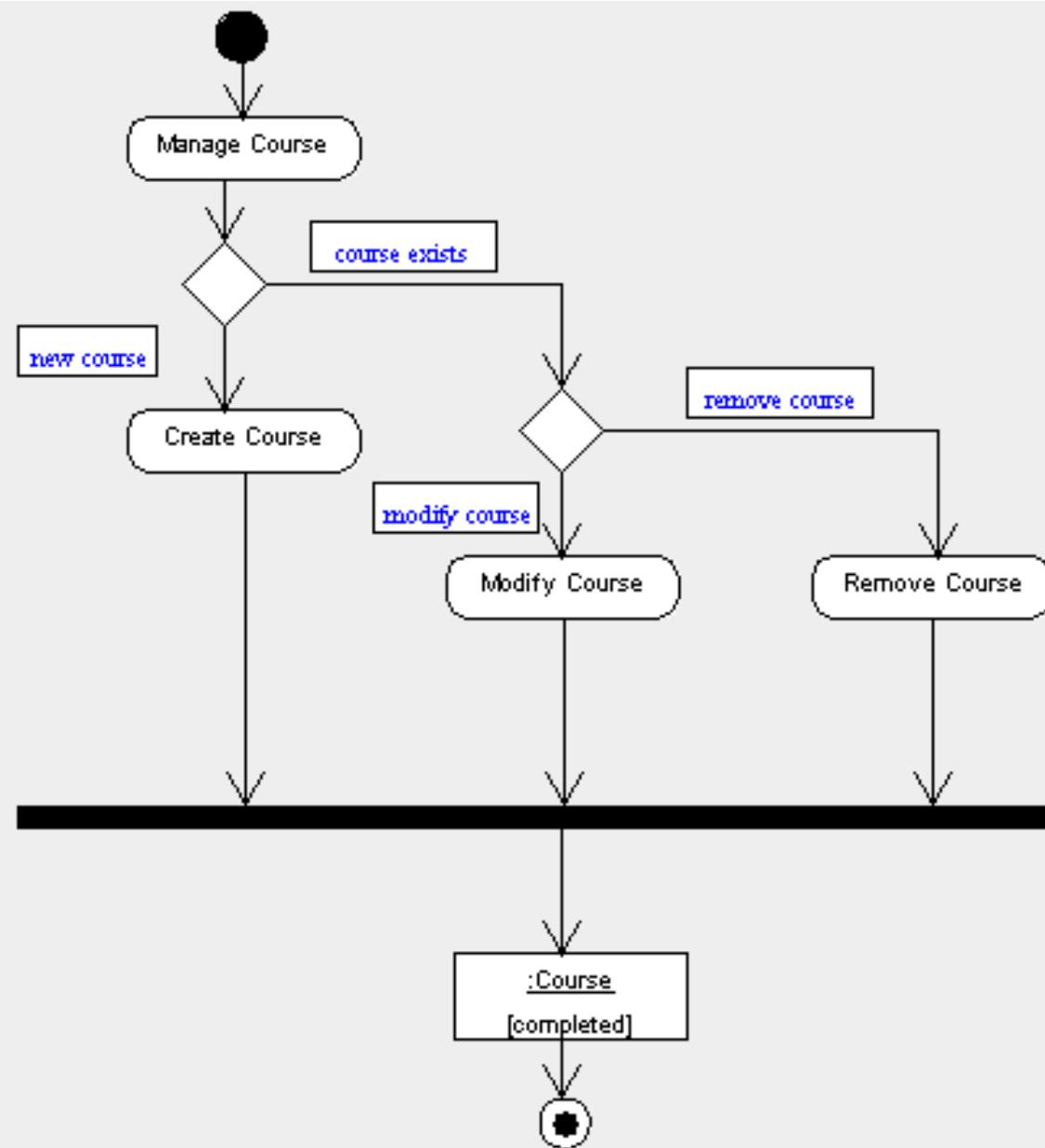
Consider the example of attending a course lecture, at 8 am.



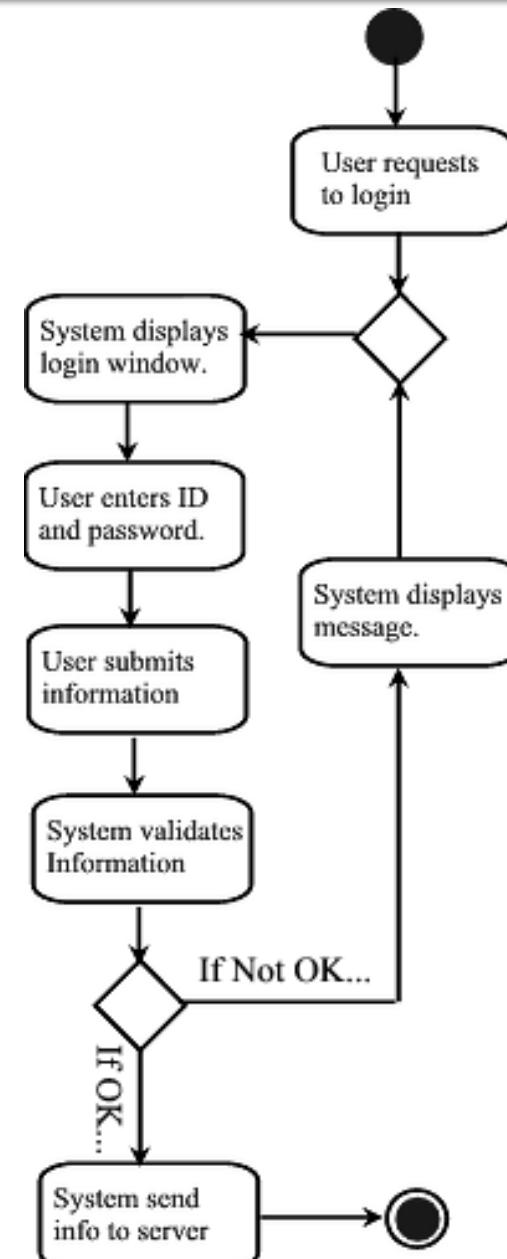


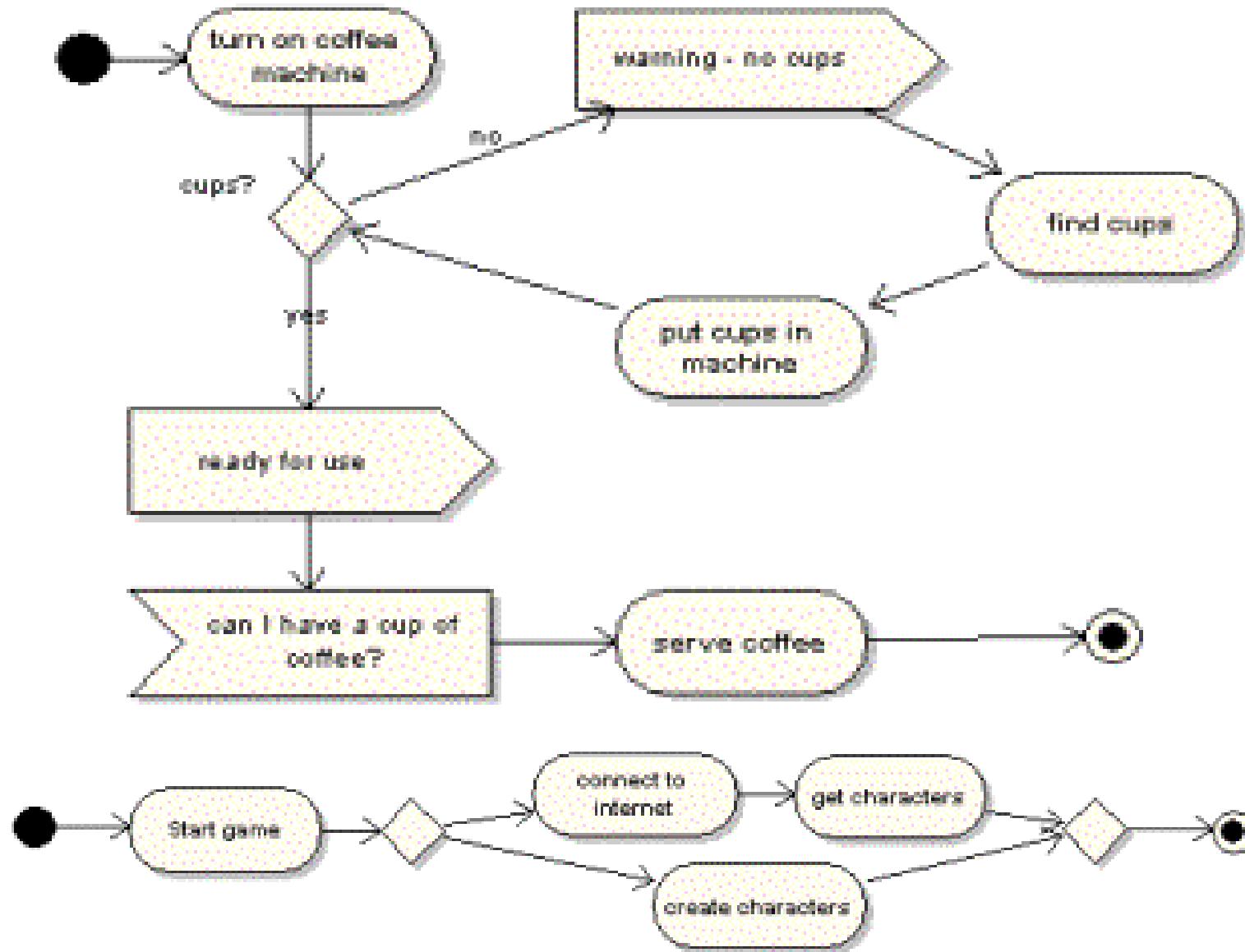
The course administrator is responsible for managing course information in a training center system, the course administrator carries out the following activities:

- Check if course exists
- If course is new, proceed to the "Create Course" step
- If course exists, check what operation is desired—whether to modify the course or remove the course
- If the modify course operation is selected by the course administrator, the "Modify Course" activity is performed
- If the remove course operation is selected by the course administrator, the "Remove Course" activity is performed



The following example shows the activity diagram for a login page.





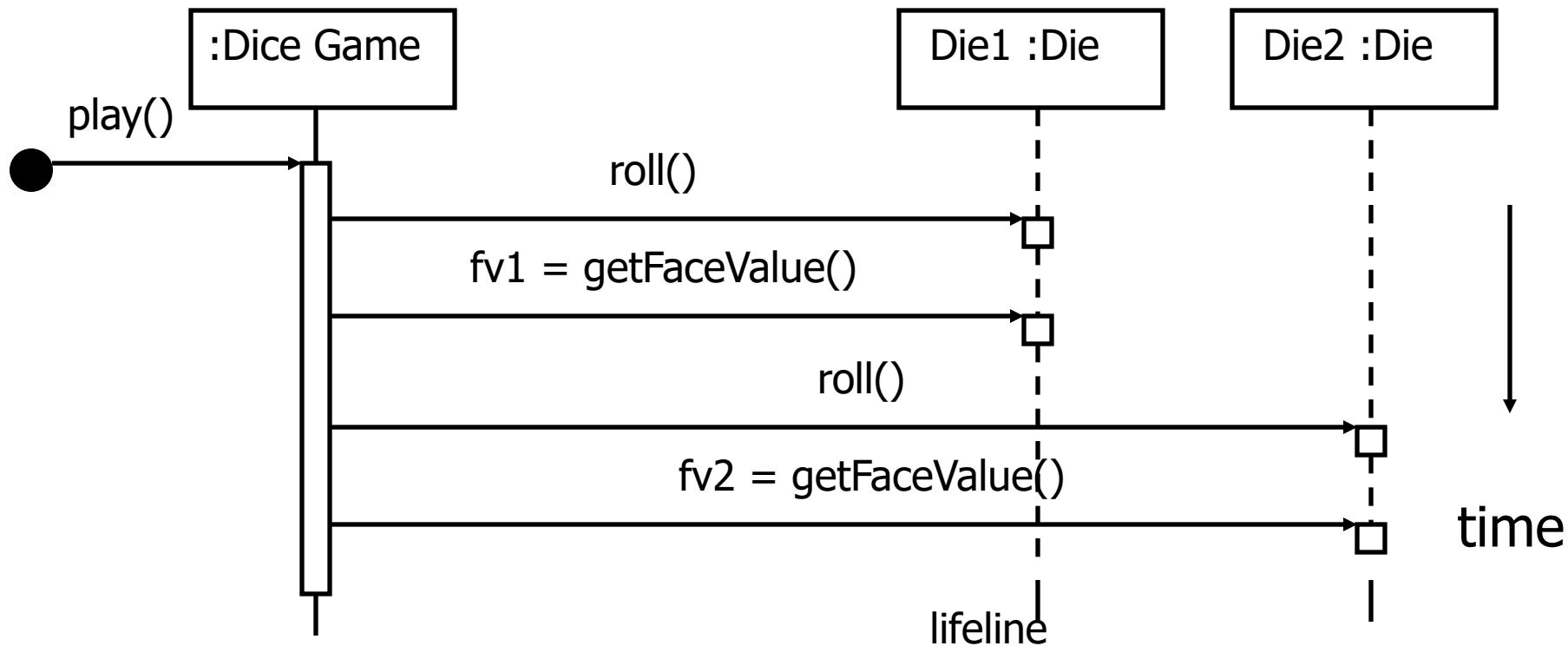
# Assignment No. 6: Draw a Activity diagram of processing orders

# Sequence Diagram in UML

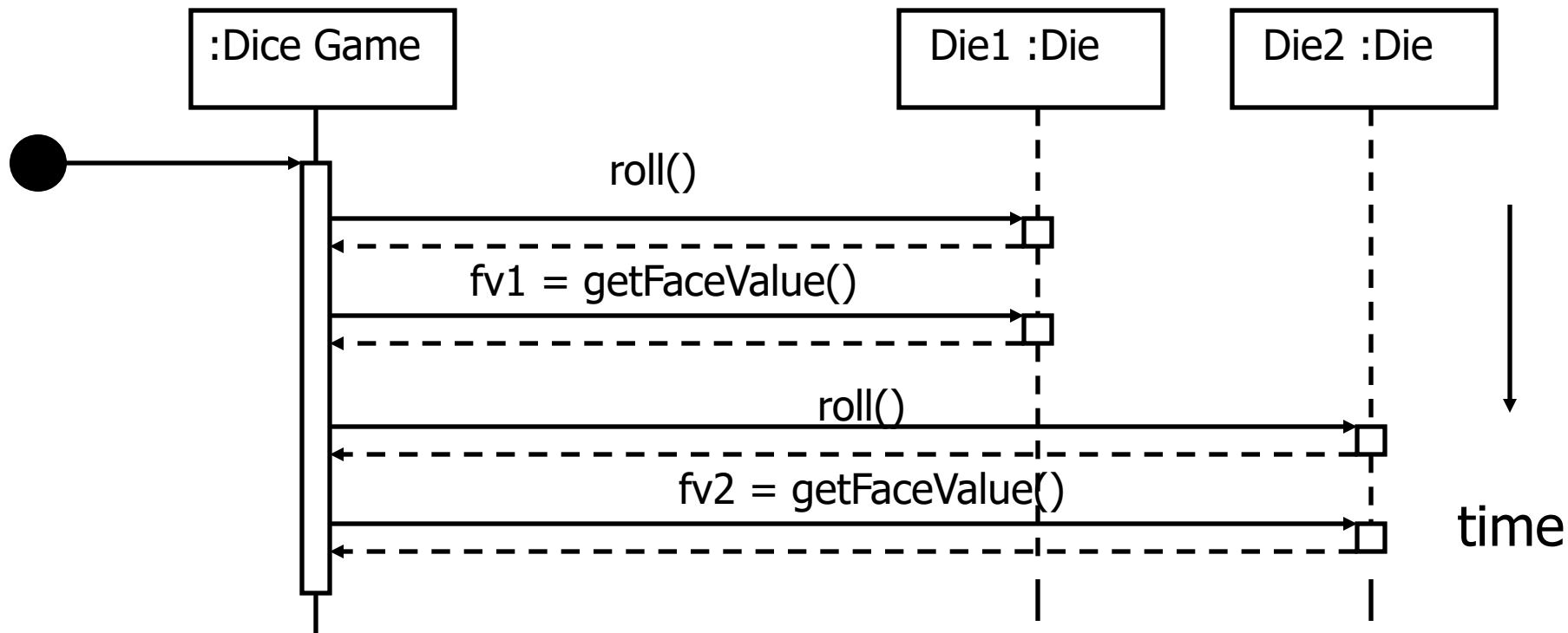
- A sequence diagram captures the behavior of a single scenario. The diagram shows a number of example objects and the messages that are passed between these objects within the use case.
- A Sequence diagram depicts the sequence of actions that occur in a system.
- The invocation of methods in each object, and the order in which the invocation occurs is captured in a Sequence diagram.
- A Sequence diagram is two-dimensional in nature. On the horizontal axis, it shows the life of the object that it represents, while on the vertical axis, it shows the sequence of the creation or invocation of these objects.

# Defining a sequence Diagram

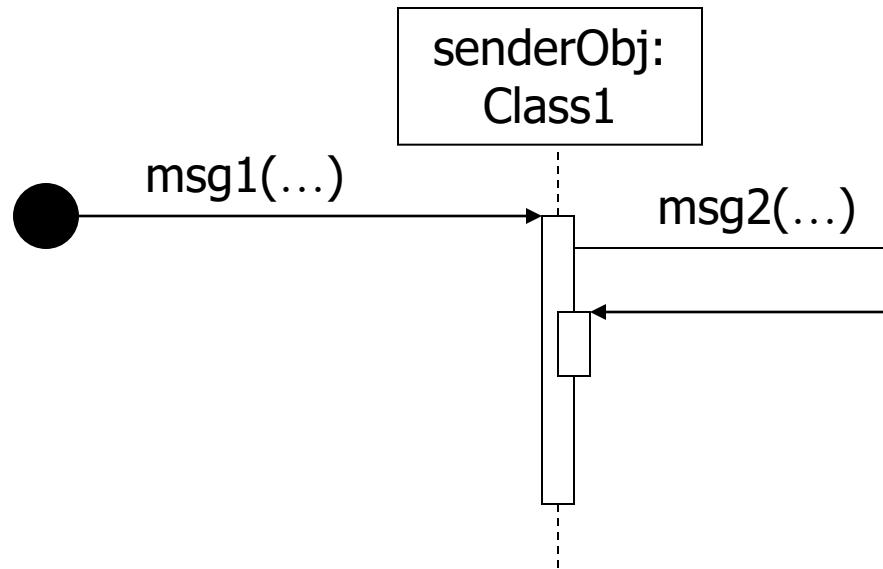
- A sequence diagram is made up of objects and messages. Objects are represented as rectangles with the underlined class name within the rectangle.

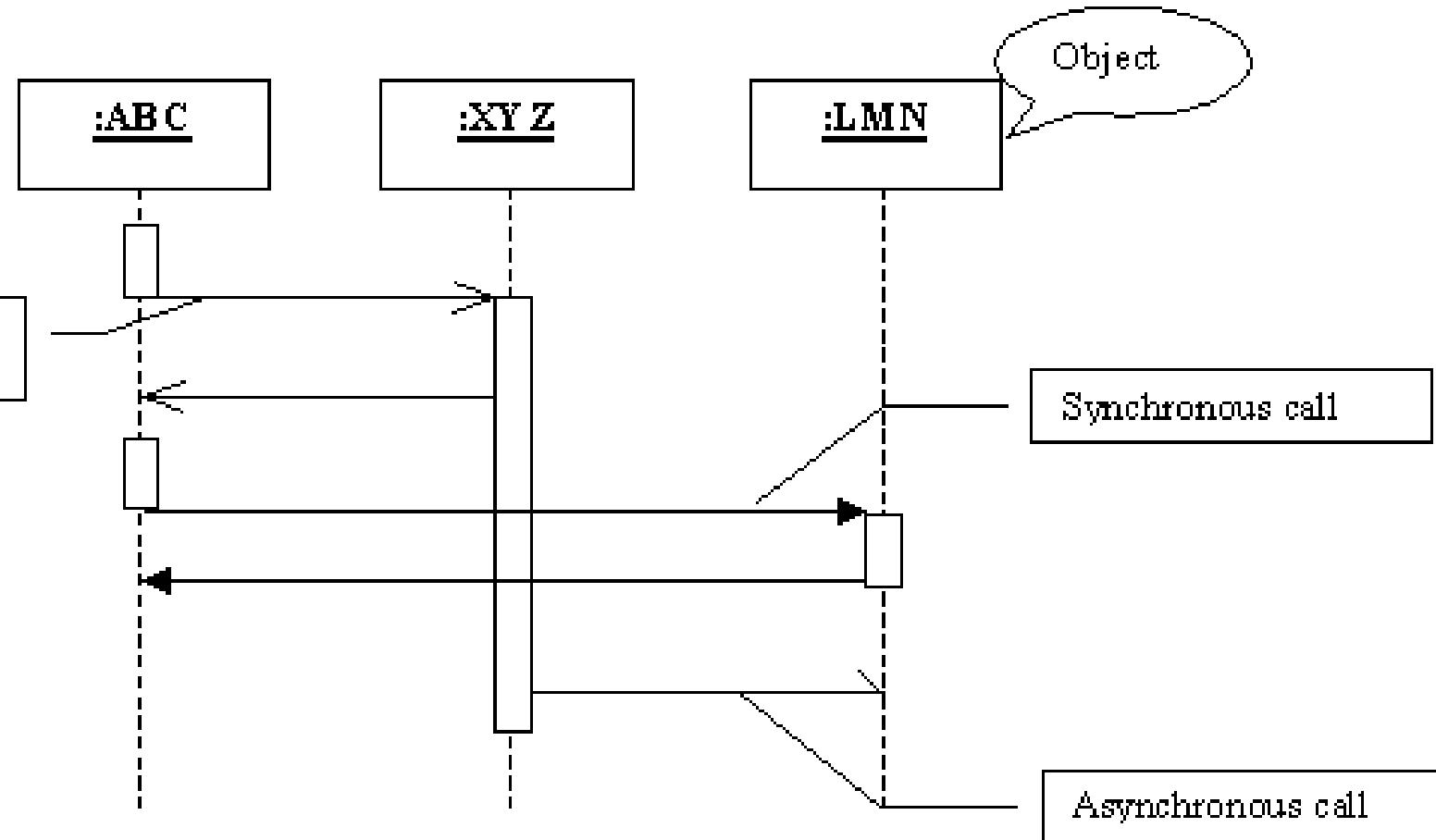


## Illustrating the return:



## Messages to Self:



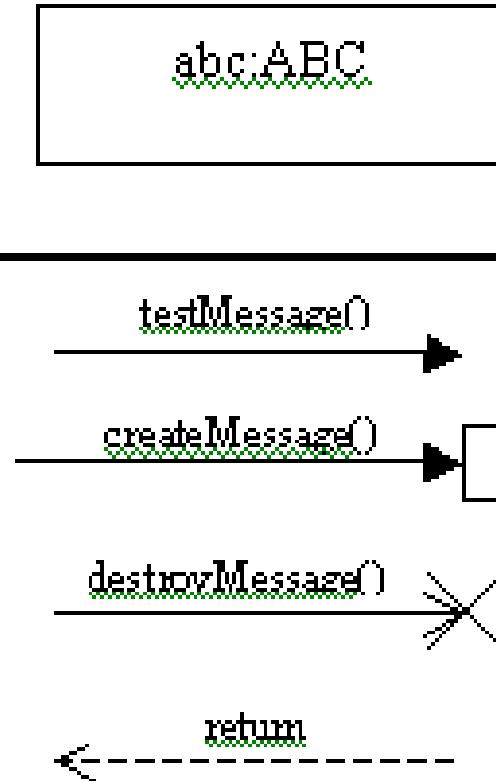


# Elements of a Sequence Diagram

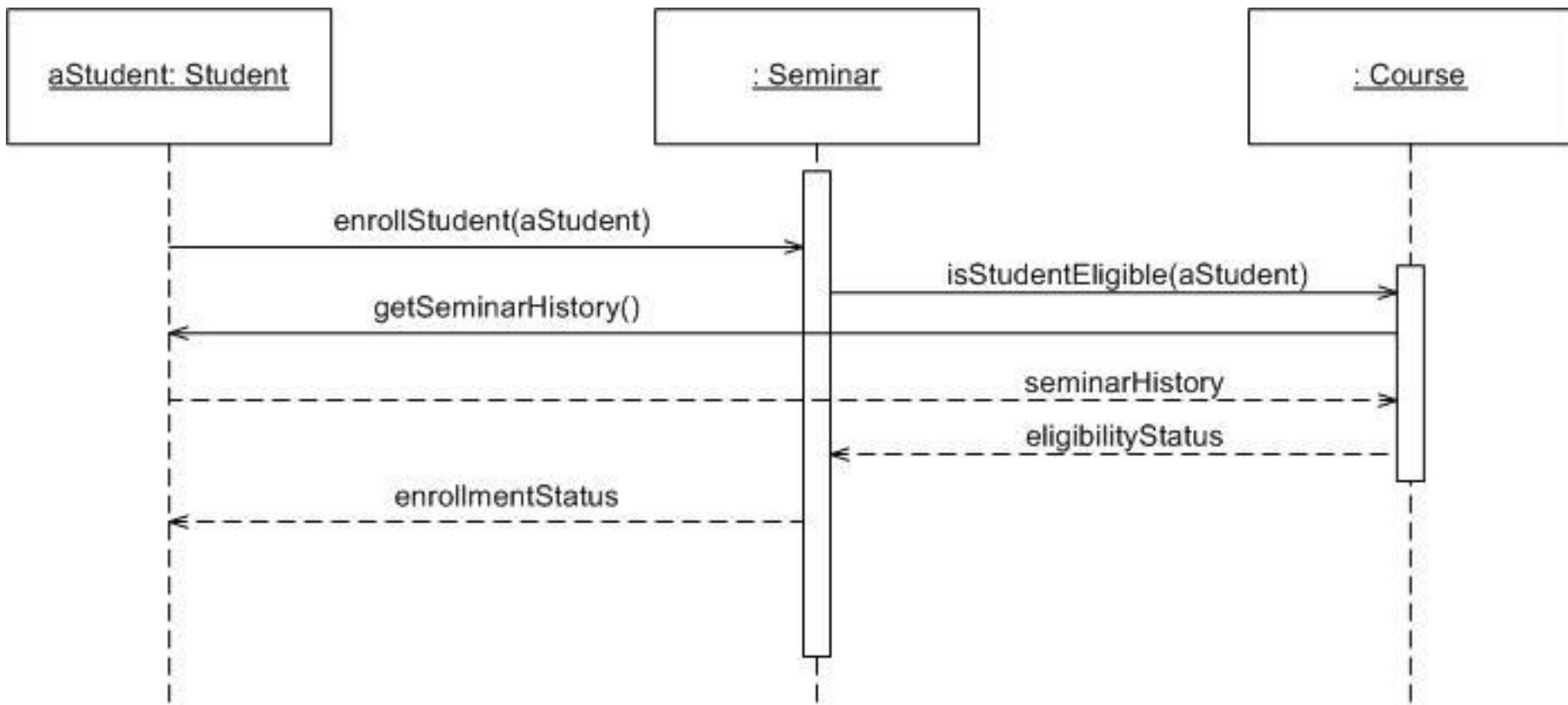
A Sequence diagram consists of the following behavioral elements:

**Object:** The primary element involved in a sequence diagram is an Object. A Sequence diagram consists of sequences of interaction among different objects over a period of time.

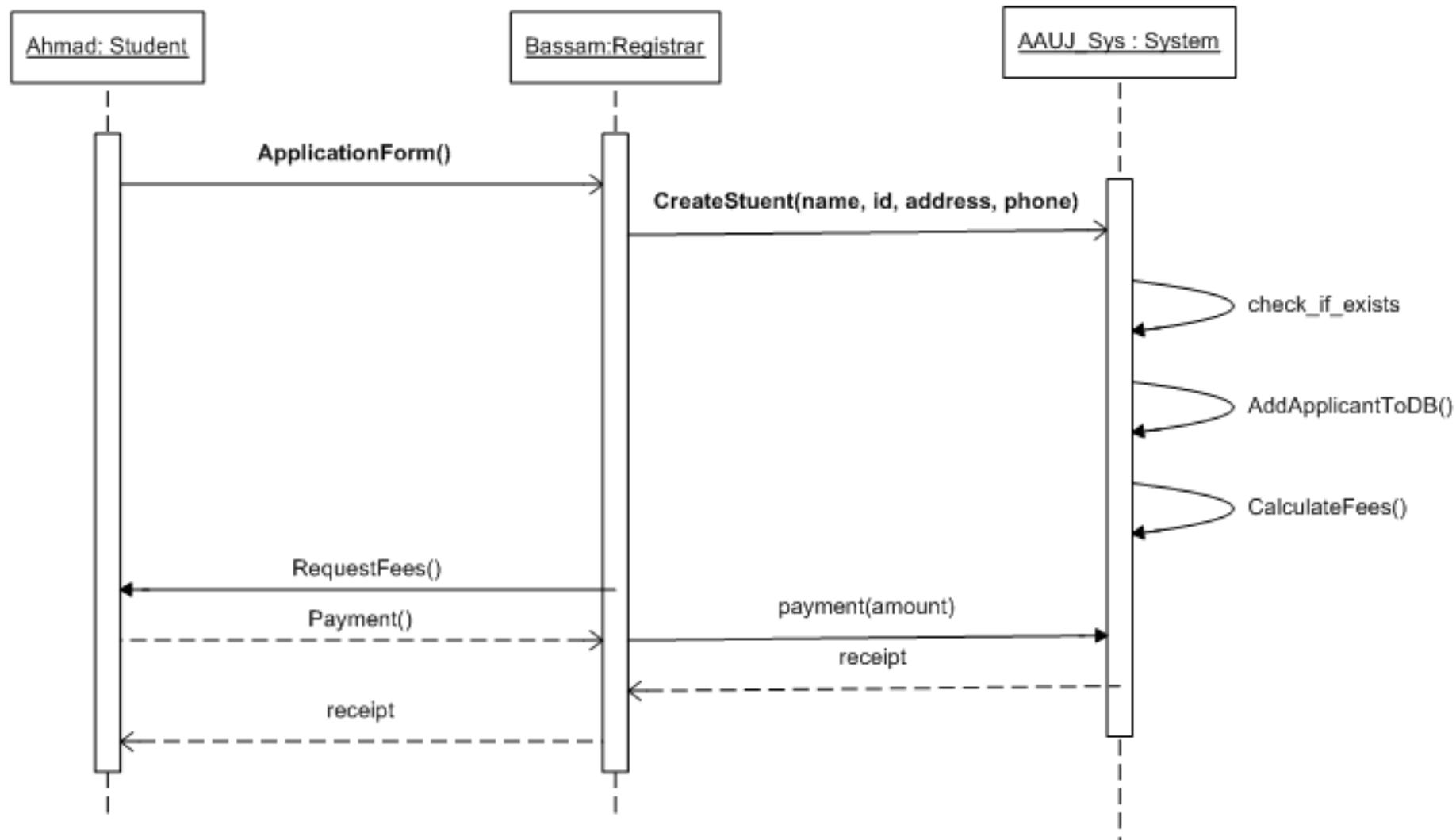
**Message:** The interaction between different objects in a sequence diagram is represented as messages. A message is represented by a directed arrow.



The following example shows the logic of how to enroll in a seminar.

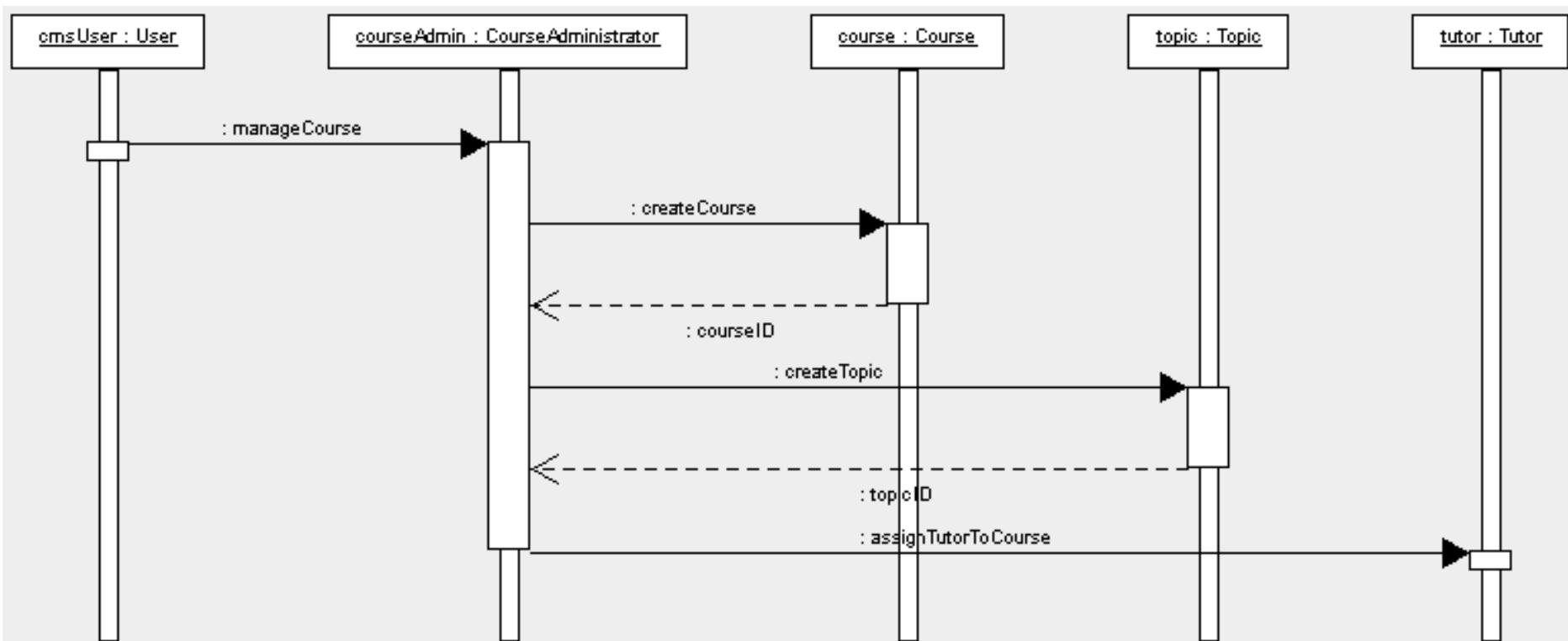


# Assignment No. 7: Sequence Diagram that for the Enroll in University Use Case



## **Identifying the activities and transitions for managing course information**

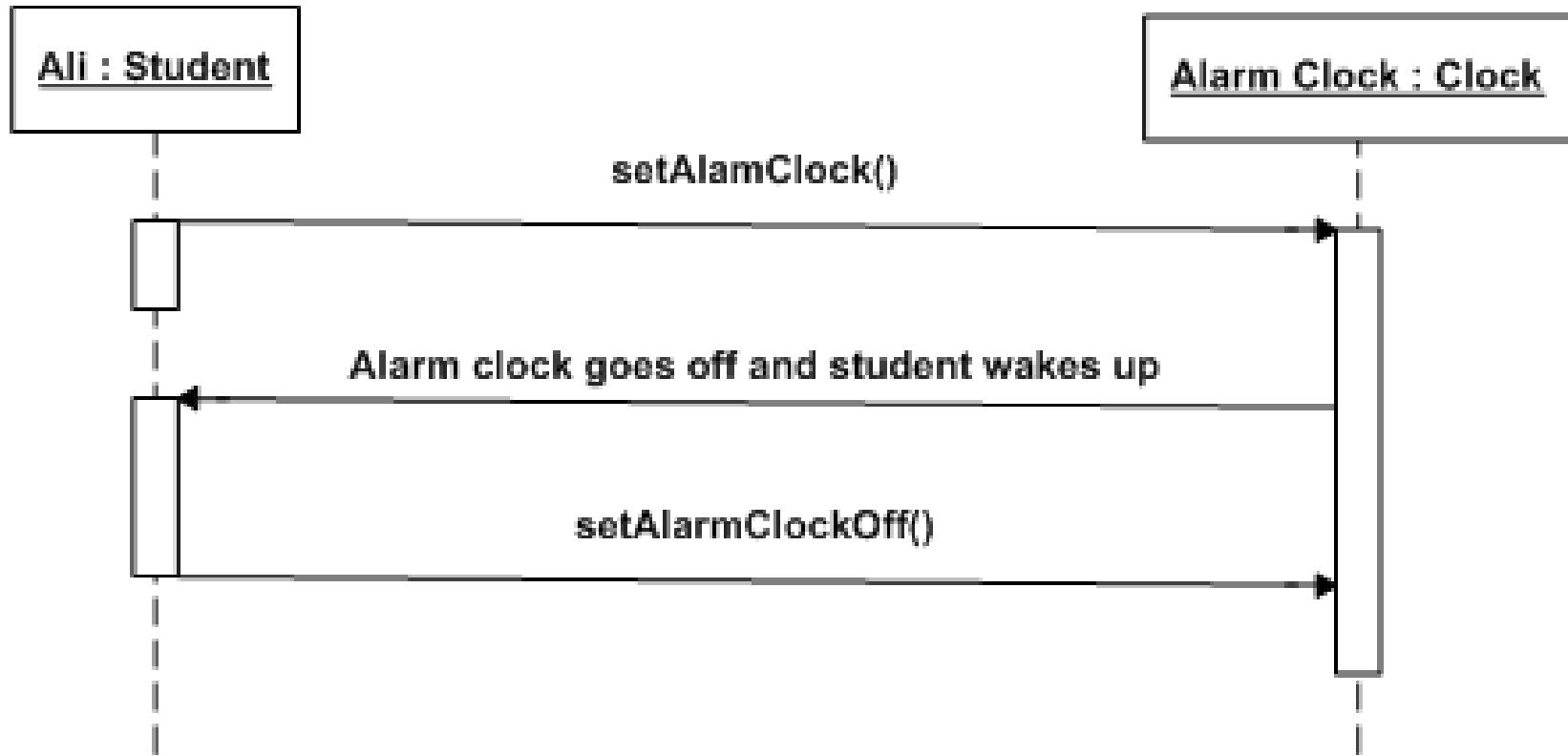
- A user who is a course administrator invokes the manage course functionality.
- The manage course functionality of the course administrator invokes either the course creation or course modification functionality of a course.
- After the course is either created or modified, the manage topic functionality of the course administrator calls the topic creation or modification functionality of a topic.
- Finally, the user invokes the assign tutor to course functionality of the course administrator to assign a tutor to the selected course.



# Assignment No. 8: Sequence Diagram that for the Enroll in University Use Case

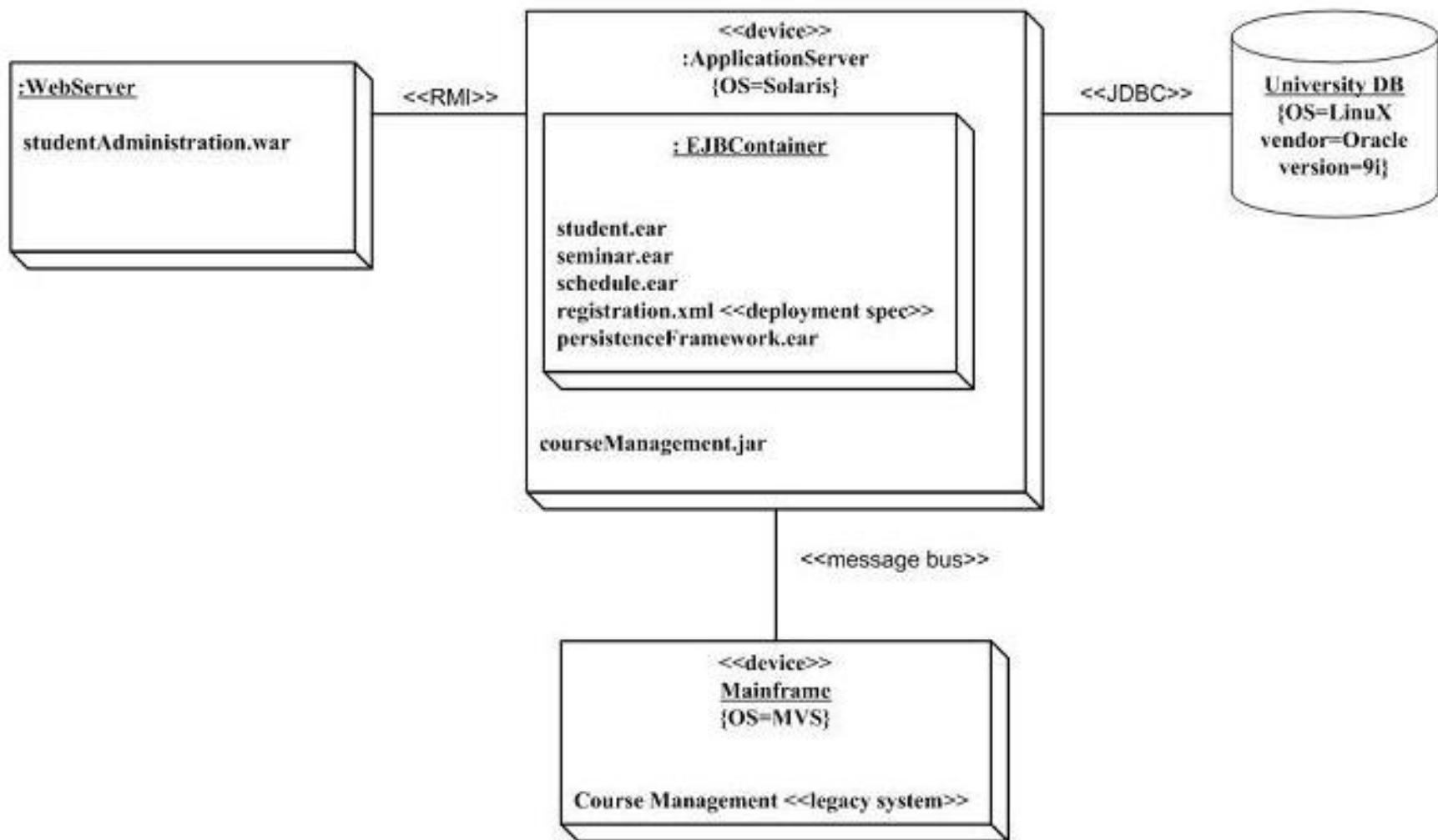
**Draw the sequence diagram for the following scenario**

- Student sets electric alarm clock to wake up time.
- Alarm clock goes off and student wakes up.
- Student sets alarm clock to off.

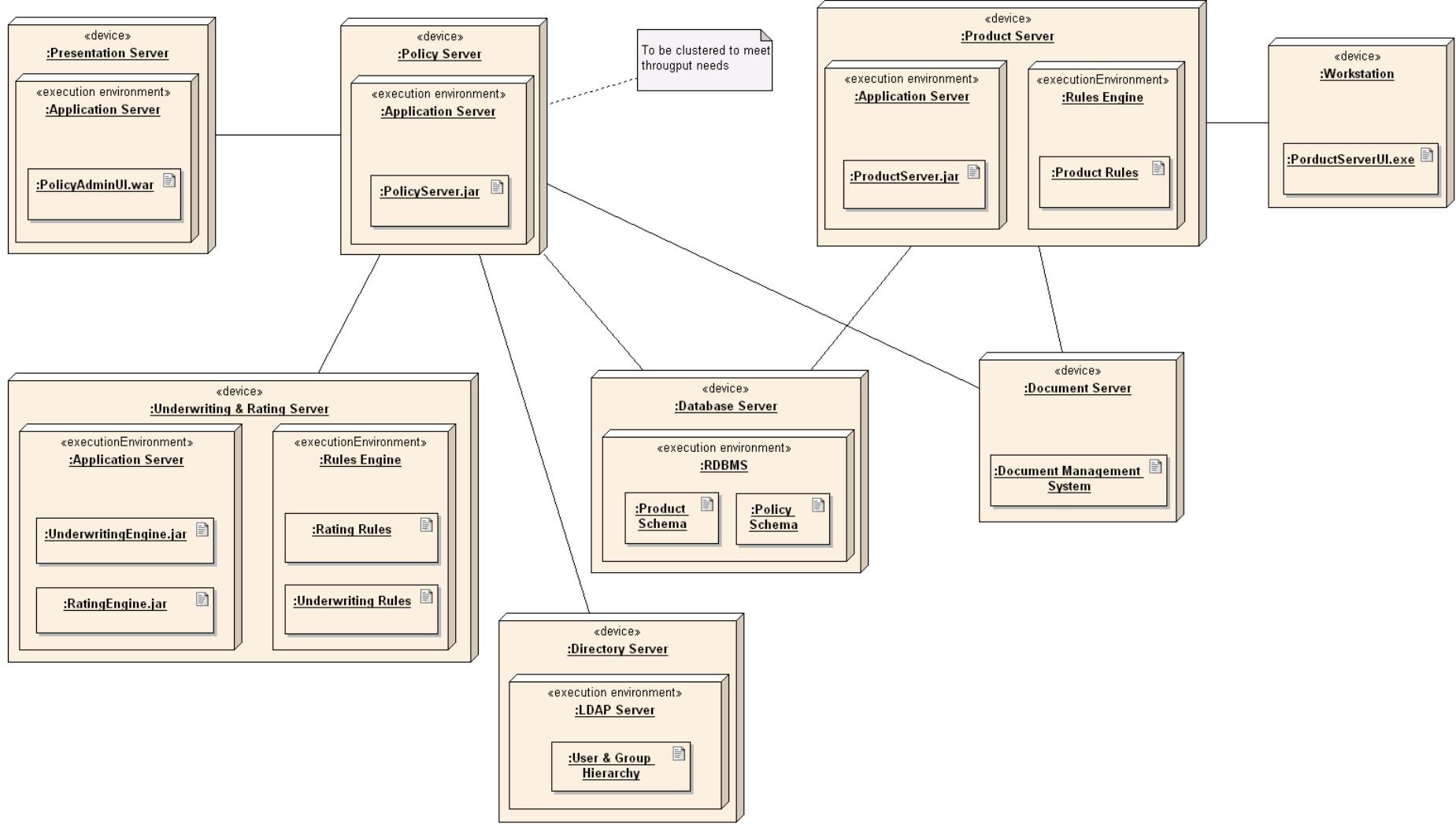


## Deployment Diagrams

- A **deployment diagram** in the [Unified Modeling Language](#) models the *physical* deployment of [artifacts](#) on [nodes](#).<sup>[1]</sup> To describe a web site, for example, a deployment diagram would show what hardware components ("nodes") exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node (e.g., web application, database), and how the different pieces are connected (e.g. JDBC, REST, RMI).
- The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have subnodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.
- There are two types of Nodes:
  - Device Node
  - Execution Environment Node
- Device nodes are physical computing resources with processing memory and services to execute software, such as typical computers or mobile phones. An execution environment node (EEN) is a software computing resource that runs within an outer node and which itself provides a service to host and execute other executable software elements.



## dd Deployment of Components



---

# Unified Process

## Introduction and History

---

# Topics

---

Unified Process

Unified Process Workflows

UML (in general)

Use Cases

# What Is a Process?

---

Defines Who is doing What, When to do it, and How to reach a certain goal.



# What is the Unified Process

---

A popular iterative modern process model (framework) derived from the work on the UML and associated process.

The leading object-oriented methodology for the development of large-scale software

Maps out when and how to use the various UML techniques

# What is the Unified Process

---

Develop high-risk elements in early iterations

Deliver value to customer

Accommodate change early on in project

Work as one team

Adaptable methodology - can be modified for the specific software product to be developed

2-dimensional systems development process described by a set of phases and workflows

Utilizes Millers Law

# Miller's Law

---

At any one time, we can concentrate on only approximately seven *chunks* (units of information)

To handle larger amounts of information, use *stepwise refinement*

Concentrate on the aspects that are currently the most important

Postpone aspects that are currently less critical

# History of UP

---

Some roots in the “*Spiral Model*” of Barry Boehm

Core initial development around 1995-1998

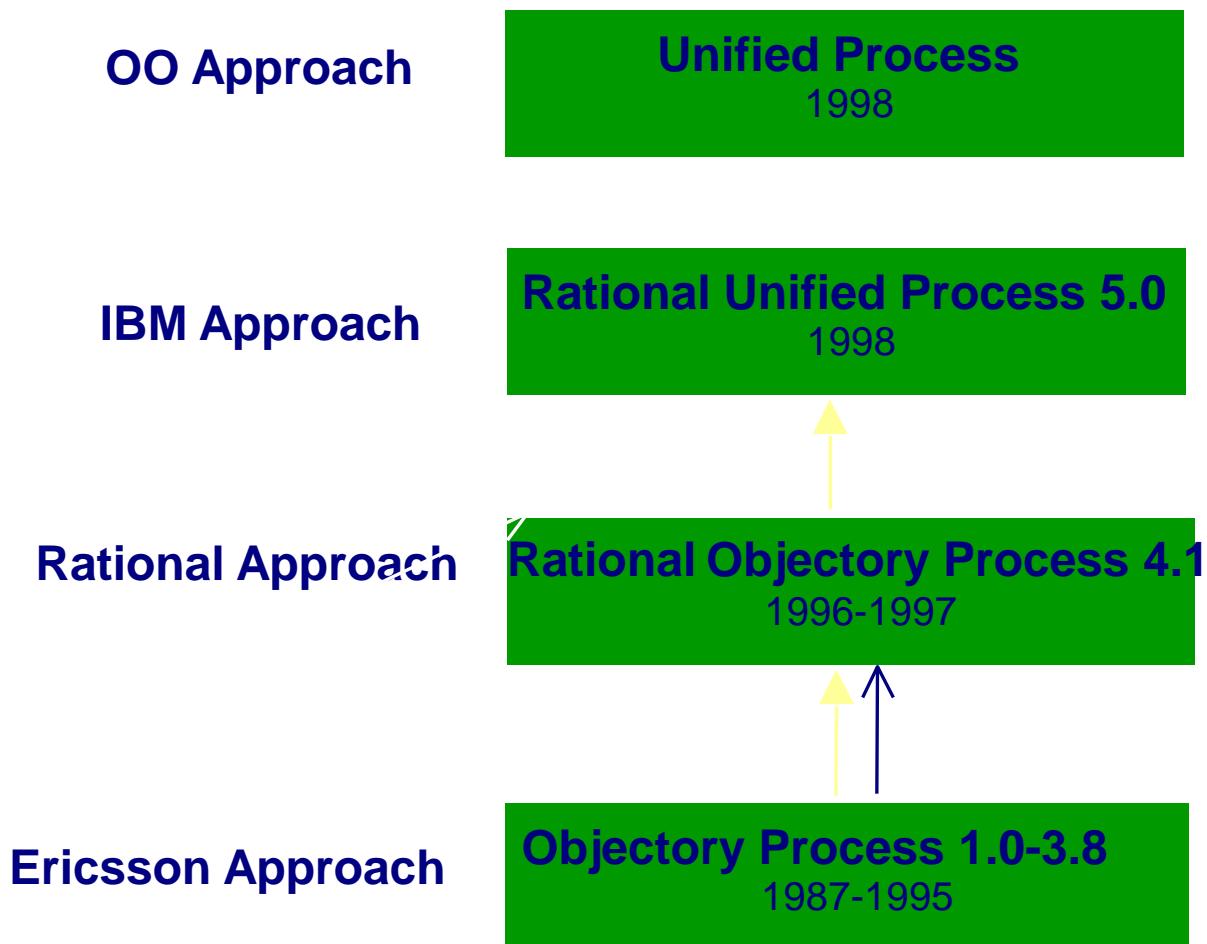
Large Canadian Air Traffic Control project as test bed

Philippe Kruchten chief architect of UP/RUP

Rational Corporation had commercial product in mind (*RUP, now owned by IBM*) but also reached out to public domain (UP)

# Creating the Unified Process

---



# The Rational Unified Process

---

RUP is a method of managing OO Software Development

It can be viewed as a Software Development Framework which is extensible and features:

- Iterative Development
- Requirements Management
- Component-Based Architectural Vision
- Visual Modeling of Systems
- Quality Management
- Change Control Management

# The Unified Process

---

## In Perspective:

Most of the world is NOT object oriented and doesn't use the process we're presenting here.

However, in practice, they do something very similar that works for them.

In 1999, Booch, Jacobson, and Rumbaugh published a complete object-oriented analysis and design methodology that unified their three separate methodologies. Called the Unified Process.

The Unified Process is an adaptable methodology

It has to be modified for the specific software product to be developed

# The Unified Process (contd)

---

UML is graphical

A picture is worth a thousand words

UML diagrams enable software engineers to communicate quickly and accurately

The Unified Process is a modeling technique

A *model* is a set of UML diagrams that represent various aspects of the software product we want to develop

UML stands for unified *modeling* language

UML is the tool that we use to represent (model) the target software product

The object-oriented paradigm is iterative and incremental in nature

There is no alternative to repeated iteration and incrementation until the UML diagrams are satisfactory

# Iteration and Incrementation

---

We cannot learn the complete Unified Process in one semester or quarter

- Extensive study and unending practice are needed

- The Unified Process has too many features

- A case study of a large-scale software product is huge

In this book, we therefore cover much, but not all, of the Unified Process

- The topics covered are adequate for smaller products

To work on larger software products, experience is needed

- This must be followed by training in the more complex aspects of the Unified Process

# Unified Process Phases

# Basic Characteristics of the Unified Process

---

Object-oriented

Use-case driven

Architecture centric

Iteration and incrementation

# Basic Characteristics of the Unified Process

---

## Object-oriented

Utilizes object oriented technologies.

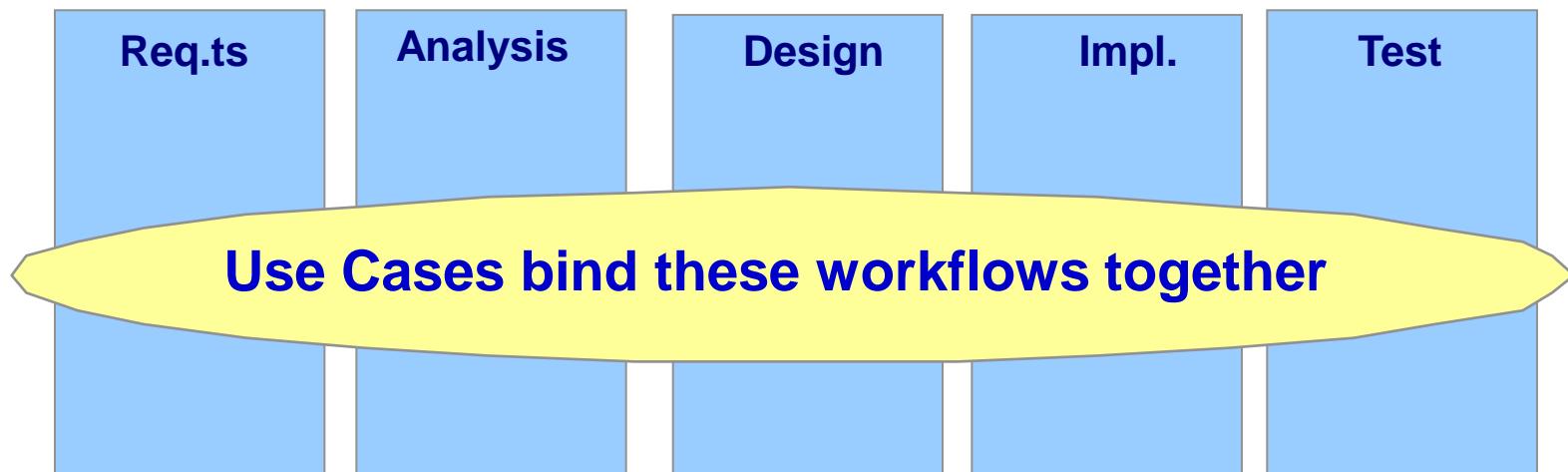
Classes are extracted during object-oriented analysis and designed during object-oriented design.

# Basic Characteristics of the Unified Process

---

## Use-case driven

Utilizes use case model to describe complete functionality of the system



# Basic Characteristics of the Unified Process

---

## Architecture centric

Embodies the most significant aspects of the system

View of the whole design with the important characteristics made more visible

Expressed with class diagram

# Basic Characteristics of the Unified Process

---

## Iteration and incrementation

Way to divide the work

Iterations are steps in the process, and increments are growth of the product

The basic software development process is iterative

Each successive version is intended to be closer to its target than its predecessor

# The Rational Unified Process

---

RUP is a method of managing OO Software Development  
It can be viewed as a Software Development Framework  
which is extensible and features:

Iterative Development

Requirements Management

Component-Based Architectural Vision

Visual Modeling of Systems

Quality Management

Change Control Management

# An Iterative Development Process...

---

Recognizes the reality of changing requirements

Caspers Jones's research on 8000 projects

40% of final requirements arrived after the analysis phase, after development had already begun

Promotes early risk mitigation, by breaking down the system into mini-projects and focusing on the riskier elements first

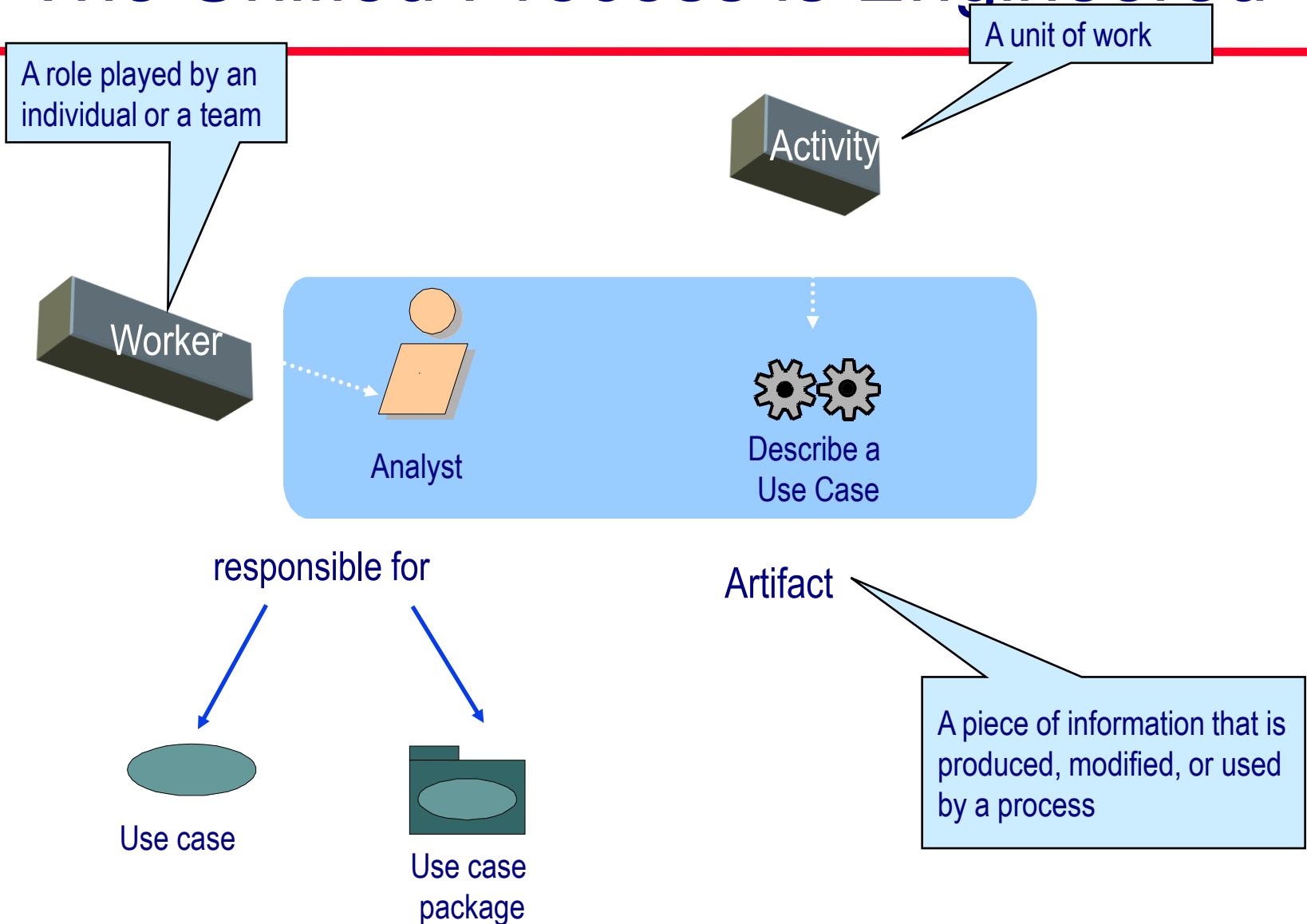
Allows you to “plan a little, design a little, and code a little”

Encourages all participants, including testers, integrators, and technical writers to be involved earlier on

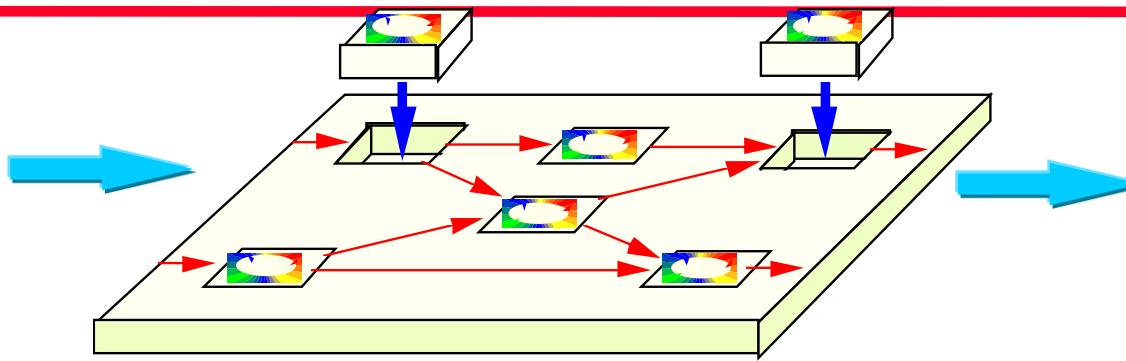
Allows the process itself to modulate with each iteration, allowing you to correct errors sooner and put into practice lessons learned in the prior iteration

Focuses on component architectures, not final big bang deployments

# The Unified Process is Engineered



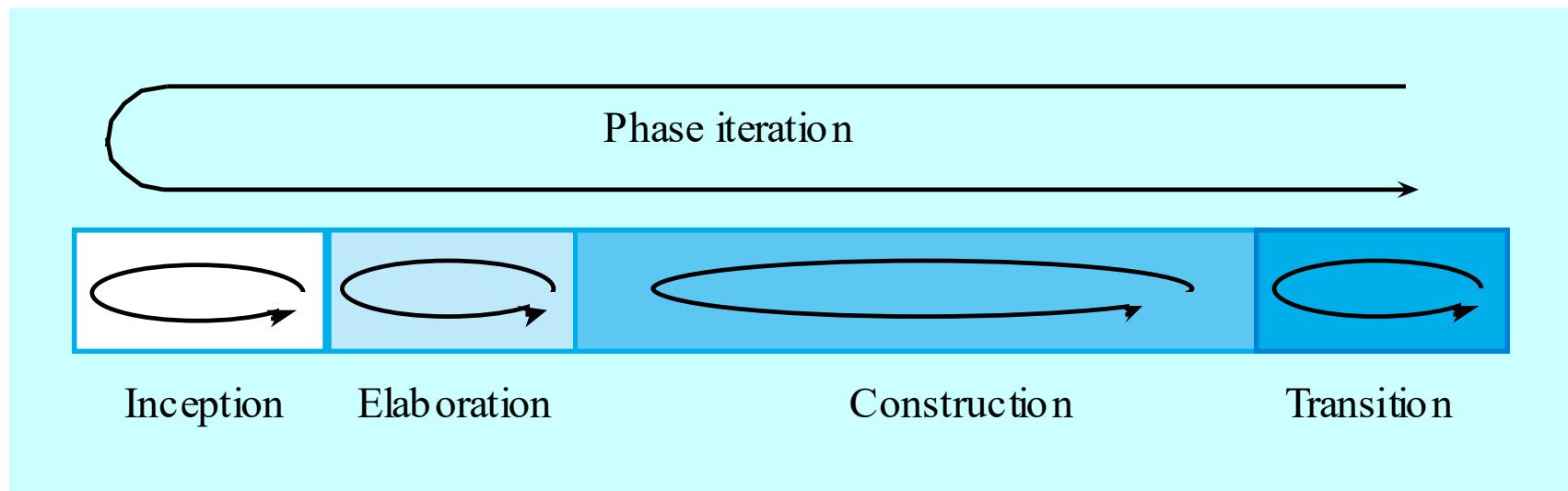
# The Unified Process is a Process Framework



There is NO Universal Process!

- The Unified Process is designed for flexibility and extensibility
  - » allows a variety of lifecycle strategies
  - » selects what artifacts to produce
  - » defines activities and workers
  - » models concepts

# Unified Process Model



# Goals and Features of Each Iteration

---

The primary goal of each iteration is to slowly chip away at the risk facing the project, namely:

- performance risks

- integration risks (different vendors, tools, etc.)

- conceptual risks (ferret out analysis and design flaws)

Perform a “minewaterfall” project that ends with a delivery of something tangible in code, available for scrutiny by the interested parties, which produces validation or correctives

Each iteration is risk-driven

The result of a single iteration is an increment--an incremental improvement of the system, yielding an evolutionary approach

# Unified Process Phases

Inception

Elaboration

Construction

Transition

## Inception

Establish the business case for the system, define risks, obtain 10% of the requirements, estimate next phase effort.

## Elaboration

Develop an understanding of the problem domain and the system architecture, risk significant portions may be coded/tested, 80% major requirements identified.

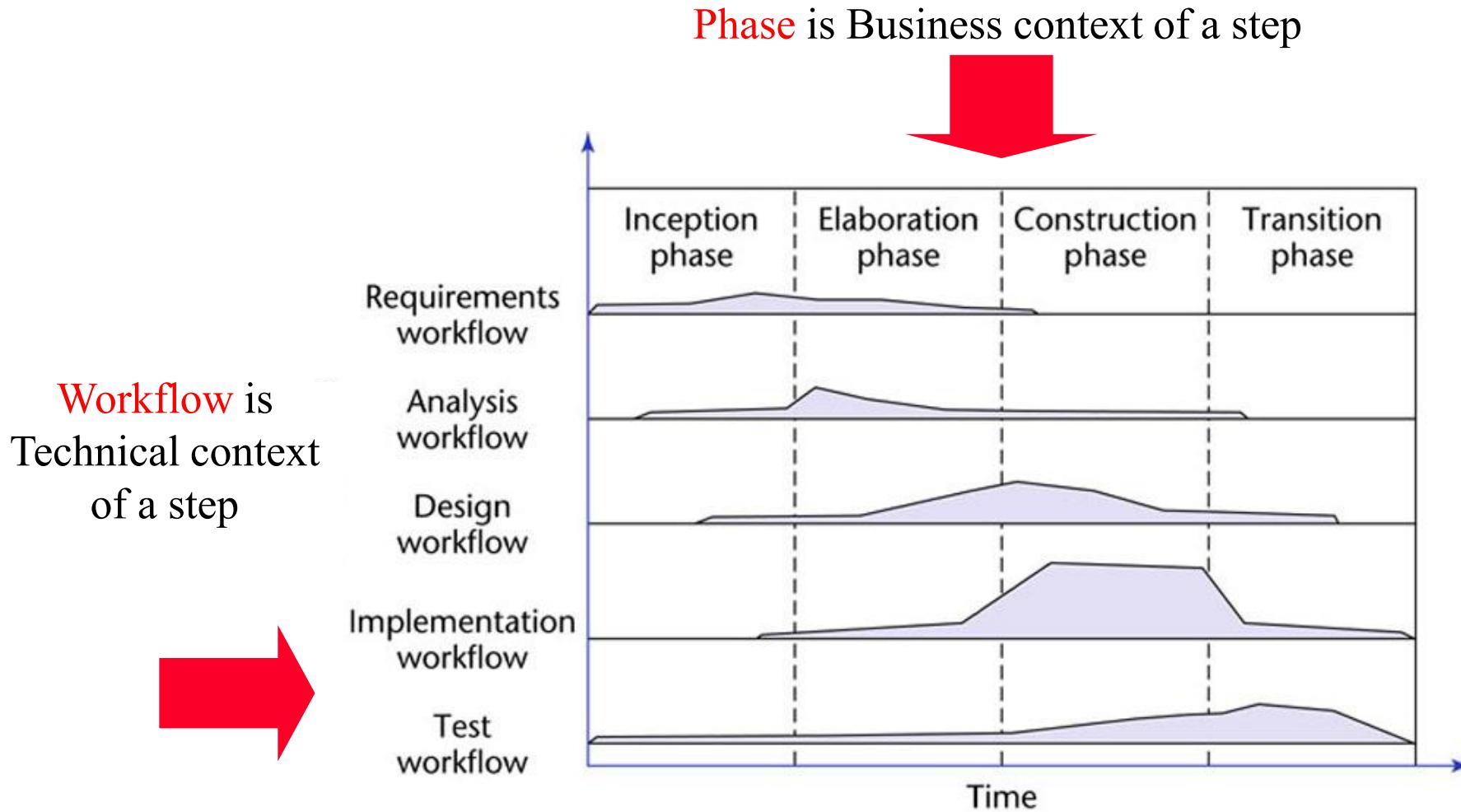
## Construction

System design, programming and testing. Building the remaining system in short iterations.

## Transition

Deploy the system in its operating environment. Deliver releases for feedback and deployment.

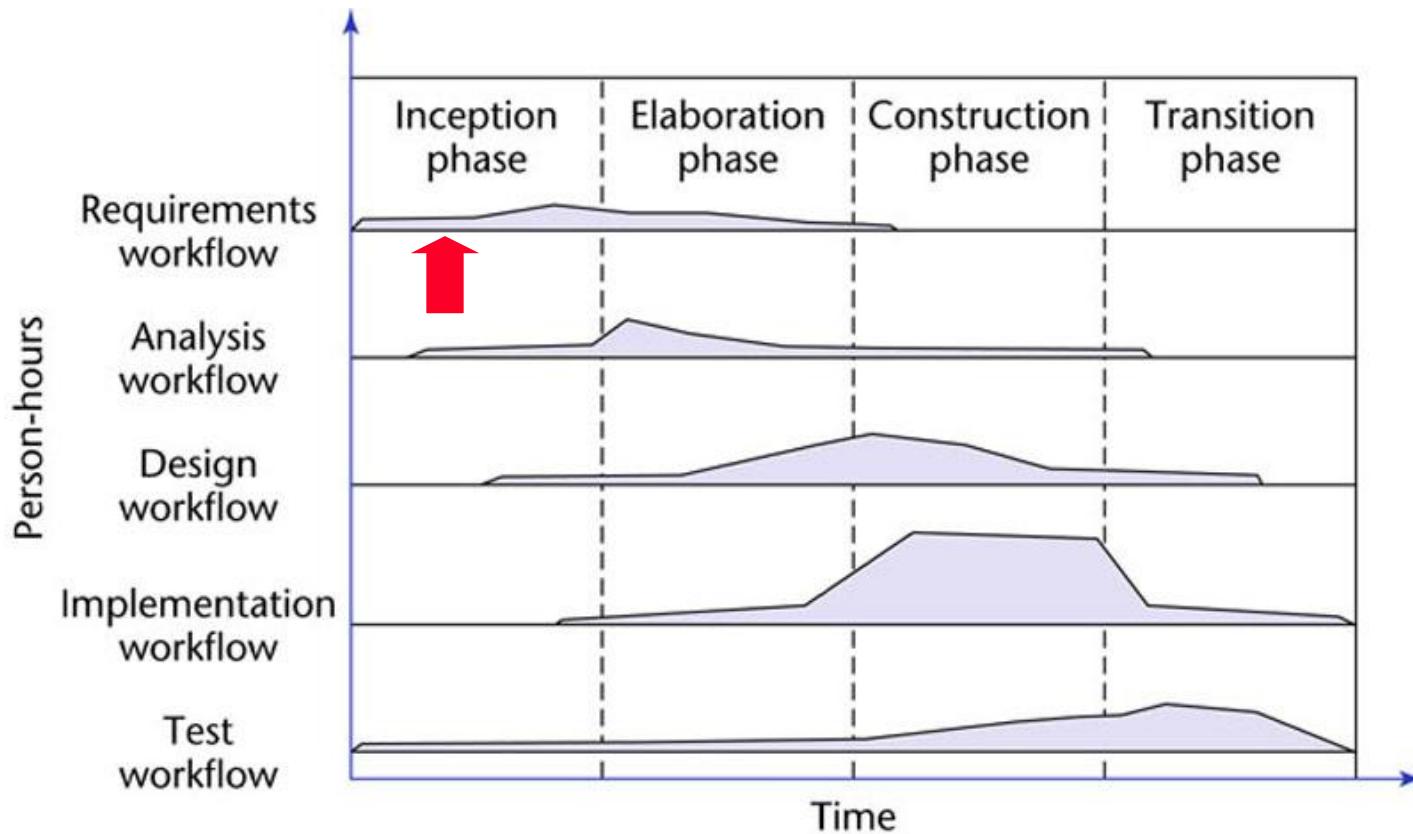
# The Phases/Workflows of the Unified Process



# The Phases/Workflows of the Unified Process

**NOTE:** Most of the requirement s work or workflow is done in the inception phase.

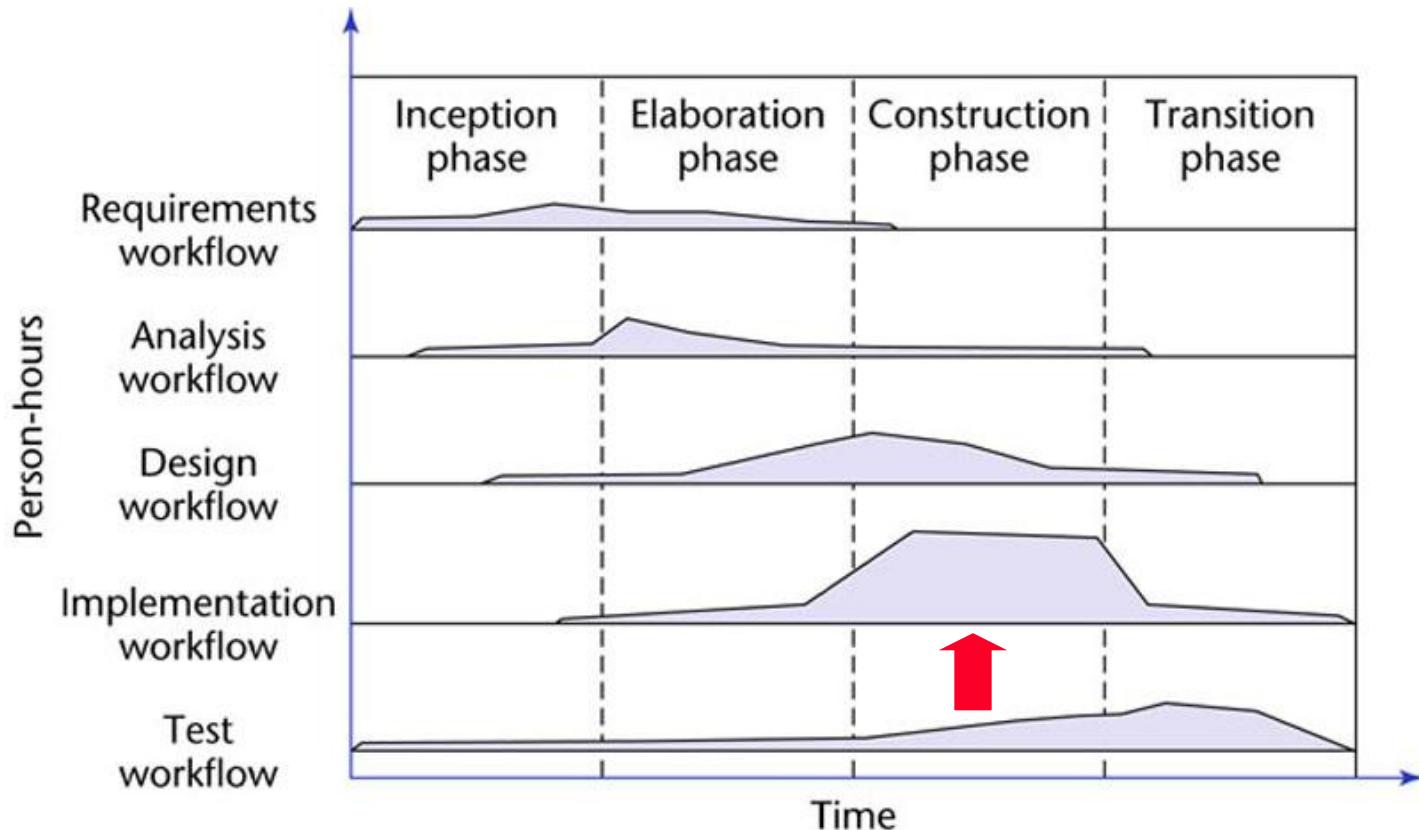
However some is done later.



# The Phases/Workflows of the Unified Process

**NOTE:** Most of the implementation work or workflow is done in construction

However some is done earlier and some later.



# Unified Process – Inception

Inception

## OVERVIEW

*Establish the business case for the system, define risks, obtain 10% to 20% of the requirements, estimate next phase effort.*

Primary Goal

Obtain buy-in from all interested parties

# Unified Process – Inception Objectives

## Inception

- Gain an understanding of the domain.
- Delimit the scope of the proposed project with a focus on the subset of the business model that is covered by the proposed software product
- Define an initial business case for the proposed system including costs, schedules, risks, priorities, and the development plan.
- Define any needed prototypes to mitigate risks.
- Obtain stakeholder concurrence on scope definition, expenditures, cost/schedule estimates, risks, development plan and priorities.

# Unified Process – Inception Activities

## Inception

- Project Initiation activities that allow new ideas to be evaluated for potential software development
- Project Planning work activities to build the team and perform the initial project planning activities
- Requirements work activities to define the business case for this potential software.
- Analysis and maybe Design work activities to define and refine costs, risks, scope, candidate architecture.
- Testing activities to define evaluation criteria for end-product vision

# Unified Process – Inception Activities

Inception

## Project Initiation

- Start with an idea
- Specify the end-product vision
- Analyze the project to assess scope
- Work the business case for the project including overall costs and schedule, and known risks
- Identify Stakeholders
- Obtain funding

# Unified Process – Inception Activities

## Inception

### Project Planning

- Build Team
- Define initial iteration
- Assess project risks and risk mitigation plan

There is insufficient information at the beginning of the inception phase to plan the entire development

The only planning that is done at the start of the project is the planning for the inception phase itself

For the same reason, the only planning that can be done at the end of the inception phase is the plan for just the next phase, the elaboration phase

# Unified Process – Inception Activities

## Inception

### Requirements

- Define or Refine Project Scope
- Begin to identify business model critical use cases of the system. (10% to 20% complete)
- Synthesize and exhibit least one candidate architectures by evaluating trade-offs, design, buy/reuse/build to refine costs.
- Prepare the supporting environment.
- Prepare development environment, selecting tools, deciding which parts of the process to improve
- Revisit estimation of overall costs and schedule.

### Analysis and maybe Design

- Define or refine costs, risks, scope, candidate architecture.
- Testing
  - Define evaluation criteria for end-product vision

# Unified Process – Inception Activities

## Inception

### Risk Assessment Activities

What are the risks involved in developing the software product, and

How can these risks be mitigated?

Does the team who will develop the proposed software product have the necessary experience?

Is new hardware needed for this software product?

If so, is there a risk that it will not be delivered in time?

If so, is there a way to mitigate that risk, perhaps by ordering back-up hardware from another supplier?

Are software tools needed?

Are they currently available?

Do they have all the necessary functionality?

# Unified Process – Inception Activities

## Inception

### Risk Assessment Activities

There are three major risk categories:

Technical risks

See earlier slide

The risk of not getting the requirements right

Mitigated by performing the requirements workflow correctly

The risk of not getting the architecture right

The architecture may not be sufficiently robust

To mitigate all three classes of risks

The risks need to be ranked so that the critical risks are mitigated first

# Unified Process – Inception Deliverables

Inception

## Primary deliverables:

- A vision document
- Initial version of the environment adoption (candidate)
- Any needed models or artifacts such as a domain model, business model, or requirements and analysis artifacts.
- Project plan, with phases and iterations with a more detailed plan for the elaboration phase.
- A project glossary
- One or several prototypes.

# Unified Process – Inception Deliverables

Inception

## Primary deliverables:

A vision document

NOTE: we use IEEE SRS Sec I,II

A general vision of the project's core requirements, key features and main constraints. Sets the scope of the project, identifies the primary requirements and constraints, sets up an initial project plan, and describes the feasibility of and risks associated with the project

Any needed models or artifacts such as a domain model, business model, or requirements and analysis artifacts.

An use-case model (10%-20% complete) – all Use Cases and Actors that can be identified so far with initial ordering.

An initial business case, which includes business context, success criteria (revenue projection, market recognition, and so on), and financial forecast;

A risk assessment analysis;

# Unified Process – Inception Questions

## Inception

- Is the proposed software product cost effective?
- How long will it take to obtain a return on investment?
- Alternatively, what will be the cost if the company decides not to develop the proposed software product?
- If the software product is to be sold in the marketplace, have the necessary marketing studies been performed?
- Can the proposed software product be delivered in time?
- If the software product is to be developed to support the client organization's own activities, what will be the impact if the proposed software product is delivered late?

# Unified Process – Elaboration Phase

Elaboration

## Elaboration

*Develop an understanding of the problem domain and the system architecture, risk significant portions may be coded/tested, 80% major requirements identified.*

The goal of the elaboration phase is to baseline the most significant requirements.

# Unified Process – Elaboration Objectives

Elaboration

## Elaboration Objectives

- *To refine the initial requirements and business case*
- *To ensure architecture, requirements, and plans are stable*
- *To monitor and address all architecturally significant risks of the project*
- *To refine or establish a baselined architecture*
- *To produce an evolutionary, throwaway, or exploratory prototypes*
- *To demonstrate that the baselined architecture will support the requirements of the system at a reasonable cost and in a reasonable time.*
- *To establish a supporting environment.*
- *To produce the project management plan*

# Unified Process – Elaboration Activities

## Elaboration

### Elaboration Essential Workflow Progress

- *All but completing the requirements workflow*
- *Performing virtually the entire analysis workflow*
- *Starting the design workflow by performing the architecture design*
- *Performing any construction workflows needed for prototypes to eliminate risks*

# Unified Process – Elaboration Activities

Elaboration

## Elaboration Essential Activities

- Analyze the problem domain.
- Define, validate and baseline the architecture
- Refine the Vision to understand the most critical Use Cases
- Create and baseline iteration plans for construction phase.
- Refine the development business case
- Put in place the development environment,
- Refine component architecture and decide build/buy/reuse
- Develop a project plan and schedule.
- Mitigate high-risk elements identified in the previous phase.

# Unified Process – Elaboration Deliverables

Elaboration

## Primary deliverables:

- Requirements model for the system
- The completed domain model (use cases, classes)
- The completed business model (costs, benefits, risks)
- The completed requirements artifacts
- The completed analysis artifacts
- Updated Architectural model
- Software project management plan

# Unified Process – Elaboration Outcomes

## Elaboration

Use Case model (at least 80% complete).

- All Use Cases identified.

- All Actors identified.

- Most Use-Case descriptions developed.

Supplementary requirements.

- (non-functional or not associated with a Use Case)

Software architecture description.

Executable architectural prototype.

Revised risk list and revised business case.

Development plan for overall project.

- coarse grained project plan, with iterations and evaluation criteria for each iteration.

Updated development case that specifies process to be used.

Preliminary user manual (optional).

# Unified Process – Elaboration Questions

Elaboration

Is the vision of the product stable?

Is the architecture stable?

Does the executable demonstration show that the major risk elements have been addressed and credibly resolved?

Is the plan for the construction phase sufficiently detailed and accurate?

Do all stakeholders agree that the current vision can be achieved if the current plan is executed to develop the complete system, in the context of the current architecture?

Is the actual resource expenditure versus planned expenditure acceptable?

# Unified Process – Construction Phase

Construction

## Construction

*System design, programming and testing. Building the remaining system in short iterations.*

The goal of the construction phase is to clarify the remaining requirements and complete the development of the first operational quality version of the software product.

# Unified Process – Construction Objectives

Construction

## Construction Objectives

- Minimizing development costs.
- Achieving adequate quality as rapidly as practical
- Achieving useful versions as rapidly as practical
- Complete analysis, design, development and testing of functionality.
- To iteratively and incrementally develop a complete product
- To decide if the software, sites, and users are deployment ready.
- To achieve parallelism in the work of development teams.

# Unified Process – Construction Activities

Construction

## Construction Essential Activities

- Complete component development and testing (beta release)
- Assess product releases against acceptance criteria for the vision. (Unit, Integration, Functional and System testing)
- Integrate all remaining components and features into the product
- Assure resource management control and process optimization

# Unified Process – Construction Deliverables

Construction

## Primary deliverables

Working software system (beta release version)

Associated documentation

Acceptance testing documentation

Updated project management deliverables (plan, risks, business case)

User Manuals

# Unified Process – Construction Outcomes

Construction

- A product ready to put into the hands of end users.
- The software product integrated on the adequate platforms.
- The user manuals.
- A description of the current release.

# Unified Process – Construction Questions

Construction

- Is this product (beta test version) release stable and mature enough to be deployed in the user community?
- Are all stakeholders ready for the transition into the user community?
- Are the actual resource expenditures versus planned expenditures still acceptable?

*Transition may have to be postponed by one release if the project fails to reach this milestone.*

# Unified Process – Transition Phase

Transition

## Transition

*Deploy the system in its operating environment.  
Deliver releases for feedback and deployment.*

The focus of the Transition Phase is to ensure that software is available for its end users and meets their needs. The Transition Phase can span several iterations, and includes testing the product in preparation for release, and making minor adjustments based on user feedback.

# Unified Process – Transition Objectives

Transition

## Transition Objectives

Assess deployment baselines against acceptance criteria

Achieve user self-supportability

Achieving stakeholder concurrence of acceptance

# Unified Process – Transition Activities

Transition

## Transition Essential Activities

- Finalize end-user support material
- Test the deliverable product at the development site
- Validate beta test to assure user expectations met
- Fine-tune the product based on feedback
- Perform parallel operation of replaced legacy system
- Convert operational databases
- Train of users and maintainers
- Roll-out to the marketing, distribution and sales forces
- Perform deployment engineering (cutover, roll-out performance tuning)

# Unified Process – Transition Deliverables

Transition

## Primary deliverable

Final product onto a production platform

## Other deliverables

All the artifacts (final versions)

Completed manual

# Phase Deliverables

Inception Phase	Elaboration Phase	Construction Phase	Transition Phase
<ul style="list-style-type: none"><li>• The initial version of the domain model</li><li>• The initial version of the business model</li><li>• The initial version of the requirements artifacts</li><li>• A preliminary version of the analysis artifacts</li><li>• A preliminary version of the architecture</li><li>• The initial list of risks</li><li>• The initial ordering of the use cases</li><li>• The plan for the elaboration phase</li><li>• The initial version of the business case</li></ul>	<ul style="list-style-type: none"><li>• The completed domain model</li><li>• The completed business model</li><li>• The completed requirements artifacts</li><li>• The completed analysis artifacts</li><li>• An updated version of the architecture</li><li>• An updated list of risks</li><li>• The project management plan (for the rest of the project)</li><li>• The completed business case</li></ul>	<ul style="list-style-type: none"><li>• The initial user manual and other manuals, as appropriate</li><li>• All the artifacts (beta release versions)</li><li>• The completed architecture</li><li>• The updated risk list</li><li>• The project management plan (for the remainder of the project)</li><li>• If necessary, the updated business case</li></ul>	<ul style="list-style-type: none"><li>• All the artifacts (final versions)</li><li>• The completed manuals</li></ul>

# UP Life cycle in four phases

---

- Inception
- Elaboration
- Construction
- Transition

The Enterprise Unified Process (EUP) adds two more phases to this:

*Production:* keep system useful/productive after deployment to customer

*Retirement:* archive, remove, or reuse etc.

# Example roles in UP

---

*Stake Holder:* customer, product manager, etc.

*Software Architect:* established and maintains architectural vision

*Process Engineer:* leads definition and refinement of Development Case

*Graphic Artist:* assists in user interface design, etc.

# Some UP guidelines

---

Attack risks early on and continuously so, before they will attack you

Stay focused on *developing executable software* in early iterations

Prefer component-oriented architectures and reuse existing components

Quality is a way of life, not an afterthought

# Six best “must” UP practices

---

Time-boxed iterations: *avoid speculative powerpoint architectures*

*Strive for cohesive architecture* and reuse existing components:

e.g. core architecture developed by small, co-located team

then early team members divide into sub-project leaders

# Six best “must” UP practices

---

Continuously verify quality: test early & often, and realistically by integrating all software at each iteration

Visual modeling: prior to programming, do at least some visual modeling to explore creative design ideas

# Six best “must” UP practices

---

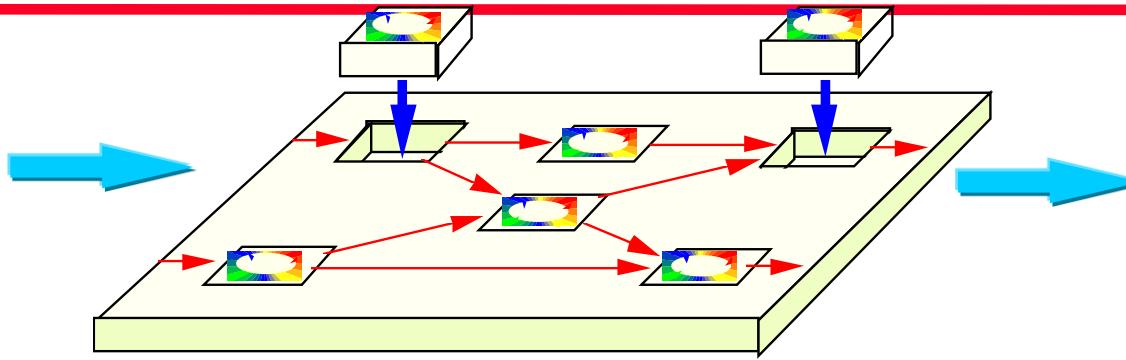
Manage requirements: find, organize, and track requirements through skillful means

Manage change:

disciplined configuration management protocol, version control,  
change request protocol  
baselined releases at iteration ends

# Unified Process Workflows

# The Unified Process is a Process Framework



While the Unified Process is widely used, there is NO Universal Process!

- The Unified Process is designed for flexibility and extensibility
  - » allows a variety of lifecycle strategies
  - » selects what artifacts to produce
  - » defines activities and workers
  - » models concepts
  - » **IT IS A PROCESS FRAMEWORK for development**

# The Unified Process

---

The Unified Process IS A  
2-dimensional systems development  
process described by a  
set of phases and (dimension one)  
Workflows (dimension two)

# The Unified Process

---

## Phases

Describe the business steps needed to develop, buy, and pay for software development.

The business increments are identified as phases

## Workflows

Describe the tasks or activities that a developer performs to evolve an information system over time

# Why a Two-Dimensional Model?

---

In an ideal world, each workflow would be completed before the next workflow is started

In reality, the development task is too big for this

As a consequence of Miller's Law

The development task has to be divided into increments (phases) Within each increment, iteration is performed until the task is complete

At the beginning of the process, there is not enough information about the software product to carry out the requirements workflow

Similarly for the other core workflows

# Why a Two-Dimensional Model?

---

A software product has to be broken into subsystems. Even subsystems can be too large at times. Modules may be all that can be handled until a fuller understanding of all the parts of the product as a whole has been obtained

The Unified Process handles the inevitable changes well

- The moving target problem

- The inevitable mistakes

The Unified Process works for treating a large problem as a set of smaller, largely independent sub problems

- It provides a framework for incrementation and iteration

- In the future, it will inevitably be superseded by some better methodology

# Process Overview

	Phases (time)			
Workflow (tasks)	Inception	Elaboration	Construction	Transition
Requirements				
Analysis				
Design				
Implementation				
Test				

# Static workflows

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Test	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow managed changes to the system (see Chapter 29).
Project management	This supporting workflow manages the system development (see Chapter 5).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

# Primary Workflows

---

The Unified Process

## PRIMARY WORKFLOWS

**Requirements workflow**

**Analysis workflow**

**Design workflow**

**Implementation workflow**

**Test workflow**

**Post delivery maintenance workflow**

## Supplemental Workflows

**Planning Workflow**

# Planning Workflow

---

Define scope of Project

Define scope of next iteration

Identify Stakeholders

Capture Stakeholders expectation

Build team

Assess Risks

Plan work for the iteration

Plan work for Project

Develop Criteria for iteration/project closure/success

UML concepts used: initial Business Model, using class diagram

# Requirements Workflow

---

## Primary focus

To determine the client's needs by eliciting both functional and nonfunctional requirements

Gain an understanding of the *application domain*

Described in the language of the customer

# Requirements Workflow

---

The aim is to determine the client's needs

First, gain an understanding of the *domain*

How does the specific business environment work

Second, build a business model

Use UML to describe the client's business processes

If at any time the client does not feel that the cost is justified,  
development terminates immediately

It is vital to determine the client's constraints

Deadline -- Nowadays software products are often mission critical

Parallel running

Portability

Reliability

Rapid response time

Cost

The aim of this *concept exploration* is to determine

What the client needs, and

*Not* what the client wants

# Requirements Workflow

---

List candidate requirements

textual feature list

Understand system context

domain model describing important concepts of the context

business modeling specifying what processes have to be supported by the system using Activity Diagram

Capture functional and nonfunctional requirements

Use Case Model

Supplementary requirements

physical, interface, design constraints, implementation constraints

# Analysis Workflow

---

## Primary focus

Analyzing and refining the requirements to achieve a detailed understanding of the requirements essential for developing a software product correctly

To ensure that both the developer and user organizations understand the underlying problem and its domain

Written in a more precise language

# Analysis Workflow

The aim of the analysis workflow

To analyze and refine the requirements

Two separate workflows are needed

The requirements artifacts must be expressed in the language of the client

The analysis artifacts must be precise, and complete enough for the designers

Specification document (“specifications”)

Constitutes a contract

It must not have imprecise phrases like “optimal,” or “98 percent complete”

Having complete and correct specifications is essential for

Testing, and

Maintenance

The specification document must not have

Contradictions

Omissions

Incompleteness

# Analysis Workflow

Structure the Use Cases

Start reasoning about the internal of the system

Develop Analysis Model: Class Diagram and State Diagram

Focus on what is the problem not how to solve it

Understand the main concepts of the problem

Three main types of classes stereotypes may be used:

Boundary Classes: used to model interaction between system and actors

Entity Classes: used to model information and associated behavior directly derived from real-world concept

Control Class: used to model business logic, computations transactions or coordination.

The specification document must not have

Contradictions

Omissions

Incompleteness

# Design Workflow

---

The aim of the design workflow is to refine the analysis workflow until the material is in a form that can be implemented by the programmers

Determines the internal structure of the software product

# Design Workflow

---

The goal is to refine the analysis workflow until the material is in a form that can be implemented by the programmers

Many nonfunctional requirements need to be finalized at this time, including: Choice of programming language, Reuse issues, Portability issues.

## Classical Design

### Architectural design

Decompose the product into modules

### Detailed design

Design each module using data structures and algorithms

## Object Oriented Design

Classes are extracted during the object-oriented analysis workflow, and

Designed during the design workflow

# Design Workflow

## General Design

Refine the Class Diagram

Structure system with Subsystems, Interfaces, Classes

Define subsystems dependencies

Capture major interfaces between subsystems

Assign responsibilities to new design classes

Describe realization of Use Cases

Assign visibility to class attributes

Design Databases and needed Data Structures

Define Methods signature

Develop state diagram for relevant design classes

Use Interaction Diagram to distribute behavior among classes

Use Design Patterns for parts of the system

# Design Workflow

## Architectural Design

### Identify Design Mechanisms

Refine Analysis based on implementation environment

Characterize needs for specific mechanisms (inter-process communication, real-time computation, access to legacy system, persistence, ...)

Assess existing implementation mechanisms

### Identify Design Classes and Subsystems

A Subsystem is a special kind of Package which has behavioral semantics (realizes one or more interfaces)

Refine analysis classes

Group classes into Packages

Identify Subsystems when analysis classes are complex

- Look for strong interactions between classes
- Try to organize the UI classes into a subsystem
- Separate functionality used by different actors in different subsystems
- Separate subsystems based on the distribution needs

Identify Interfaces of the subsystems

# Implementation Workflow

---

The aim of the implementation workflow is to implement the target software product in the selected implementation language

# Implementation Workflow

---

Distribute the system by mapping executable components onto nodes in the deployment model

Implement Design Classes and subsystems through packaging mechanism:

package in Java, Project in VB, files  
directory in C++

Acquire external components realizing needed interfaces

Unit test the components

Integrate via builds

Requirements

Analysis

Design

Implementation

Testing

# Test Workflow

---

Carried out in parallel with other workflows

Primary purpose

To increase the quality of the evolving system

The test workflow is the responsibility of

Every developer and maintainer

Quality assurance group

# Test Workflow

---

Develop set of test cases that specify what to test in the system

many for each Use Case  
each test case will verify one scenario of the use case  
based on Sequence Diagram

Develop test procedures specifying how to perform test cases

Develop test component that automates test procedures

# Deployment Workflow

---

Activities include

Software packaging

Distribution

Installation

Beta testing

# Deployment Workflow

---

## Producing the Software

Output of implementation is tested executables.

Must be associated with other artifacts to constitute a complete product:

- Installation scripts

- User documentation

- Configuration data

- Additional programs for migration: data conversion.

In some cases:

- different executables needed for different user configurations

- different sets of artifacts needed for different classes of users:

  - new users versus existing users,

  - variants by country or language

# Deployment Workflow

---

Producing the Software (continued)

For distributed software, different sets may have to be produced for different computing nodes in the network

Packaging the Software

Distributing the Software

Installing the Software

Migration

Providing Help and Assistance to Users

Acceptance

# Iterations and Workflow

## Core Workflows

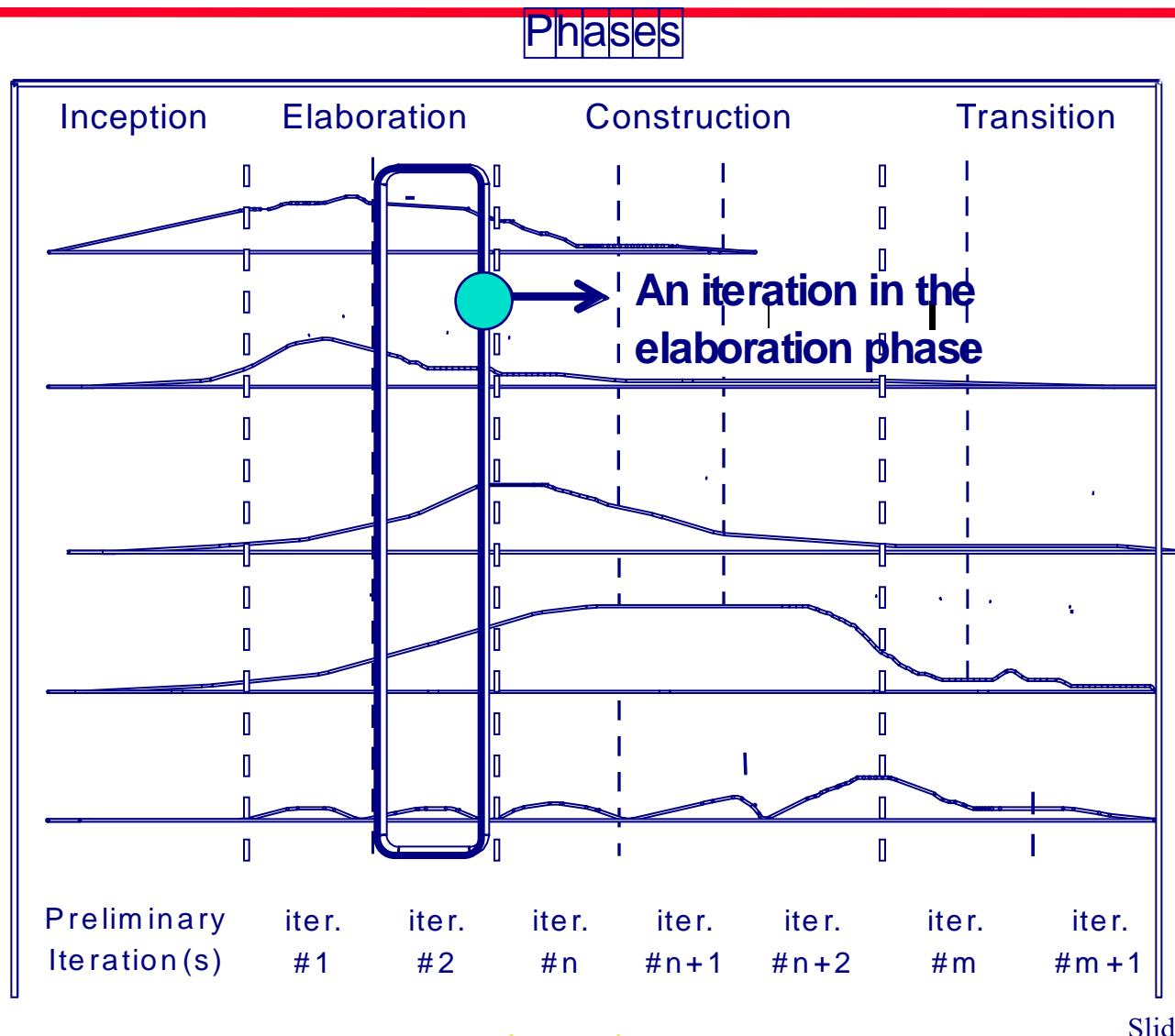
Requirements

Analysis

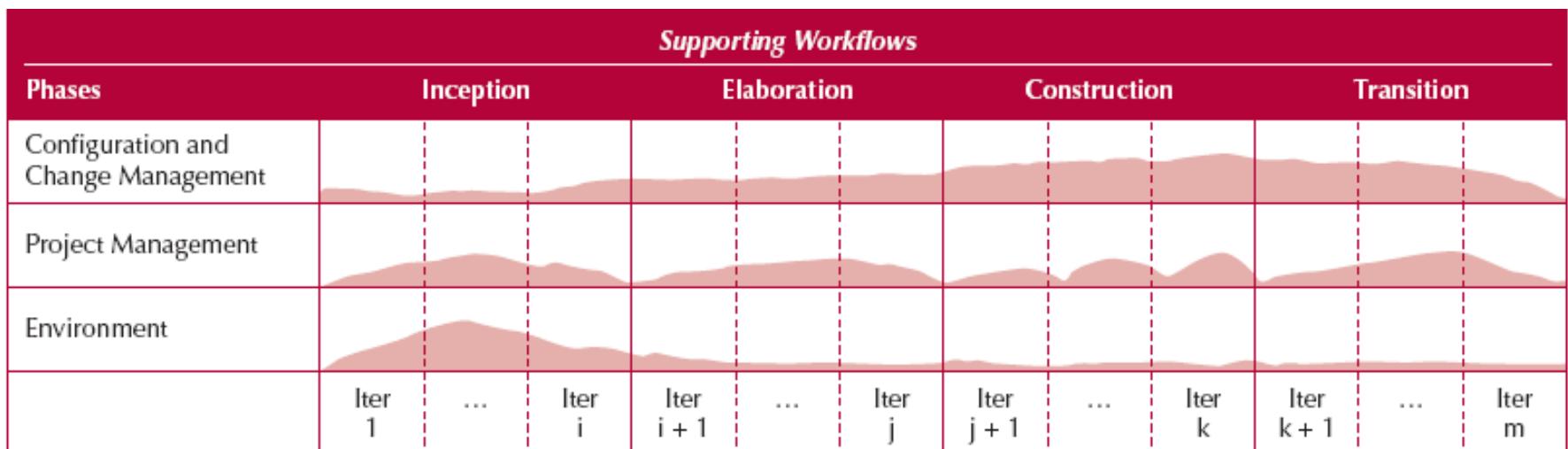
Design

Implementation

Test



# Supporting Workflows of The Unified Process



# Software Project Management Plan

---

Once the client has signed off the specifications, detailed planning and estimating begins

We draw up the software project management plan, including

- Cost estimate
- Duration estimate
- Deliverables
- Milestones
- Budget

This is the earliest possible time for the SPMP

# Post delivery Maintenance

---

Post delivery maintenance is an essential component of software development

More money is spent on post delivery maintenance than on all other activities combined

Problems can be caused by

Lack of documentation of all kinds

Two types of testing are needed

Testing the changes made during post delivery maintenance

Regression testing

All previous test cases (and their expected outcomes) need to be retained

# Retirement

---

Software can be made unmaintainable because

- A drastic change in design has occurred

- The product must be implemented on a totally new hardware/operating system

- Documentation is missing or inaccurate

- Hardware is to be changed—it may be cheaper to rewrite the software from scratch than to modify it

These are instances of maintenance (rewriting of existing software)

True retirement is a rare event

It occurs when the client organization no longer needs the functionality provided by the product

# What to Read...

---

- Dean Leffingwell, Don Widrig, Managing Software Requirements, Addison-Wesley, 2000, 491p.
- Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2001, 270p.
- Alan W. Brown (ed.), Component-Based Software Engineering, IEEE Computer Society, Los Alamitos, CA, 1996, pp.140.
- Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard, Object-Oriented Software Engineering-A Use Case Driven Approach, Wokingham, England, Addison-Wesley, 1992, 582p.

# Recommended Reading

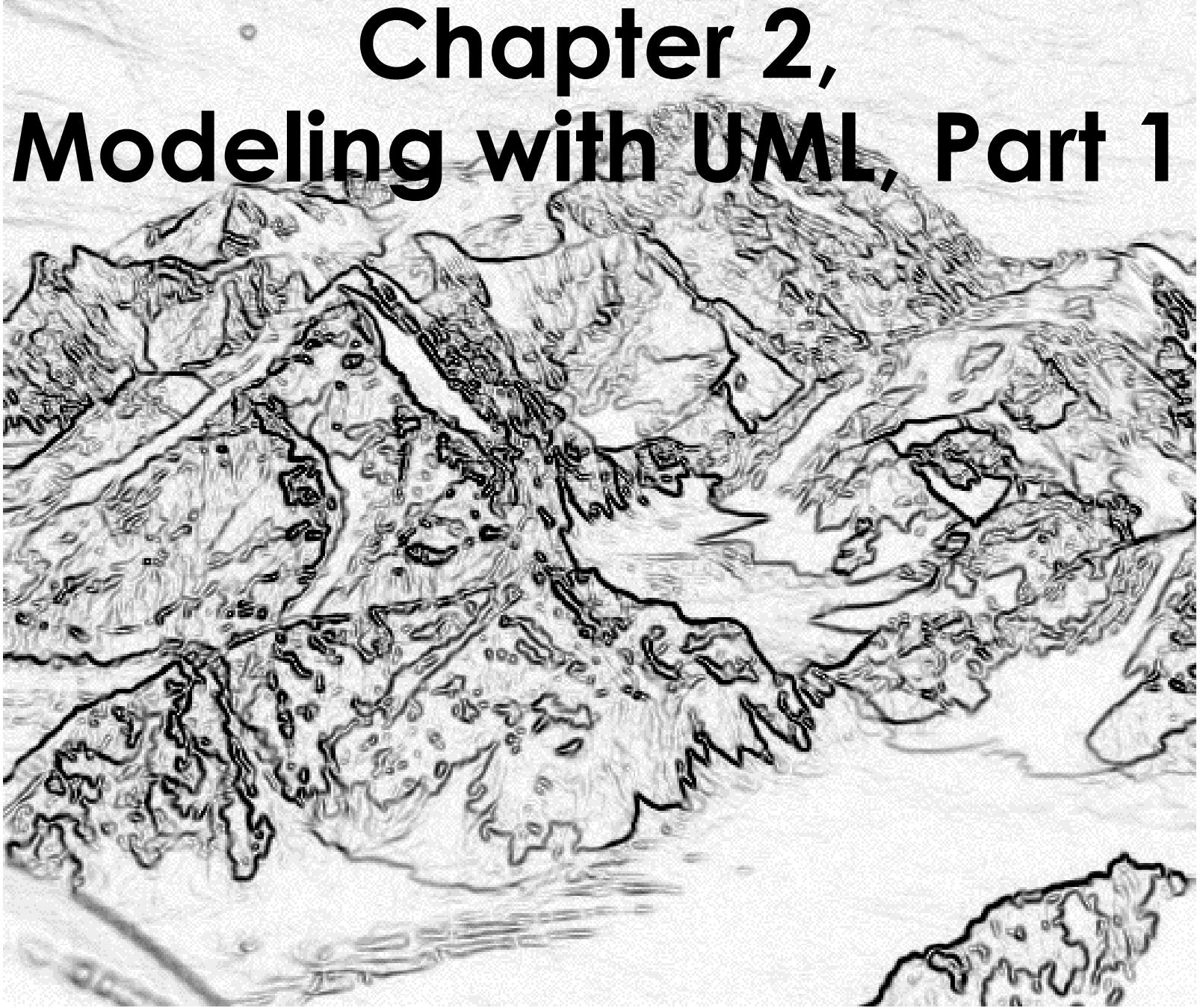
---

Applying UML and Patterns: An Introduction to OOA/D and the Unified Process, Prentice Hall, 2002, by G. Larman

The Rational Unified Process - An Introduction, Addison-Wesley Professional, 2002, by its lead architect Ph. Kruchten

# Object-Oriented Software Engineering

Using UML, Patterns, and Java



## Chapter 2, Modeling with UML, Part 1

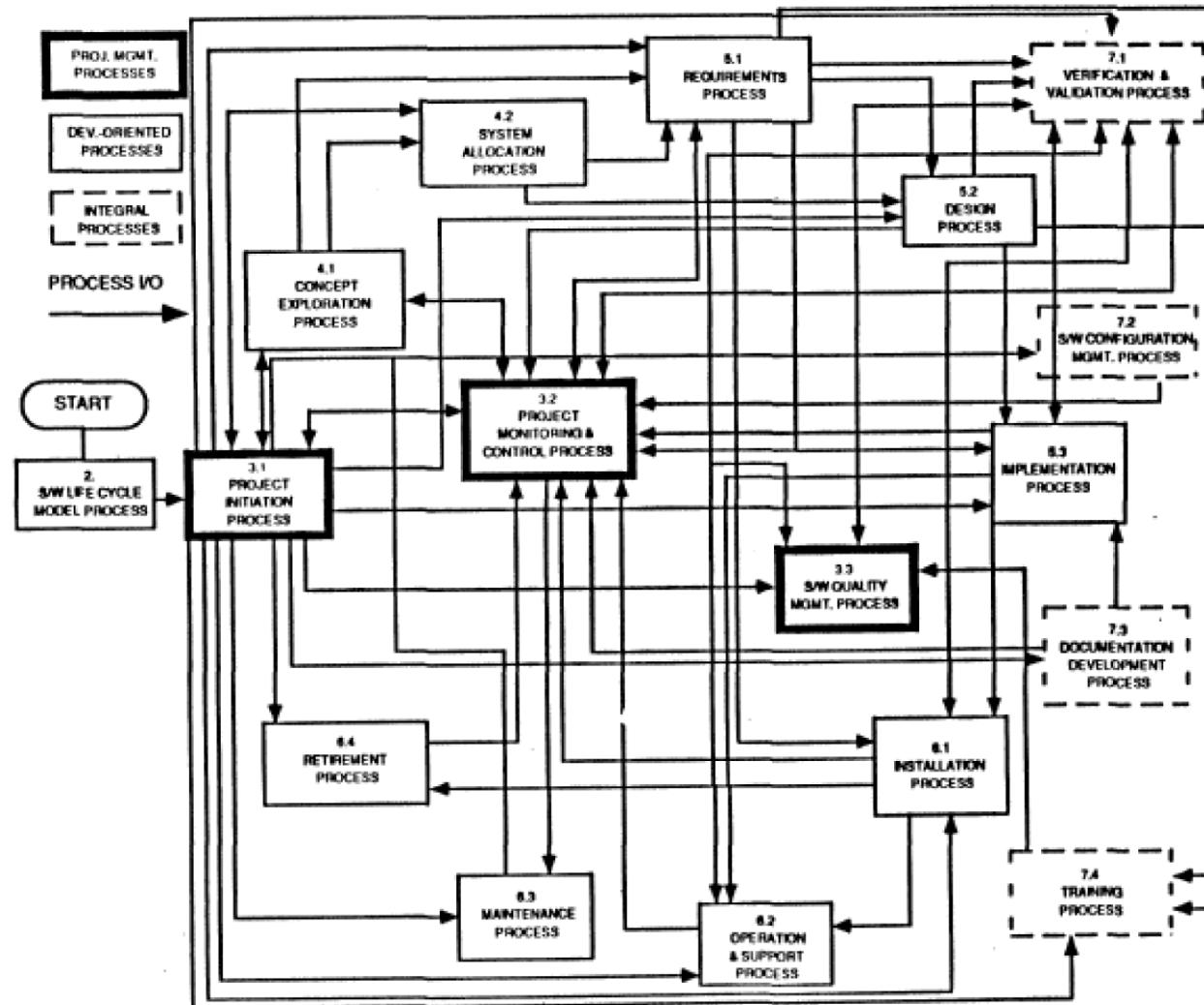
# Odds and Ends (2)

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

# Overview for the Lecture

- *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

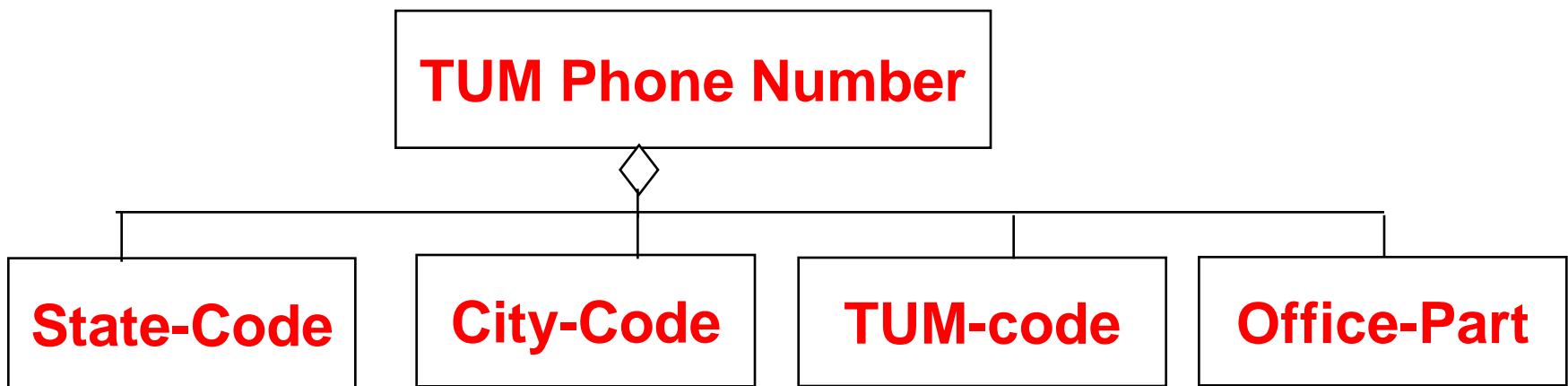
- *Complex systems are hard to understand*
  - *The 7 +- 2 phenomena*
    - *Our short term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain*
    - *TUM Phone Number: 498928918204*

# 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*
    - *TUM Phone Number: 498928918204*
- *Chunking:*
  - *Group collection of objects to reduce complexity*
  - *4 chunks:*
    - *State-code, city-code, TUM-code, Office-Part*

# 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*
    - *TUM Phone Number: 498928918204*
- *Chunking:*
  - *Group collection of objects to reduce complexity*
  - *State-code, city-code, TUM-code, Office-Part*



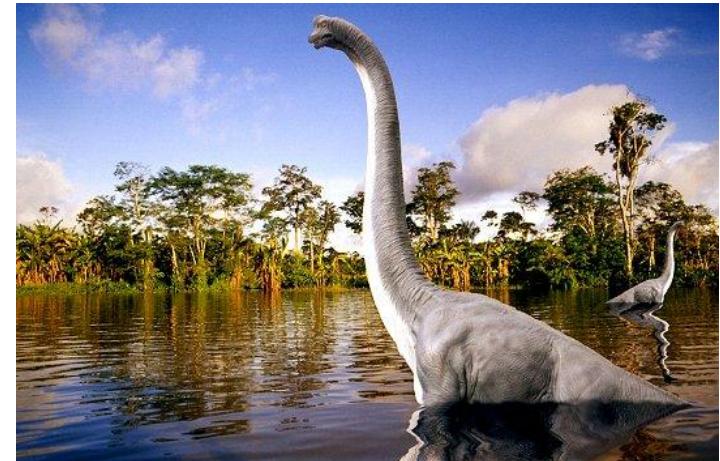
# Abstraction

- *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 as entity**
  - *Abstraction is the resulting idea of a thought process where an idea has been distanced from an object*
- *Ideas can be expressed by models*



# Model

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



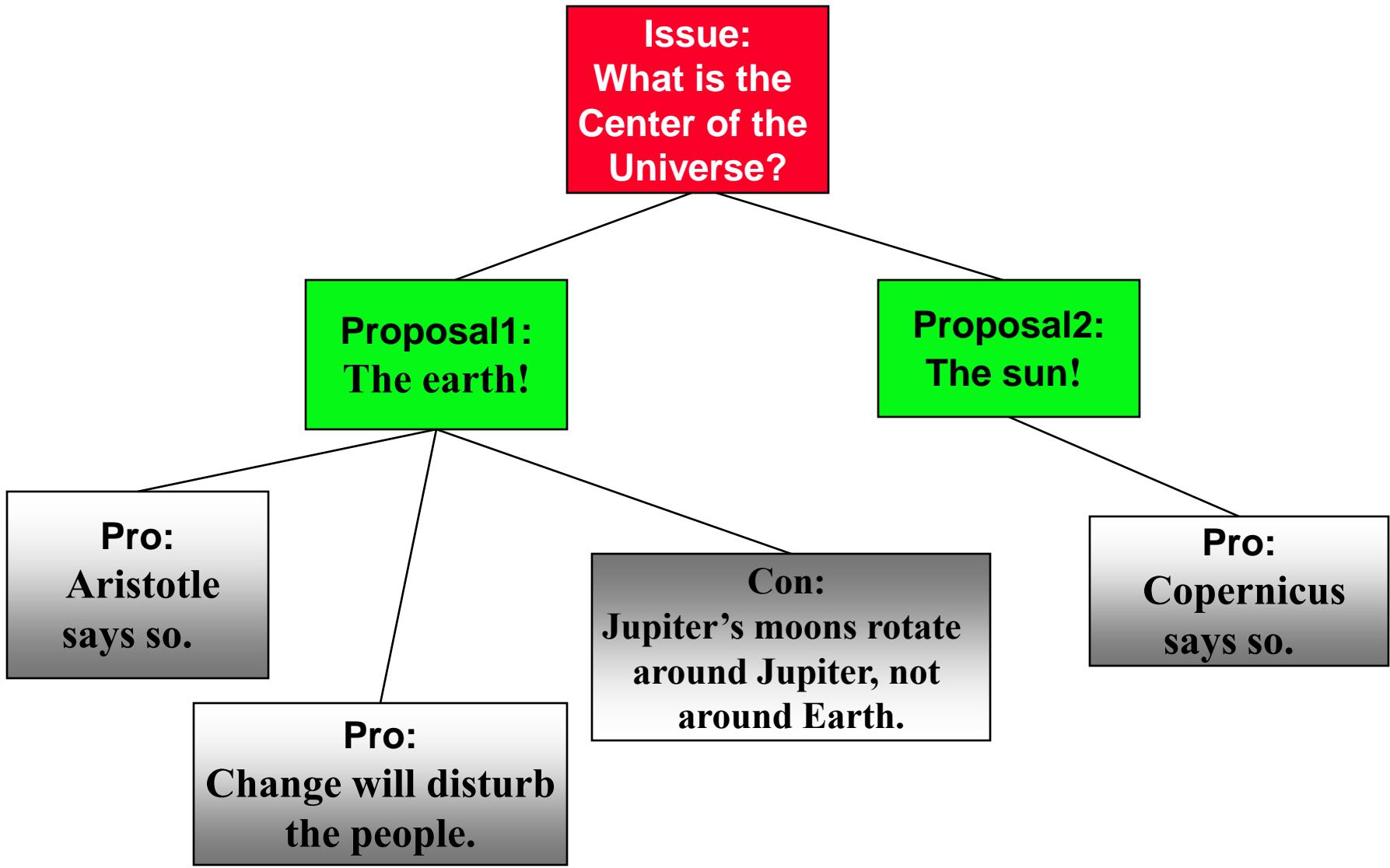
# We use Models to describe Software Systems

- *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

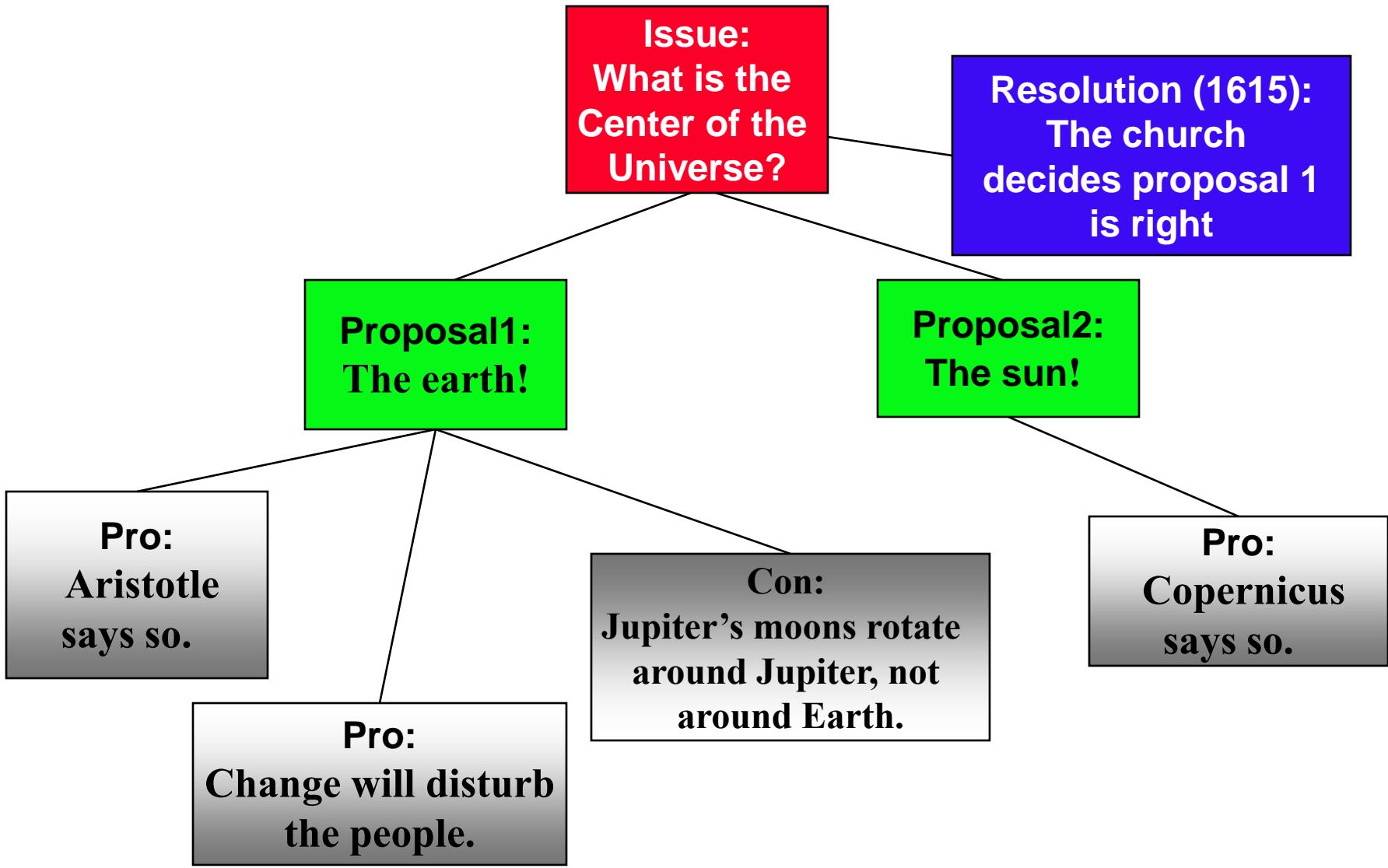
# Other models used to describe Software System Development

- *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

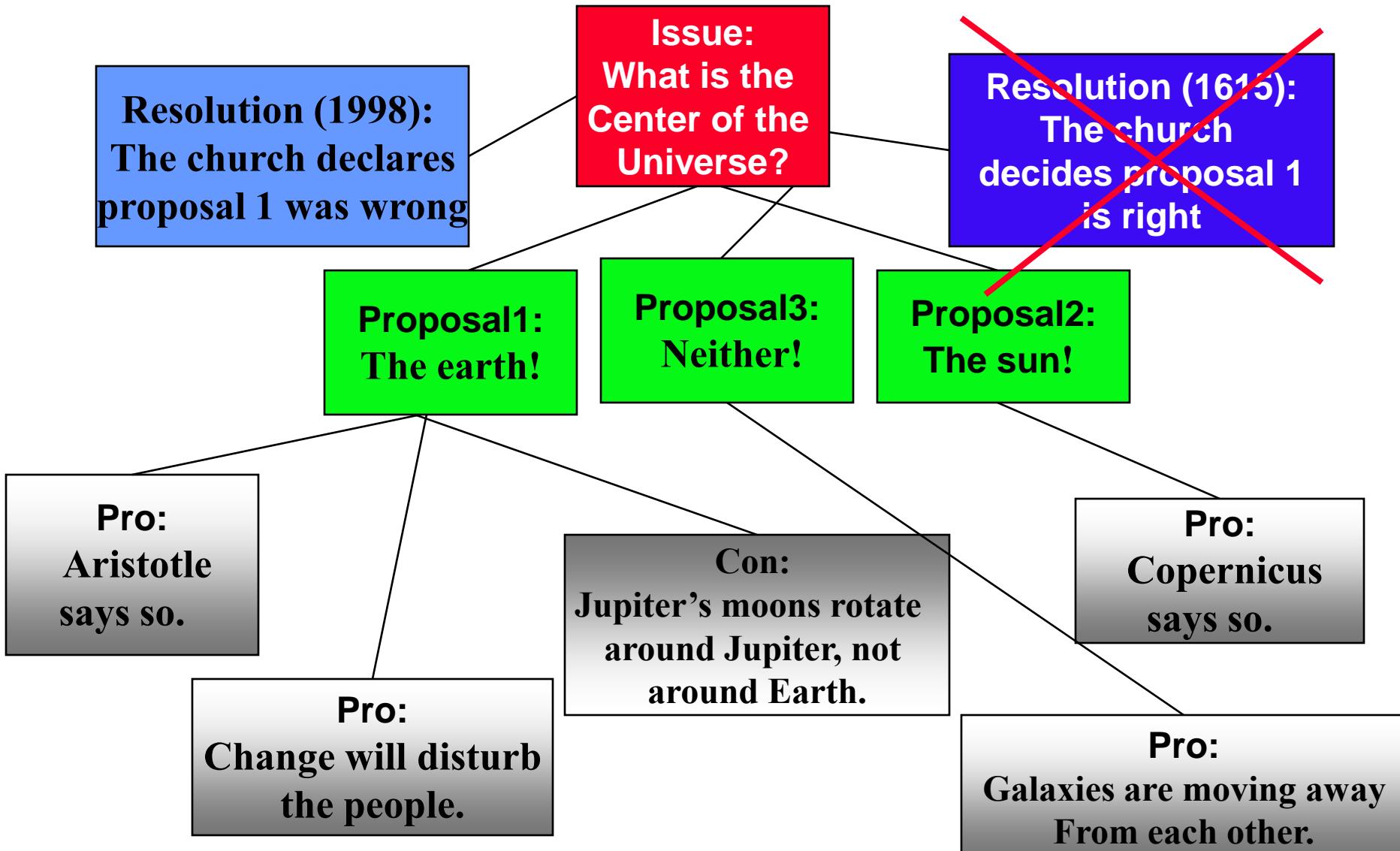
# Issue-Modeling



# Issue-Modeling



# Issue-Modeling



## 2. Technique to deal with Complexity: Decomposition

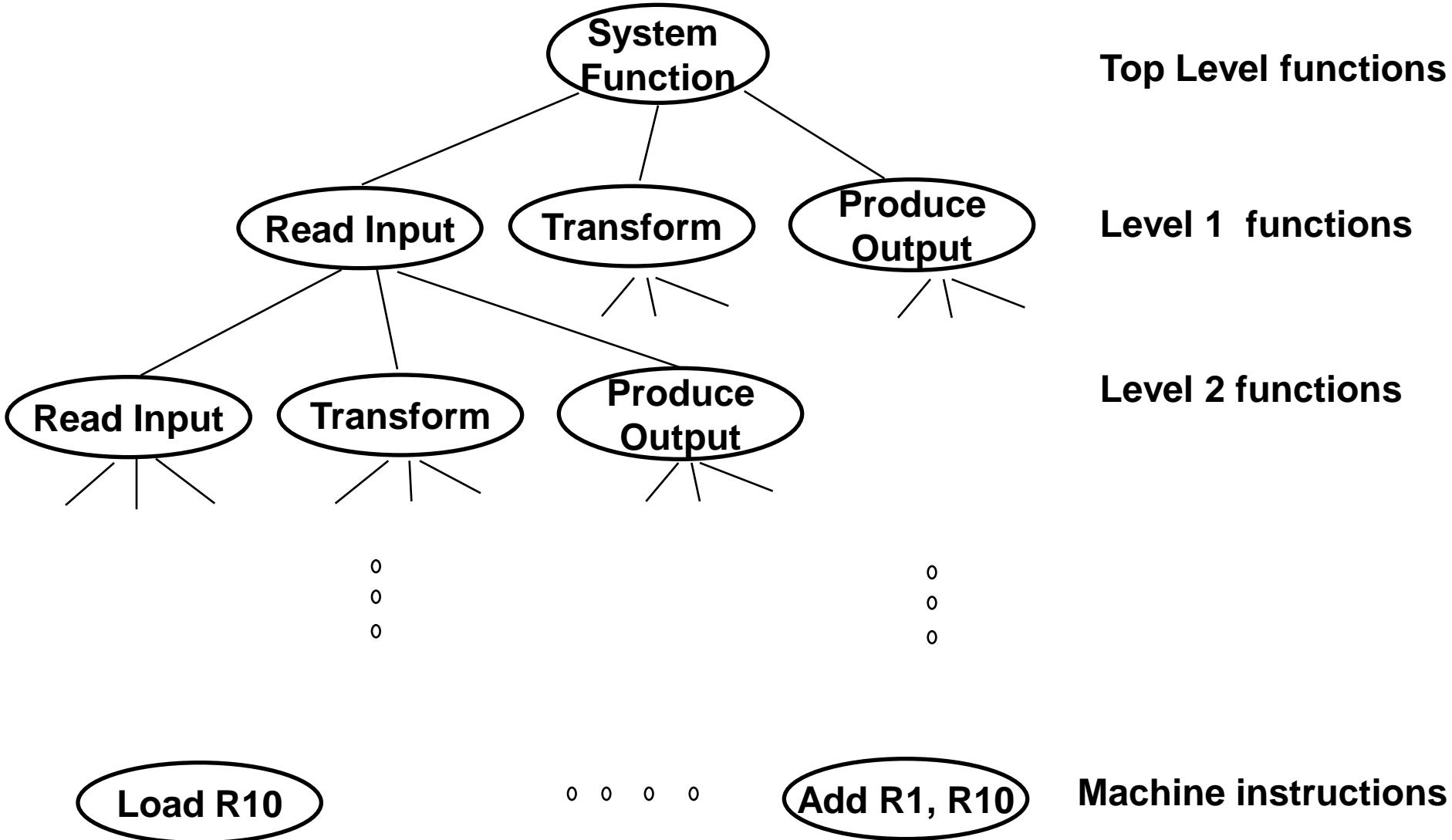
- A *technique used to master complexity ("divide and conquer")*
- Two *major types of decomposition*
  - *Functional decomposition*
  - *Object-oriented decomposition*
- *Functional decomposition*
  - *The system is decomposed into modules*
  - *Each module is a major function in the application domain*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

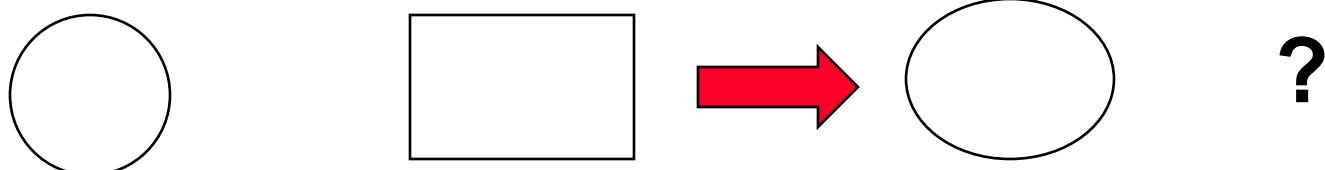


# Functional Decomposition

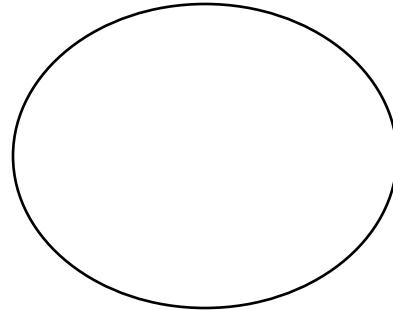
- *The functionality is spread all over the system*
- *Maintainer must understand the whole system to make a single change to the system*
- *Consequence:*
  - *Source code is hard to understand*
  - *Source code is complex and impossible to maintain*
  - *User interface is often awkward and non-intuitive.*

# Functional Decomposition

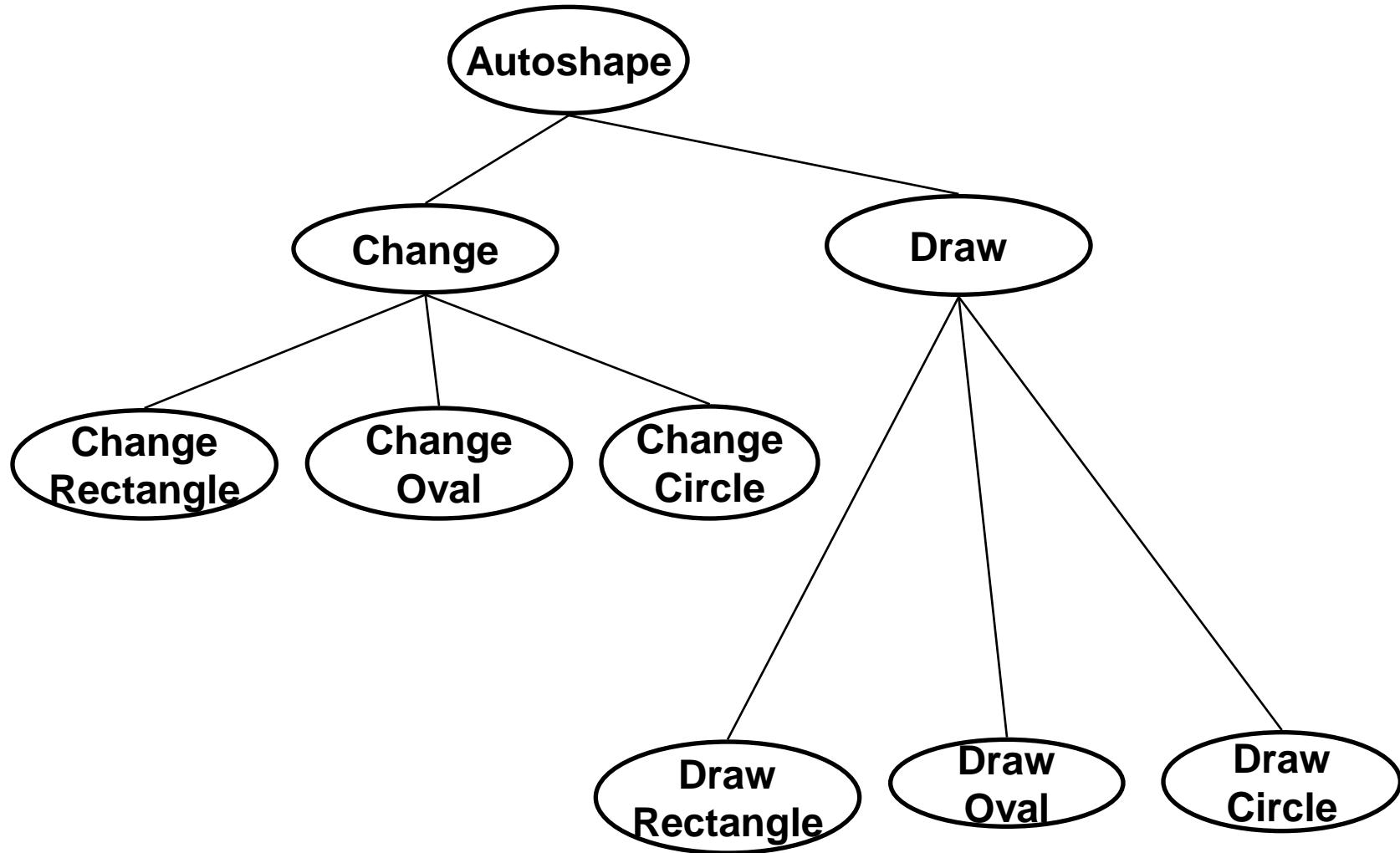
- *The functionality is spread all over the system*
- *Maintainer must understand the whole system to make a single change to the system*
- *Consequence:*
  - *Source code is hard to understand*
  - *Source code is complex and impossible to maintain*
  - *User interface is often awkward and non-intuitive*
- *Example: Microsoft Powerpoint's Autoshapes*
  - *How do I change a square into a circle?*



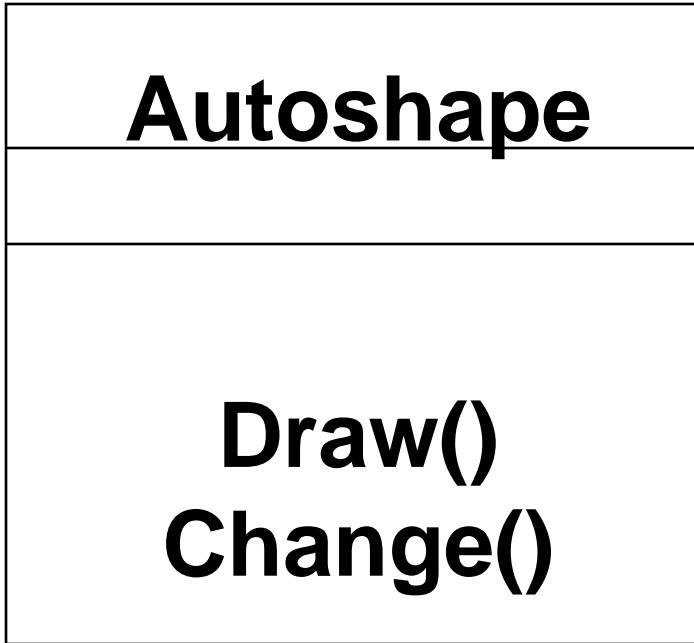
# Changing a Square into a Circle



# Functional Decomposition: Autoshape

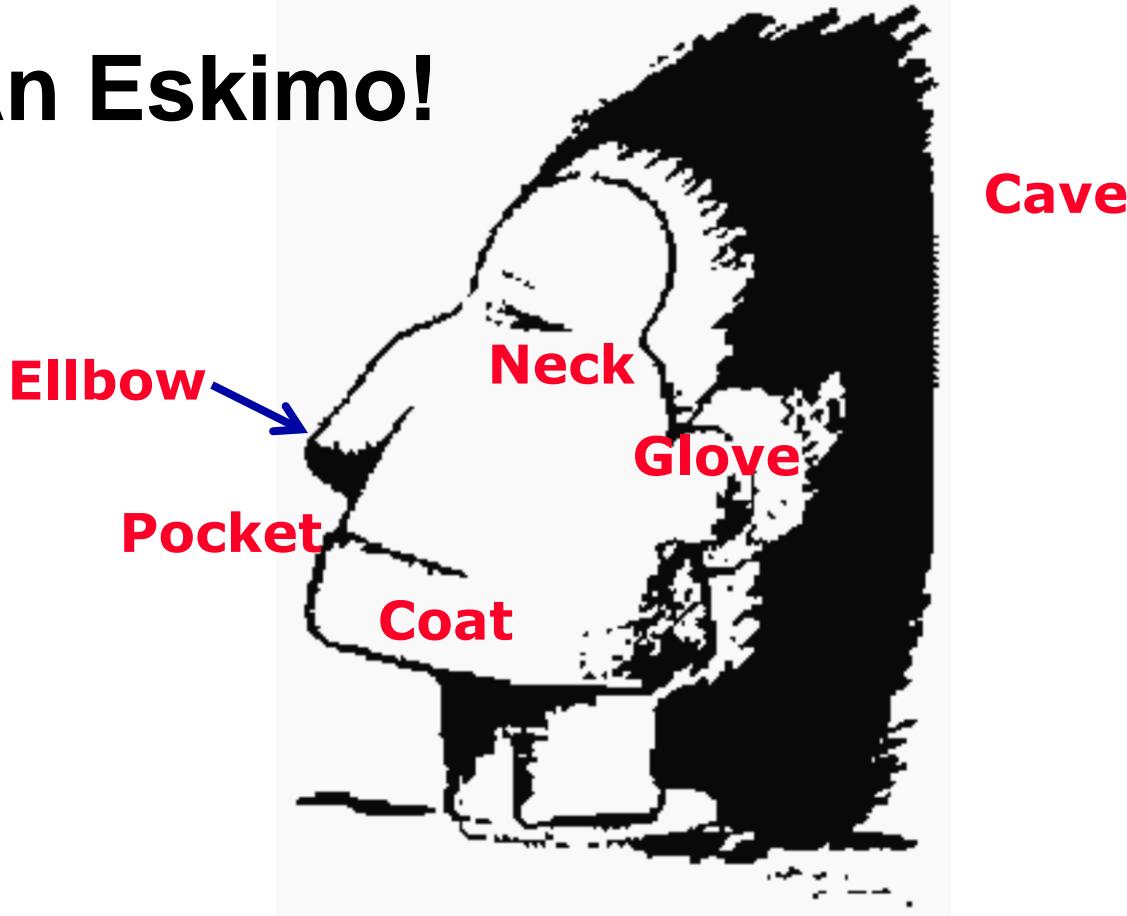


# Object-Oriented View

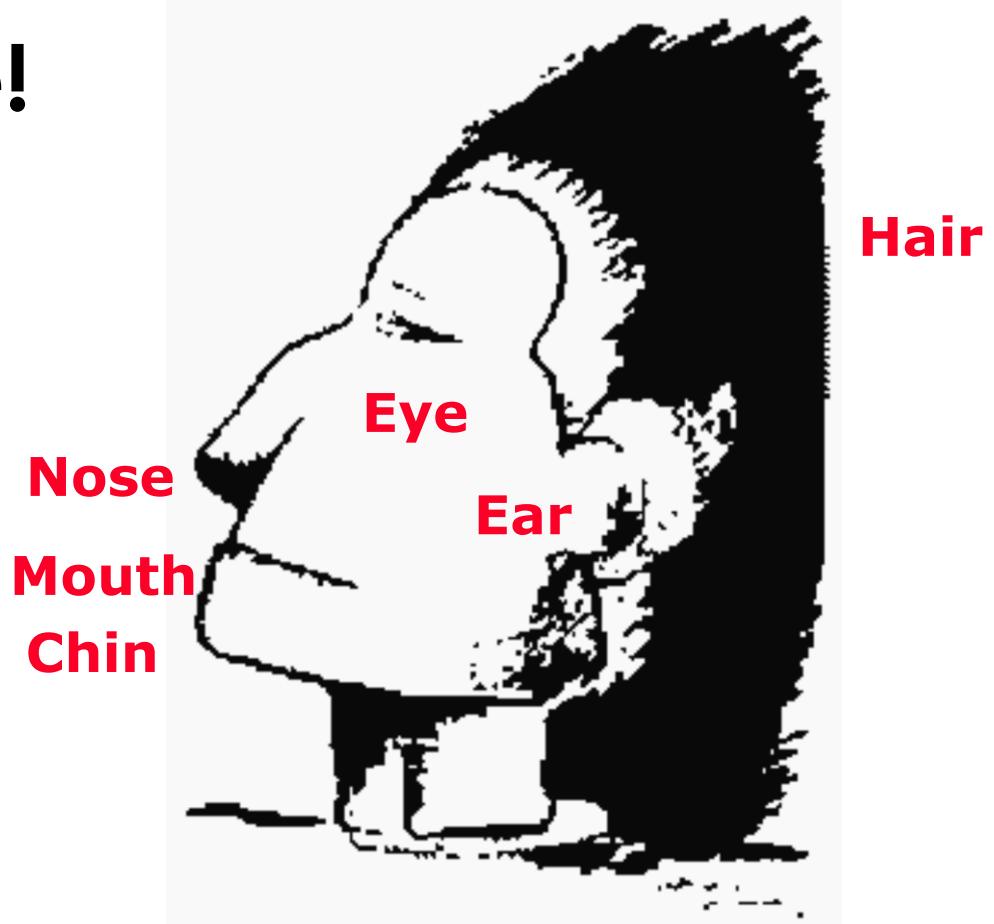


# What is This?

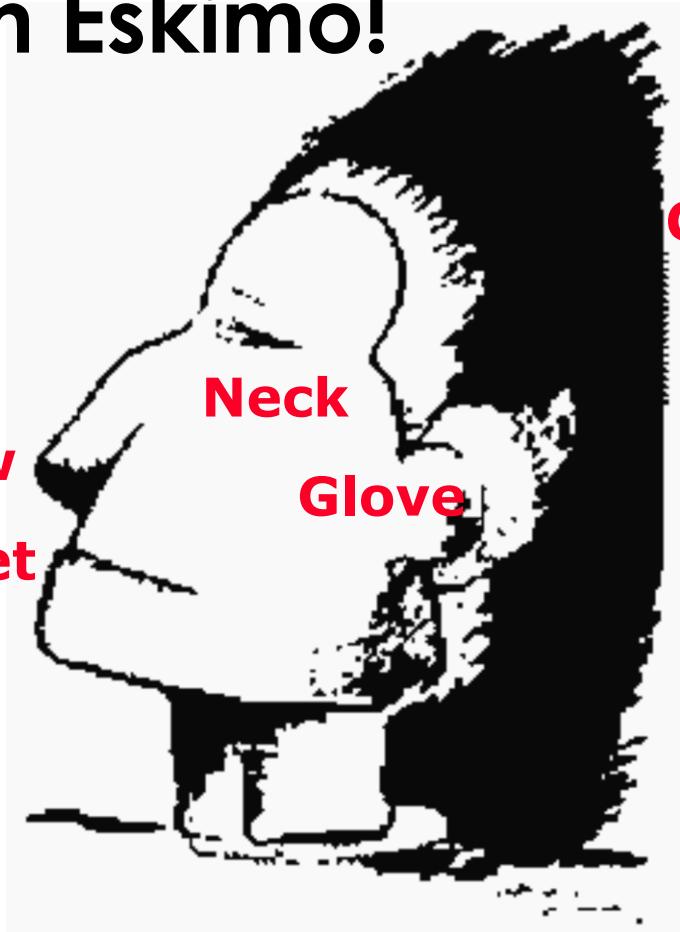
An Eskimo!



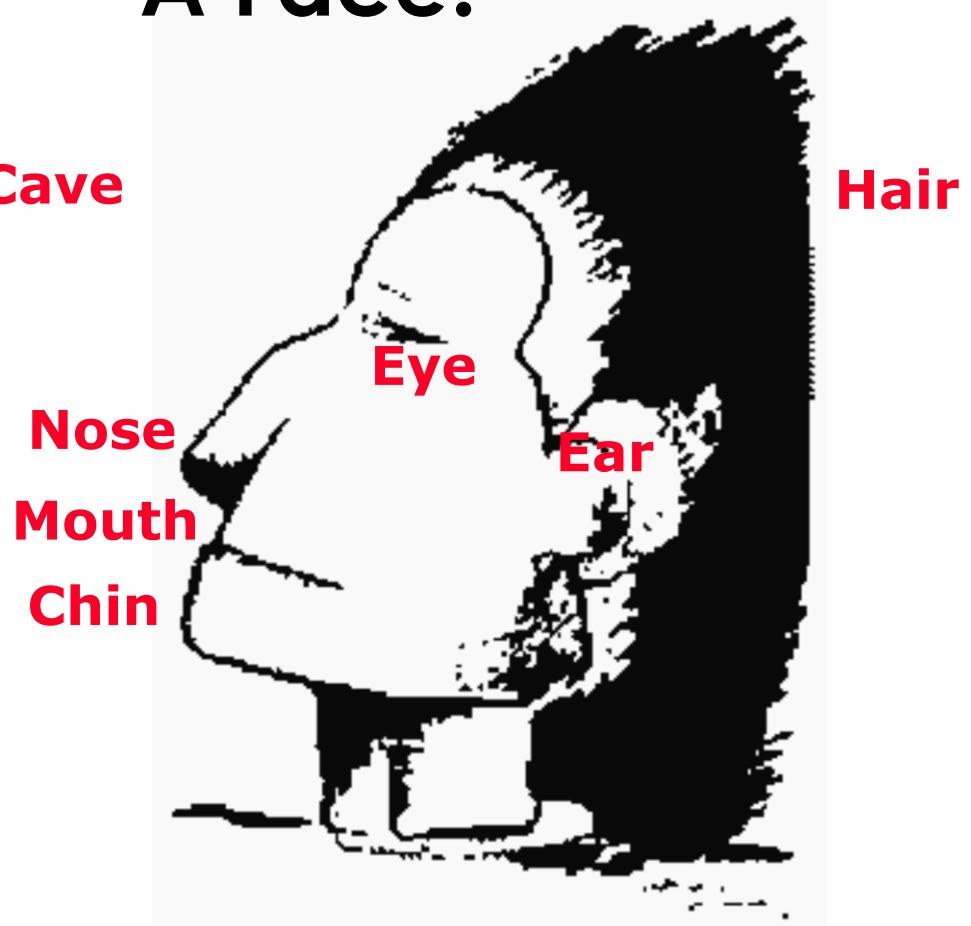
# A Face!



# 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)

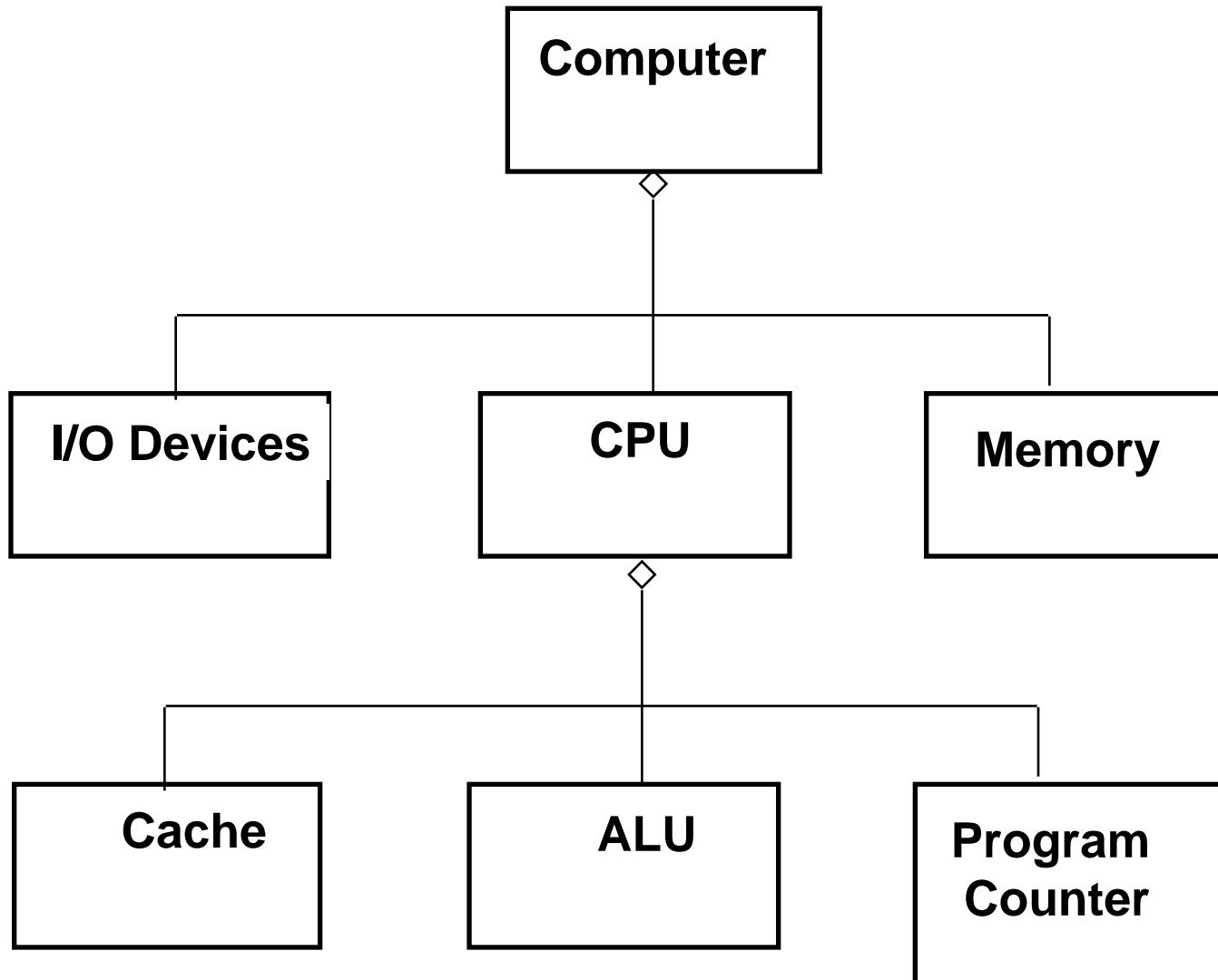
- **Why can we do this?**
  - *Philosophy, science, experimental evidence*
- **What are the limitations?**
  - *Depending on the purpose of the system, different objects might be found*
- **Crucial**  
*Identify the purpose of a system*



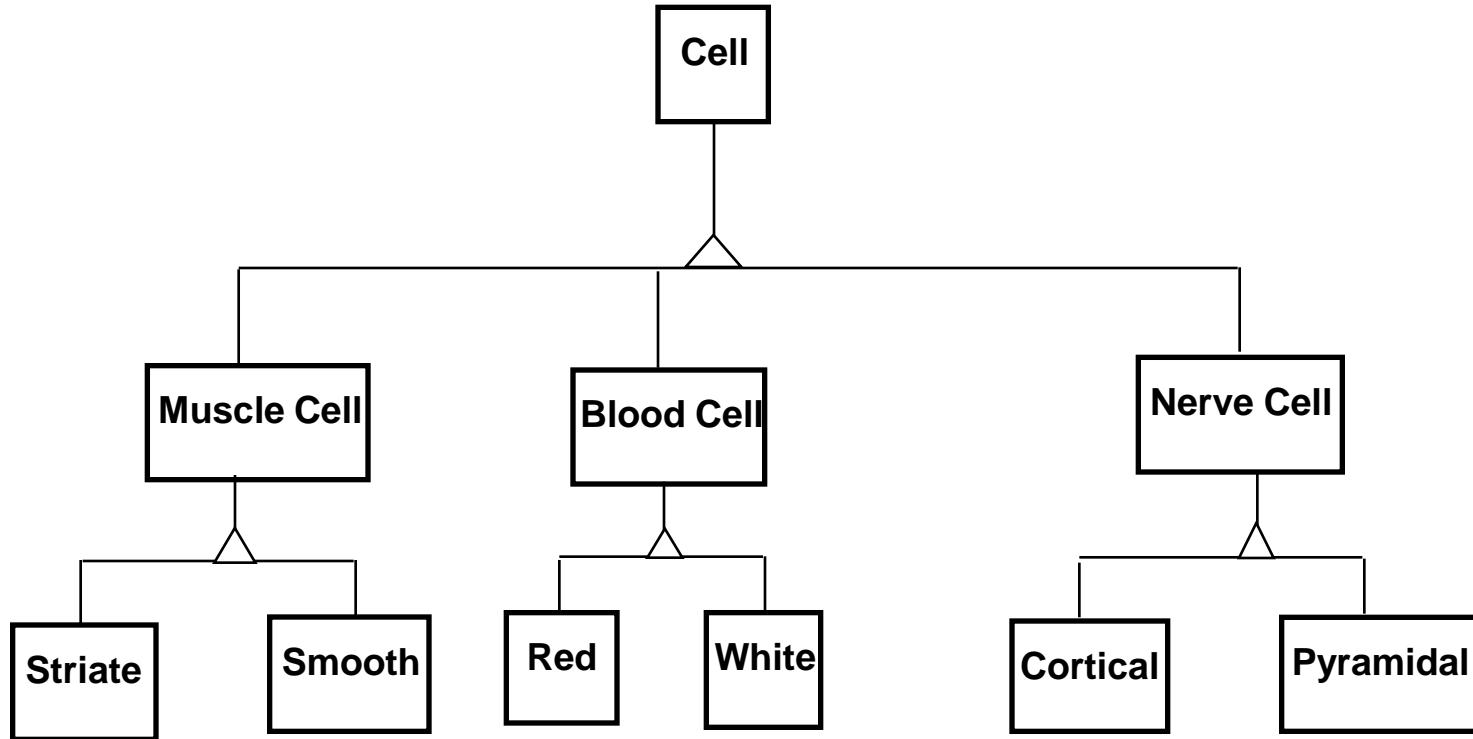
# 3. Hierarchy

- *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)



# 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*

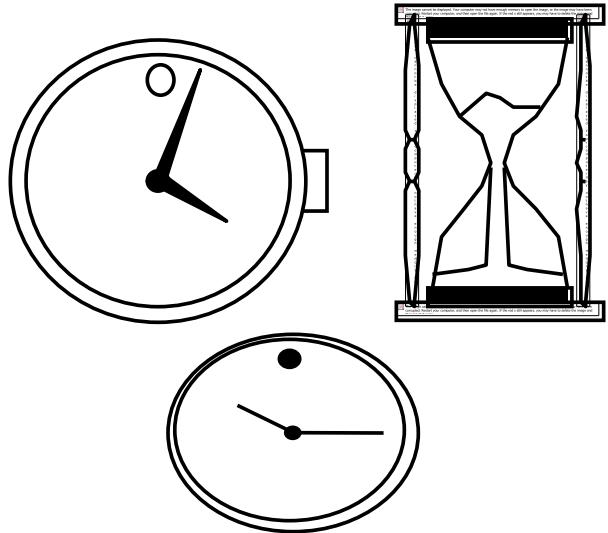
# Models must be falsifiable

- *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.

# 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
Watch	A device that measures time.	

## *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*

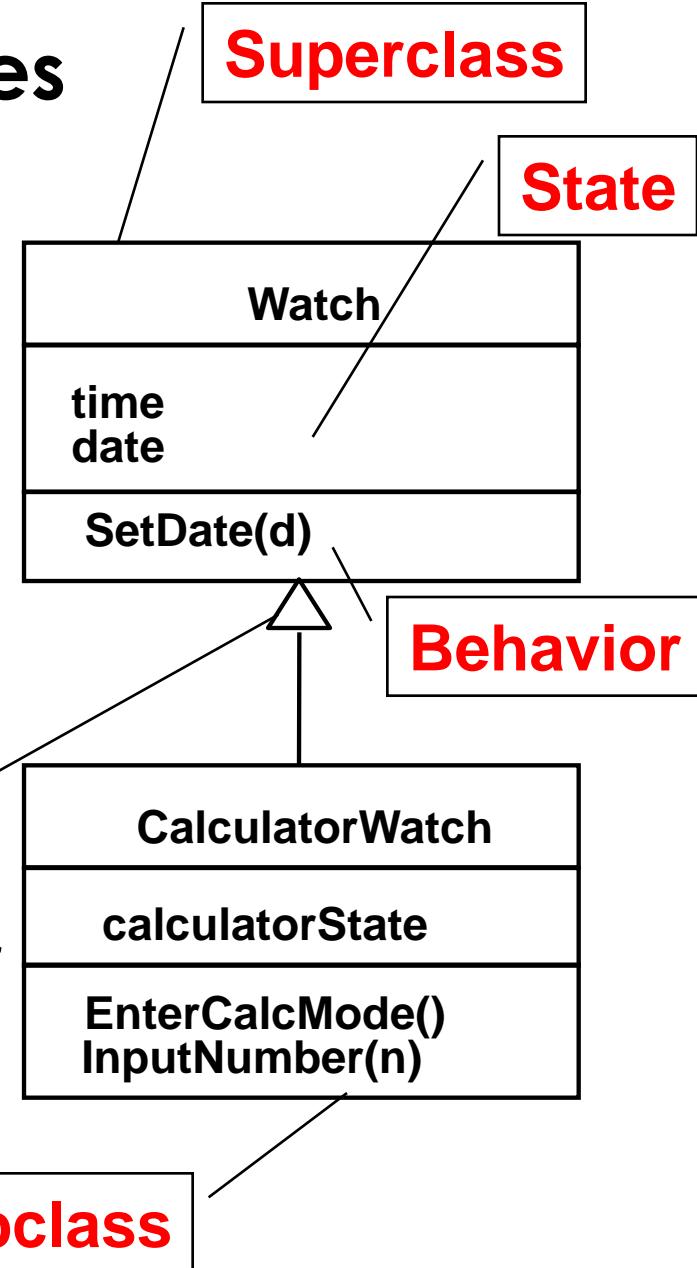
- **Class:**

- *An abstraction in the context of object-oriented languages*
- *A class encapsulates state and behavior*
  - *Example: Watch*

**Inheritance**

*Unlike abstract data types, 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:*
  - Member of a specific type
- *The type of a variable represents all possible instances of the variable*

*The following relationships are similar:*

*Type      <->   Variable*

*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.

# 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:

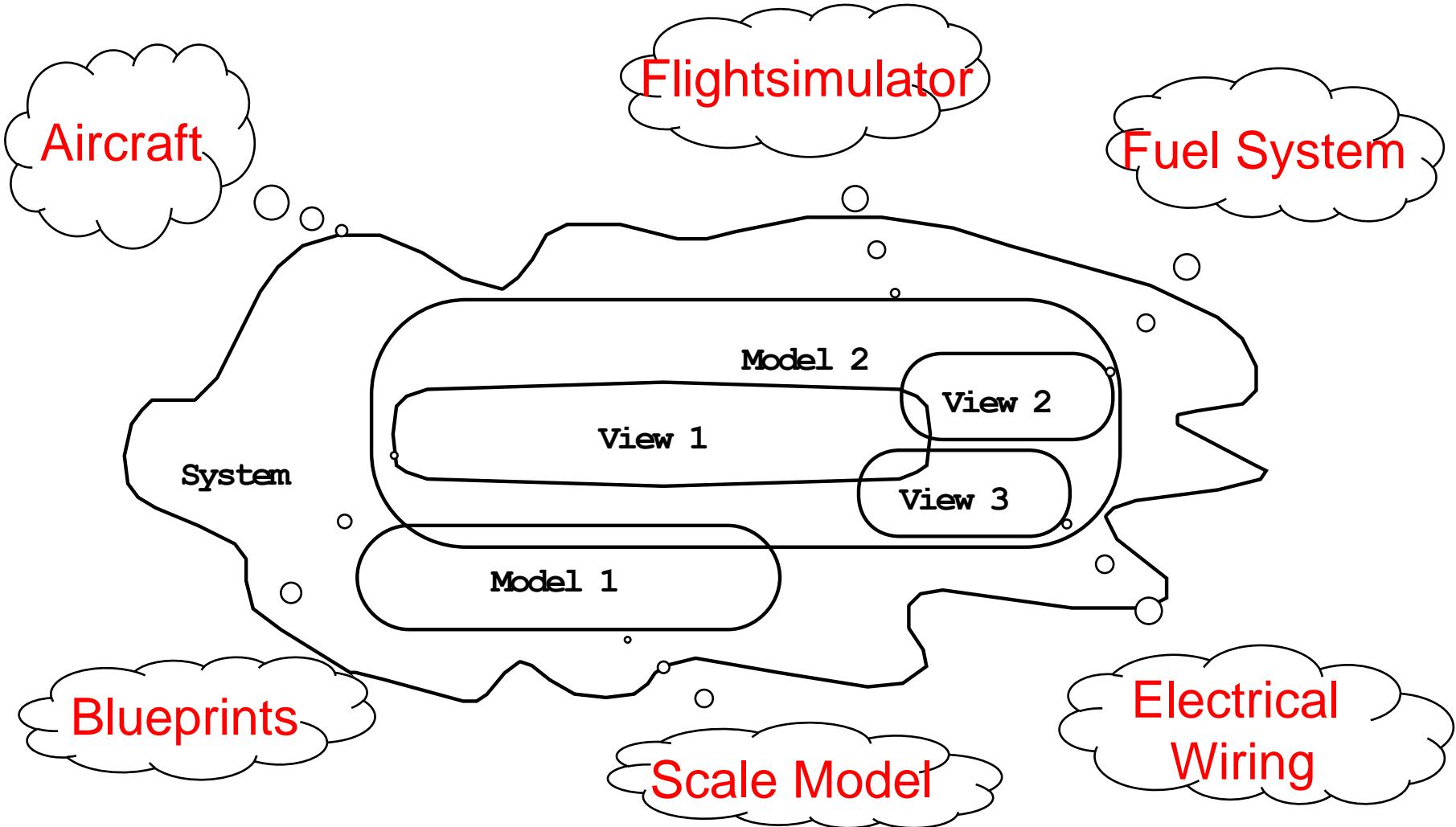
*Flight simulator  
Scale model*

### Views:

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



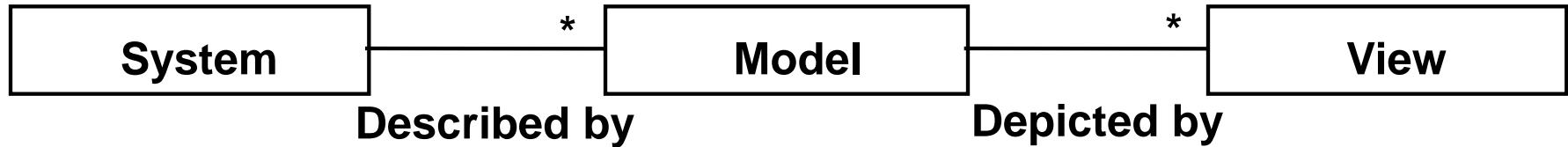
# Systems, Models and Views (“Napkin” Notation)



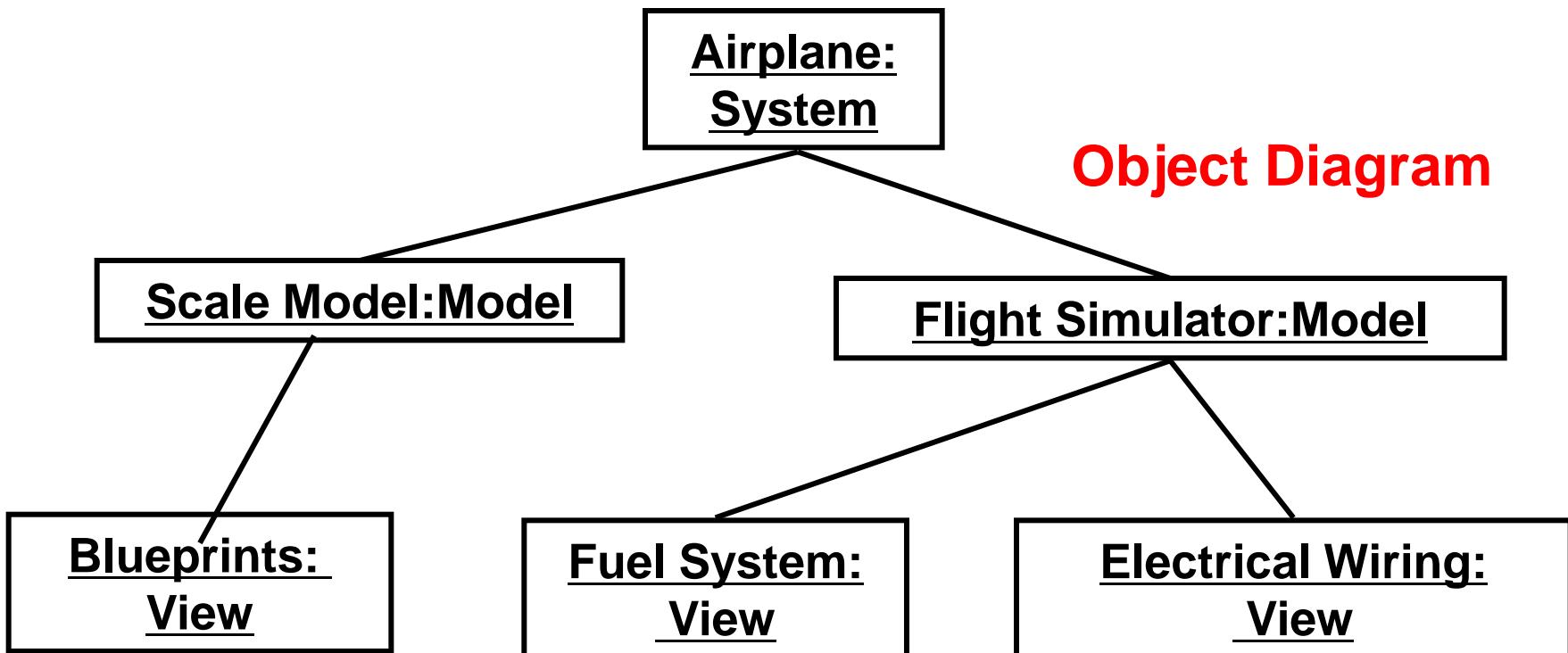
*Views and models of a complex system usually overlap*

# Systems, Models and Views (UML Notation)

## Class Diagram



## Object Diagram



# 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();  
};
```

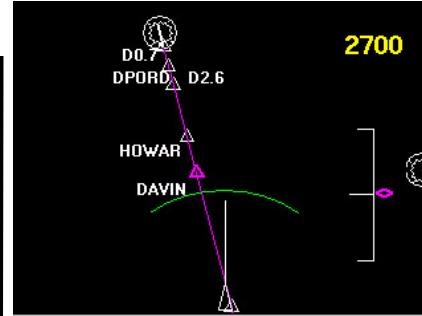
# Application vs Solution Domain

- *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



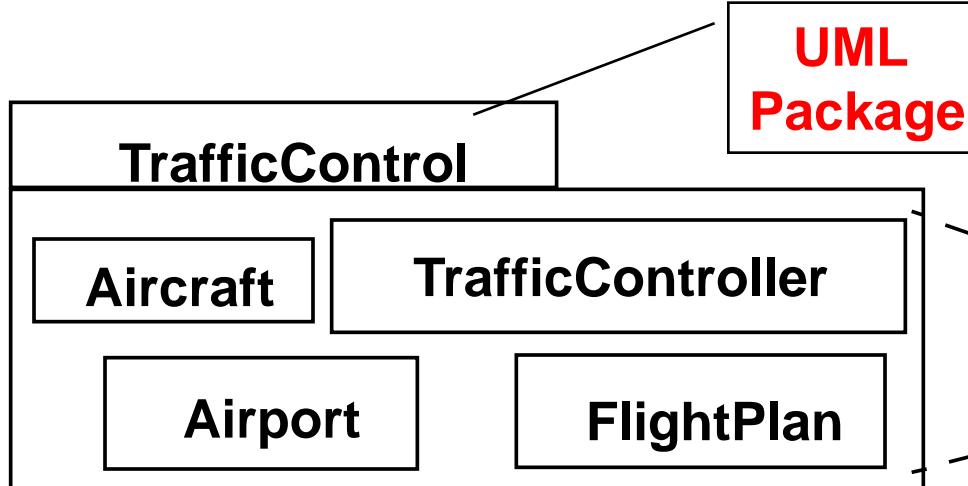
**Application Domain  
(Phenomena)**



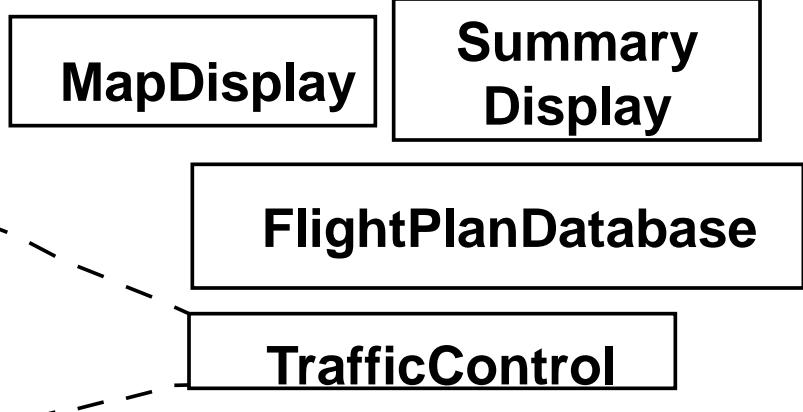
SP/PCB	Processor	Mem	SSD	Power	Size (W)	SP/PCB	Processor	Mem	SSD	Power	Size (W)
1	i7-11700K	32GB	1TB	300W	120	1	i7-11700K	32GB	1TB	300W	120
2	i7-11700K	32GB	1TB	300W	120	2	i7-11700K	32GB	1TB	300W	120
3	i7-11700K	32GB	1TB	300W	120	3	i7-11700K	32GB	1TB	300W	120
4	i7-11700K	32GB	1TB	300W	120	4	i7-11700K	32GB	1TB	300W	120
5	i7-11700K	32GB	1TB	300W	120	5	i7-11700K	32GB	1TB	300W	120
6	i7-11700K	32GB	1TB	300W	120	6	i7-11700K	32GB	1TB	300W	120
7	i7-11700K	32GB	1TB	300W	120	7	i7-11700K	32GB	1TB	300W	120
8	i7-11700K	32GB	1TB	300W	120	8	i7-11700K	32GB	1TB	300W	120
9	i7-11700K	32GB	1TB	300W	120	9	i7-11700K	32GB	1TB	300W	120
10	i7-11700K	32GB	1TB	300W	120	10	i7-11700K	32GB	1TB	300W	120
11	i7-11700K	32GB	1TB	300W	120	11	i7-11700K	32GB	1TB	300W	120
12	i7-11700K	32GB	1TB	300W	120	12	i7-11700K	32GB	1TB	300W	120
13	i7-11700K	32GB	1TB	300W	120	13	i7-11700K	32GB	1TB	300W	120
14	i7-11700K	32GB	1TB	300W	120	14	i7-11700K	32GB	1TB	300W	120
15	i7-11700K	32GB	1TB	300W	120	15	i7-11700K	32GB	1TB	300W	120
16	i7-11700K	32GB	1TB	300W	120	16	i7-11700K	32GB	1TB	300W	120
17	i7-11700K	32GB	1TB	300W	120	18	i7-11700K	32GB	1TB	300W	120
19	i7-11700K	32GB	1TB	300W	120	20	i7-11700K	32GB	1TB	300W	120
21	i7-11700K	32GB	1TB	300W	120	22	i7-11700K	32GB	1TB	300W	120
23	i7-11700K	32GB	1TB	300W	120	24	i7-11700K	32GB	1TB	300W	120
25	i7-11700K	32GB	1TB	300W	120	26	i7-11700K	32GB	1TB	300W	120
27	i7-11700K	32GB	1TB	300W	120	28	i7-11700K	32GB	1TB	300W	120
29	i7-11700K	32GB	1TB	300W	120	30	i7-11700K	32GB	1TB	300W	120
31	i7-11700K	32GB	1TB	300W	120	32	i7-11700K	32GB	1TB	300W	120
33	i7-11700K	32GB	1TB	300W	120	34	i7-11700K	32GB	1TB	300W	120
35	i7-11700K	32GB	1TB	300W	120	36	i7-11700K	32GB	1TB	300W	120
37	i7-11700K	32GB	1TB	300W	120	38	i7-11700K	32GB	1TB	300W	120
39	i7-11700K	32GB	1TB	300W	120	40	i7-11700K	32GB	1TB	300W	120
41	i7-11700K	32GB	1TB	300W	120	42	i7-11700K	32GB	1TB	300W	120
43	i7-11700K	32GB	1TB	300W	120	44	i7-11700K	32GB	1TB	300W	120
45	i7-11700K	32GB	1TB	300W	120	46	i7-11700K	32GB	1TB	300W	120

**Solution Domain  
(Phenomena)**

**System Model (Concepts)*(Analysis)***



**System Model (Concepts)*(Design)***



# 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 colleagues)*
    - *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)*

# 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](http://www.ephorie.de/hindle_pareto-prinzip.htm)

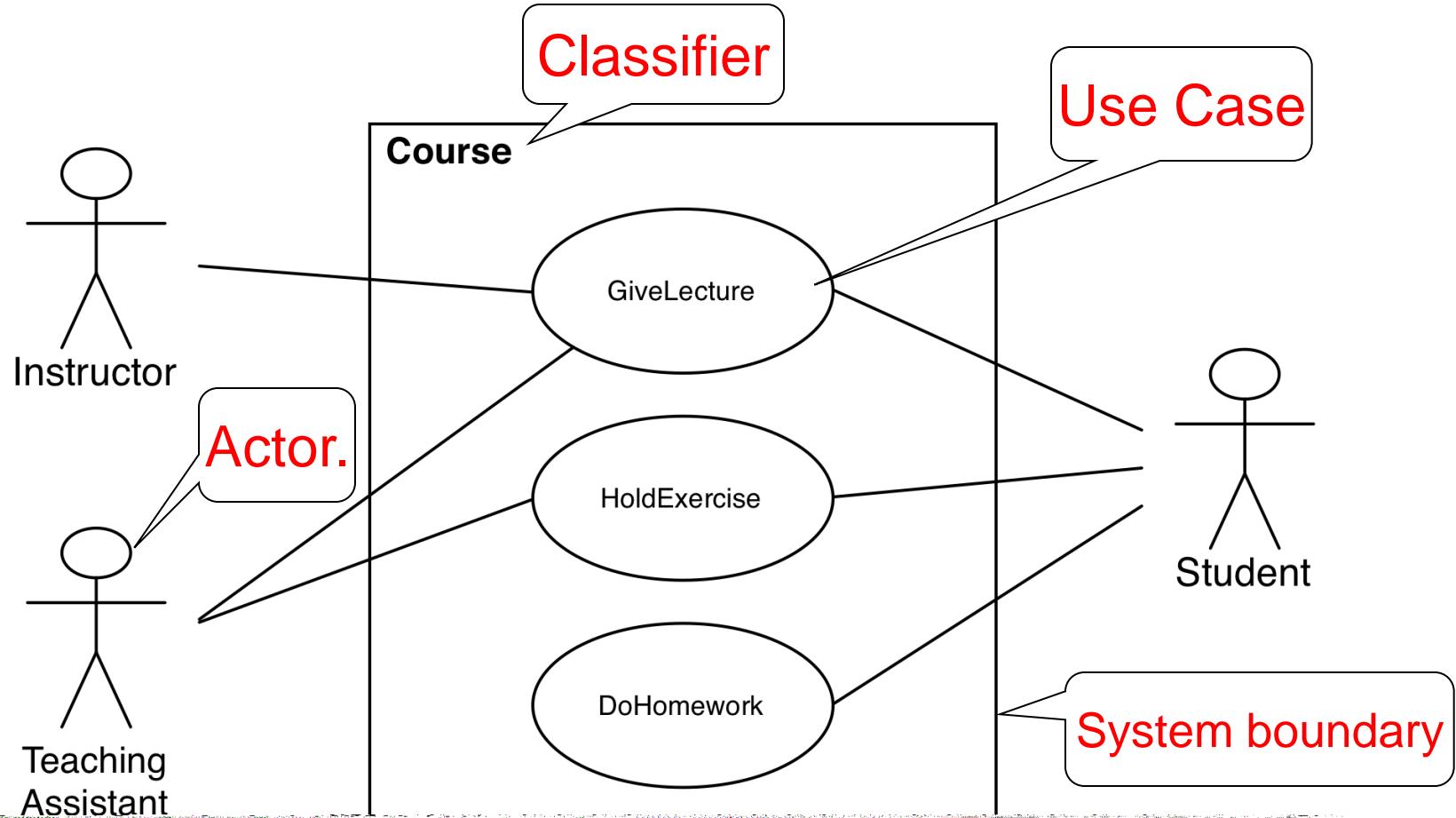
# UML First Pass

- *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

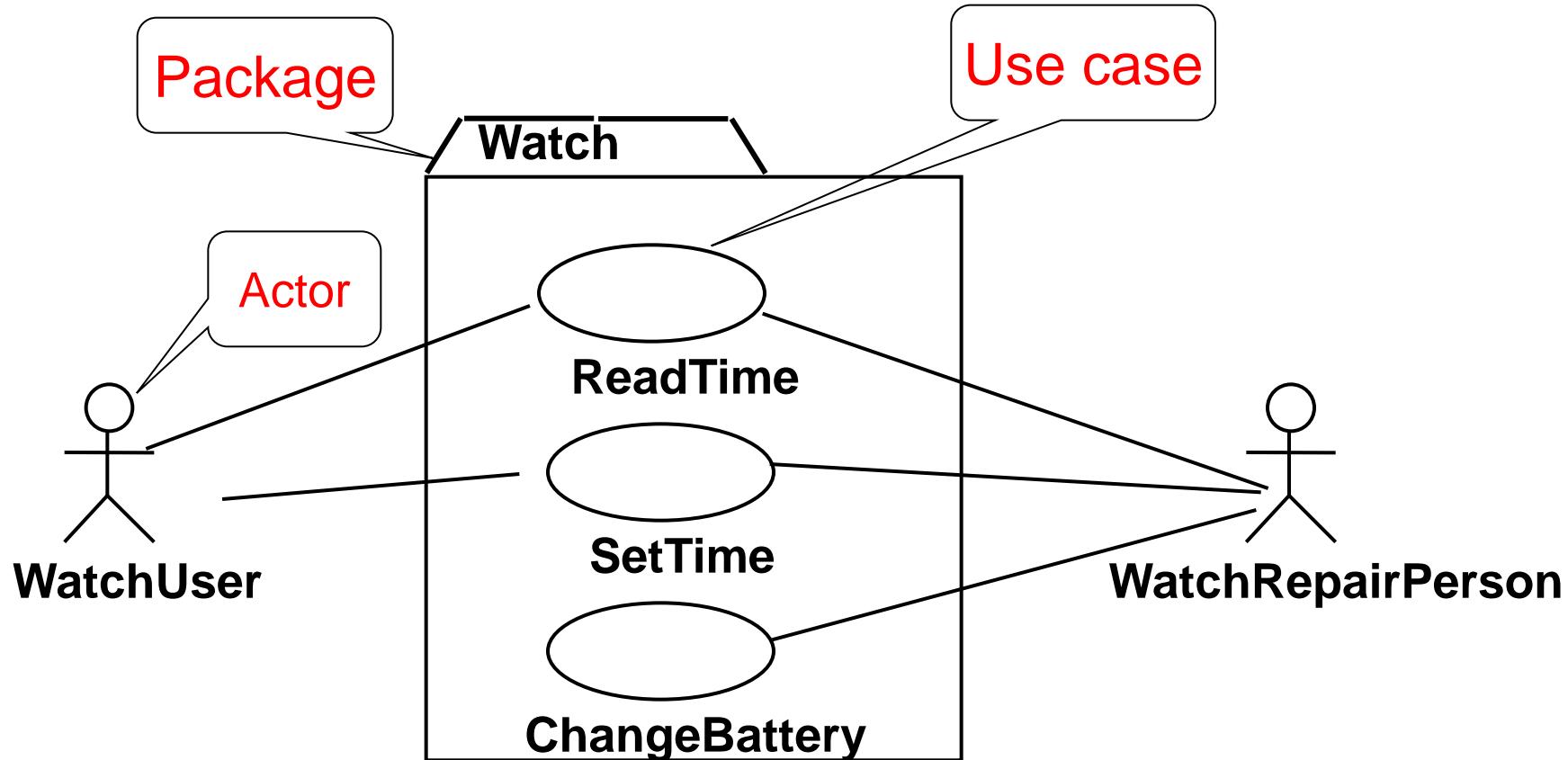
- All UML Diagrams denote graphs of nodes and edges
  - Nodes are entities and drawn as rectangles or ovals
  - Rectangles denote classes or instances
  - Ovals denote functions
- Names of Classes are not underlined
  - SimpleWatch
  - Firefighter
- Names of Instances are underlined
  - myWatch:SimpleWatch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

# UML first pass: Use case diagrams



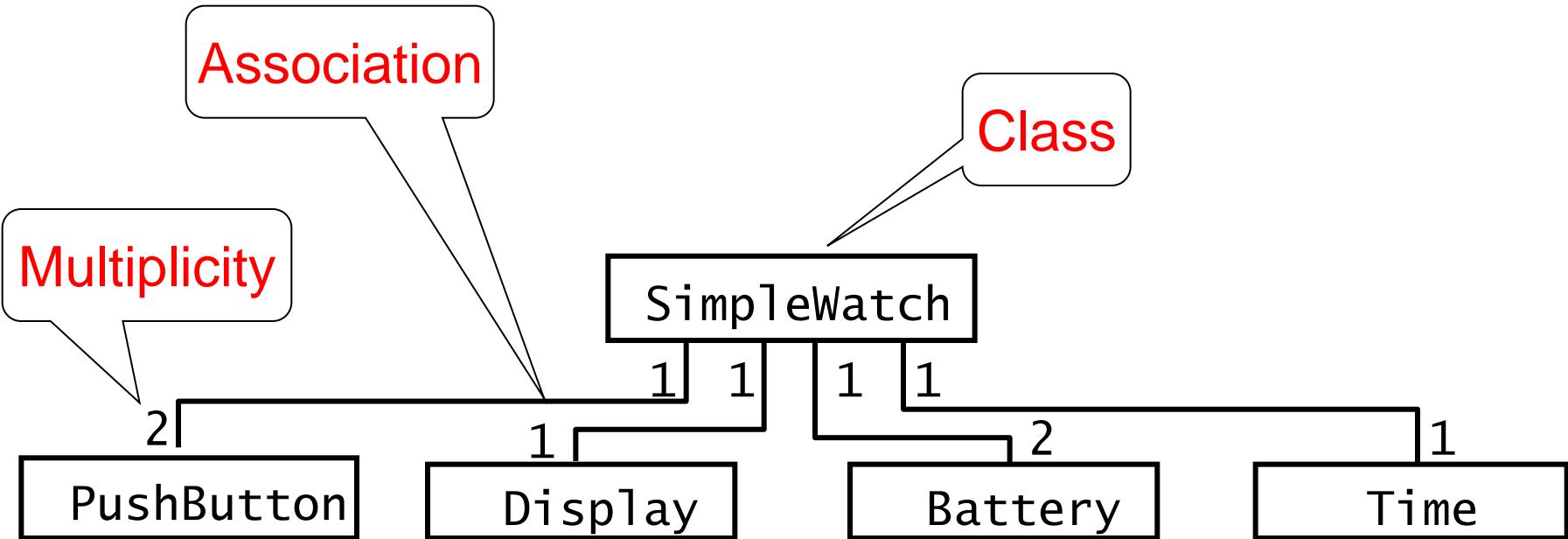
Use case diagrams represent the functionality of the system from user's point of view

# Historical Remark: UML 1 used packages



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

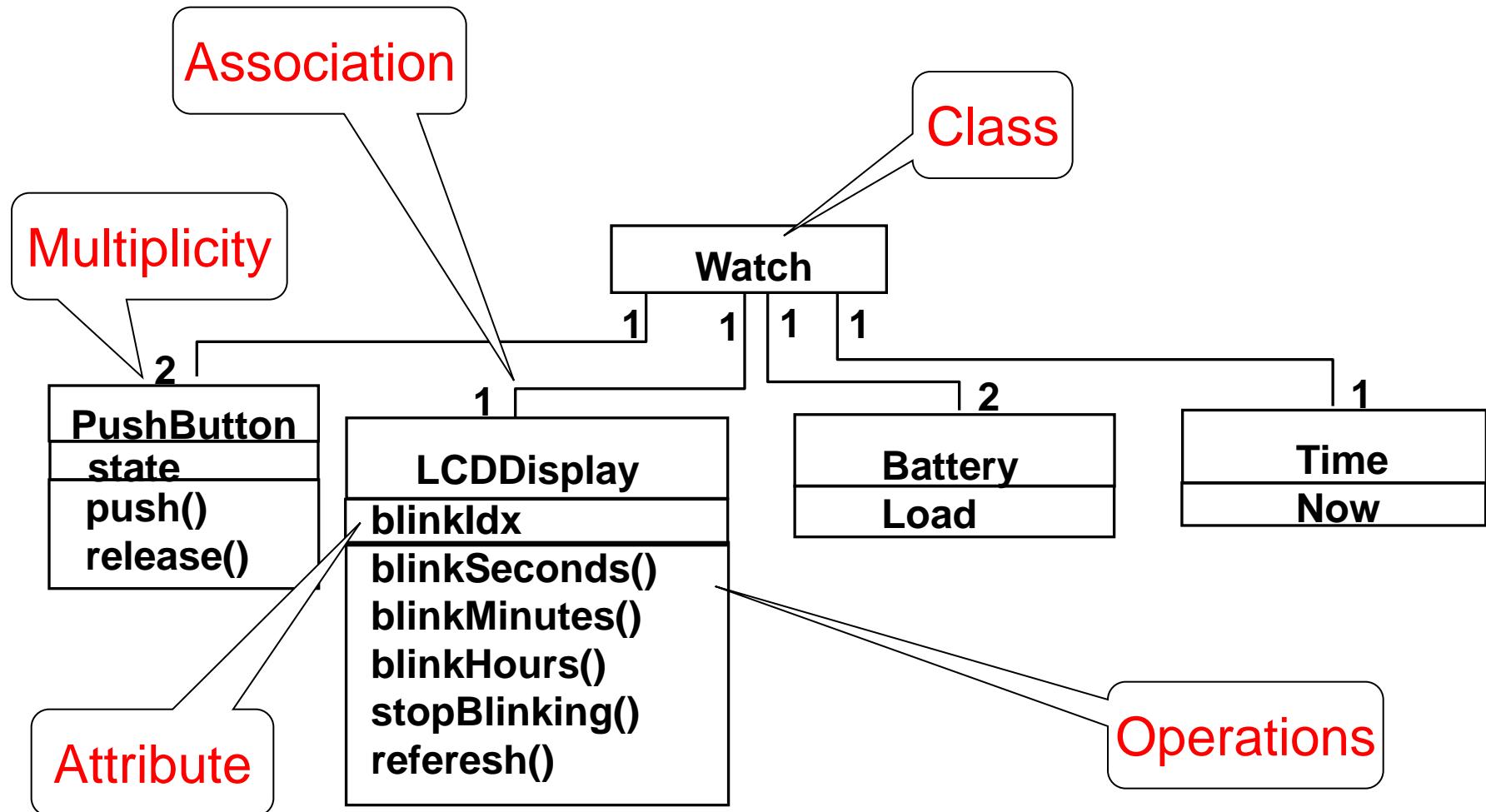
# UML first pass: Class diagrams



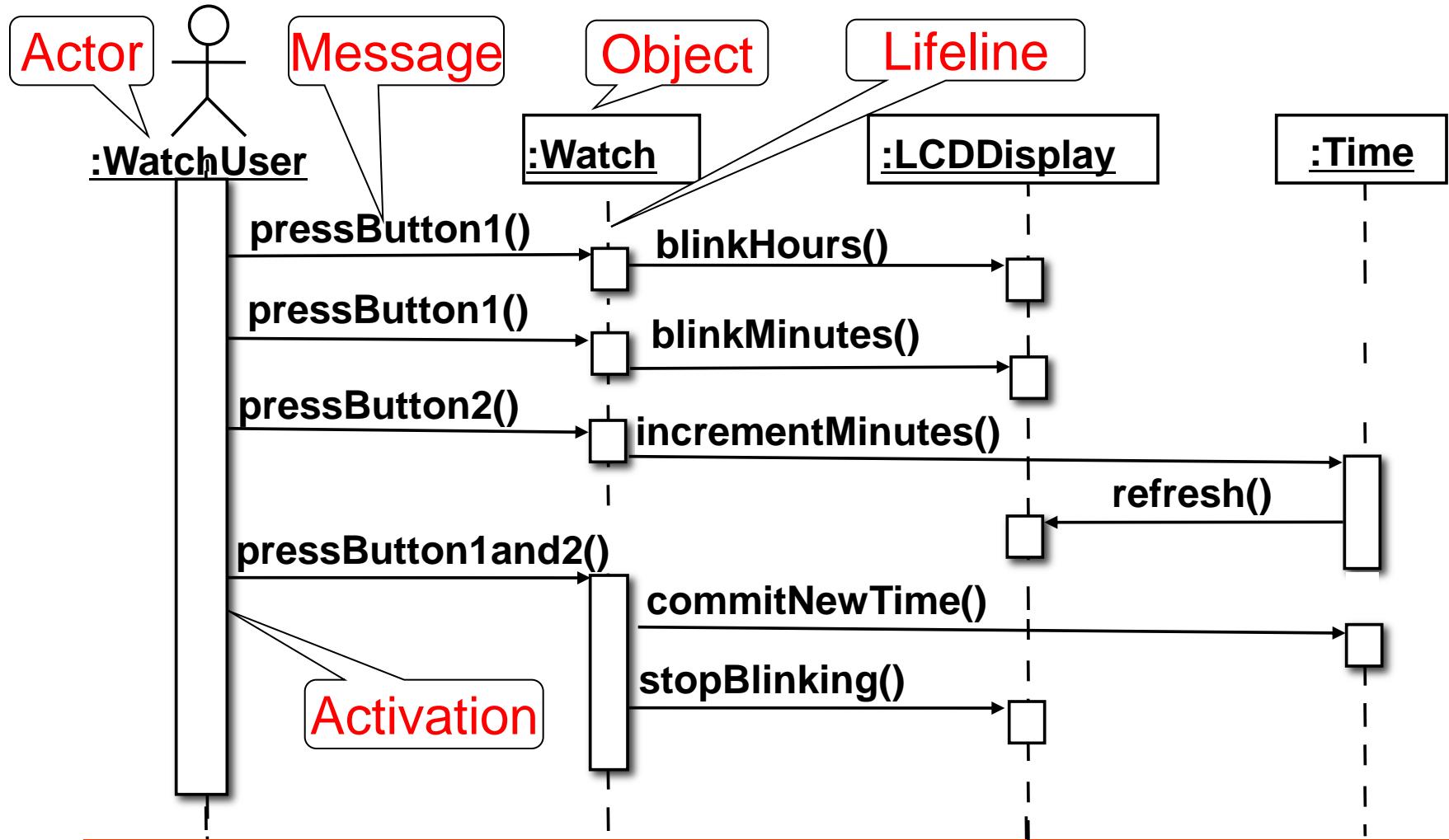
Class diagrams represent the structure of the system

# UML first pass: Class diagrams

Class diagrams represent the structure of the system

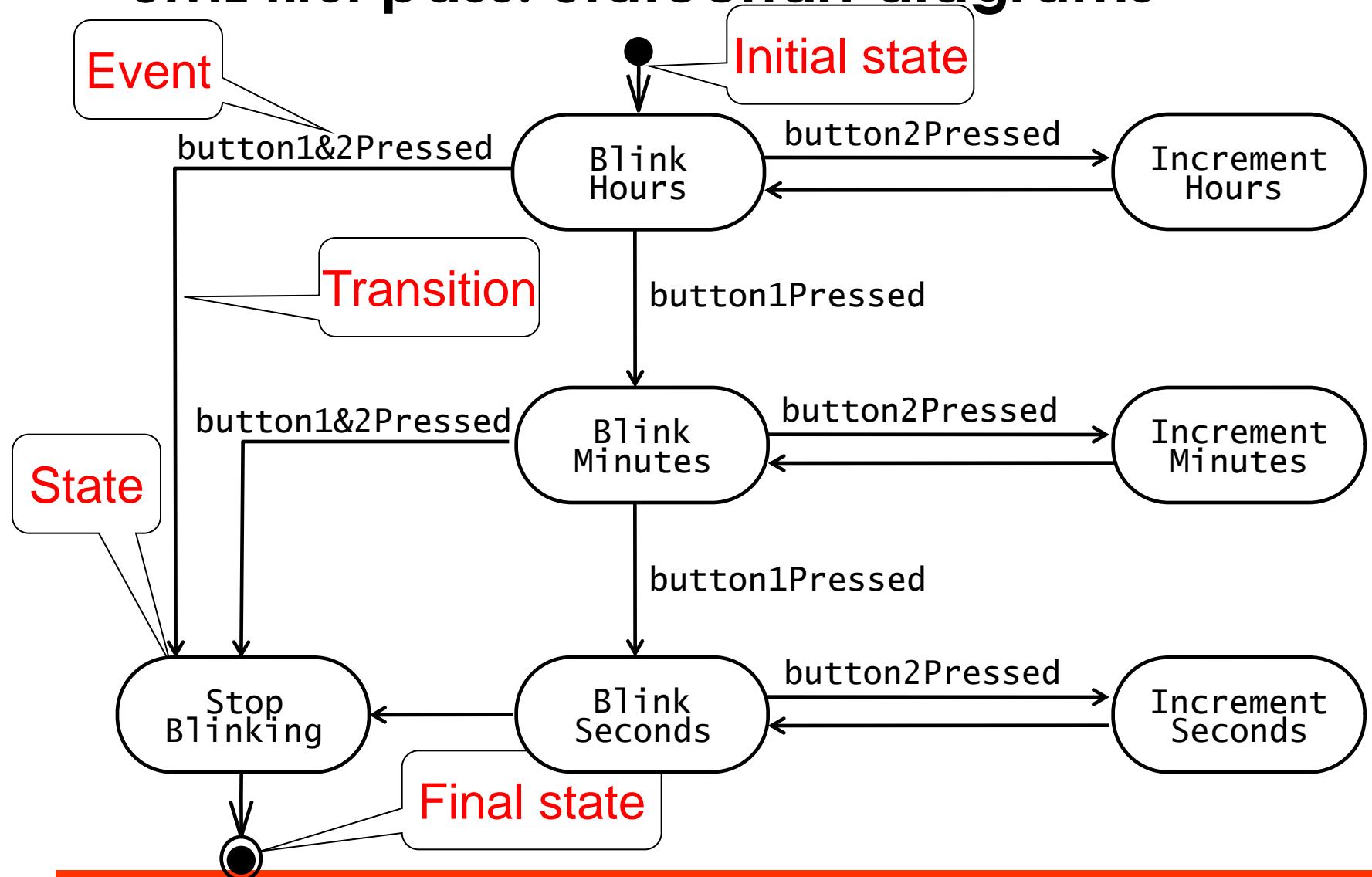


# UML first pass: Sequence diagram



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 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.*

# UML Basic Notation Summary

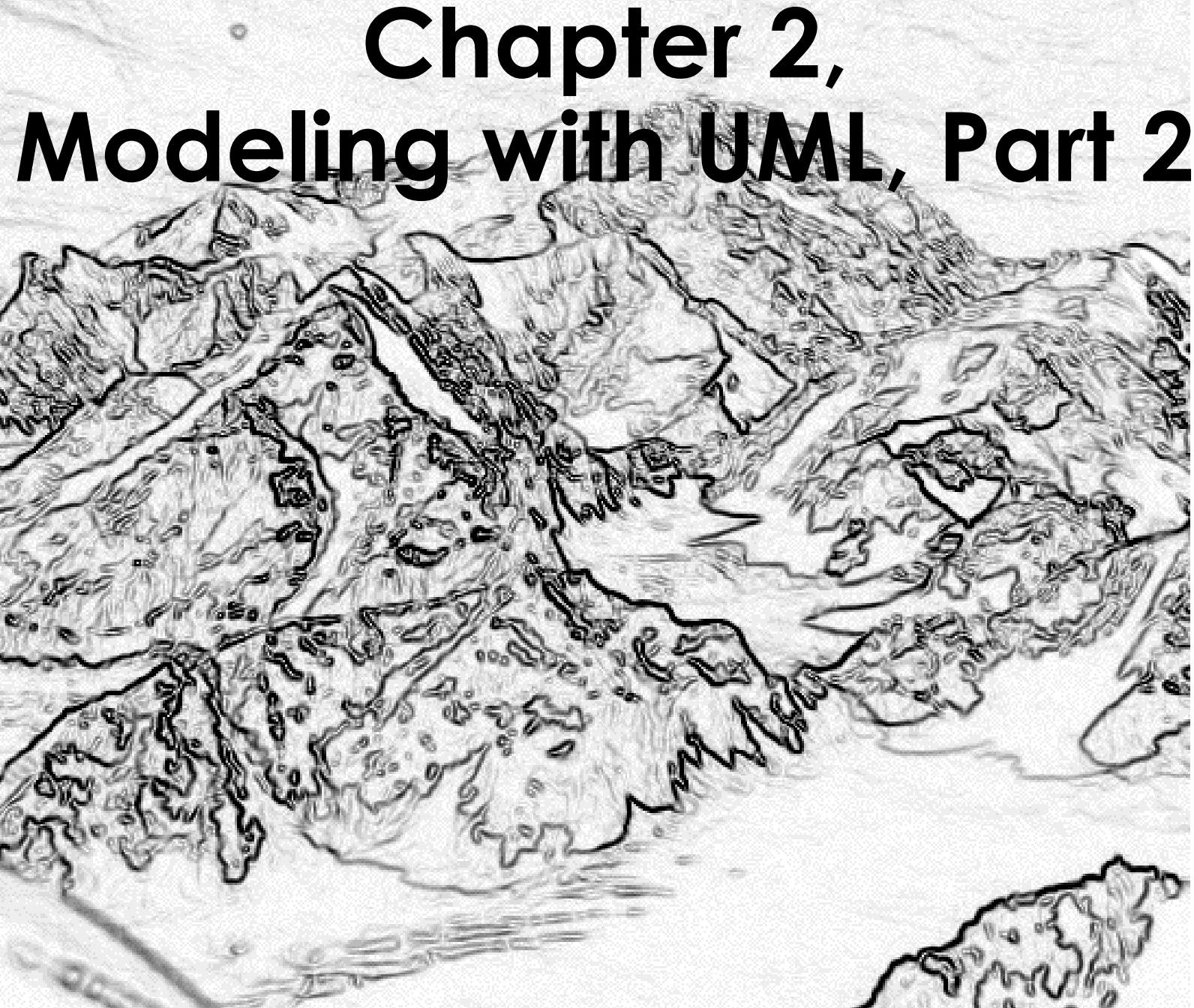
- *UML provides a wide variety of notations for modeling many aspects of software systems*
- *Today we concentrated on a few notations:*
  - *Functional model: Use case diagram*
  - *Object model: Class diagram*
  - *Dynamic model: Sequence diagrams, statechart*

# 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, 2<sup>nd</sup> edition, 2005*
- *Commercial UML tools*
  - *Rational Rose XDE for Java*
    - <http://www-306.ibm.com/software/awdtools/developer/java/>
  - *Together (Eclipse, MS Visual Studio, JBuilder)*
    - <http://www.borland.com/us/products/together/index.html>
- *Open Source UML tools*
  - <http://java-source.net/open-source/uml-modeling>
  - *ArgoUML, UMLet, Violet, ...*

# **Object-Oriented Software Engineering**

Using UML, Patterns, and Java



## **Chapter 2, Modeling with UML, Part 2**

# Outline of this Class

- *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.*

# What is UML? Unified Modeling Language

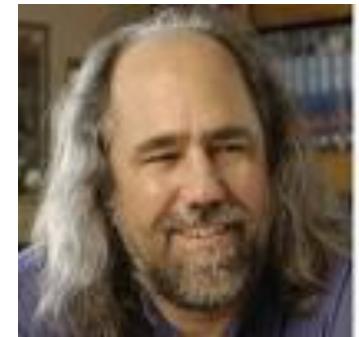
- *Convergence of different notations used in object-oriented methods, mainly*
  - *OMT (James Rumbaugh and colleagues), OOSE (Ivar Jacobson), Booch (Grady Booch)*
- *They also developed the Rational Unified Process, which became the Unified Process in 1999*



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



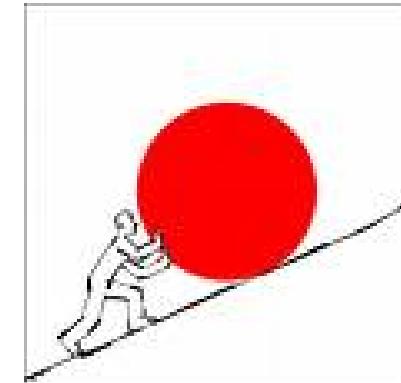
At Ericsson until 1994,  
developed use cases and the  
CASE tool Objectory, at IBM  
Rational since 1995,  
<http://www.ivarjacobson.com>



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

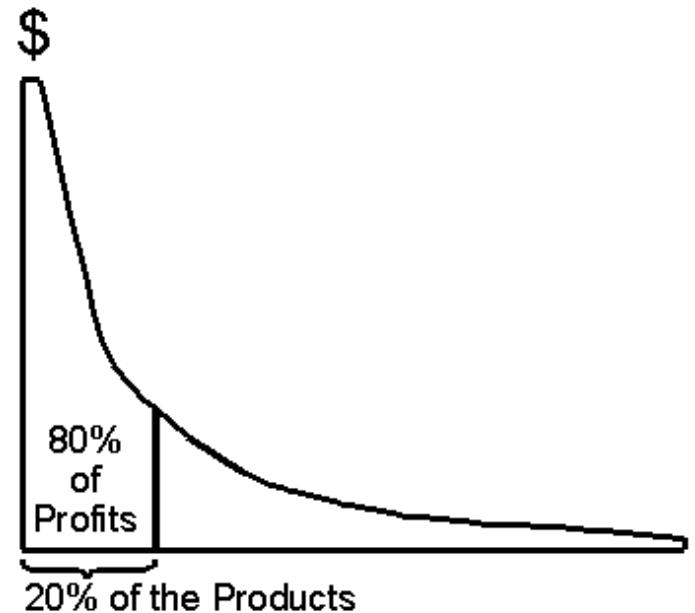
# UML

- *Nonproprietary standard for modeling systems*
- *Current Version: UML 2.2*
  - *Information at the OMG portal <http://www.uml.org/>*
- *Commercial tools:*
  - *Rational (IBM), Together (Borland), Visual Architect (Visual Paradigm), Enterprise Architect (SpaRx Systems)*
- *Open Source tools <http://www.sourceforge.net/>*
  - *ArgoUML, StarUML, Umbrello (for KDE), PoseidonUML*
- *Example of research tools: Unicase, Sysiphus*
  - *Based on a unified project model for modeling, collaboration and project organization*
  - <http://unicase.org>
  - <http://sysiphus.in.tum.de/>



# 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 (covered in Last Lecture)

- *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 Basic Notation: First Summary

- *UML provides a wide variety of notations for modeling many aspects of software systems*
- *In the first lecture we concentrated on:*
  - *Functional model: Use case diagram*
  - *Object model: Class diagram*
  - *Dynamic model: Sequence diagrams, statechart*
- *Now we go into a little bit more detail...*

# UML Use Case Diagrams

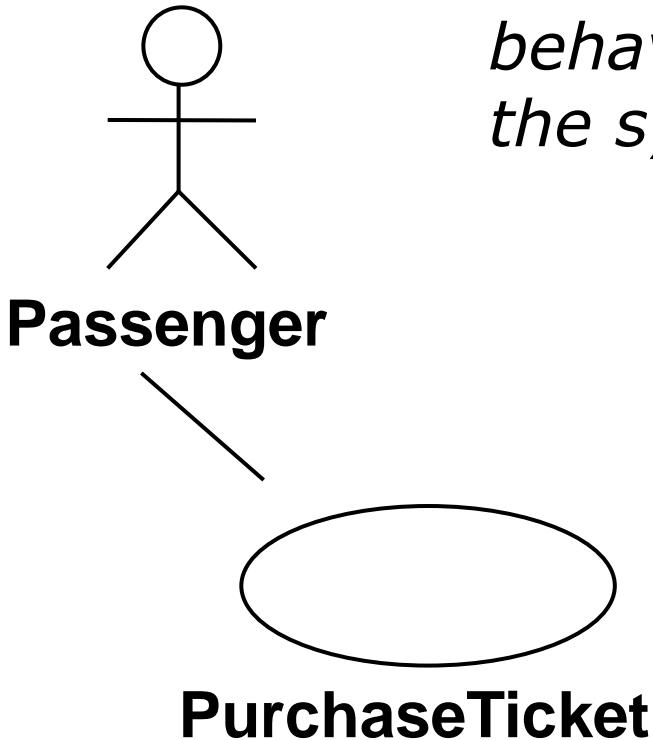
*Used during requirements elicitation and analysis to represent external behavior ("visible from the outside of the system")*

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

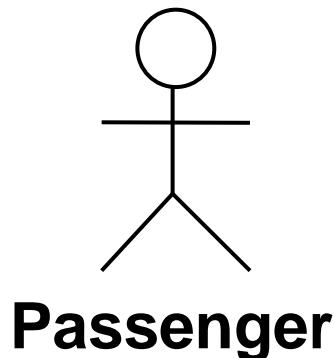
A **use case** represents a class of functionality provided by the system

**Use case model:**

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



# Actors

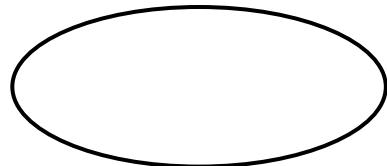


- *An actor is a model for an external entity which interacts (communicates) with the system:*
  - User
  - External system (Another system)
  - Physical environment (e.g. Weather)
- *An actor has a unique name and an optional description*
- *Examples:*
  - **Passenger:** A person in the train
  - **GPS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

Optional  
Description

Name

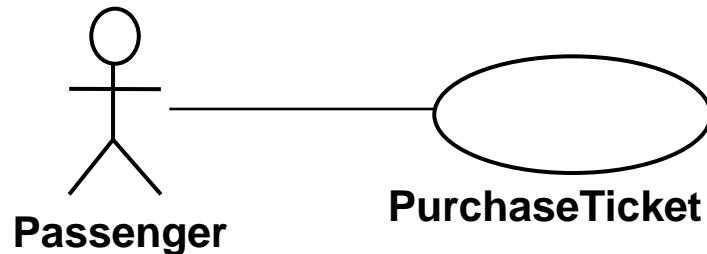
# Use Case



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:*

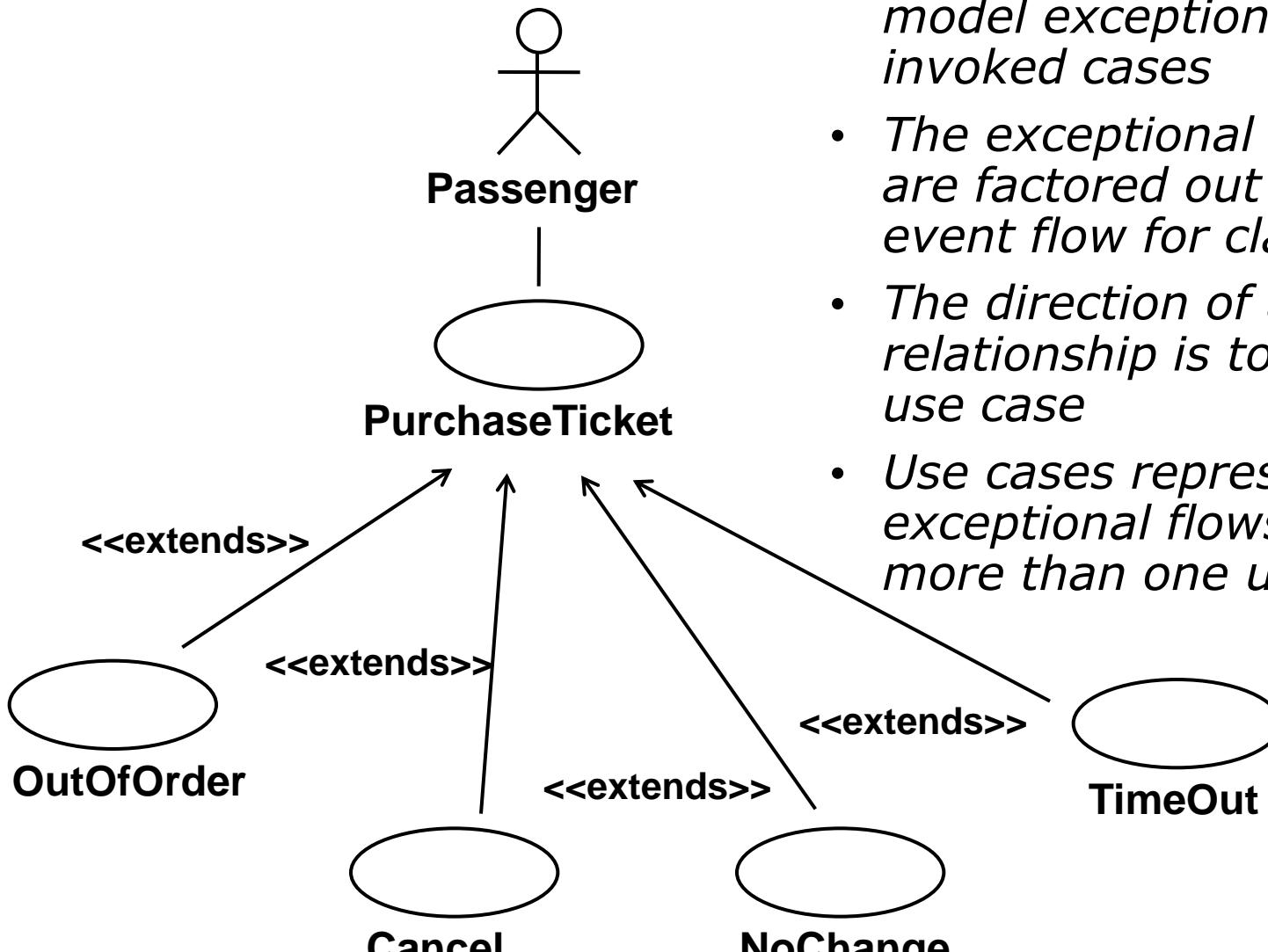
1. Passenger selects the number of zones to be traveled
2.  Ticket Distributor displays the amount due
3. Passenger inserts money, at least the amount due
4. Ticket Distributor returns change
5. Ticket Distributor issues ticket

6. *Special requirements:*  
None.

# Uses Cases can be related

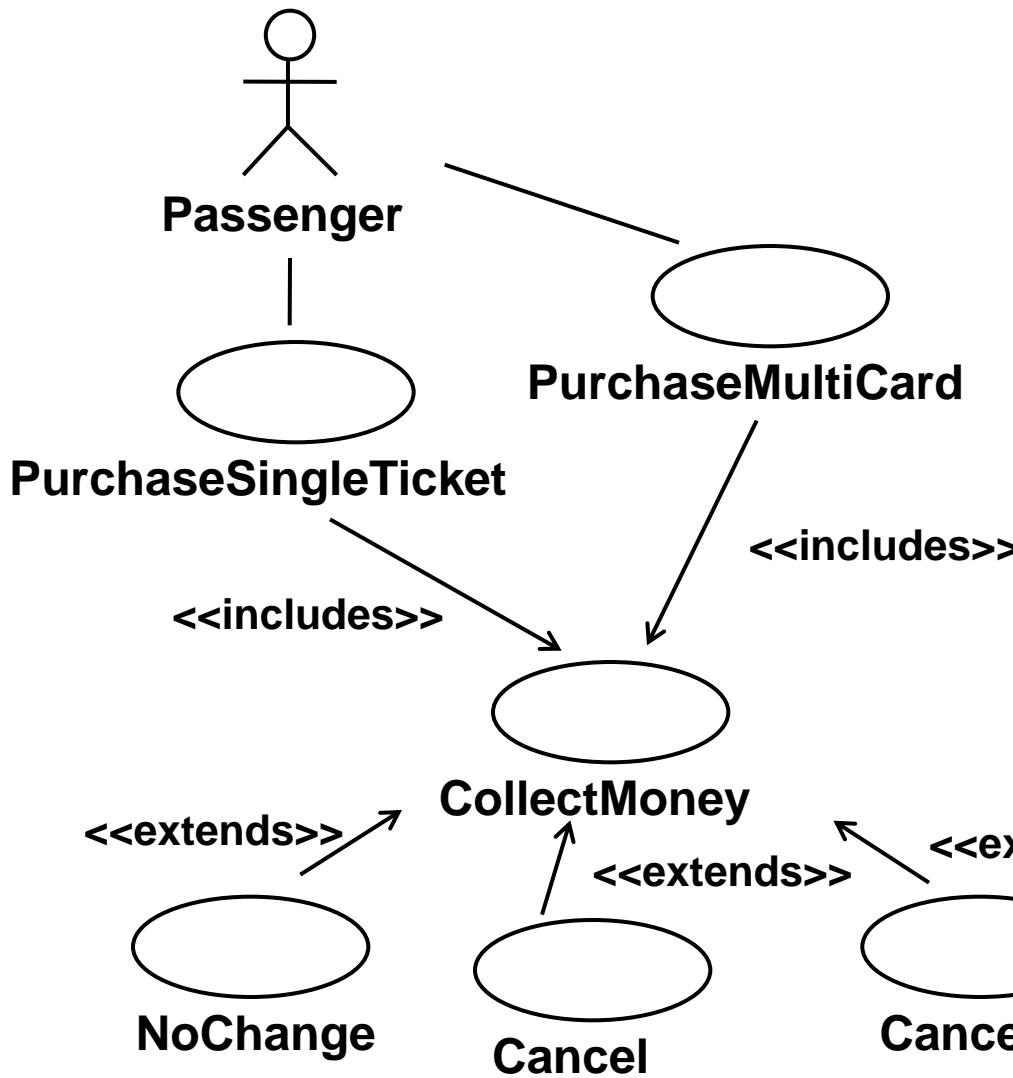
- *Extends Relationship*
  - *To represent seldom invoked use cases or exceptional functionality*
- *Includes Relationship*
  - *To represent functional behavior common to more than one use case.*

# 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.

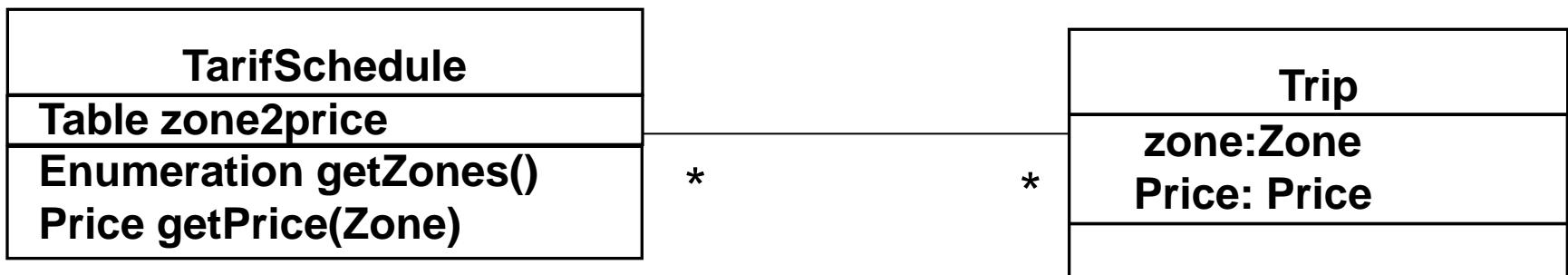
# The <<includes>> Relationship



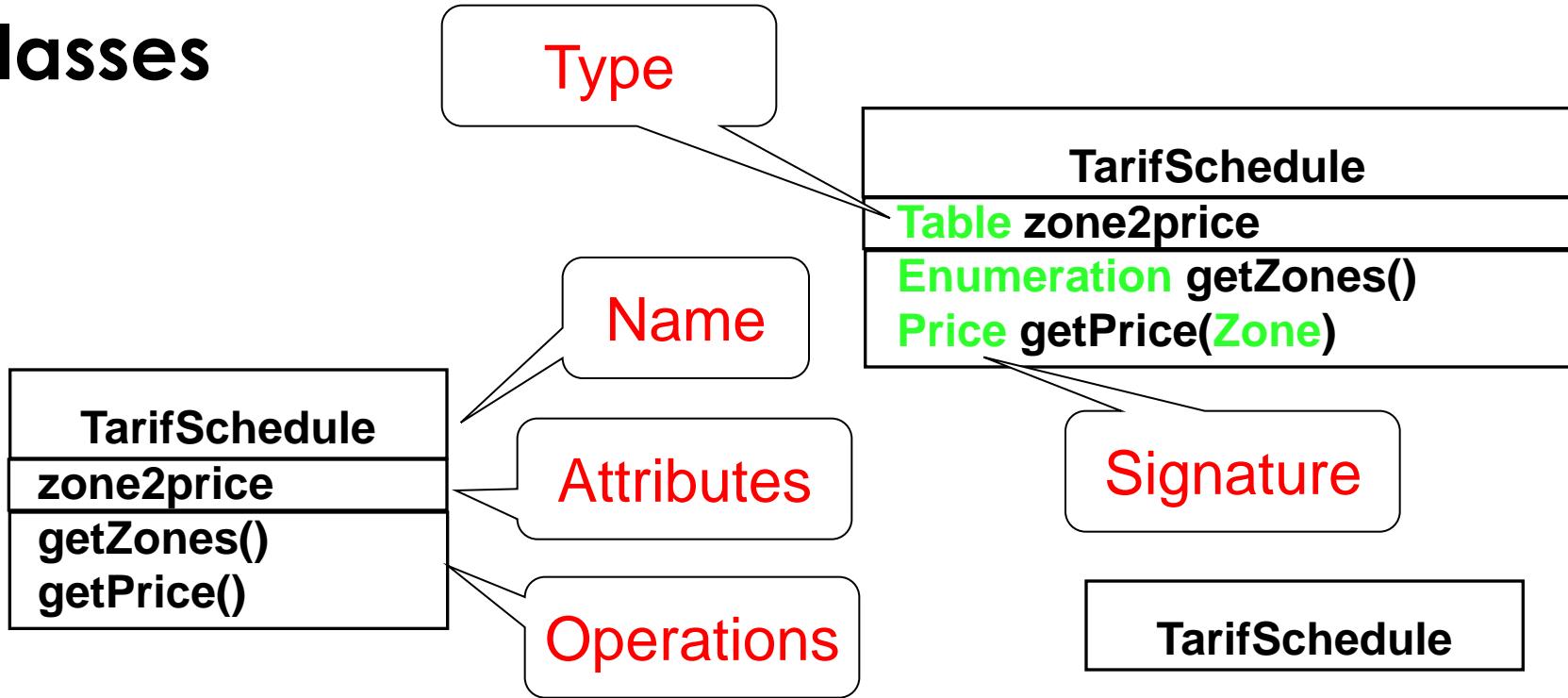
- <<includes>> relationship represents common functionality needed in more than one use case
- <<includes>> behavior is factored out for reuse, not because it is an exception
- The direction of a <<includes>> relationship is to the using use case (unlike the direction of the <<extends>> relationship).

# 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.*



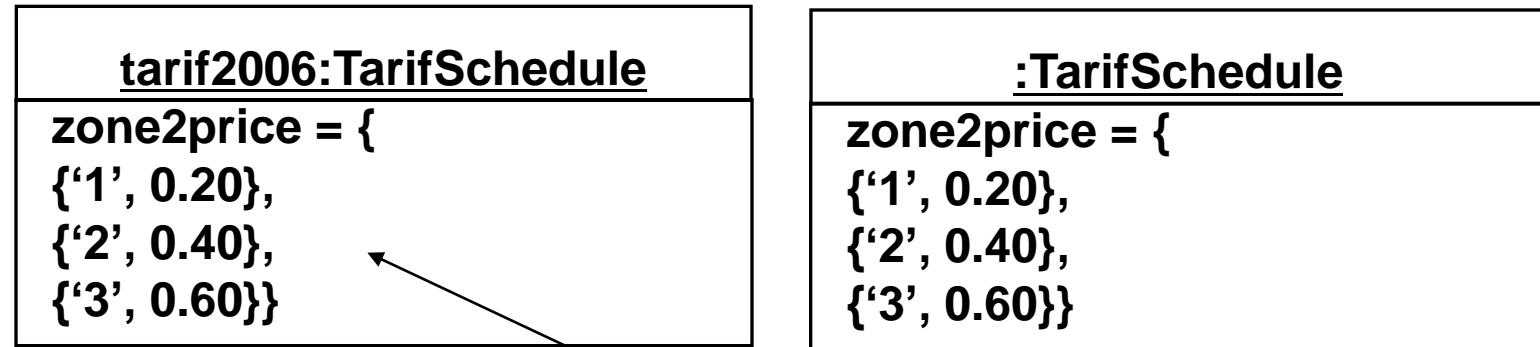
# Classes



- 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

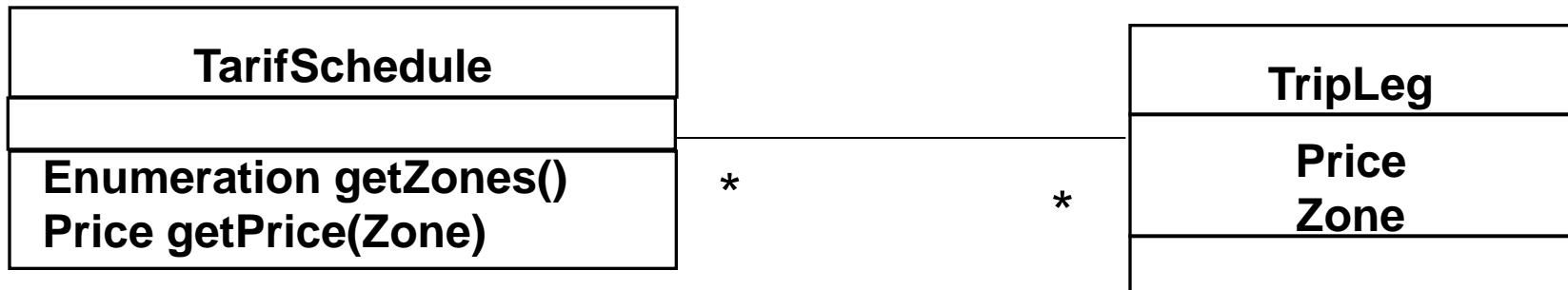


- An ***instance*** represents a phenomenon
- The attributes are represented with their ***values***
- The name of an *instance* is underlined
- The name can contain only the class name of the *instance* (anonymous *instance*)

# 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").*

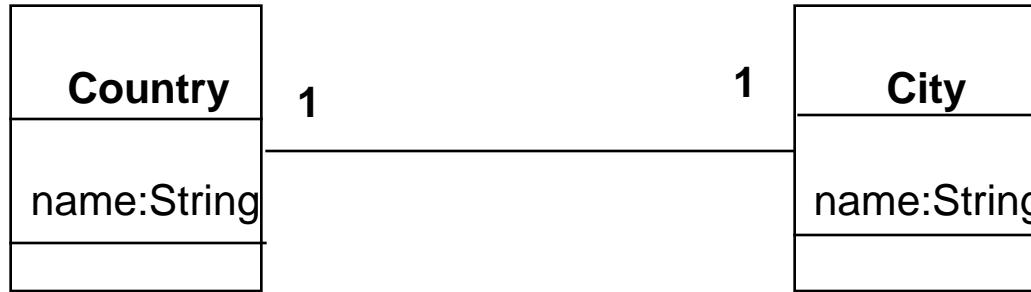
# Associations



*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



# From Problem Statement To Object Model

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

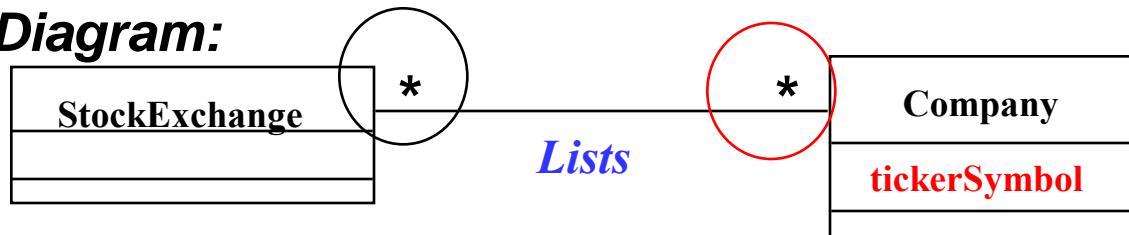
*Class Diagram:*



# 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**

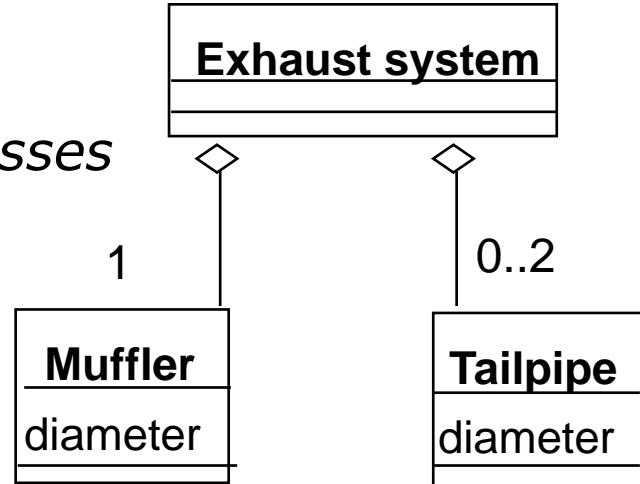
```
public class StockExchange
{
    private Vector m_Company = new Vector();
};

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

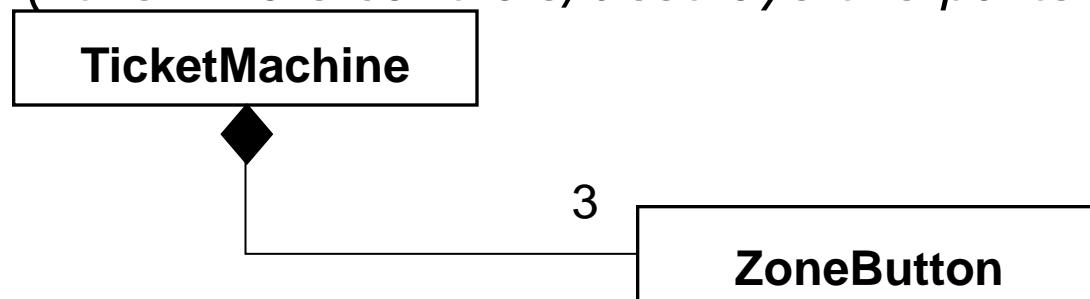
**Associations  
are mapped to  
Attributes!**

# 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

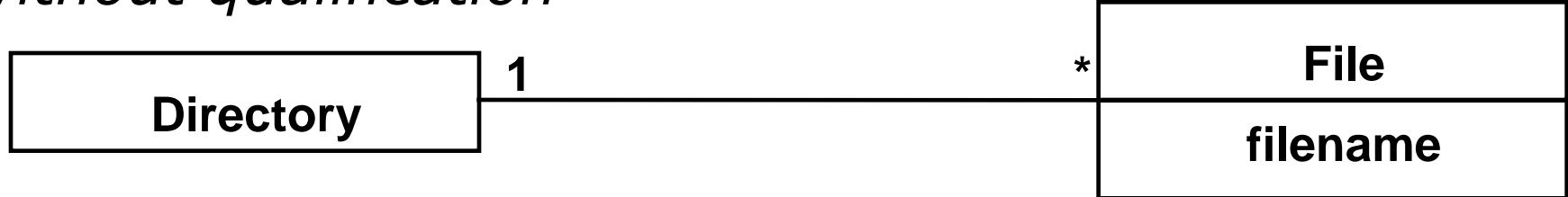


A solid diamond denotes *composition*: A strong form of aggregation where the life time of the component instances is controlled by the aggregate. That is, the parts don't exist on their own ("the whole controls/destroys the parts")

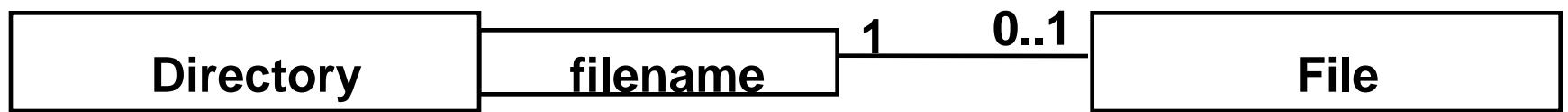


# Qualifiers

*Without qualification*

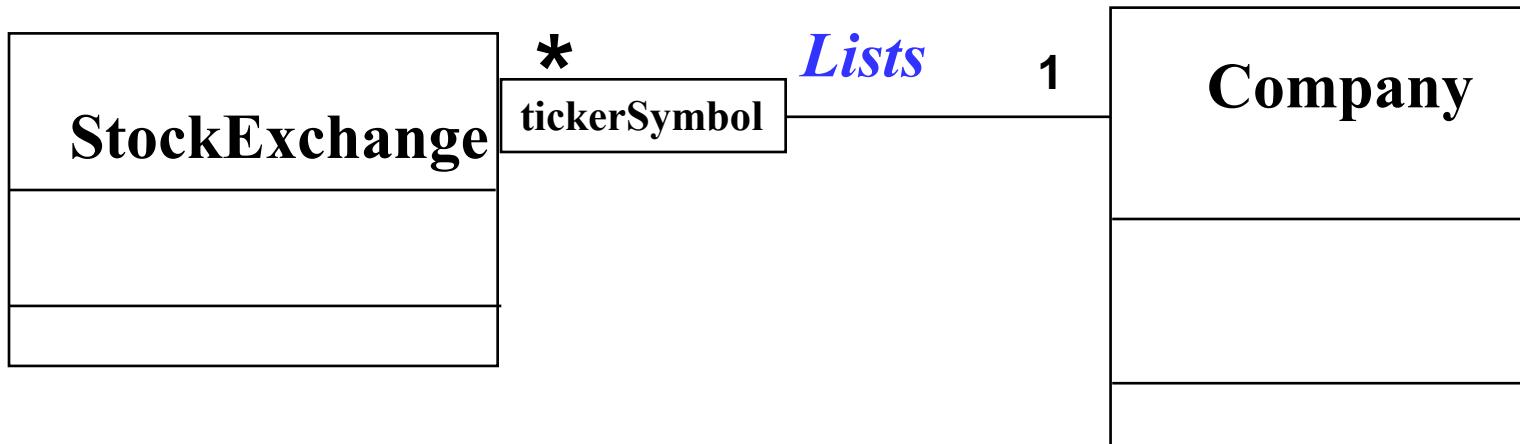


*With qualification*

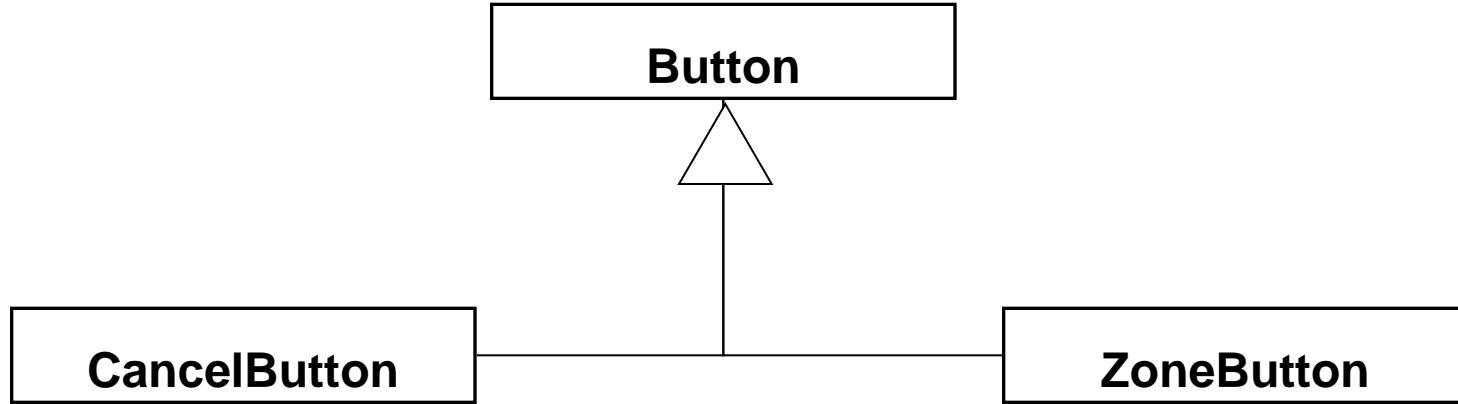


- *Qualifiers can be used to reduce the multiplicity of an association*

# Qualification: Another Example



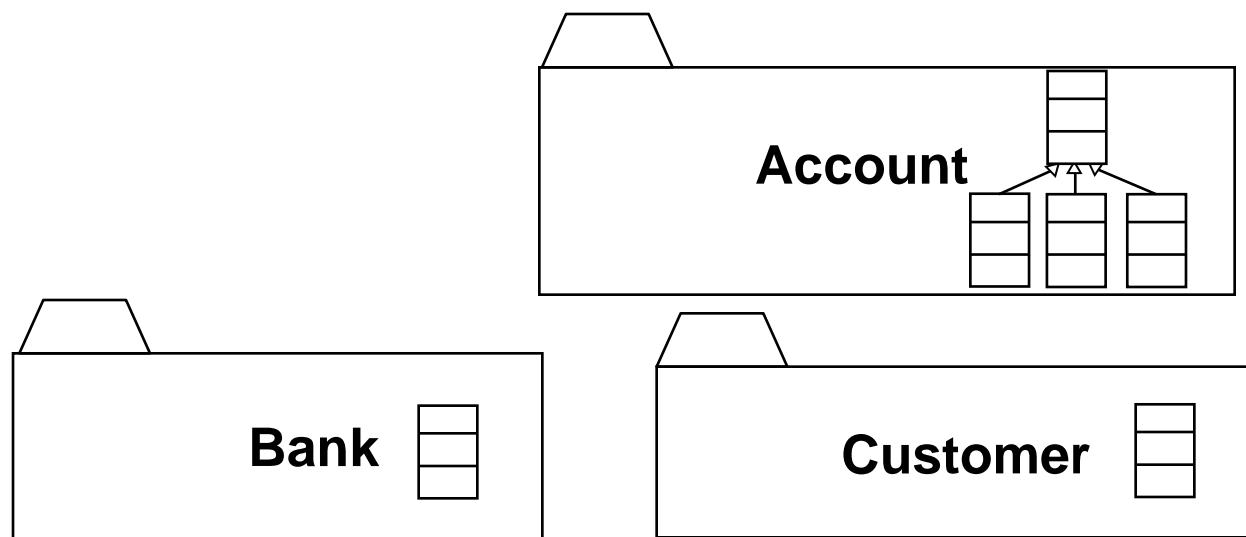
# Inheritance



- *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.**

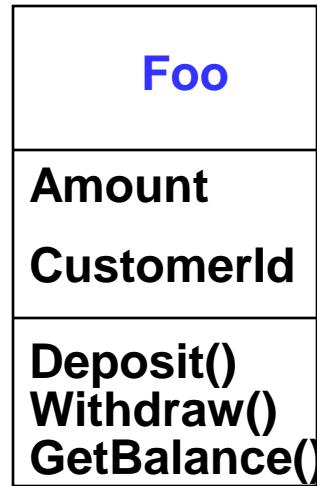
# Packages

- *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.*

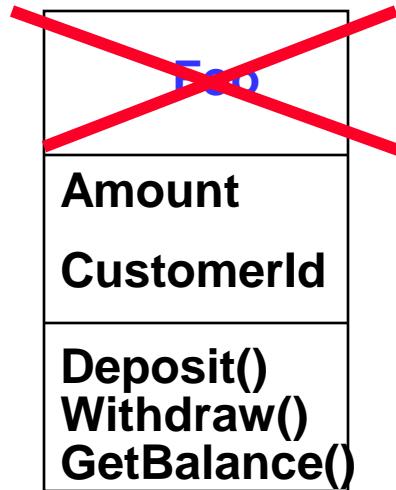
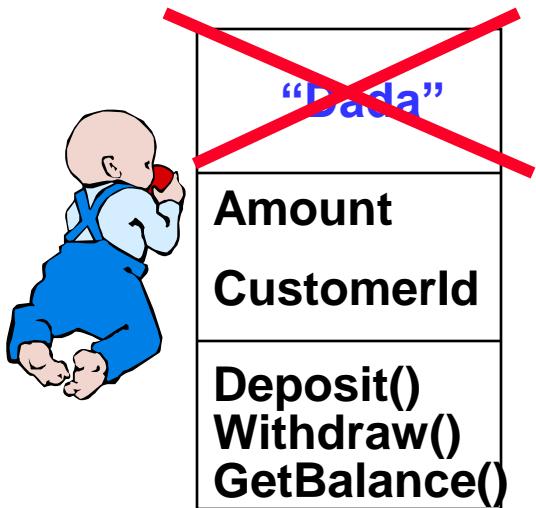
# Object Modeling in Practice



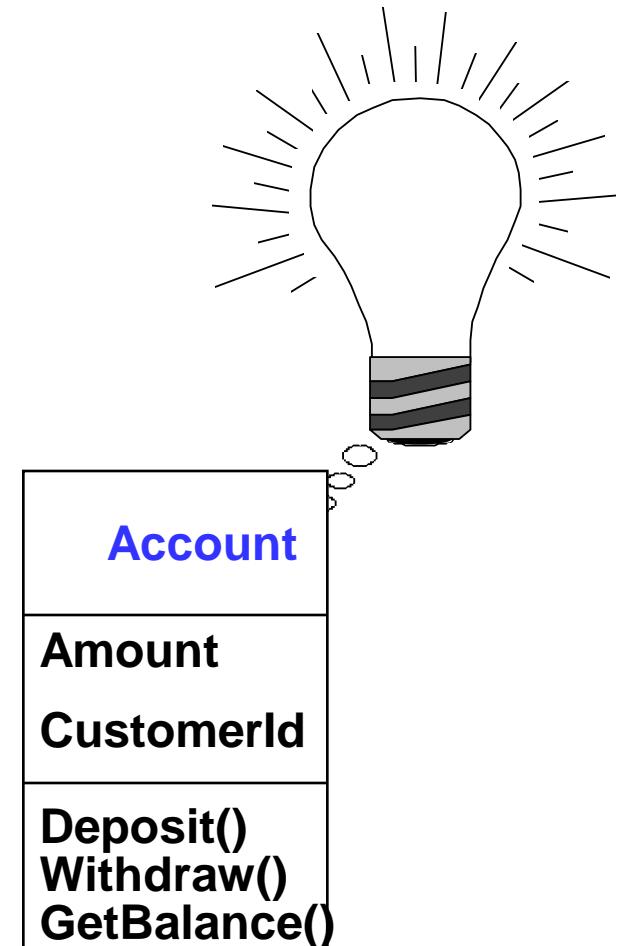
**Class Identification: Name of Class, Attributes and Methods**

Is **Foo** the right name?

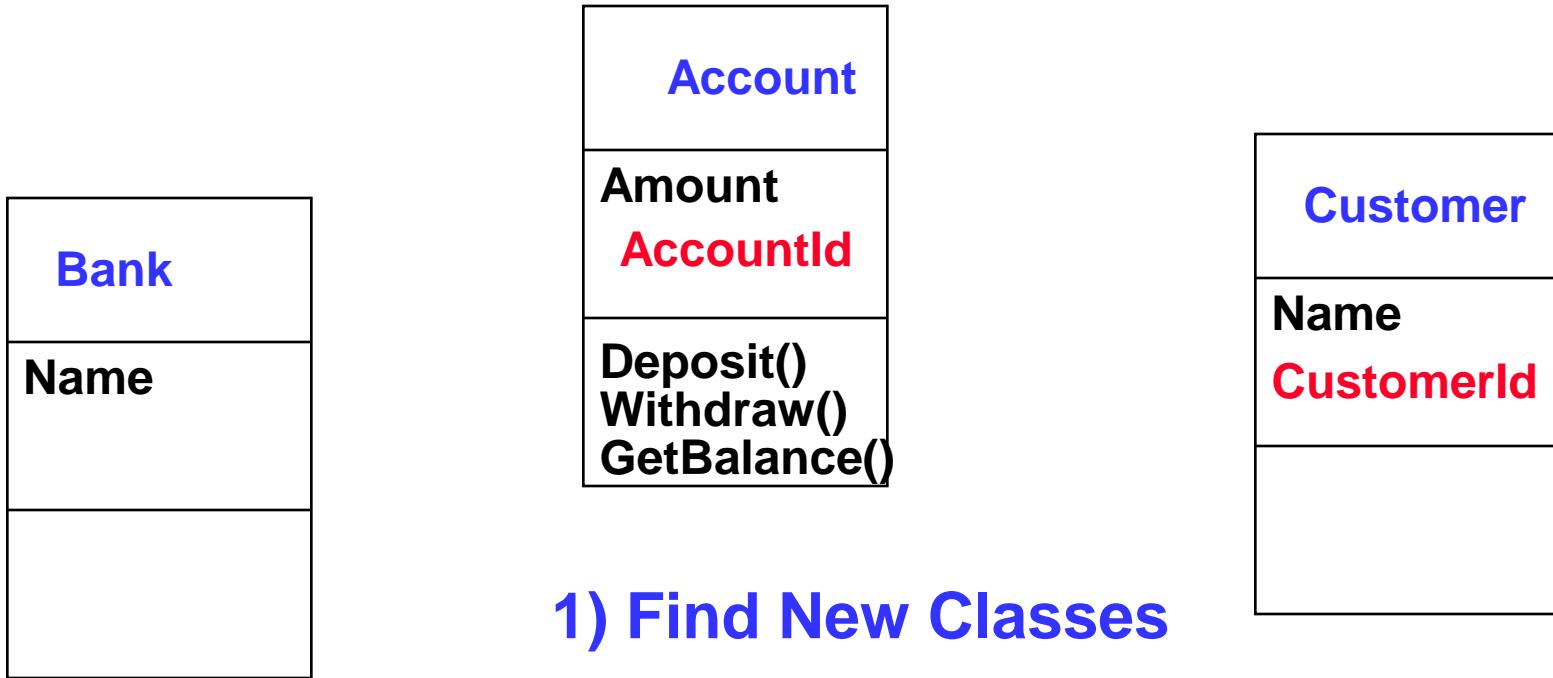
# Object Modeling in Practice: Brainstorming



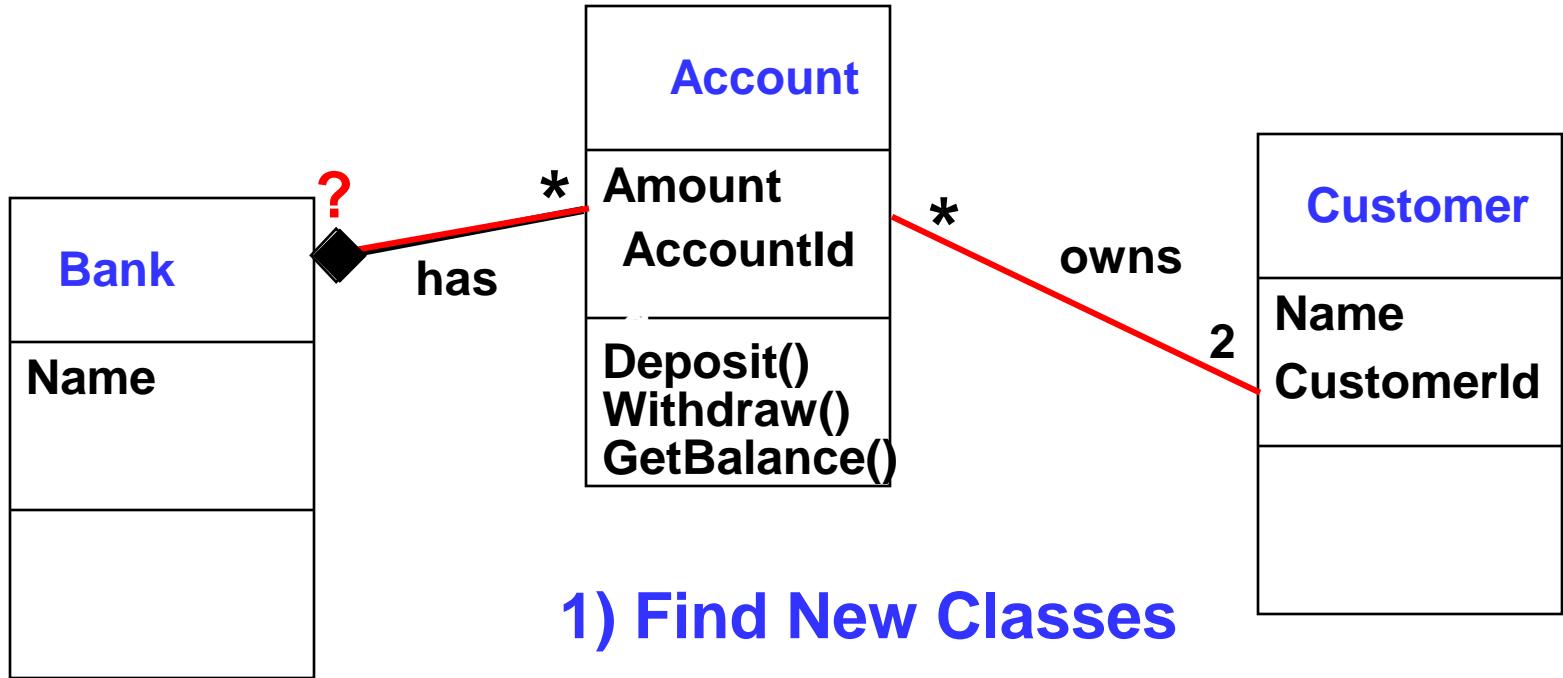
Is **Foo** the right name?



# Object Modeling in Practice: More classes



# Object Modeling in Practice: Associations



1) Find New Classes

2) Review Names, Attributes and Methods

3) Find Associations between Classes

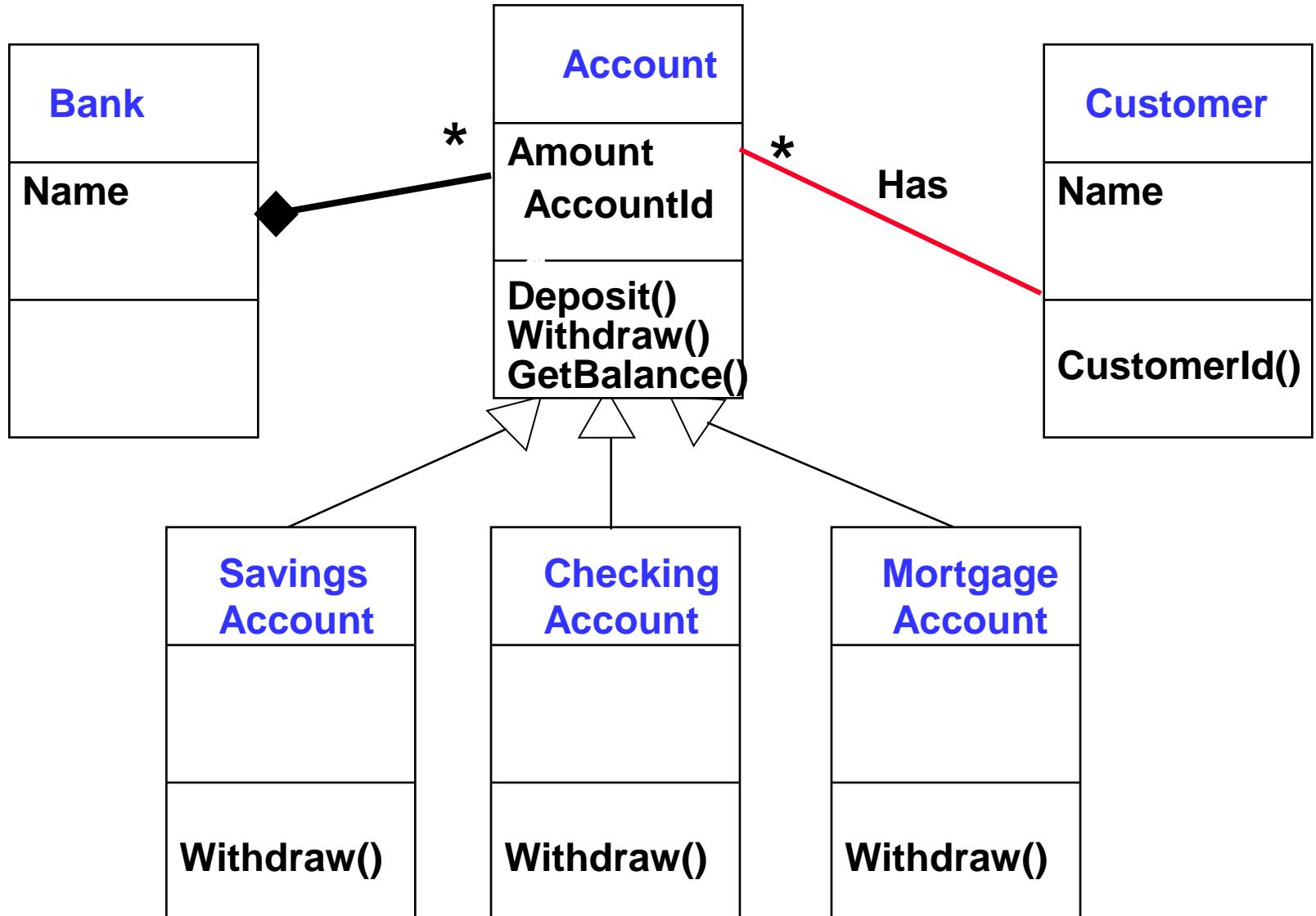
4) Label the generic associations

5) Determine the multiplicity of the associations

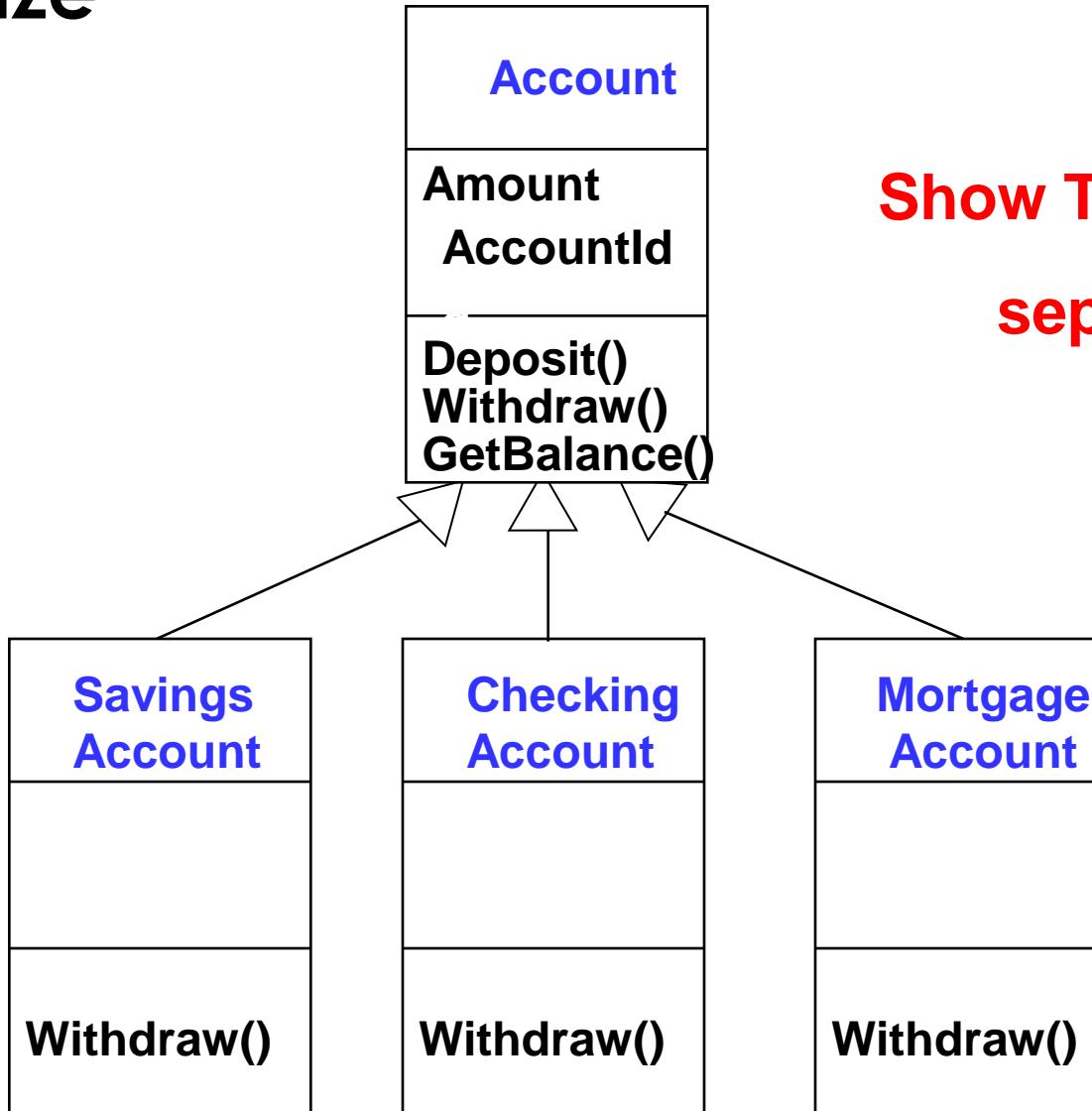
6) Review

associations

# Practice Object Modeling: Find Taxonomies

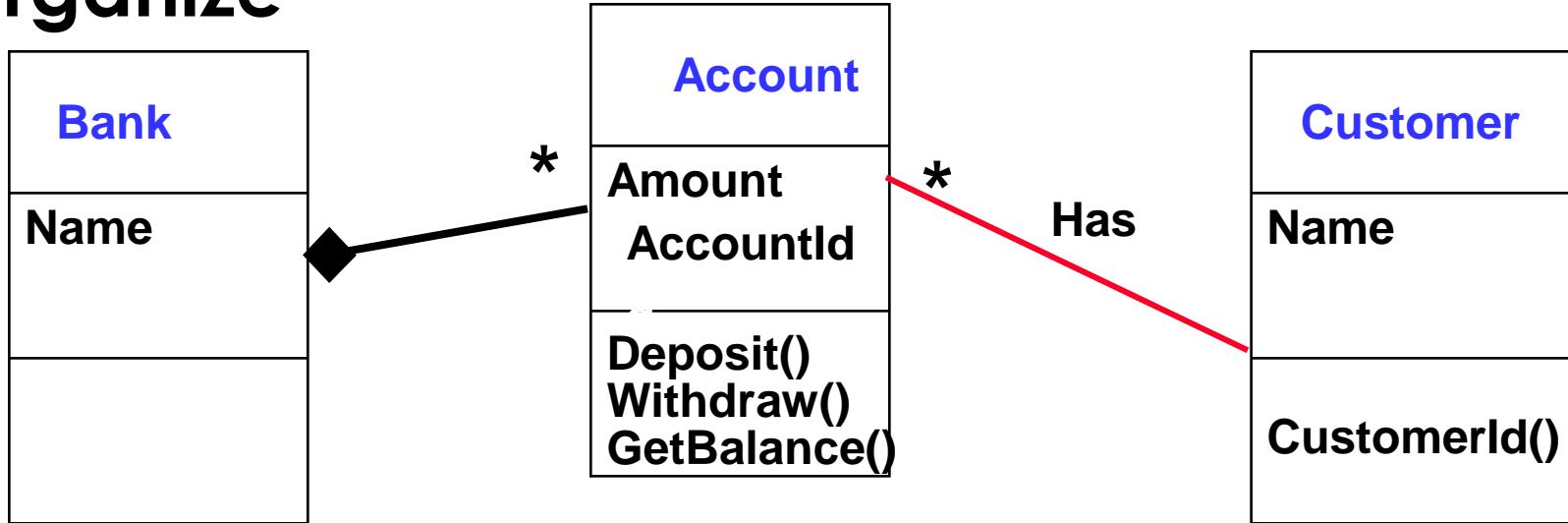


# Practice Object Modeling: Simplify, Organize



Show Taxonomies  
separately

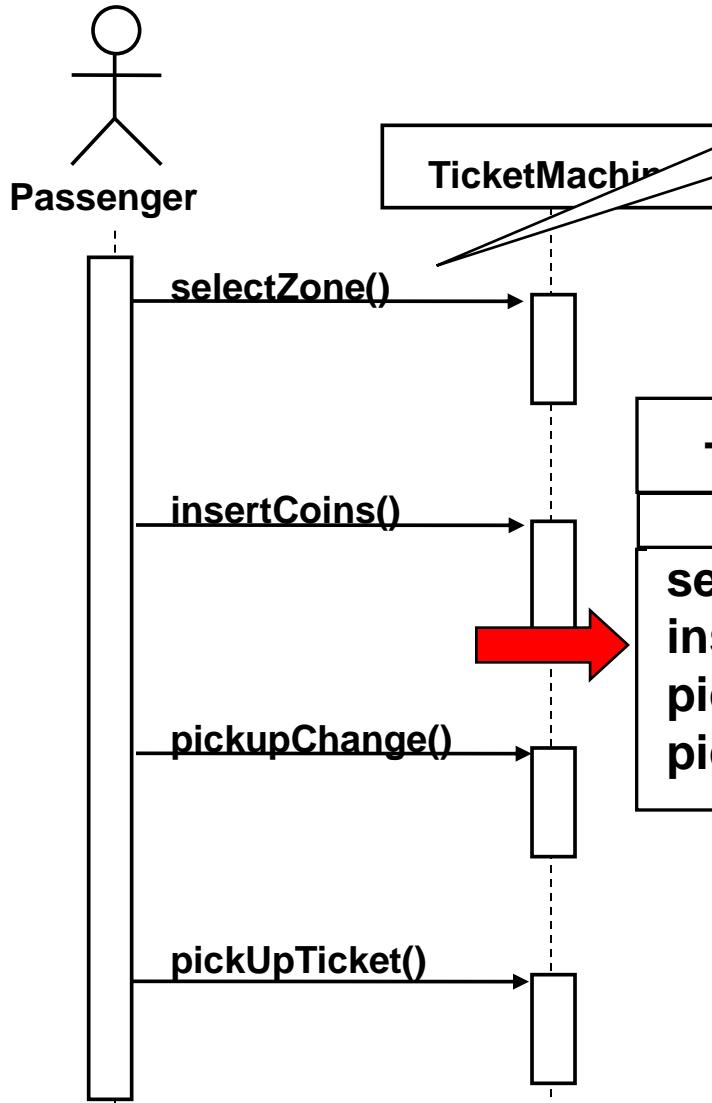
# Practice Object Modeling: Simplify, Organize



Use the 7+2 heuristics  
or better 5+2!

# Sequence Diagrams

Focus on  
Controlflow



Used during analysis

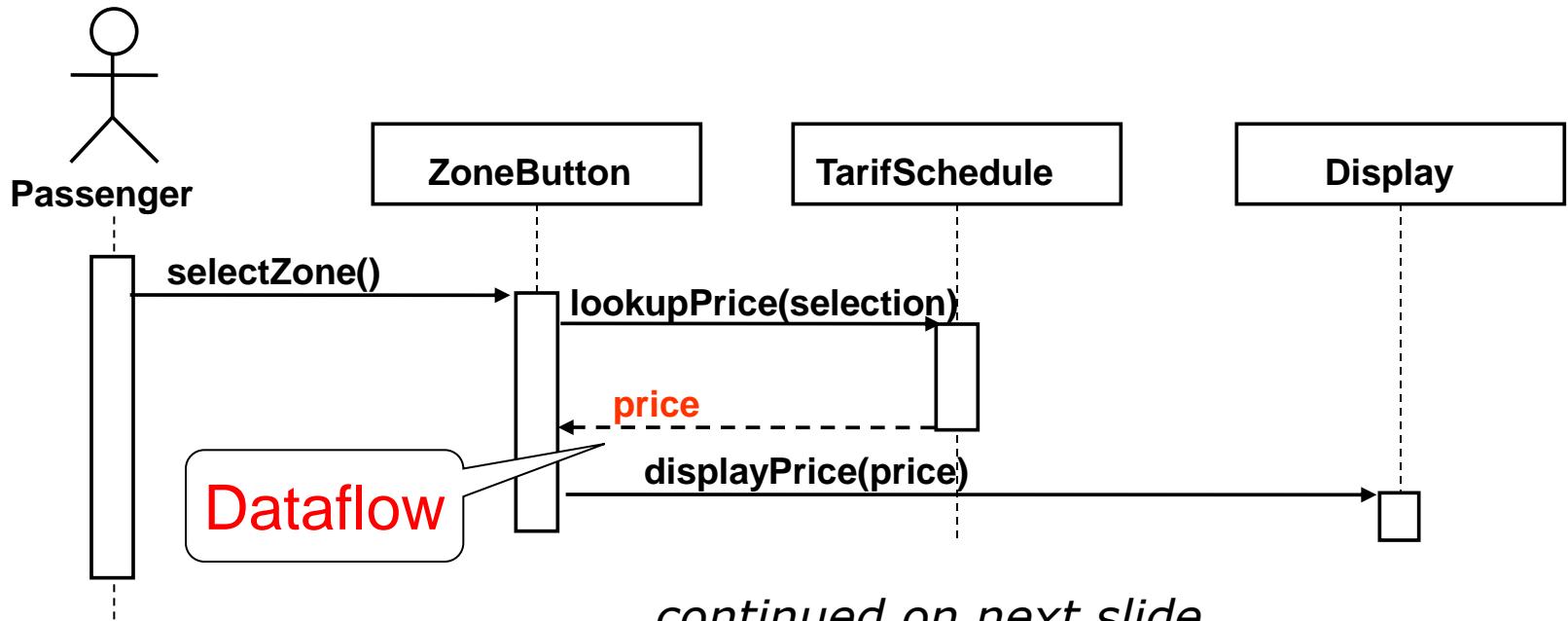
- To refine use case descriptions
- to find additional objects ("participating objects")
- Used during system design

define subsystem interfaces

Messages are operations. Activations are represented by lines

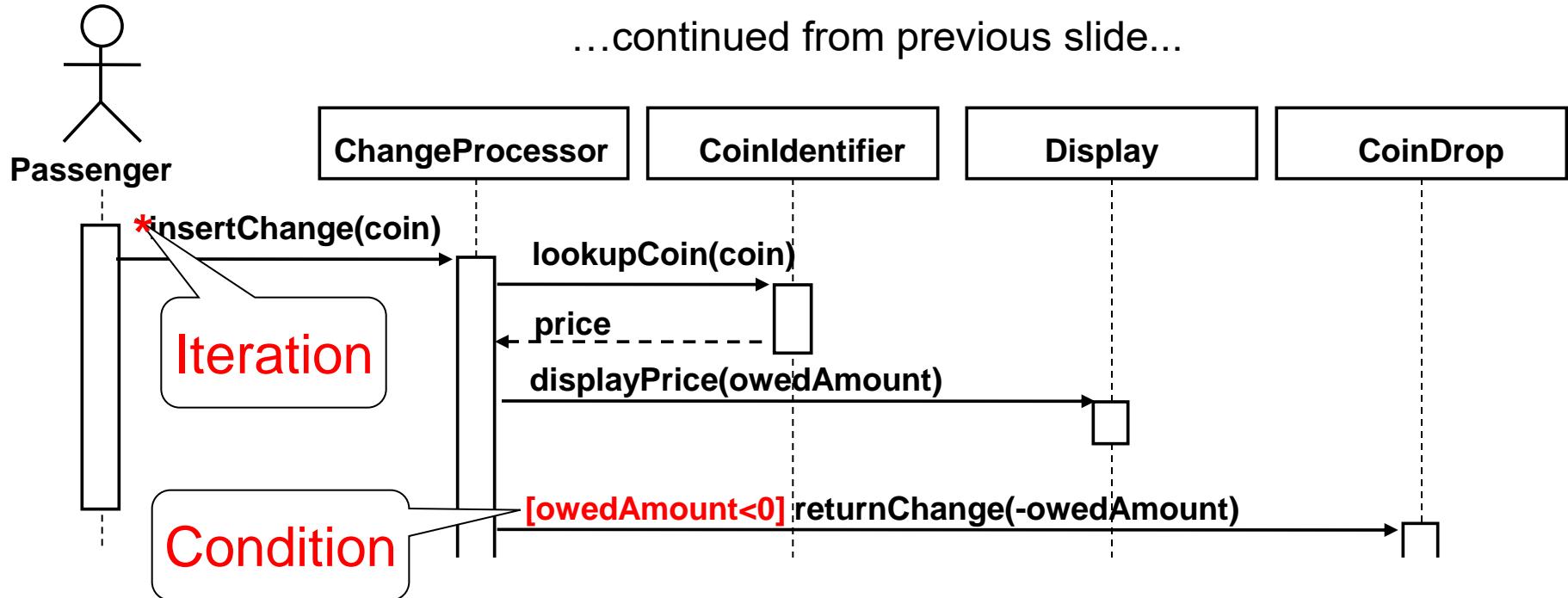
Messages are operations. Activations are represented by lines.

# 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*

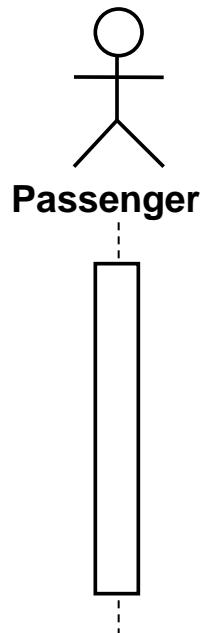
# Sequence Diagrams: Iteration & Condition



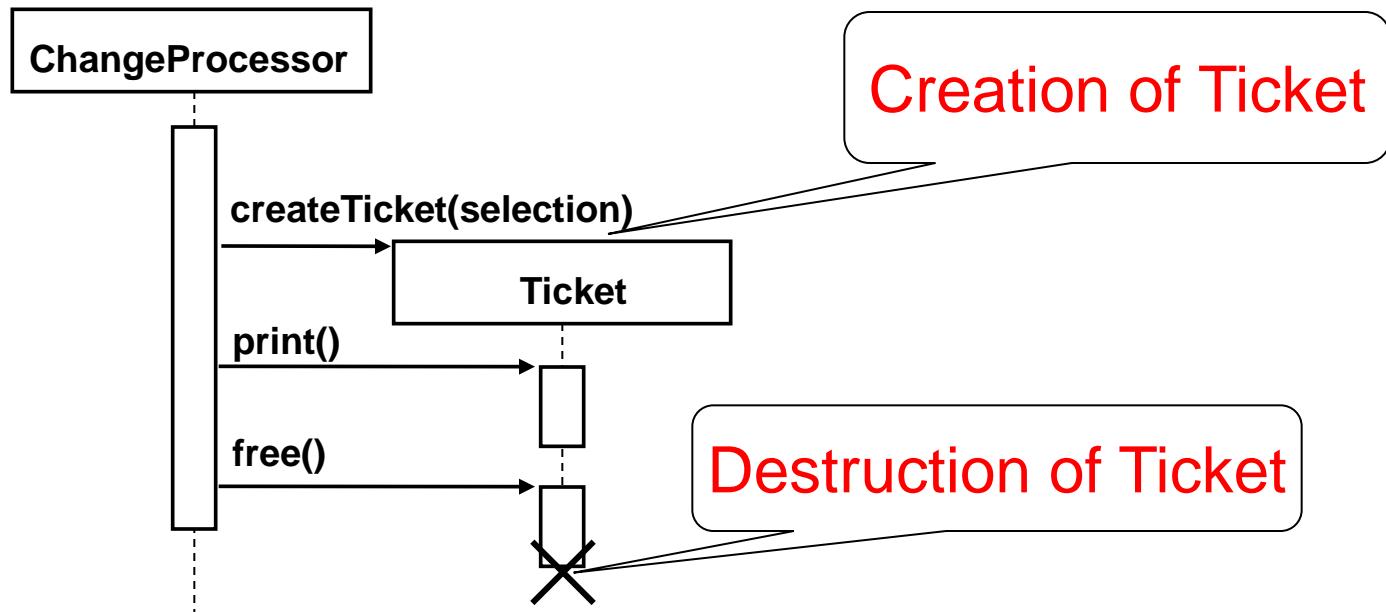
...continued on next slide...

- *Iteration is denoted by a \* preceding the message name*
- *Condition is denoted by boolean expression in [ ] before the message name*

# Creation and destruction



...continued from previous slide...



- *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).*

# Outline of this Class

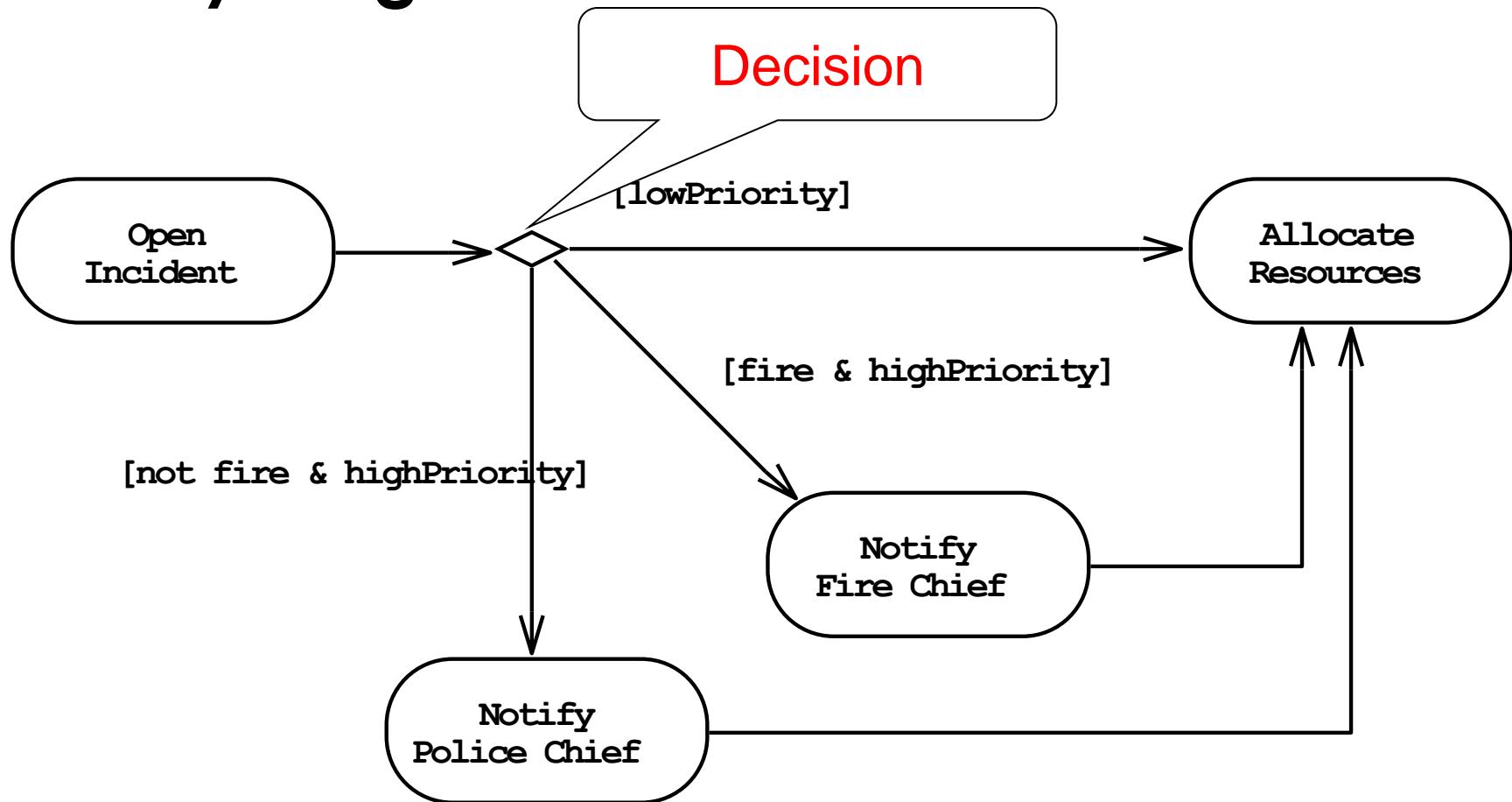
- A more detailed view on
  - ✓ *Use case diagrams*
  - ✓ *Class diagrams*
  - ✓ *Sequence diagrams*
  - *Activity diagrams*

# 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*

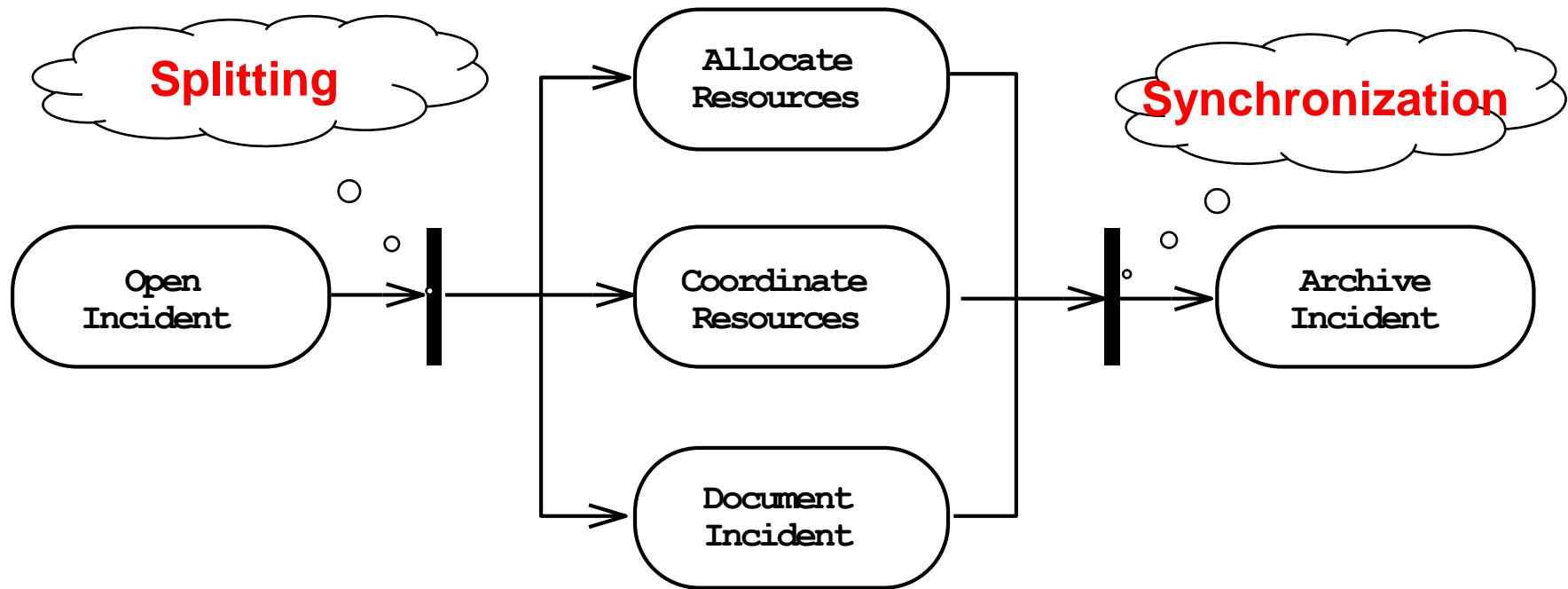


# Activity Diagrams allow to model Decisions



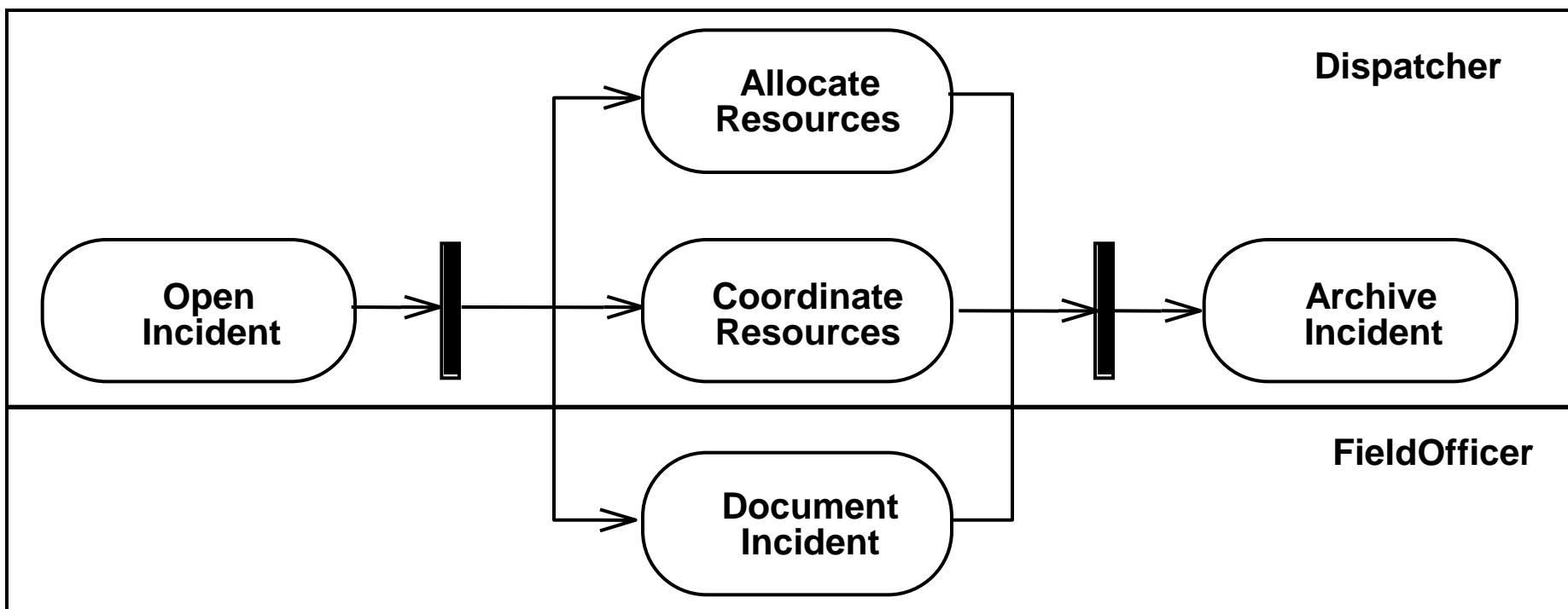
# Activity Diagrams can model Concurrency

- *Synchronization of multiple activities*
- *Splitting the flow of control into multiple threads*



# Activity Diagrams: Grouping of Activities

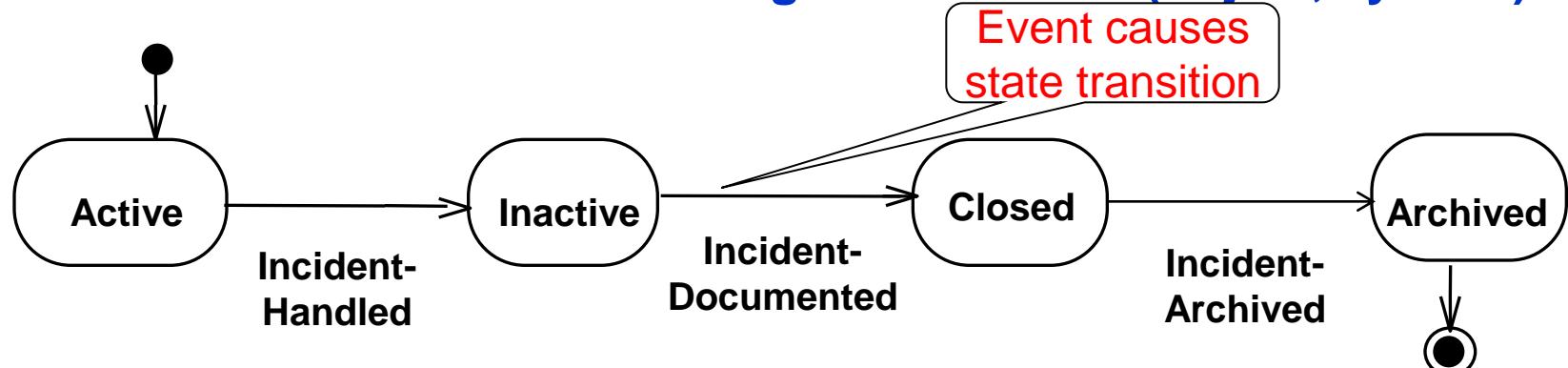
- *Activities may be grouped into **swimlanes** to denote the object or subsystem that implements the activities.*



# Activity Diagram vs. Statechart Diagram

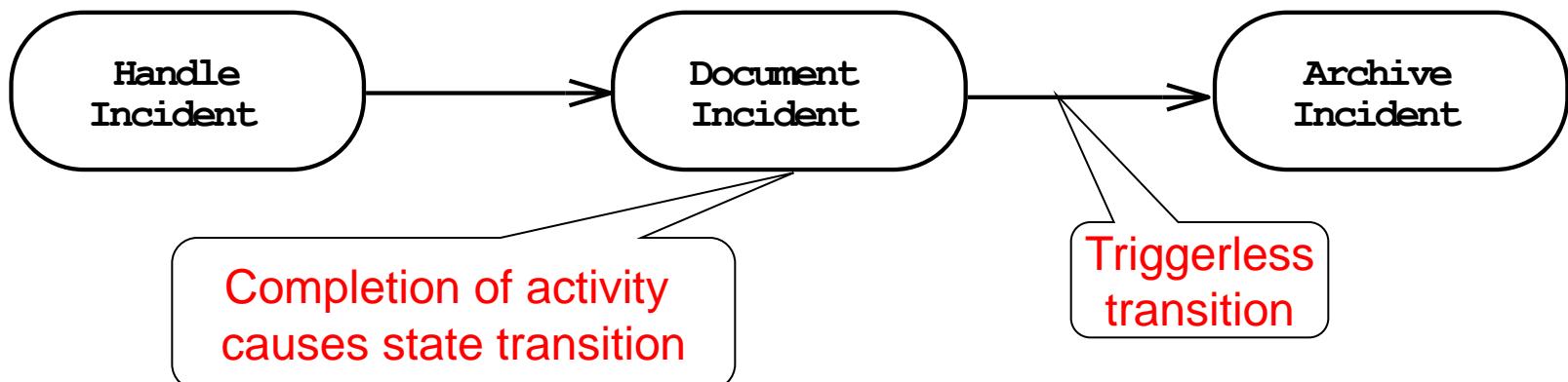
## Statechart Diagram for Incident

**Focus on the set of attributes of a single abstraction (object, system)**



## Activity Diagram for Incident

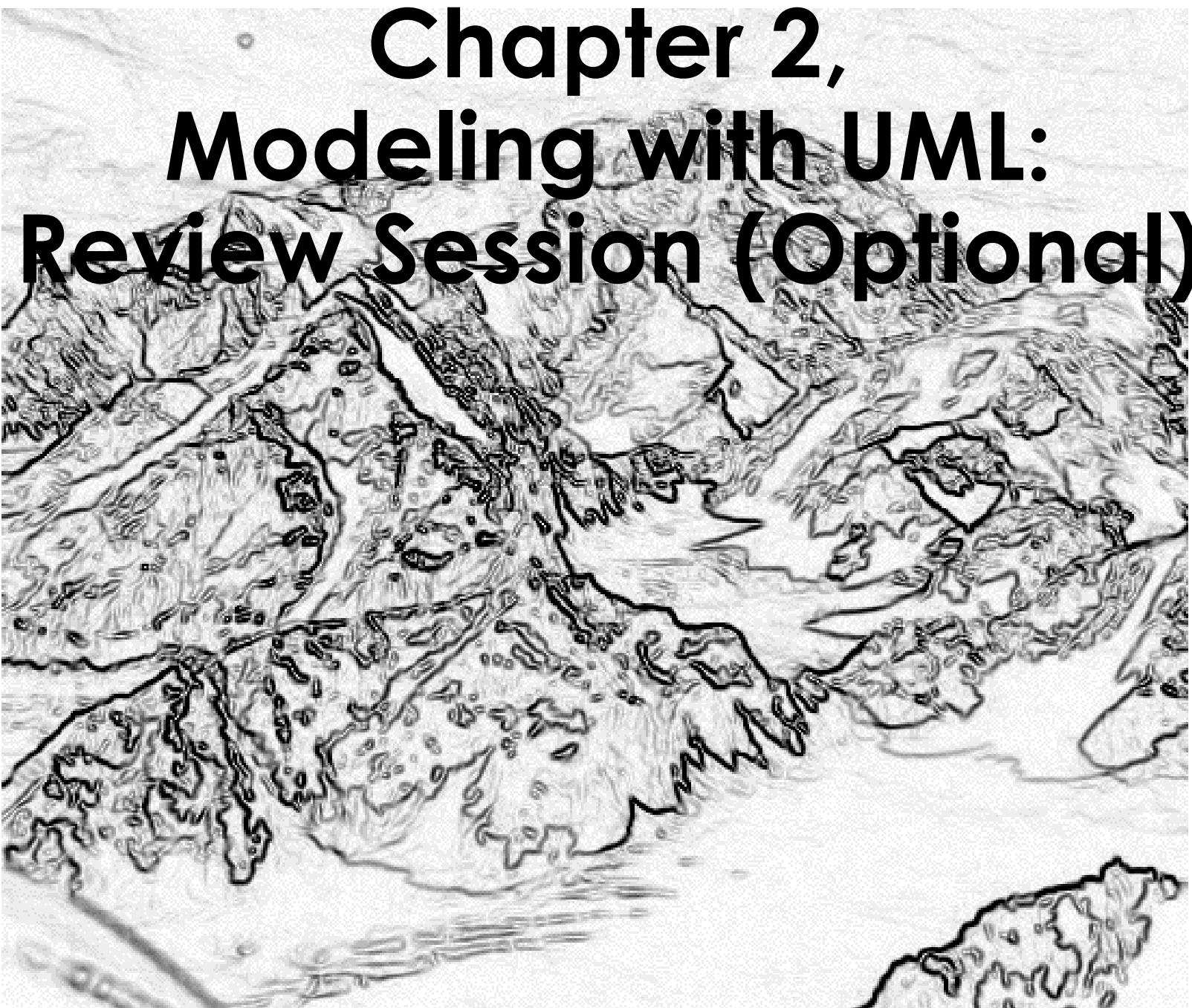
**(Focus on dataflow in a system)**



# UML Summary

- *UML provides a wide variety of notations for representing many aspects of software development*
  - Powerful, but complex
- *UML is a programming language*
  - Can be misused to generate unreadable models
  - Can be misunderstood when using too many exotic features
- *We concentrated on a few notations:*
  - Functional model: Use case diagram
  - Object model: class diagram
  - Dynamic model: sequence diagrams, statechart and activity diagrams

# Chapter 2, Modeling with UML: Review Session (Optional)



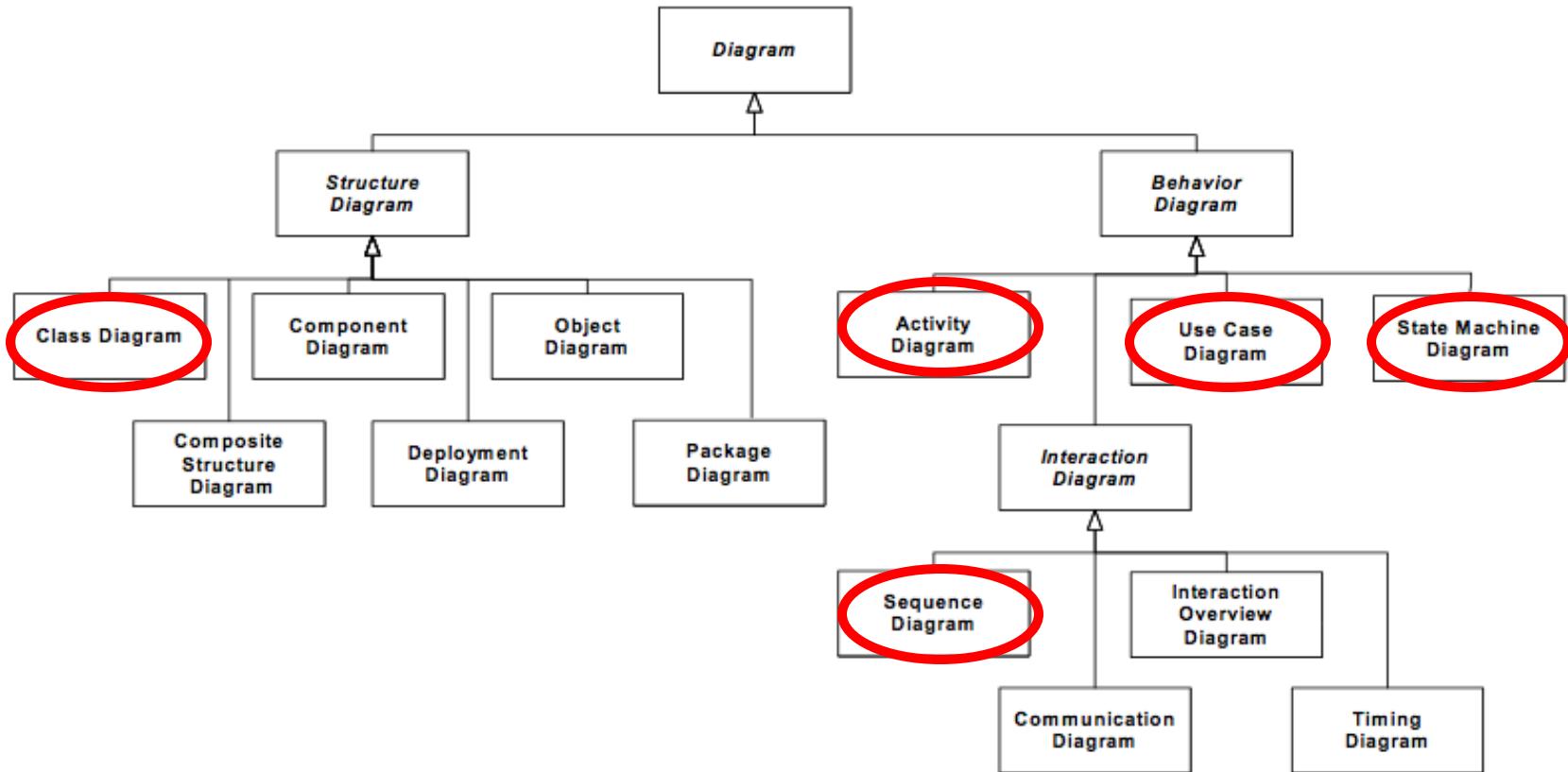
# Outline for this Week

- *Last Lecture: Modeling Functions, Structure and Behavior*
  - *Use case diagrams*
  - *Class diagrams*
  - *Sequence diagrams, State chart diagrams, Activity diagrams*
- *Today we review these concepts*
  - *Review: Why UML*
  - *Review of diagram notations*
- *Next Lecture:*
  - *Deployment diagrams*
  - *Stereotypes and Profiles*
  - *UML 2 Meta model*

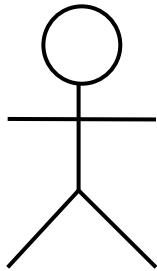
# We use Models to describe Software Systems

- *System model: Object model + functional model + dynamic model*
- *Object model: What is the structure of the system?*
  - UML Notation: Class diagrams
- *Functional model: What are the functions of the system?*
  - UML Notation: Use case diagrams
- *Dynamic model: How does the system react to external events?*
  - UML Notation: Sequence, State chart and Activity diagrams

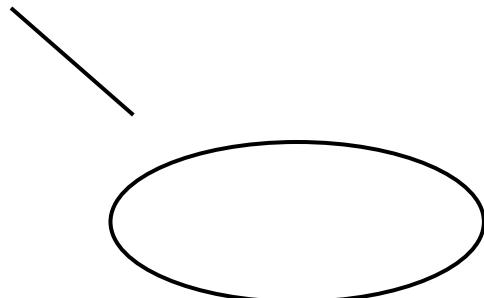
# Another view on UML Diagrams



# Review of Use Case Diagrams: 3 Important Terms



## Student



## DoHomework

*Used during requirements elicitation and analysis to represent behavior visible from the outside of the system*

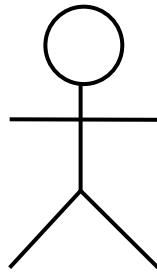
*An **actor** represents a role, that is, a type of user of the system*

*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.*

# Actor



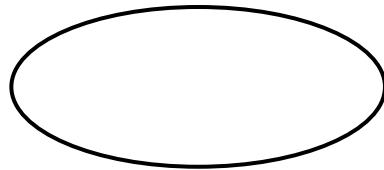
**Student**

Name

- *An actor is a model for an external entity which interacts with the system:*
  - *EndUser, Administrator*
  - *External system (Another system)*
  - *Physical environment (e.g. Weather)*
- *An actor has a unique name and an optional description*
- *Examples:*
  - **Student:** A studying person
  - **Teaching Assistant:** Member of teaching staff who supports the instructor.
  - **Random Number generator**

Optional  
Description

# Use Case

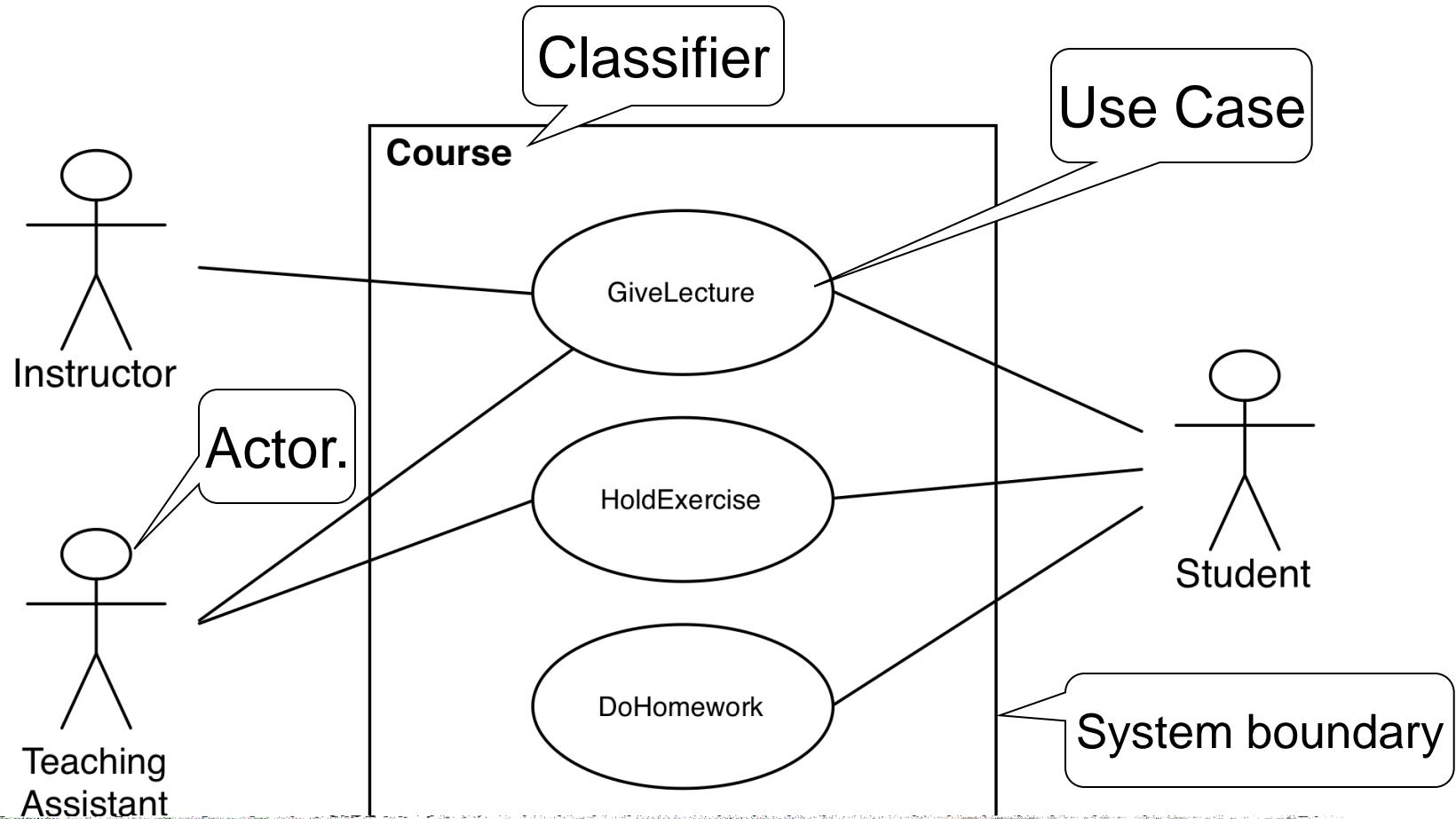


- A *use case represents a class of functionality provided by the system*
- *Use cases can be described textually, with a focus on the event flow between actor and system*

## Do Homework

- *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.

# Use Case Model

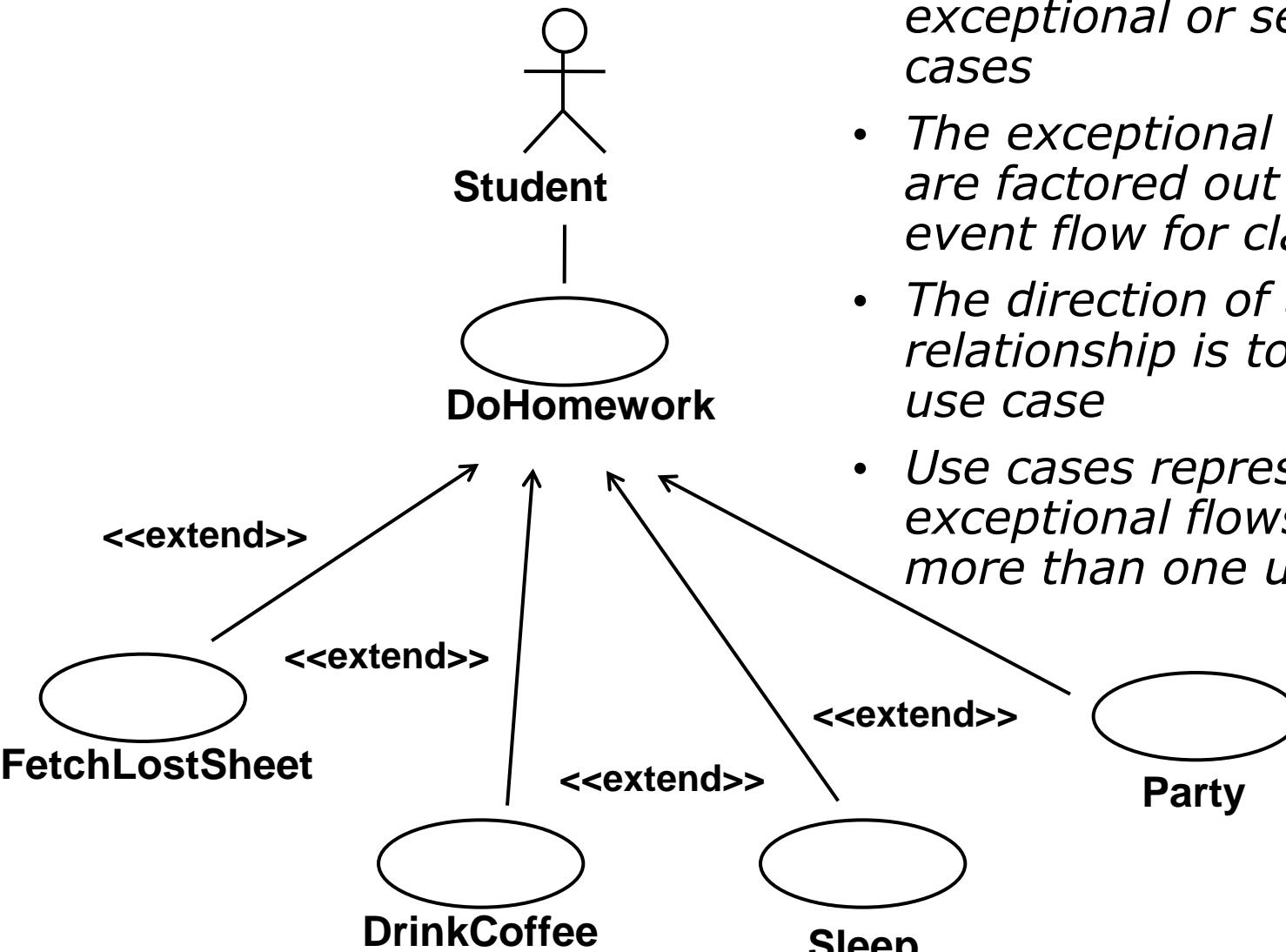


Use case diagrams represent the functionality of the system from user's point of view

# Uses Cases can be related

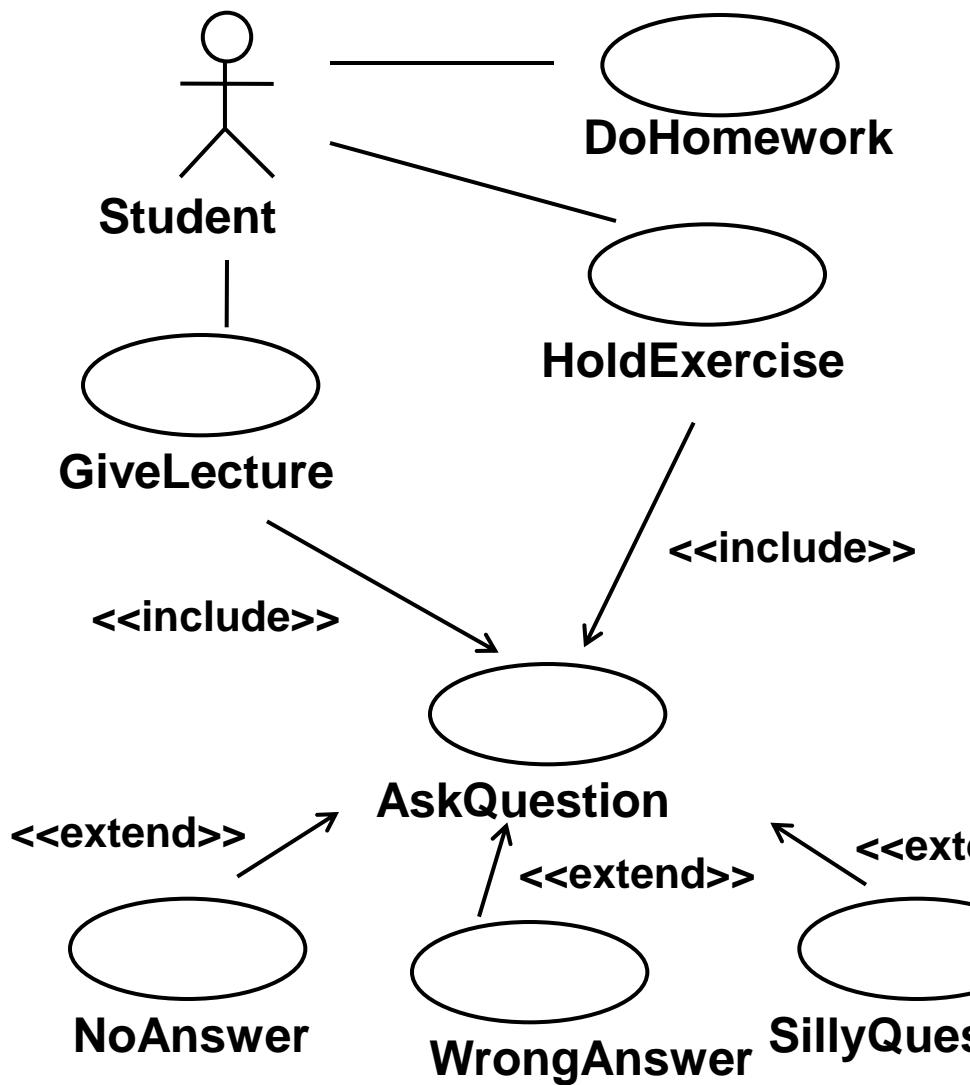
- *Extend Relationship*
  - *To represent seldom invoked use cases or exceptional functionality*
- *Include Relationship*
  - *To represent functional behavior common to more than one use case.*

# The <<extend>> Relationship



- <<extend>> 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 <<extend>> relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use case.

# The <<include>> Relationship



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

# Textual Use Case Description Example

1. *Name:* DoHomework

2. *Participating actor:* Student

3. *Entry condition:*

- Student *received exercise sheet*
- Student *is in good health*

4. *Exit condition:*

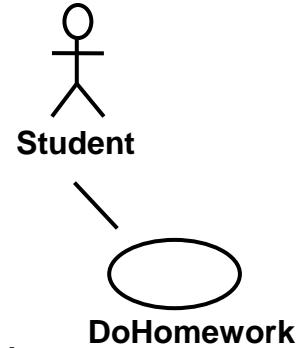
- Student *delivered solution*

5. *Flow of events:*

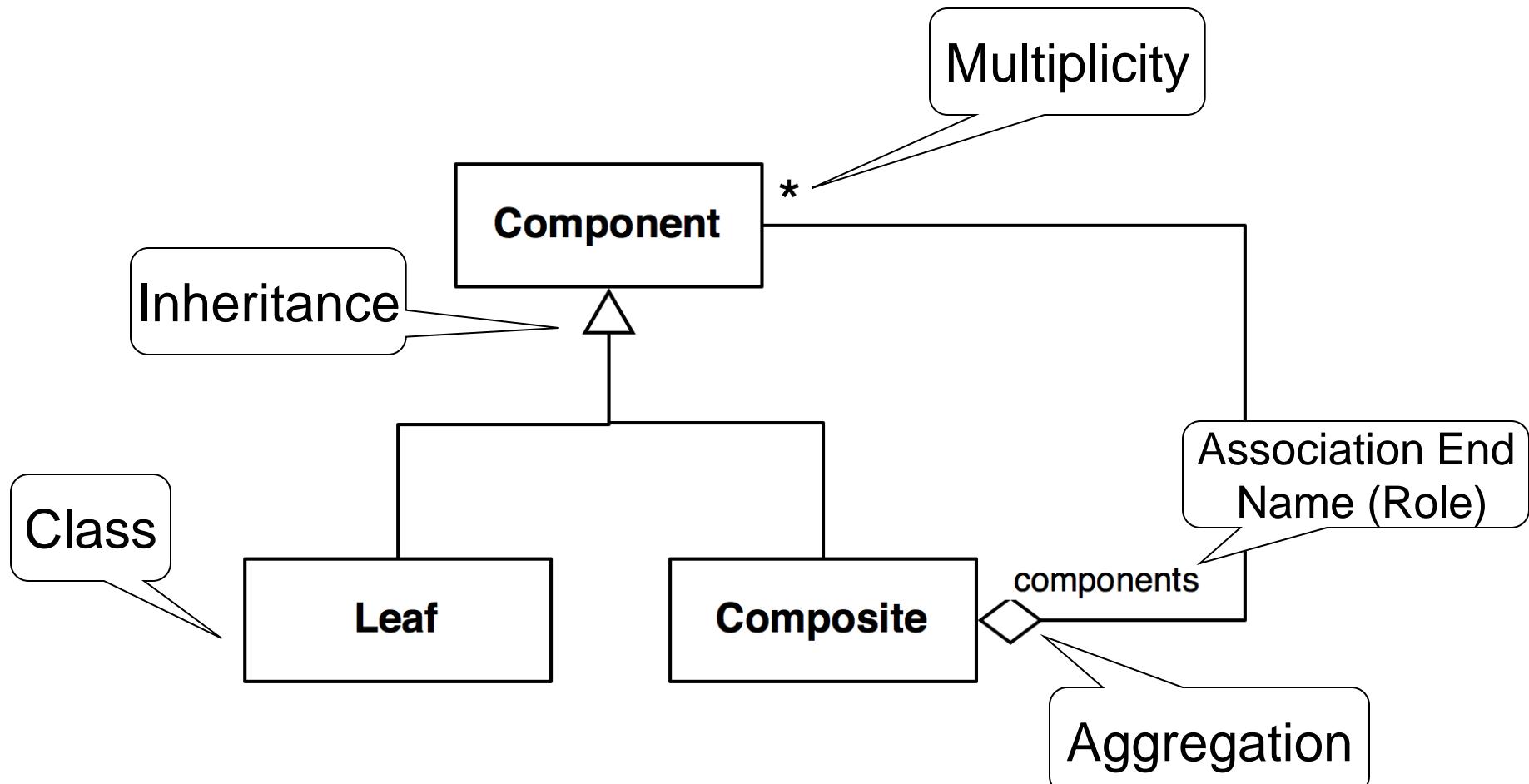
1. Student *fetches the exercise sheet*
2. Student *reads through the assignments*
3. Student *processes the assignments and types the solution in his Computer.*
4. Student *prints out the solution*
5. Student *delivers the solution in the following exercise*

6. *Special requirements:*

*None.*

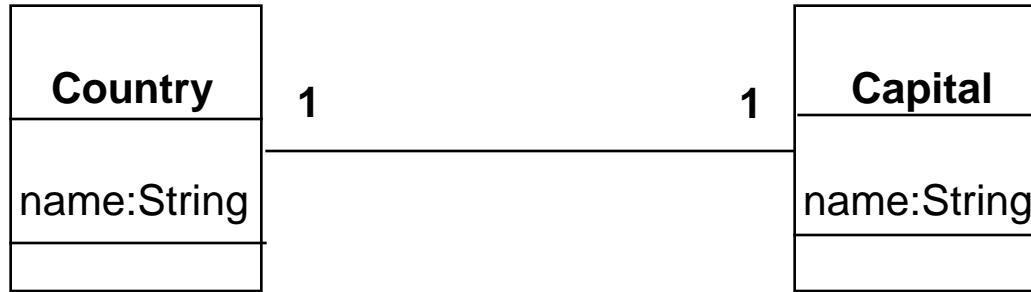


# Review of Class Diagrams



Class diagrams represent the structure of the system

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



1-to-1 association



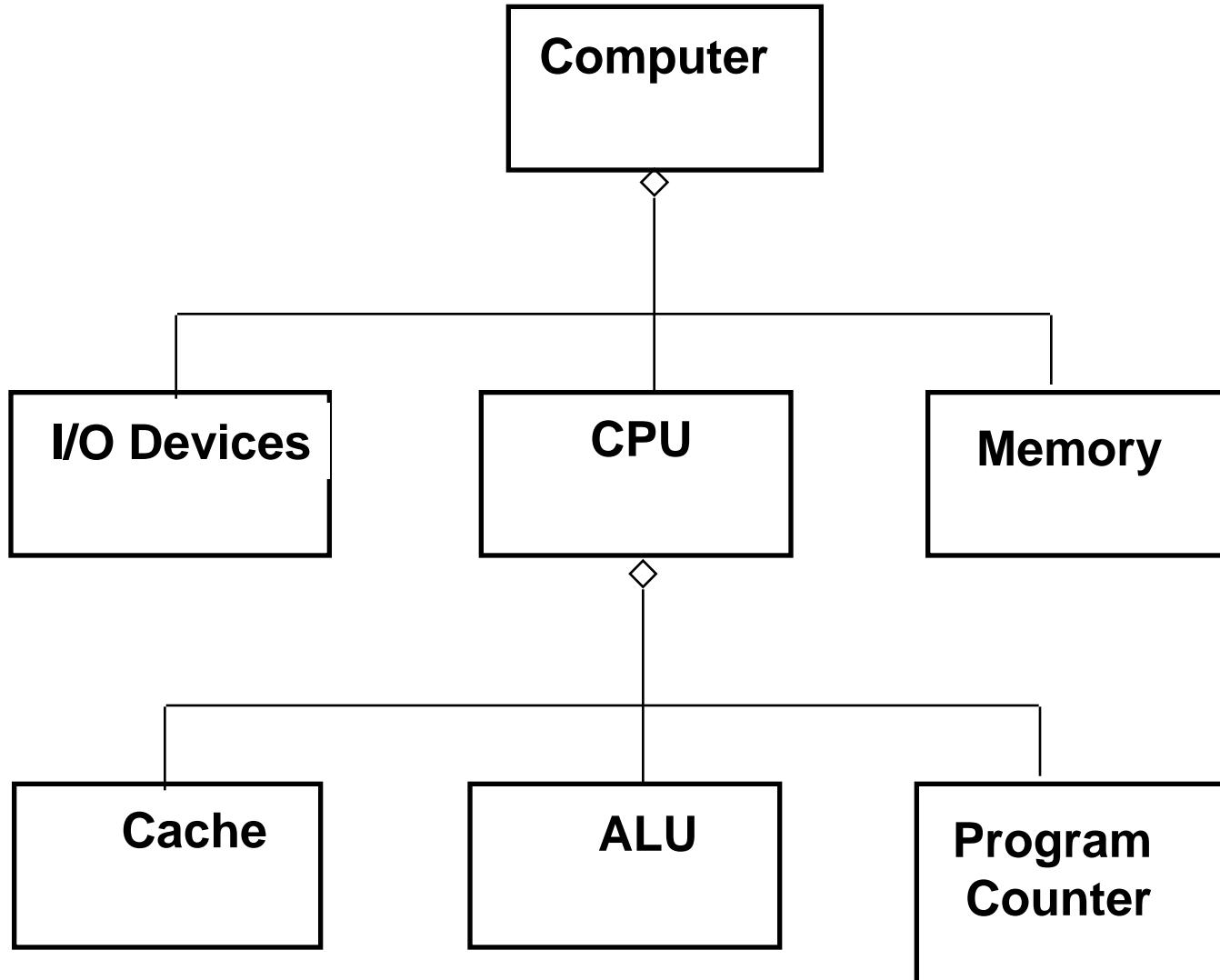
1-to-many association

# Many-to-many Associations



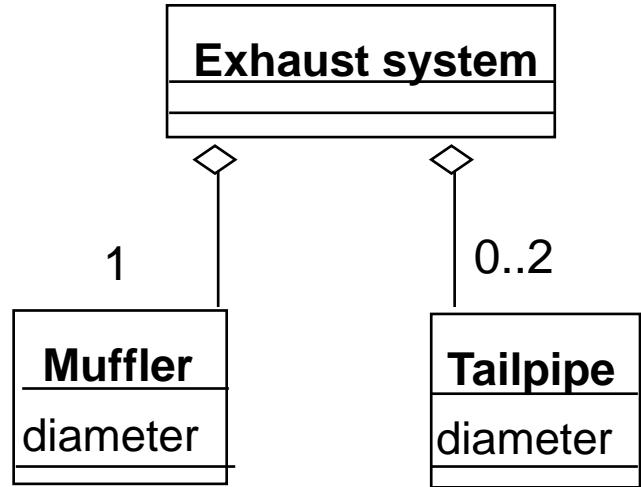
- *A stock exchange lists many companies.*
- *Each company is identified by a ticker symbol*

# Part-of Hierarchy (Aggregation)

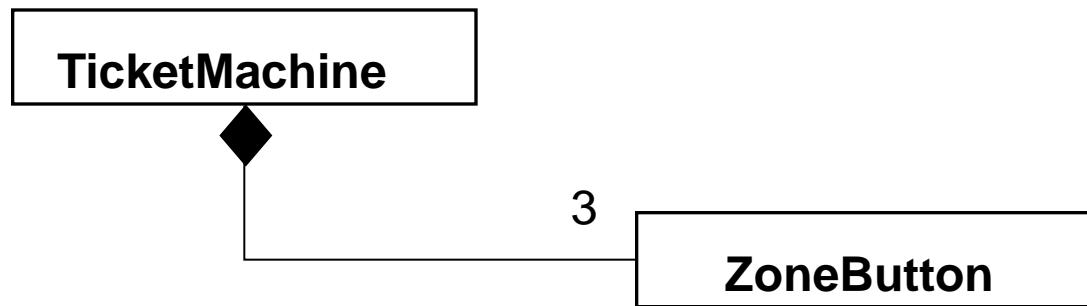


# Aggregation

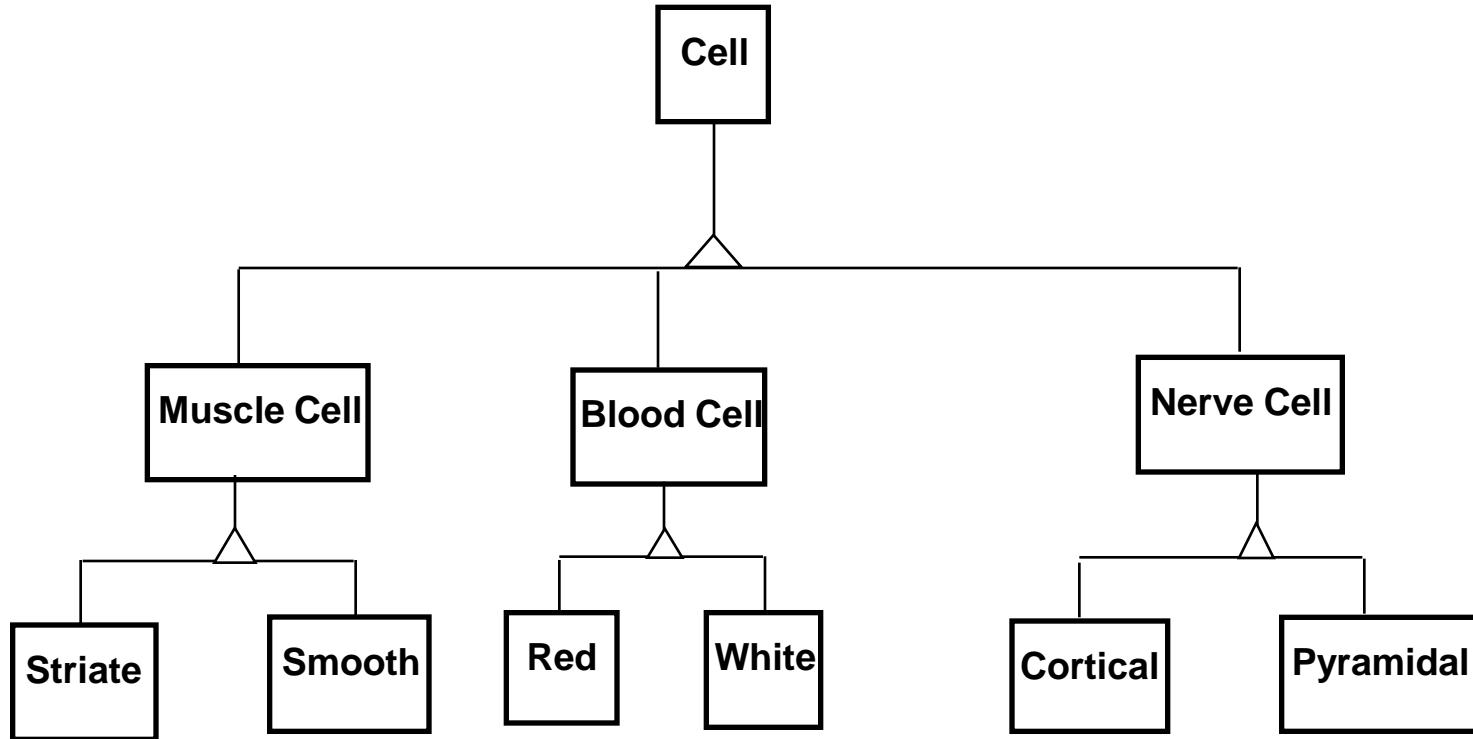
*Aggregation*



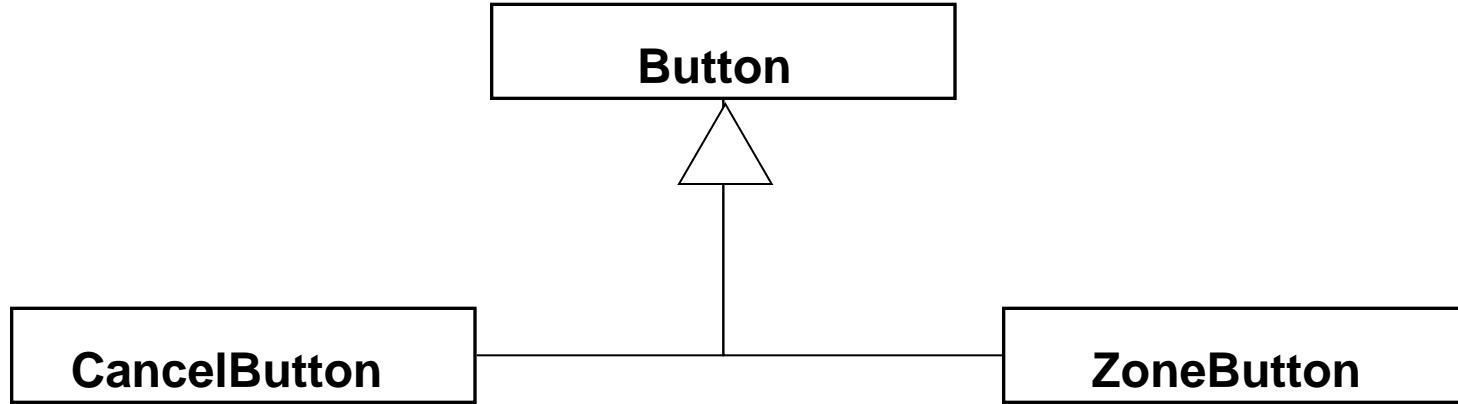
*Composition*



# Is-Kind-of Hierarchy (Taxonomy)

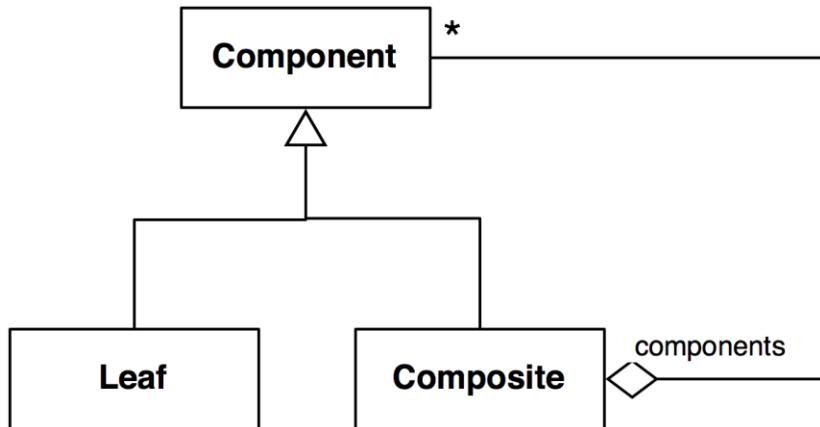


# Inheritance



- *Inheritance is another special case of an association denoting a “kind-of” hierarchy for describing taxonomies*

# Code Generation from UML to Java I



```
public class Component{ }
```

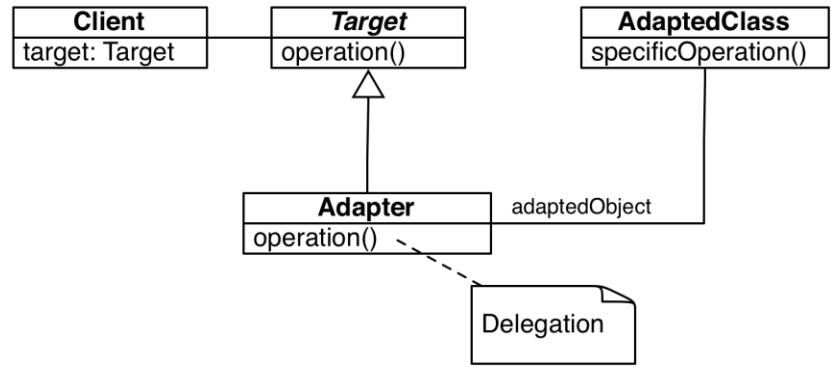
```
public class Leaf extends Component{ }
```

```
public class Composite extends Component{  
    private Collection<Component>  
    components;  
}
```

# Code Generation from UML to Java II

```
public abstract class Target{  
    public ... operation(); }
```

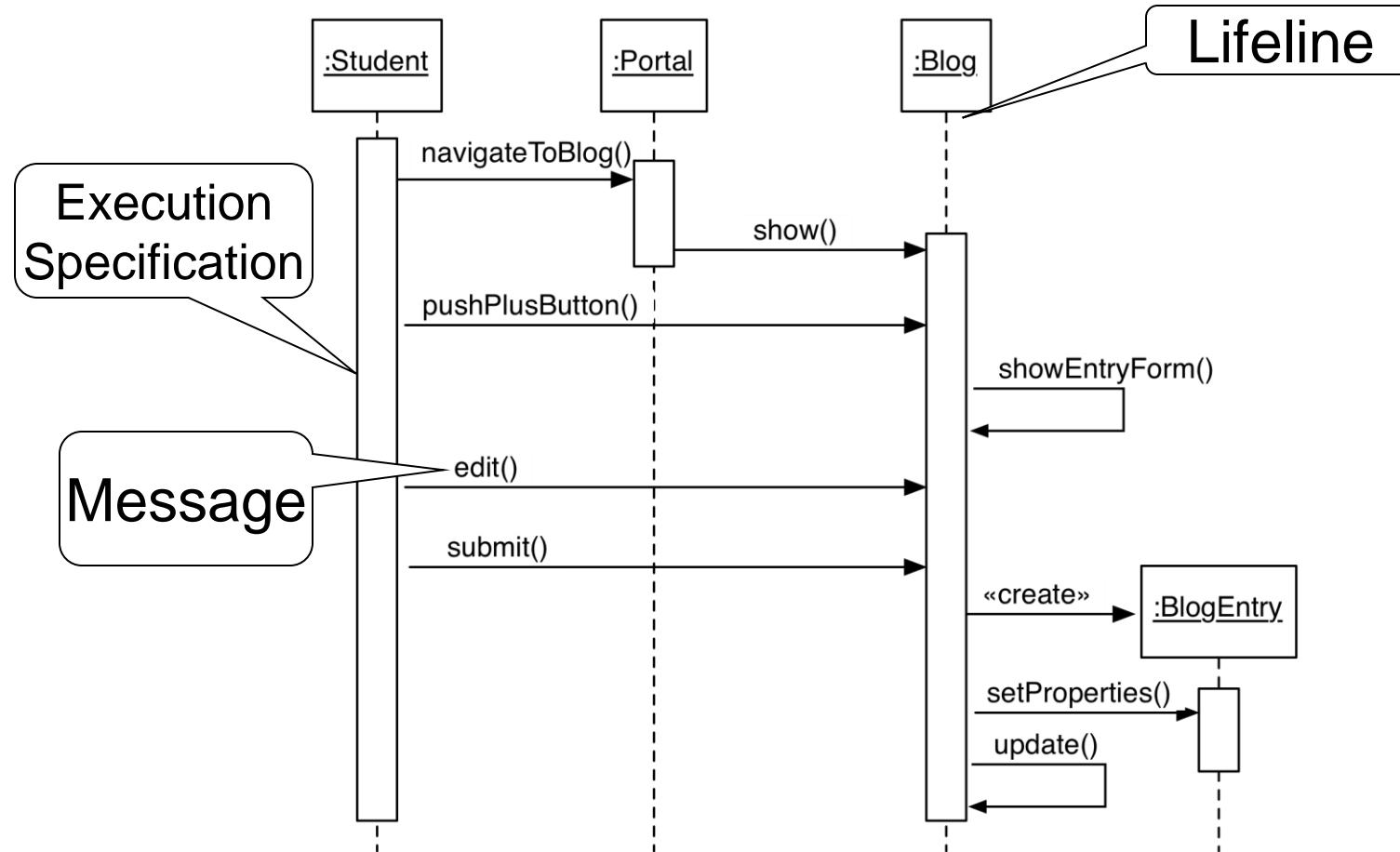
```
public class Adapter extends Target {  
    private AdaptedClass adaptedObject;  
    public ... operation(){  
        adaptedObject.specificOperation();  
    } }
```



# Where are we?

- ✓ *What is UML?*
- ✓ *Review functional modeling*
  - ✓ *Use case diagram*
- ✓ *Review object modeling*
  - ✓ *Class diagram*
- *Review dynamic modeling*
  - *Sequence diagram*
  - *State chart diagram*
  - *Activity diagram*

# Sequence diagram: Basic Notation



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

# Lifeline and Execution Specification

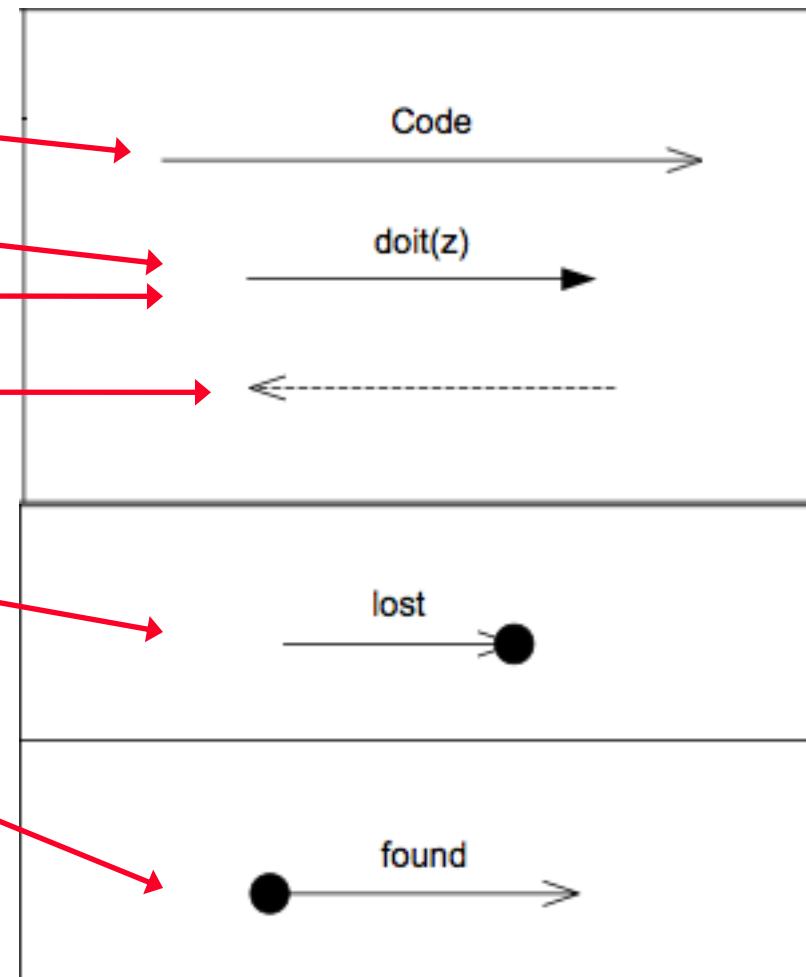
- A **lifeline** represents an *individual participant (or object) in the interaction*
- A *lifeline is shown using a symbol that consists of a rectangle forming its “head” followed by a vertical line (which may be dashed) that represents the lifetime of the participant*
- An **execution specification** specifies a *behavior or interaction within the lifeline*
- An *execution specification is represented as a thin rectangle on the lifeline.*

# Messages

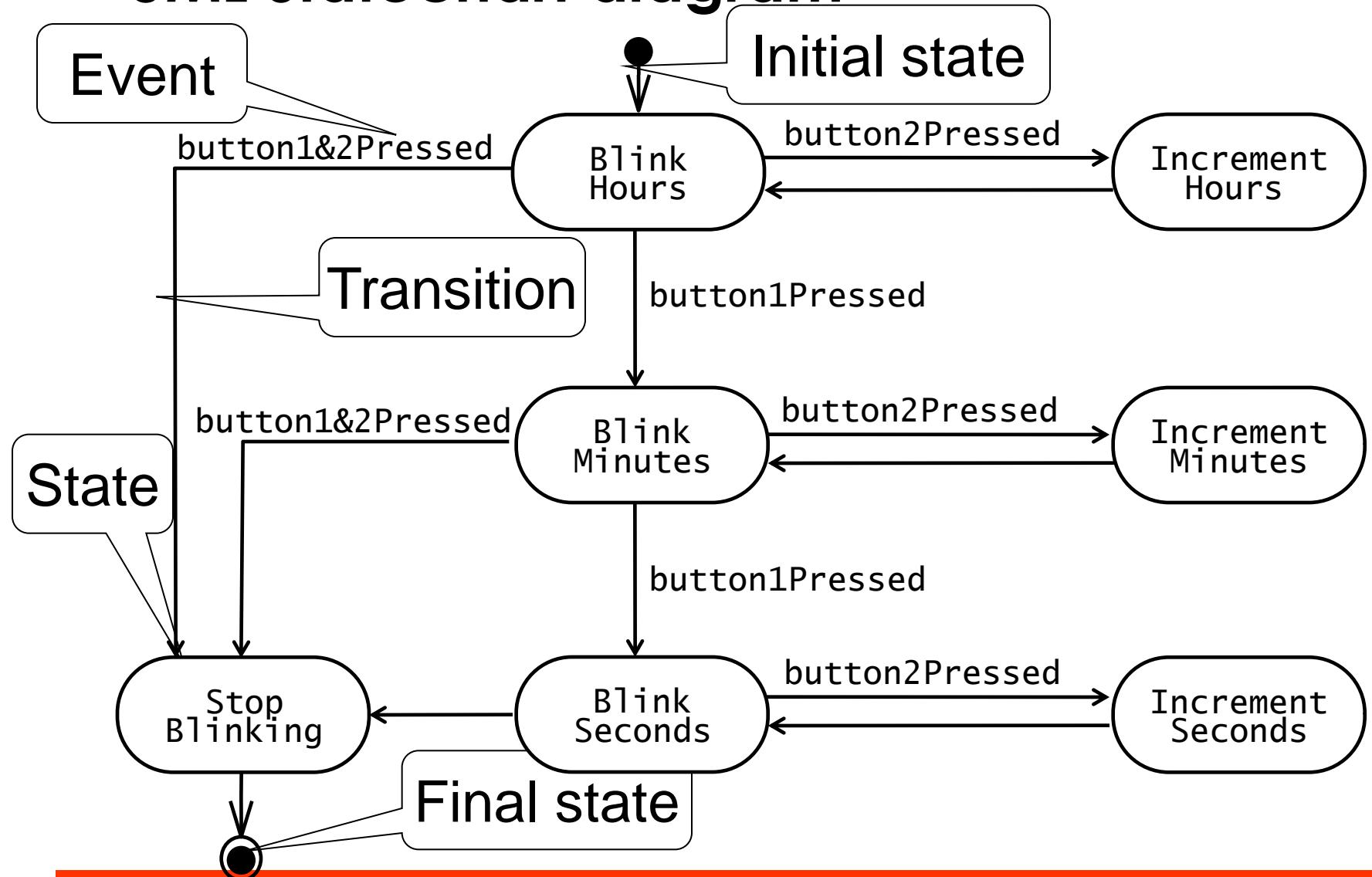
- *Define a particular communication between lifelines of an interaction*
- *Examples of communication*
  - *raising a signal*
  - *invoking an operation*
  - *creating or destroying an instance*
- *Specify (implicitly) sender and receiver*
- *are shown as a line from the sender to the receiver*
- *Form of line and arrowhead reflect message properties*

# Message Types

- *Asynchronous*
- *Synchronous*
- *Call and Object creation*
- *Reply*
- *Lost*
- *Found*



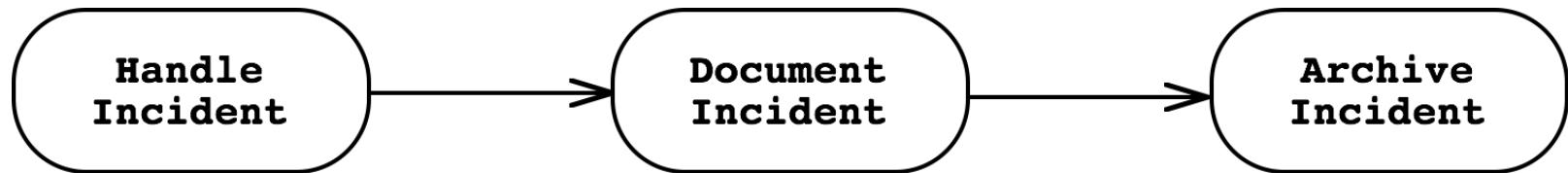
# UML Statechart diagram



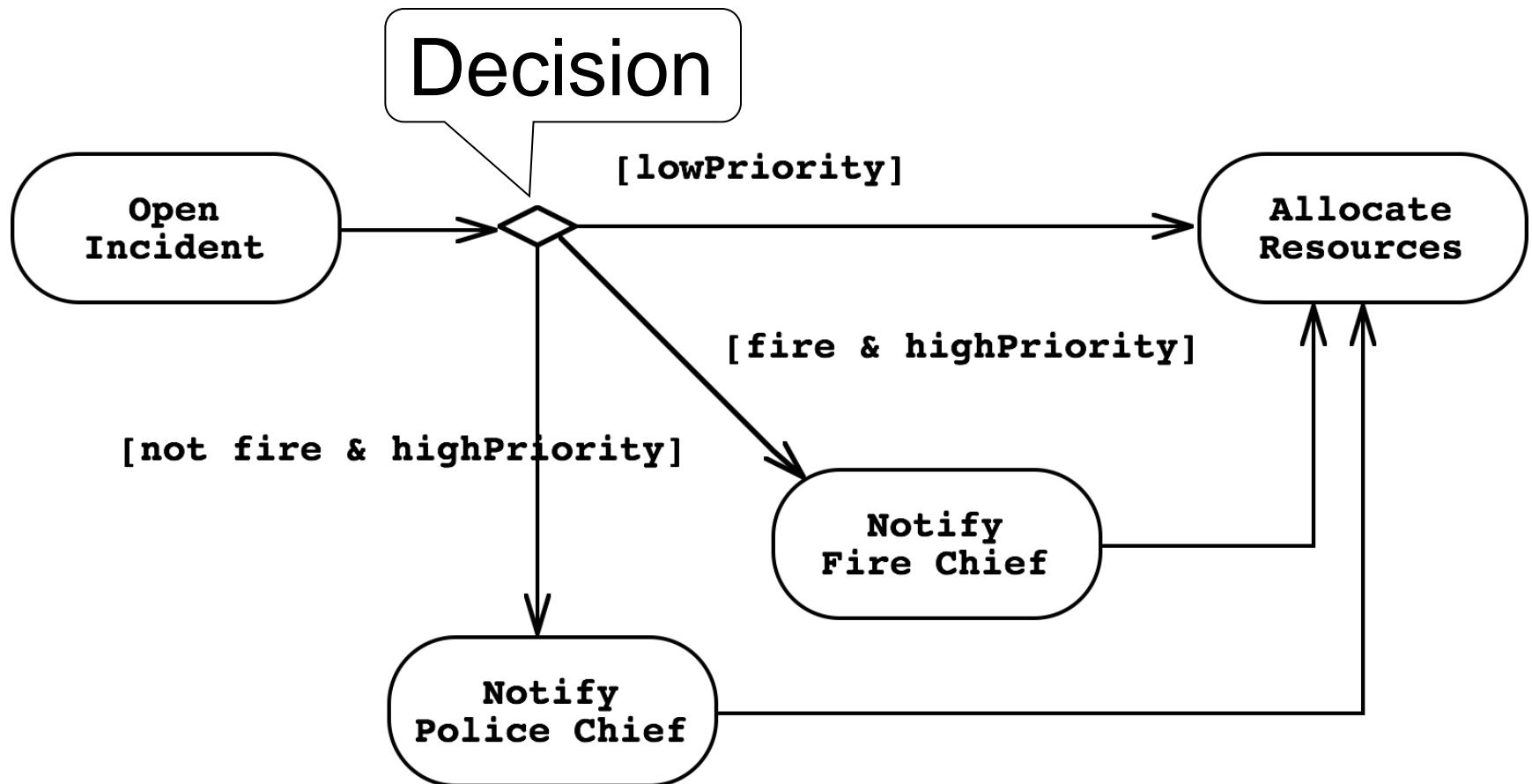
Represents behavior of a *single object* with interesting dynamic behavior.

# UML 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.*

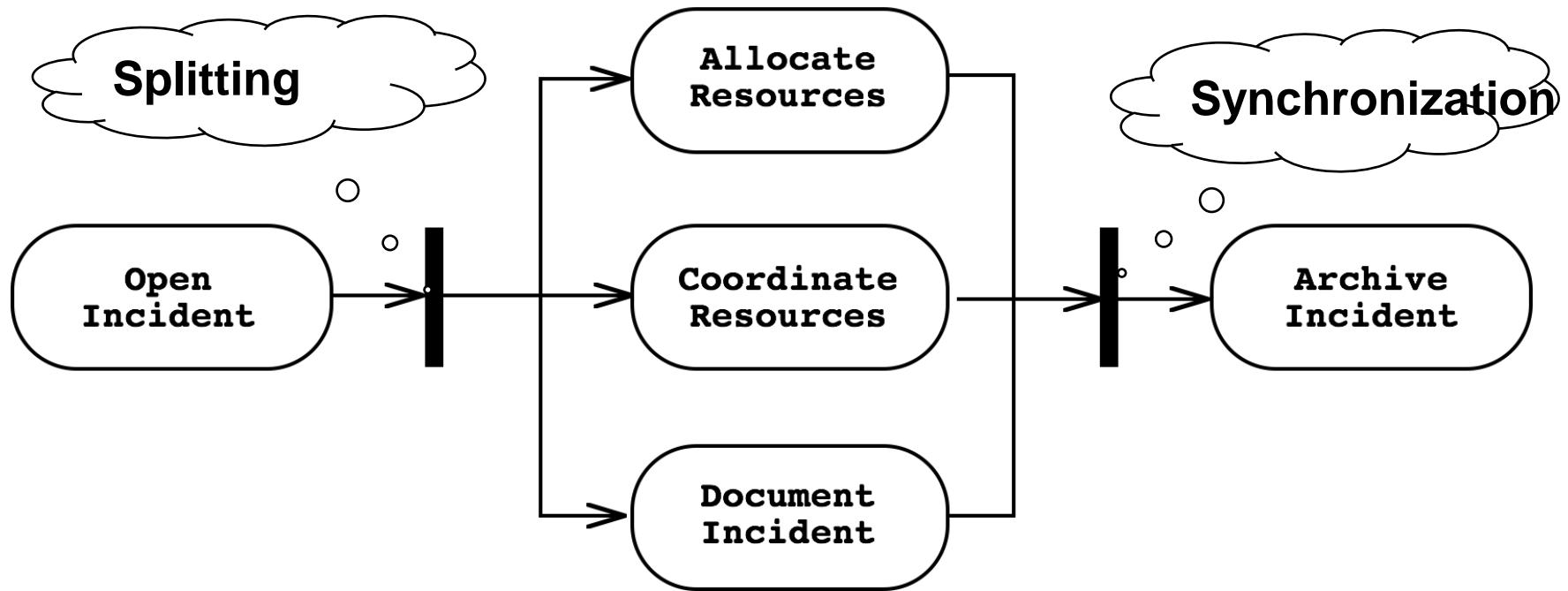


# Activity Diagrams allow to model Decisions

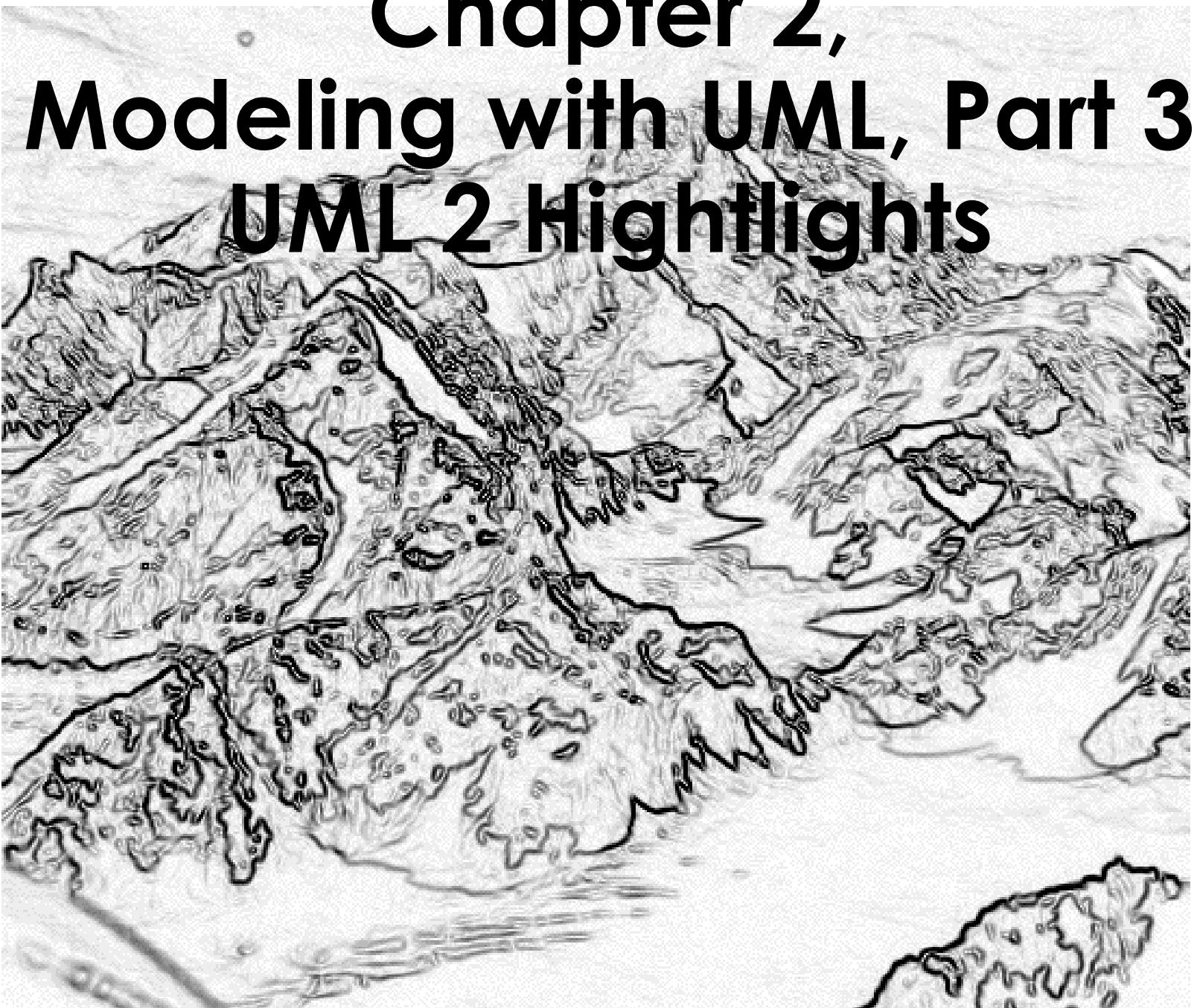


# Activity Diagrams can model Concurrency

- *Synchronization of multiple activities*
- *Splitting the flow of control into multiple threads*



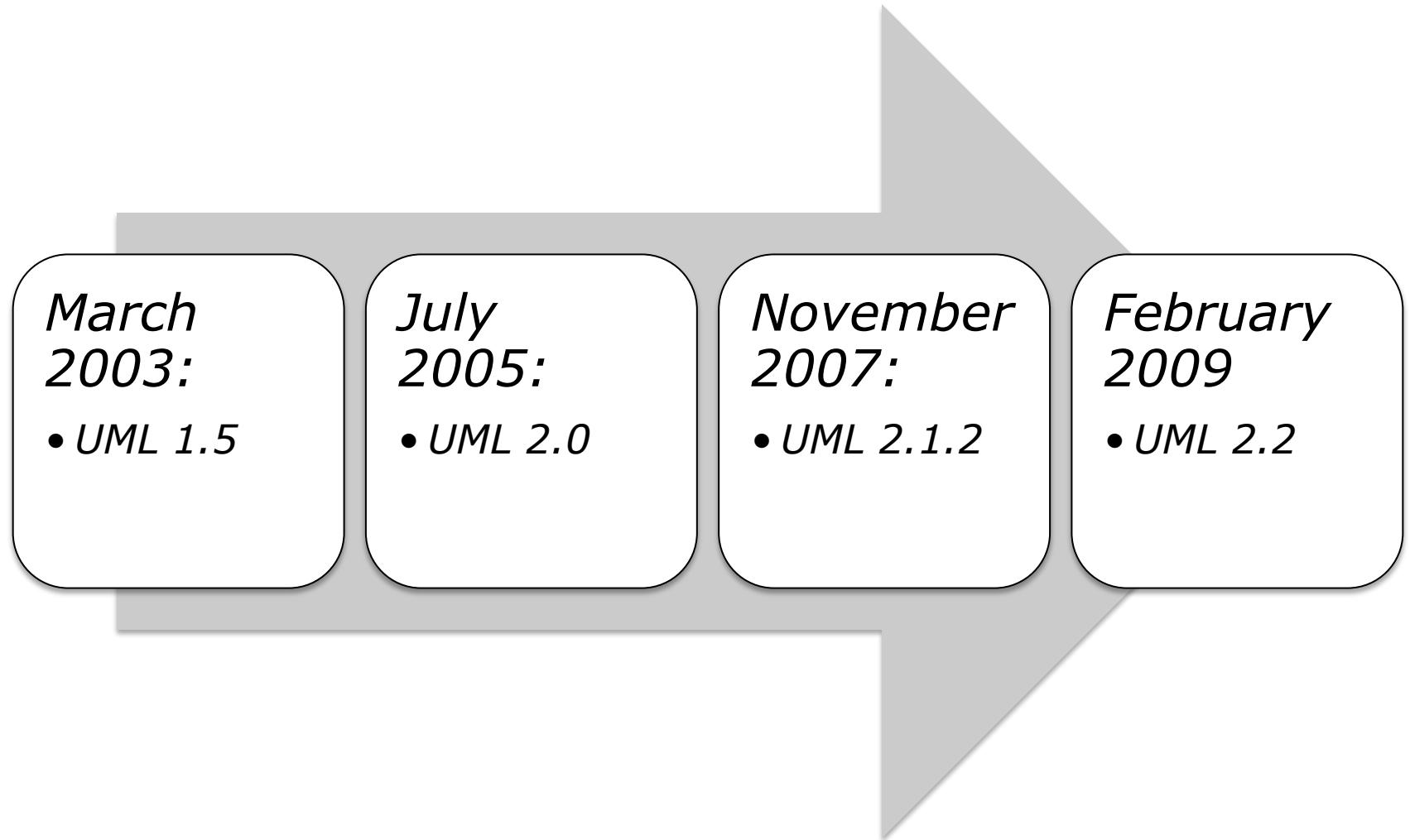
# Chapter 2, Modeling with UML, Part 3 UML2 Highlights



# Outline for today

- *UML 2: UML is a living language*
- *Overview of important changes in UML 2*
  - *Frames and nesting*
  - *Activity diagrams*
  - *Deployment diagrams*
  - *Sequence diagrams*
  - *Profiles, Stereotypes*
  - *UML Metamodel*

# Recent History of UML



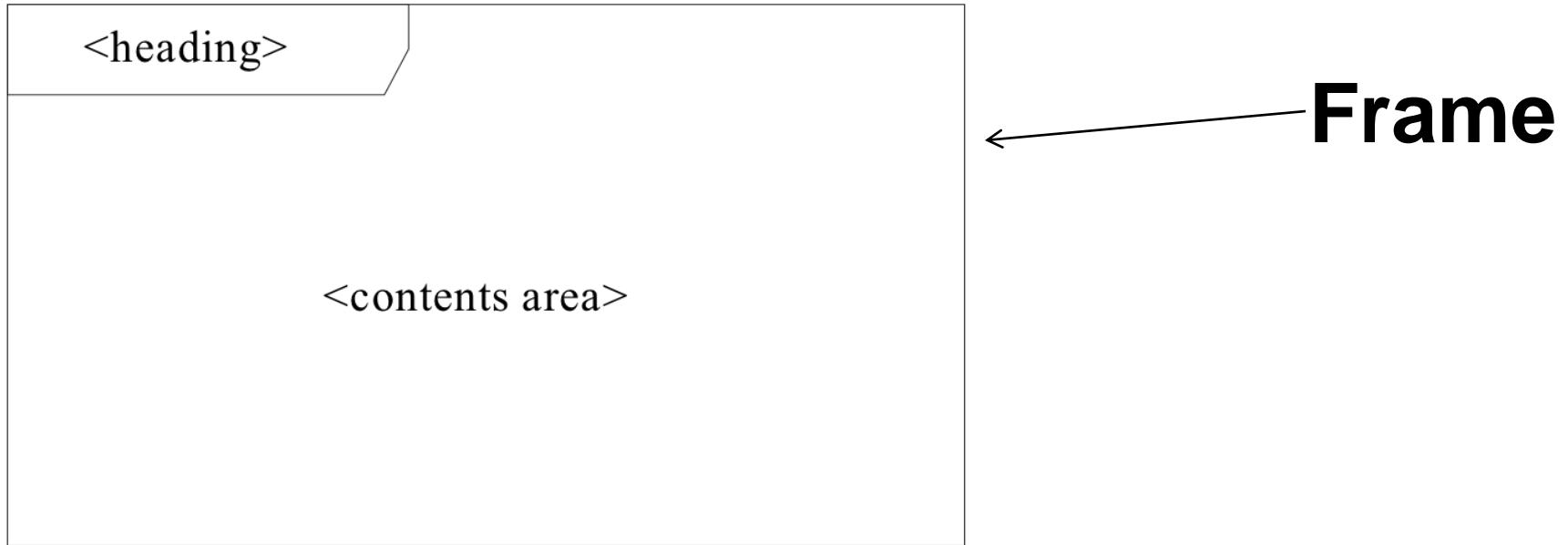
- *UML now usable with Model Driven Architecture (MDA)*
  - *Better support for the automatic transformation of a Platform Independent Model (PIM) into a Platform Specific Model (PSM)*
- *UML 2 is based on a meta model*
  - *Meta Object Facility (MOF)*

# Changes in UML 2

- *Frames and nesting*
- *Changes in diagram notation:*
- *Activity diagram*
- *Deployment diagram*
- *Sequence diagram*
- *Profiles and stereotypes*
- *New diagram types:*
- *Composite structure diagrams*
- *Timing diagrams*
- *Not covered in this lecture*

# Notation for all UML 2 Diagrams

- Mandatory is now the **contents area**
- Optional: The contents area can be surrounded with **frame** and a **heading**



*<heading> ::= [<**diagram kind**>]<name>[<parameters>]*

# Diagram Kinds

Activity diagram

Class diagram

Component diagram

Interaction diagram

Package

state machine diagram

Use case diagram

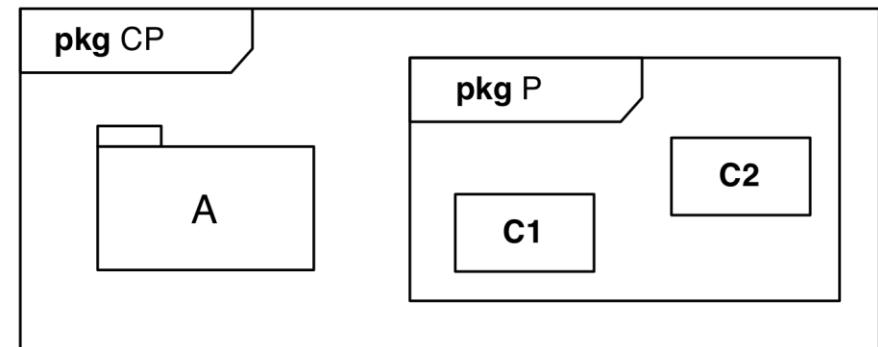
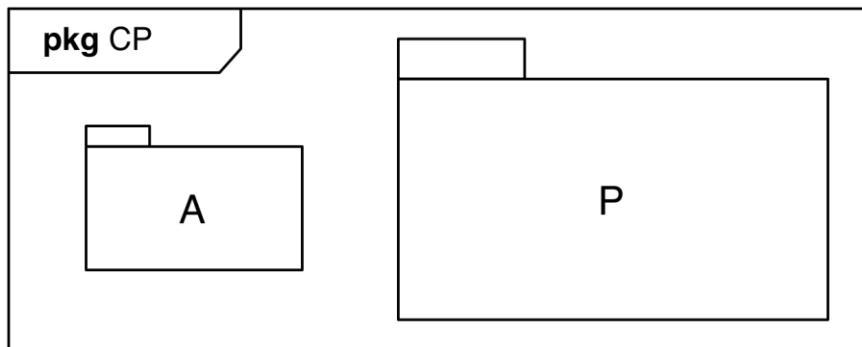
- The following abbreviations are usually used for diagram frames:
- **act** (for activity frames)
- **cmp** (for component frames)
- **sd** (for interaction frames)
- **pkg** (for package frames)
- **stm** (for state machine frames)
- **uc** (for use case frames)

Why are interaction frames abbreviated with “sd”?

Historical: sequence **diagram**

# Nested Diagrams

- *UML 2 supports nested diagrams*
  - e.g. an activity diagram inside a class
- *The frame concept visually groups elements that belong to one diagram*

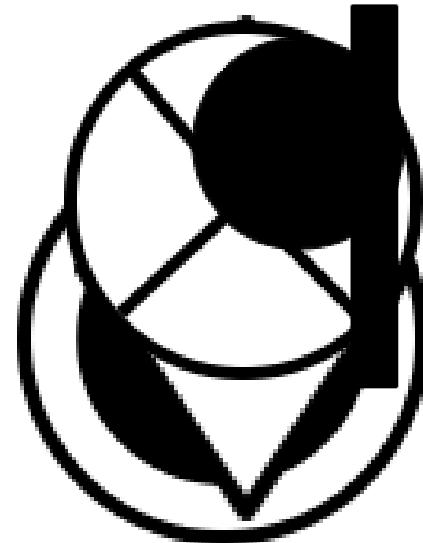


# UML 2 Activity Diagrams

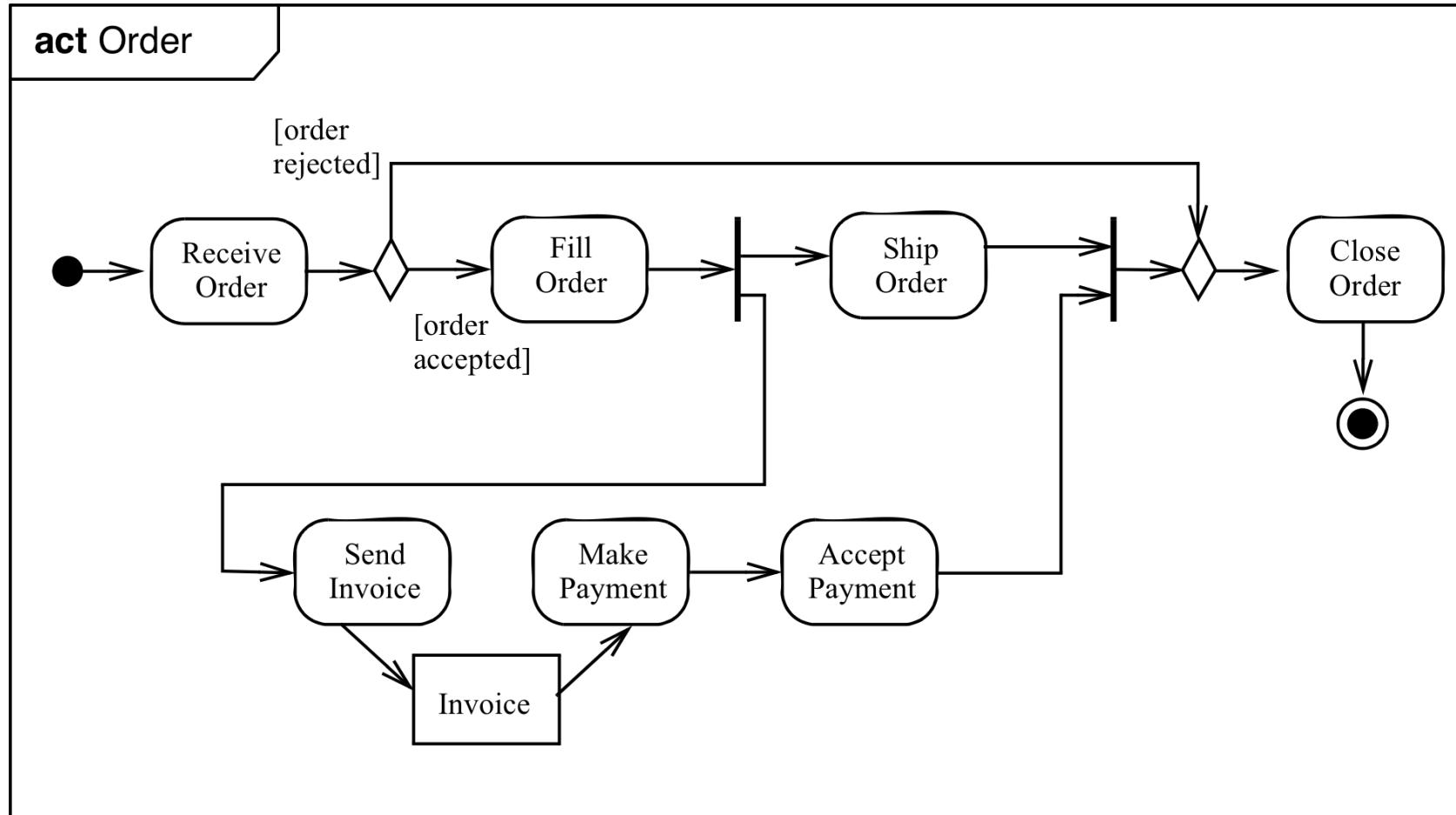
- An activity diagram consists of nodes and edges
- There are **three** types of activity nodes
  - Control nodes
    - Executable nodes
      - Most prominent: **Action**
      - Object nodes
        - E.g. a document
  - An **edge** is a directed connection between nodes
    - There are two types of edges
      - Control flow edges
      - Object flow edges

# Control Nodes in an Activity Diagram

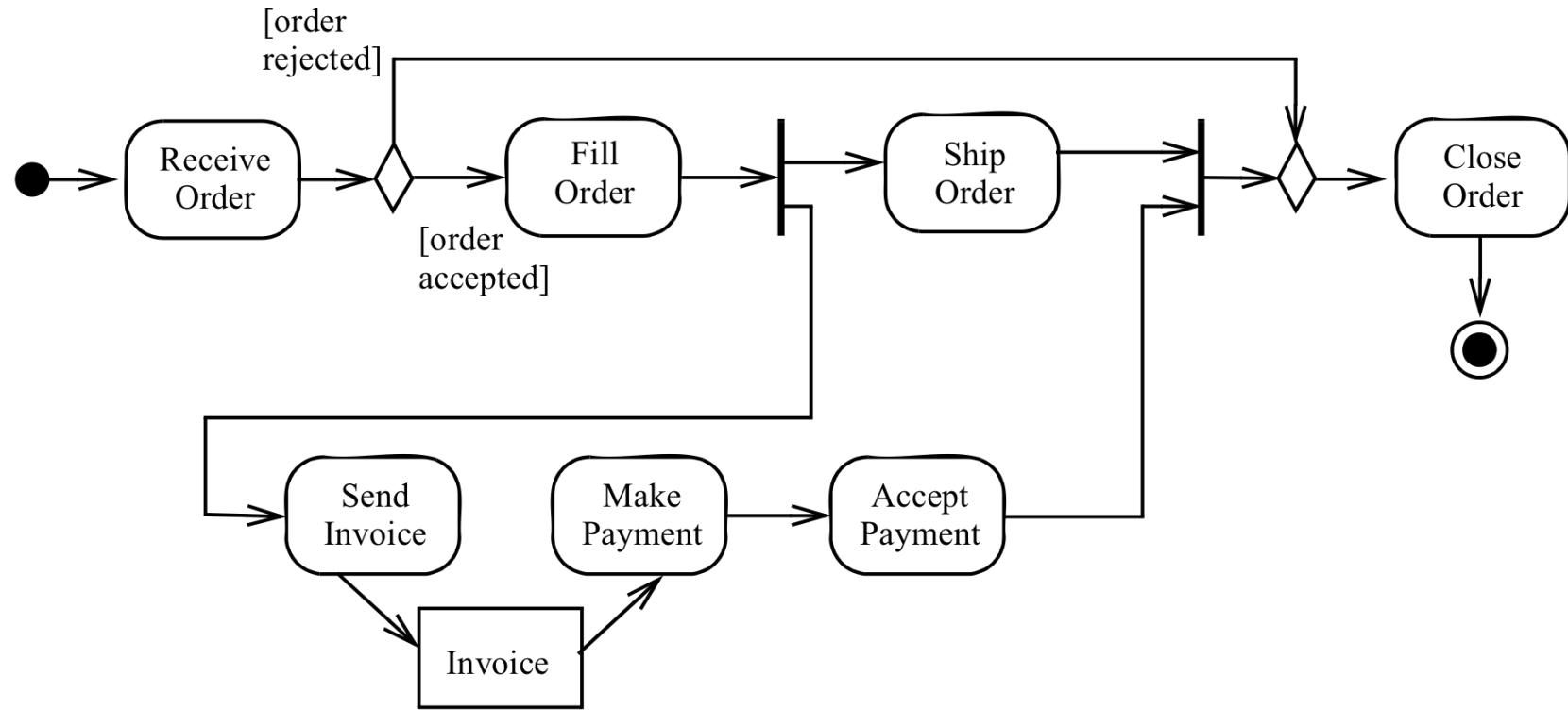
- *Initial node*
- *Final node*
  - *Activity final node*
  - *Flow final node*
- *Fork node*
- *Join node*
- *Merge node*
- *Decision node*



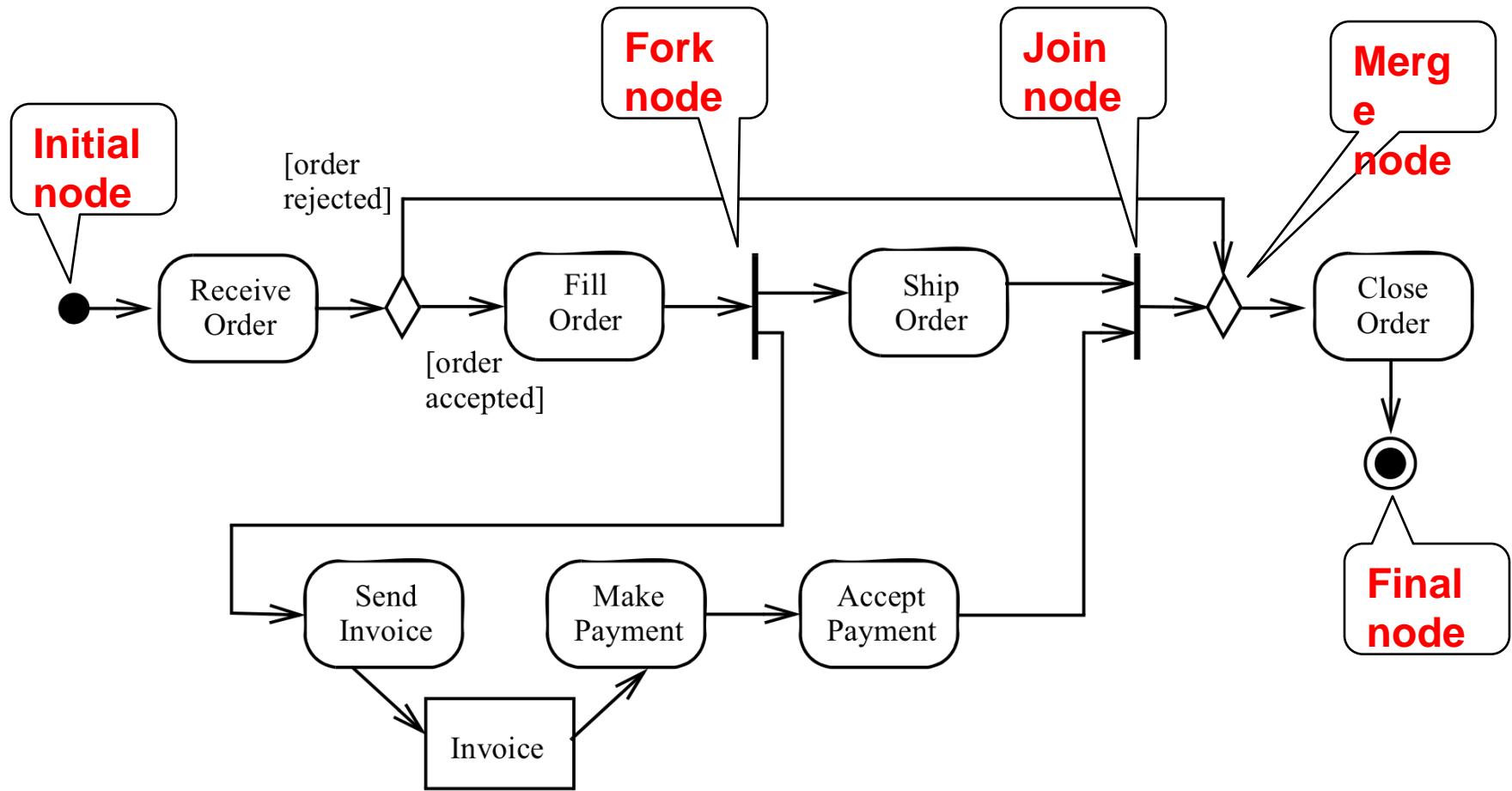
# Example of a Activity Diagram with a Frame



# The Activity Diagram without Frame



# Activity Diagram Example



# Activity Diagram: Activity Nodes & Edges

- An activity diagram consists of nodes and edges
- There are **three** types of activity nodes
  - ✓ Control nodes
  - Executable nodes
    - Most prominent: **Action**
  - Object nodes
    - E.g. a document
- An **edge** is a directed connection between nodes
  - There are two types of edges
    - Control flow edges
    - Object flow edges

# Action Nodes and Object Nodes

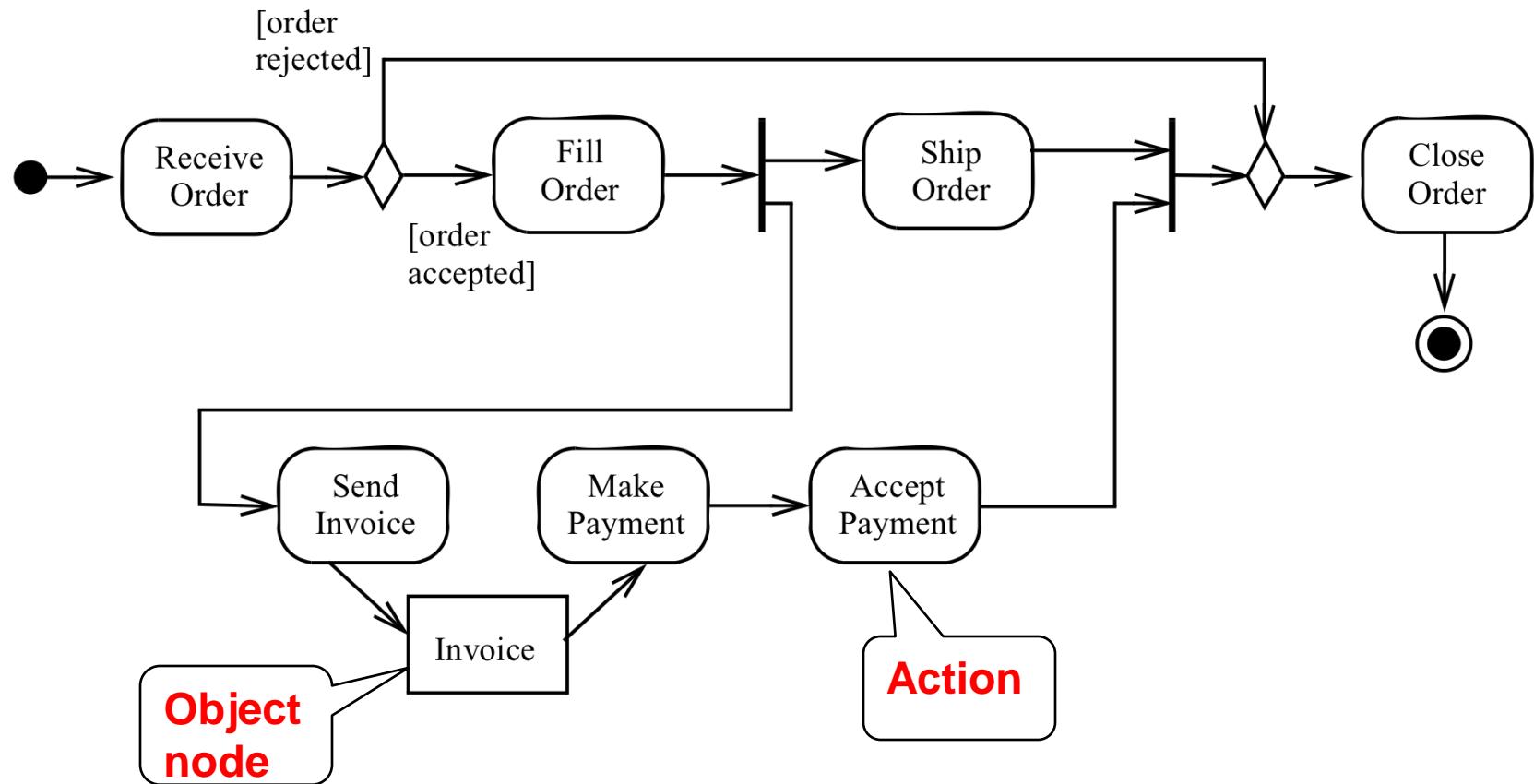
- *Action Node*
- *Object Node*



- An *action* is part of an activity which has local pre- and post conditions
- *Historical Remark:*
  - In UML 1 an action was the operation on the transition of a state machine.



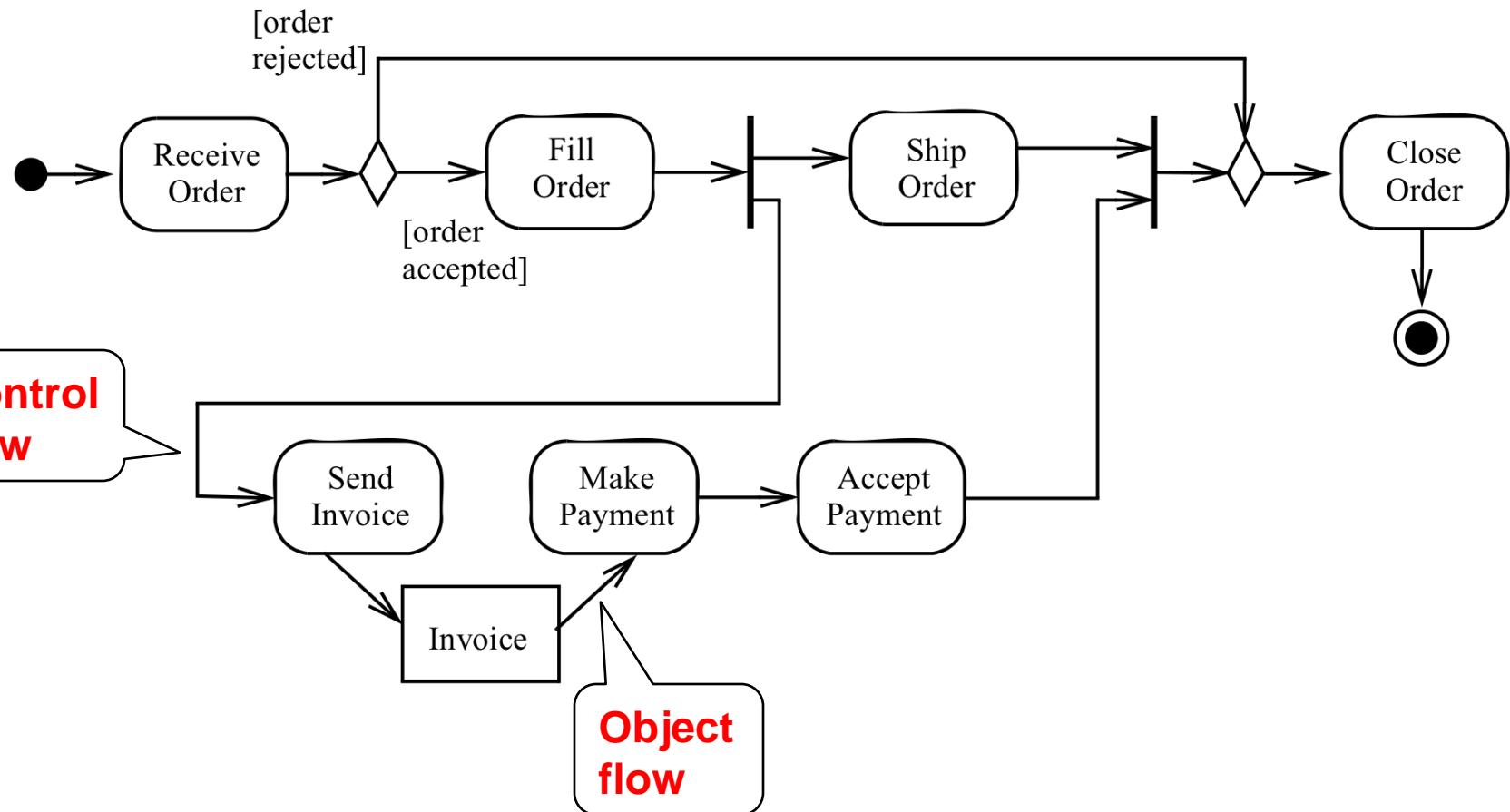
# Activity Diagram Example



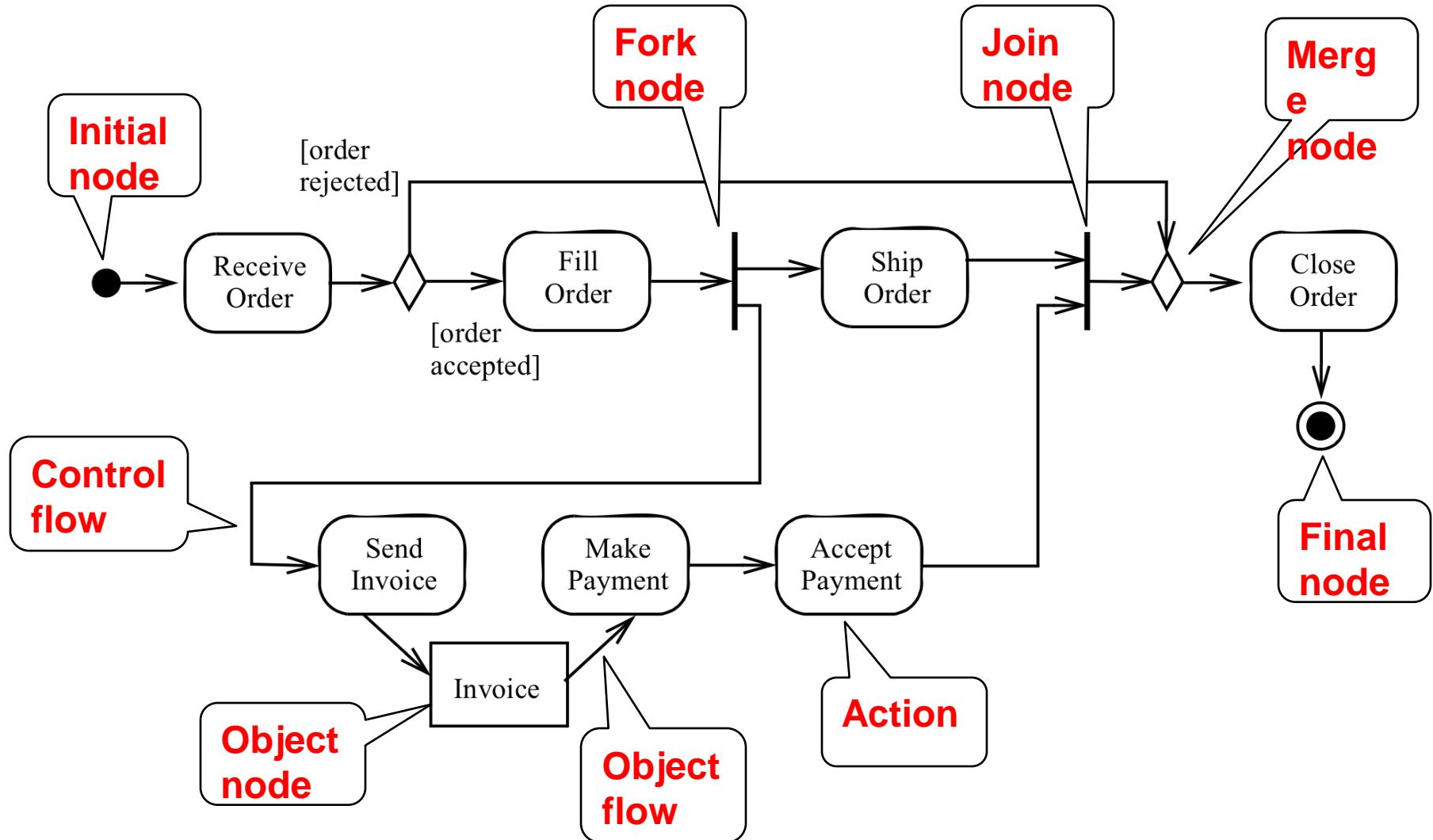
# Activity Diagram: Activity Nodes & Edges

- An activity diagram consists of nodes and edges
- There are **three** types of activity nodes
  - ✓ Control nodes
  - ✓ Executable nodes
    - Most prominent: **Action**
  - ✓ Object nodes
    - E.g. a document
- An **edge** is a directed connection between nodes
  - ➡ There are two types of edges
    - Control flow edges
    - Object flow edges

# Activity Diagram Example

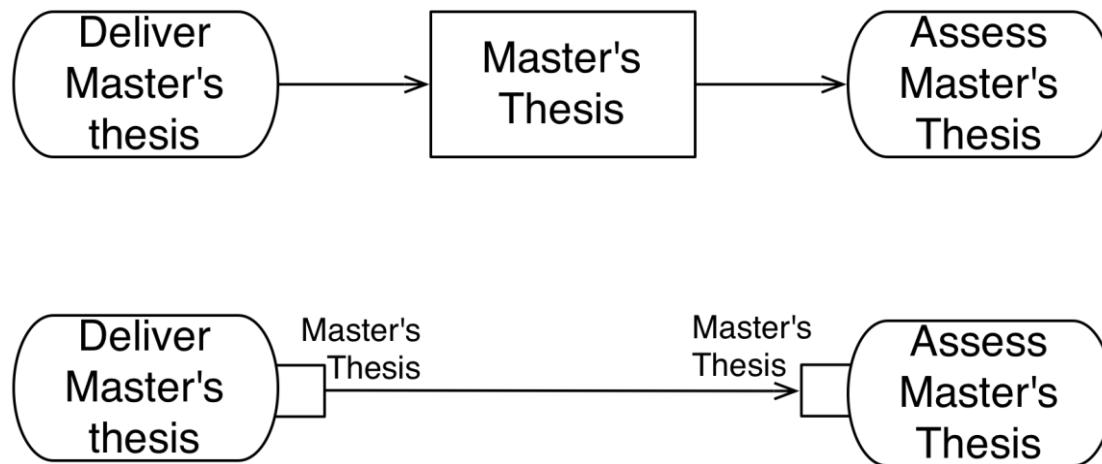


# Summary: Activity Diagram Example



# Activity Diagram: Pins

- *Pin: Abbreviated notation for an object node*
  - *Different notations with same semantics*
  - *Both notations define object flow in an activity*

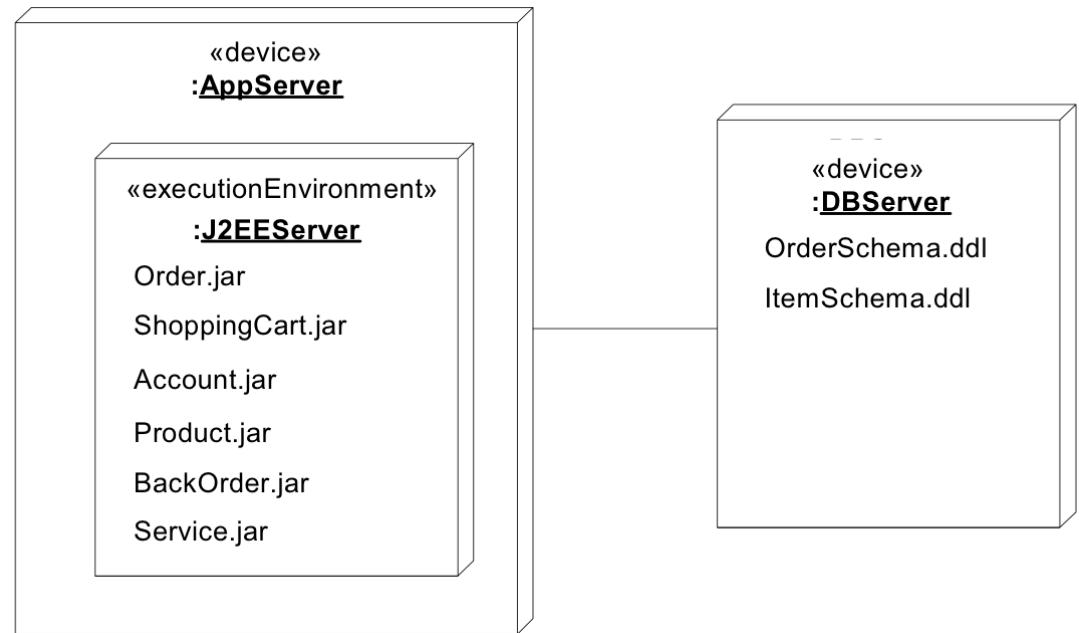


# Outline for today

- ✓ *UML 2: UML is a living language*
- ✓ *Overview of important changes in UML 2*
  - ✓ *Frames and nesting*
  - ✓ *Activity diagrams*
  - *Deployment diagrams*
    - *Sequence diagrams*
    - *Profiles, Stereotypes*
    - *UML Metamodel*

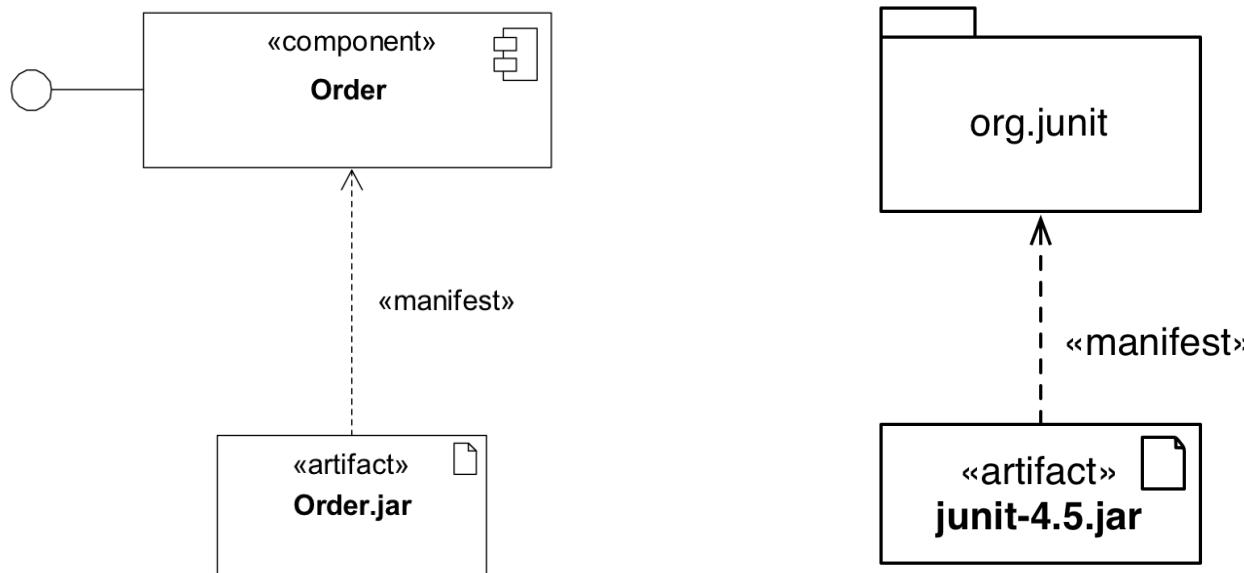
# UML 2 Deployment Diagrams

- *Two node types:*
  - *Device*
  - *Execution environment*



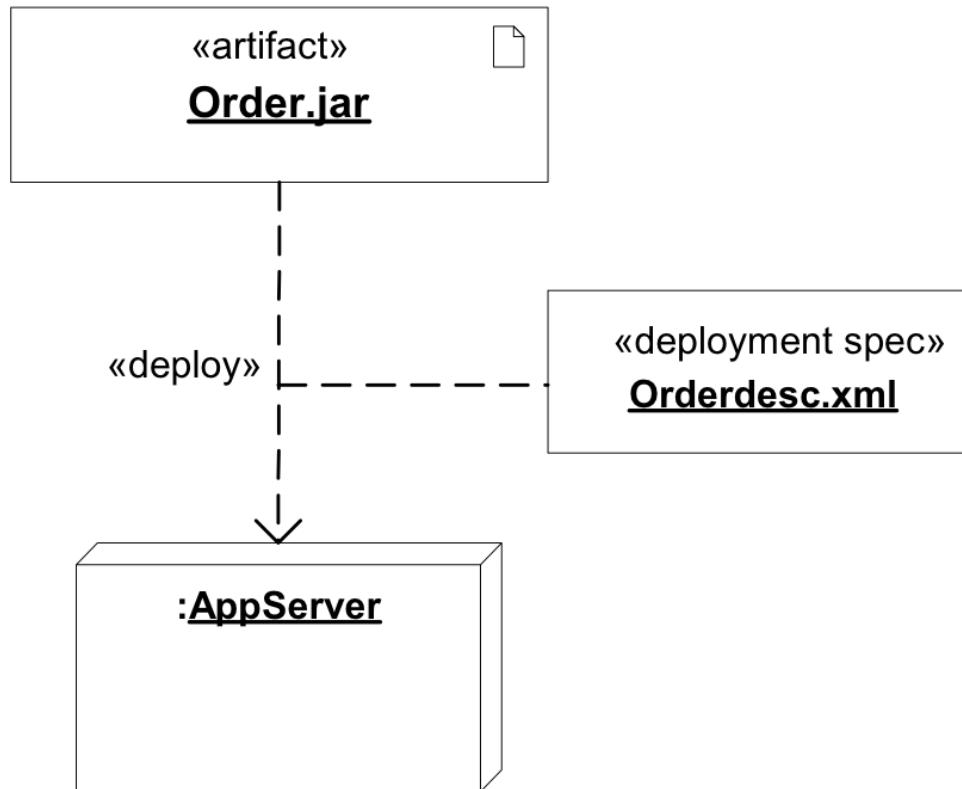
# Deployment Diagram Changes II

- *Artifacts can now manifest any packageable element, not just components*
- *Manifestation is shown by a dependency with keyword «manifest»*



# Deployment Diagram

- A *deployment diagam can have a deployment specification*



# Sequence Diagram Changes

- *New concept of **interaction fragments***
- *Before going into detail with interaction fragments, we cover the concept of an **interaction***

# Interaction

- An Interaction is a concept providing a basis for interaction diagrams:
  - Sequence diagrams
  - Communication diagrams
  - Interaction overview diagrams
  - Timing diagrams
- An *interaction* is a unit of behavior that focuses on the observable exchange of information between connectable elements.
- We only focus on the impact of interactions on sequence diagrams

# Usage of Interactions

- UML Interactions are used to get a better grip of an interaction situation
- Interactions are also used during the detailed design phase if precise inter-process communication must be set up according to formal protocols
- When testing is performed, the traces of the system can be described as interactions and compared with those of the earlier phases.

# Example of an Interaction: Sequence Diagram

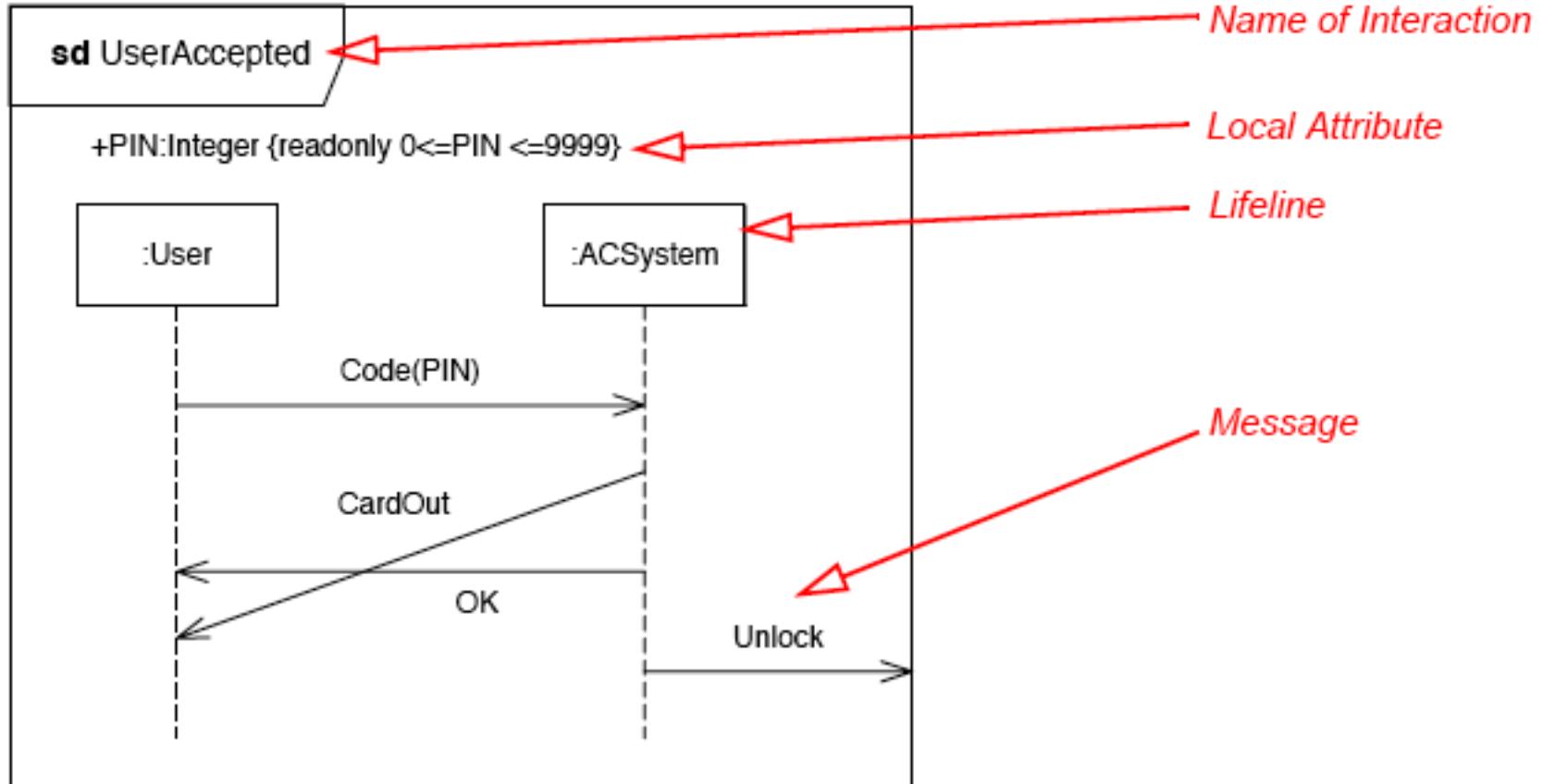


Figure 14.16 - An example of an Interaction In the form of a Sequence Diagram

# Explanation for the Previous Slide

- *The sequence diagram on the previous slide shows three messages between two anonymous lifelines of type User and ACSys tem: CodePin, CardOut and OK*
  - *The message CardOut overtakes the message OK in the way that the receiving event occurrences are in the opposite order of the sending event occurrences.*
  - *Such communication may occur when the messages are asynchronous.*
- *A fourth message Unlock is sent from the ACSys tem to the environment*
  - *Through a gate with the implicit name out\_Unlock.*

# Interaction Fragment

- *Interaction Fragment*
  - *Is a piece of an interaction*
  - *Acts like an interaction itself*
- *Combined Fragment*
  - *Is a subtype of interaction fragment*
  - *defines an expression of interaction fragments*
  - *defined by an interaction operator and corresponding interaction operands*

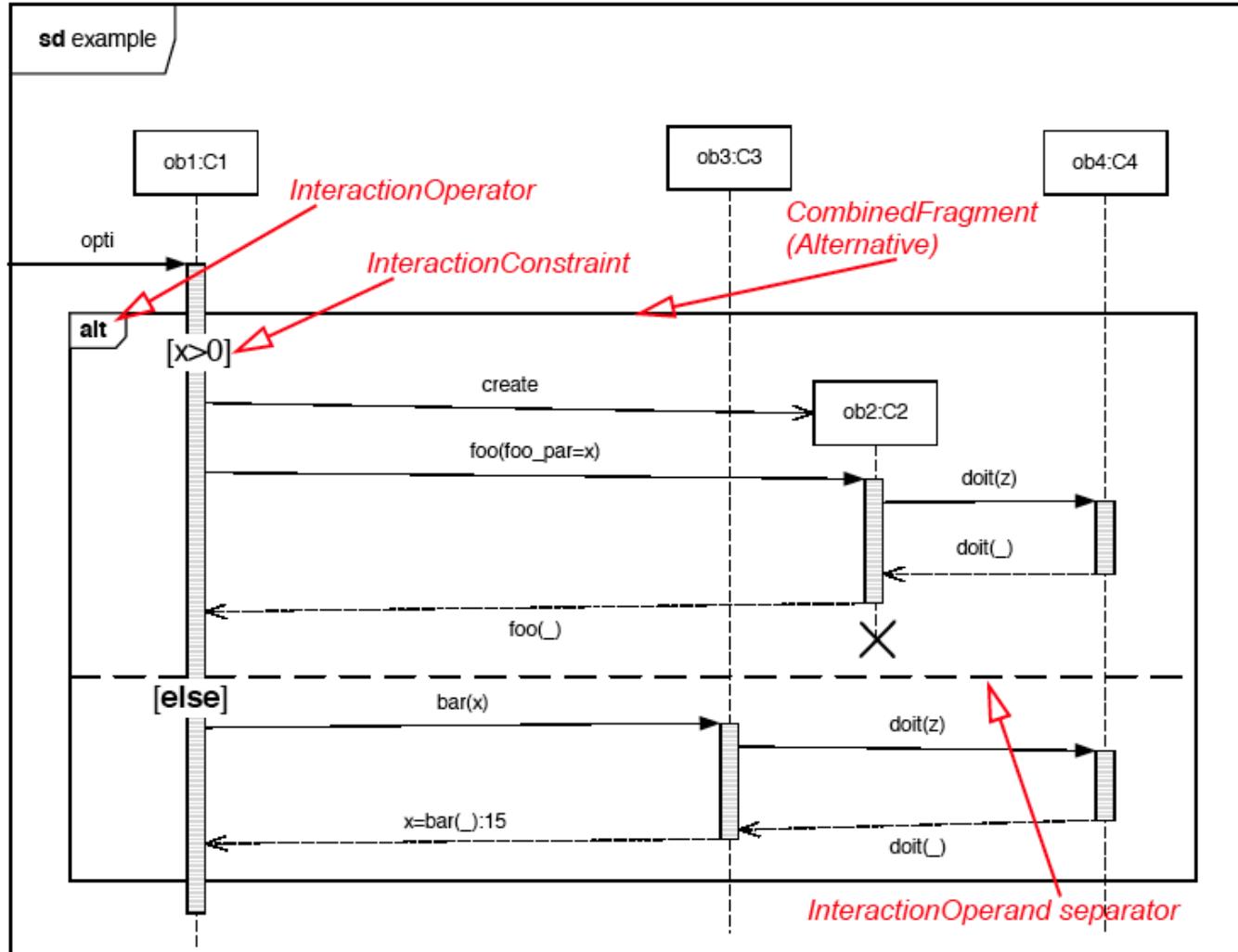
# Interaction Operators

- A combined fragment defines an expression of interaction fragments. The following operators are allowed in an combined fragment expression:
  - Alt
  - Opt
  - Par
  - Loop
  - Critical
  - Neg
  - Assert
  - Strict
  - Seq
  - Ignore
  - Consider

# Alt and Else Operators

- The interaction operator **alt** designates that the combined fragment represents a choice of behavior.
  - At most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction. An implicit true guard is implied if the operand has no guard.
  - The set of traces that defines a choice is the union of the (guarded) traces of the operands.
- An operand guarded by **else** designates a guard that is the negation of the disjunction of all other guards in the enclosing combined fragment.
  - If none of the operands has a guard that evaluates to true, none of the operands are executed and the remainder of the enclosing interaction fragment is executed.

# Example of a Combined Fragment



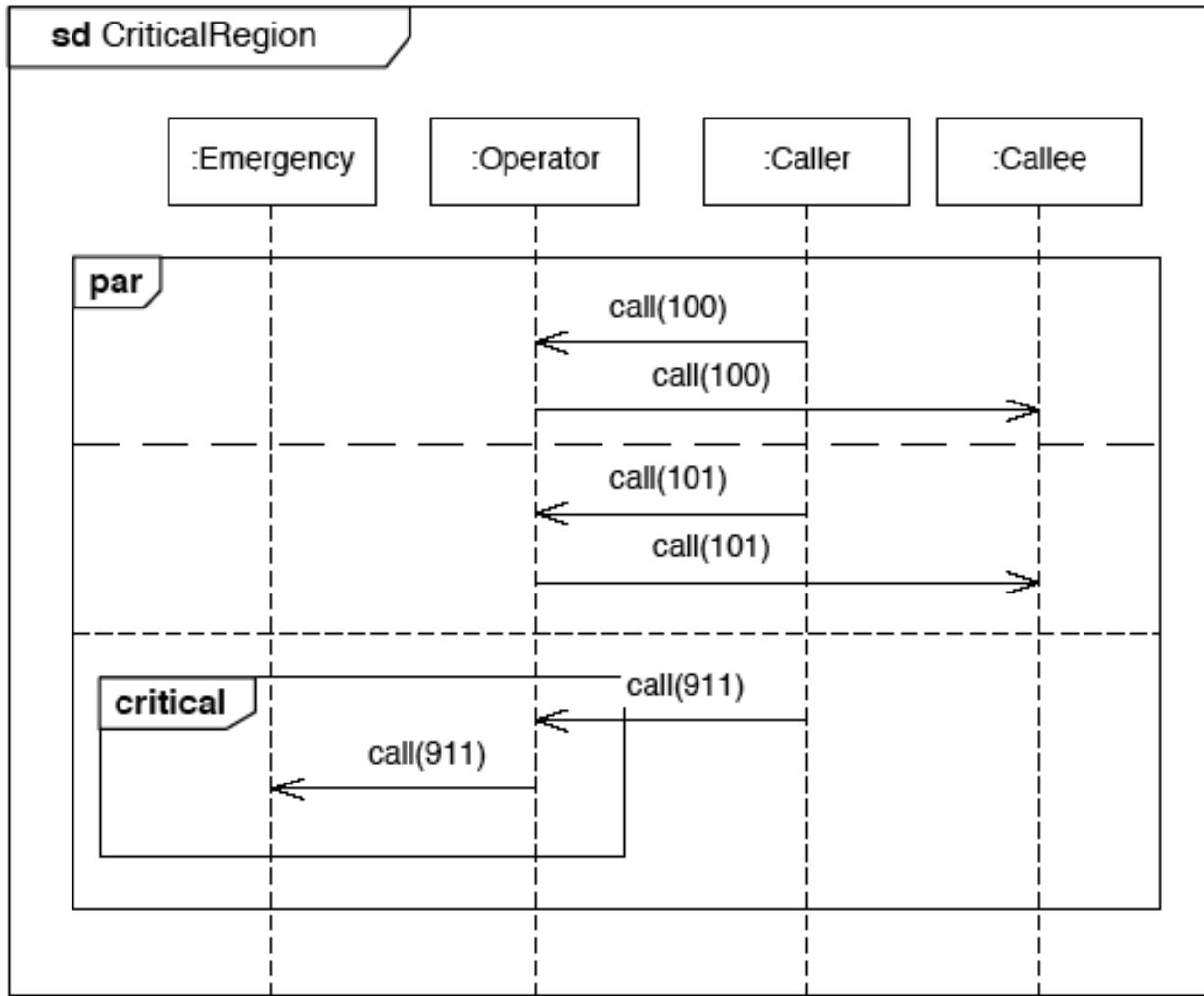
# Opt and Break Operators

- *Option:* The interaction operator **opt** designates a choice of behavior where either the (sole) operand happens or nothing happens.
- *Break:* The interaction operator **break** represents a breaking scenario: The operand is a scenario that is performed instead of the remainder of the enclosing interaction fragment.
  - A break operator with a guard is chosen when the guard is true
  - When the guard of the break operand is false, the break operand is ignored and the rest of the enclosing interaction fragment is chosen.
  - The choice between a break operand without a guard and the rest of the enclosing interaction fragment is done non-deterministically.

# Parallel and Critical Operator

- **Parallel:** The interaction operator **par** designates a parallel merge between the behaviors of the operands of a combined fragment.
  - The event occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved.
  - A parallel merge defines a set of traces that describes all the ways that event occurrences of the operands may be interleaved.
- **Critical:** The interaction operator **critical** designates that the combined fragment represents a critical region.
  - The traces of the region cannot be interleaved by other event occurrences (on the Lifelines covered by the region). This means that the region is treated atomically by the enclosing fragment.

# Example of a Critical Region



The Operator must make sure to forward a 911-call to the Emergency object before doing anything else. Normal calls can be freely interleaved.

# Time Constraint with Messages

- In this example, time constraints are associated with the duration of a Message and the duration between two event occurrences.

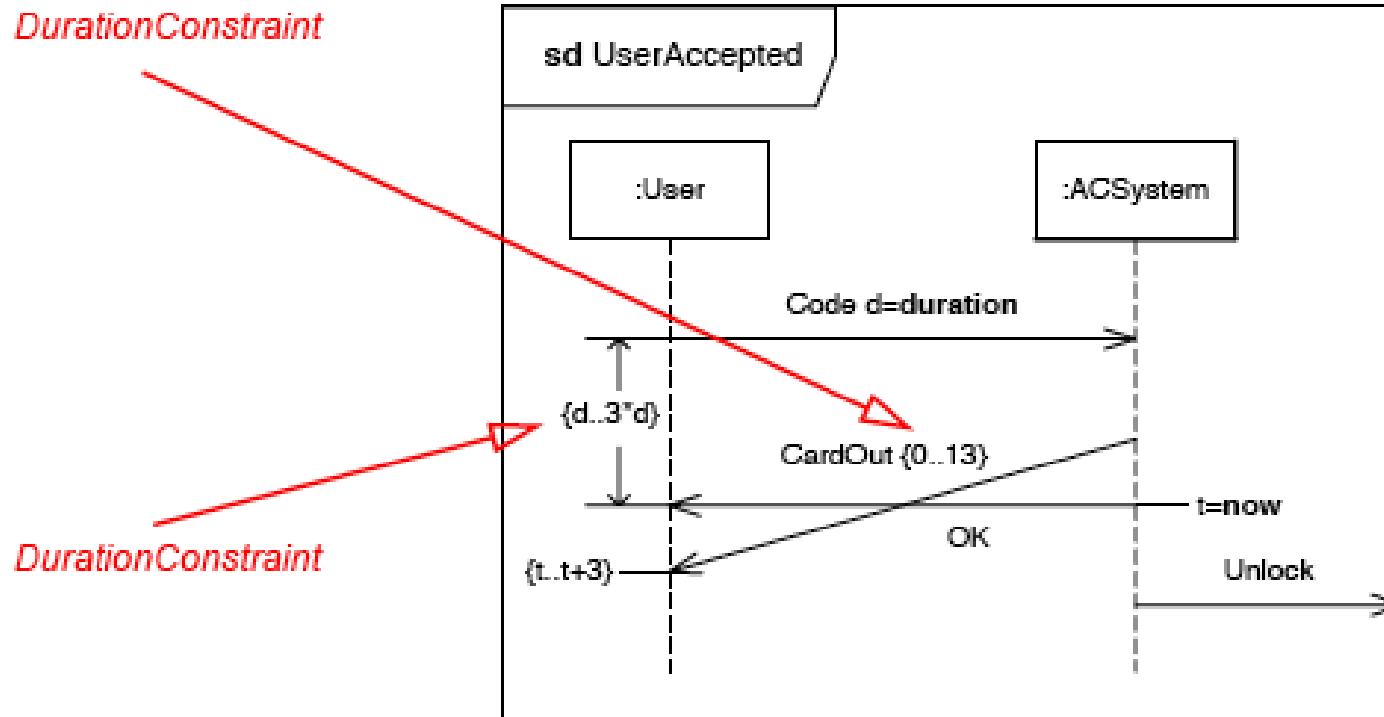


Figure 13.15 - DurationConstraint and other time-related concepts

# Where are we?

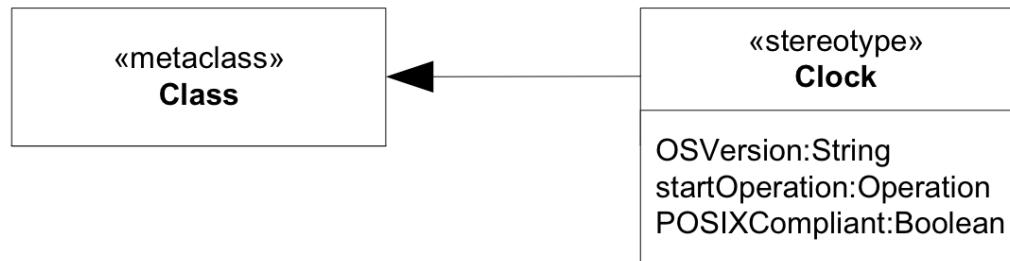
- ✓ *Introduction*
- ✓ *Frames and nesting*
- ✓ *Changes in diagram notation:*
  - ✓ *Activity diagram*
  - ✓ *Deployment diagram*
  - ✓ *Sequence diagram*
- *Profiles and stereotypes*

# Meta Modeling

- *Meta model: A model describing a model*
- *Meta class: Part of the meta model, describing the structure of a model element*
- *Why are we talking about this?*
  - *Meta models can be used to explain the UML notation*
  - *More about this in Friday's lecture but we need the term meta class already to explain UML profiles and stereotypes.*

# Stereotype

- *Defines how an existing meta class may be extended*
- *Can only be used in conjunction with a meta class*
- *A meta class may be extended by one or more stereotypes*



- *Why should we want to do this?*
  - *We may want to use platform or domain specific terminology*

# Stereotype Notations

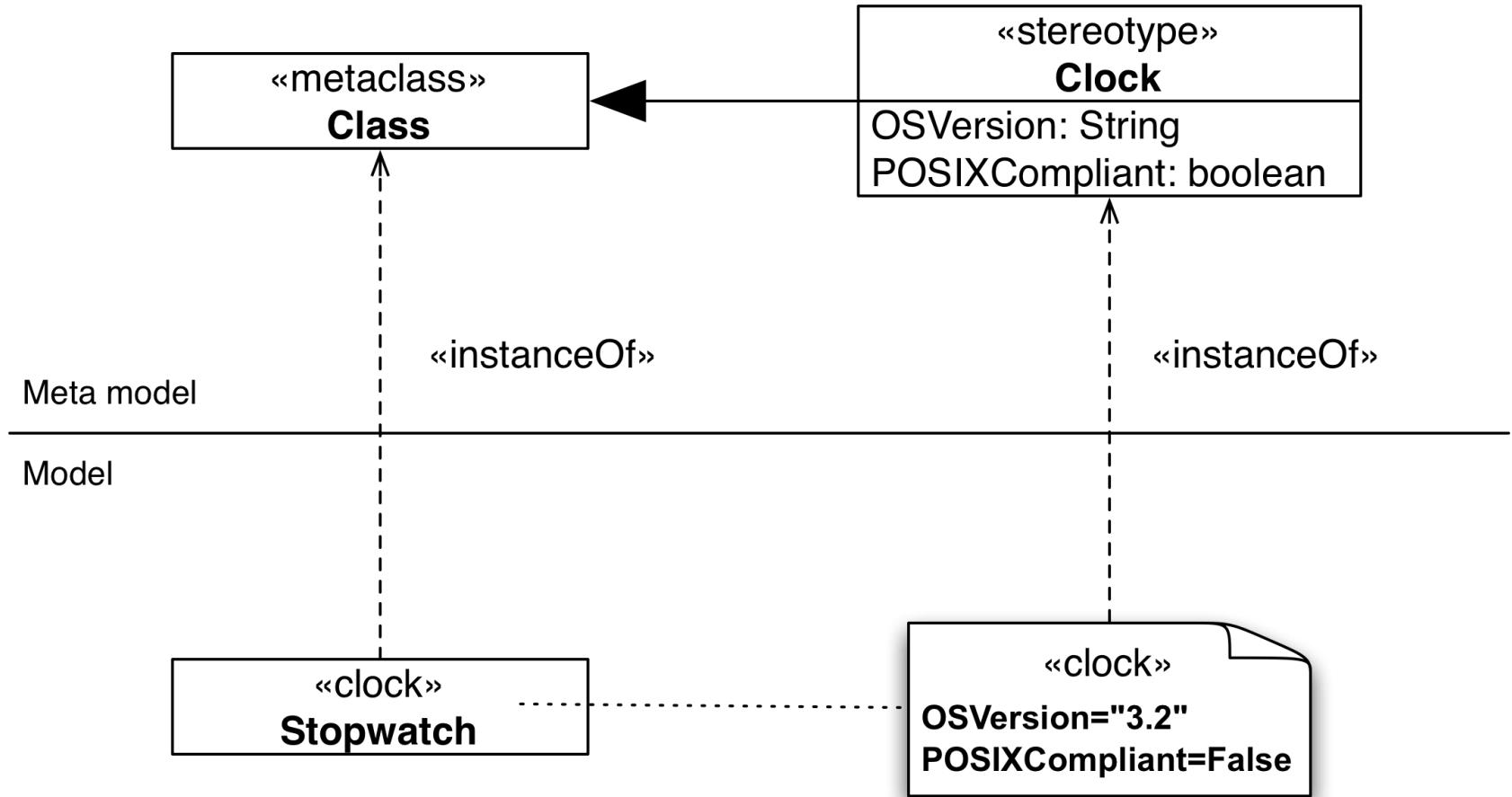
- *The application of a stereotype is shown as a string enclosed in guillemets before the classifier name*
  - *E.g. «boundary» button*
- *Stereotypes can additionally have an image*
  - *The image may replace the standard notation of the element the stereotype has been applied to*



# Applying a Stereotype

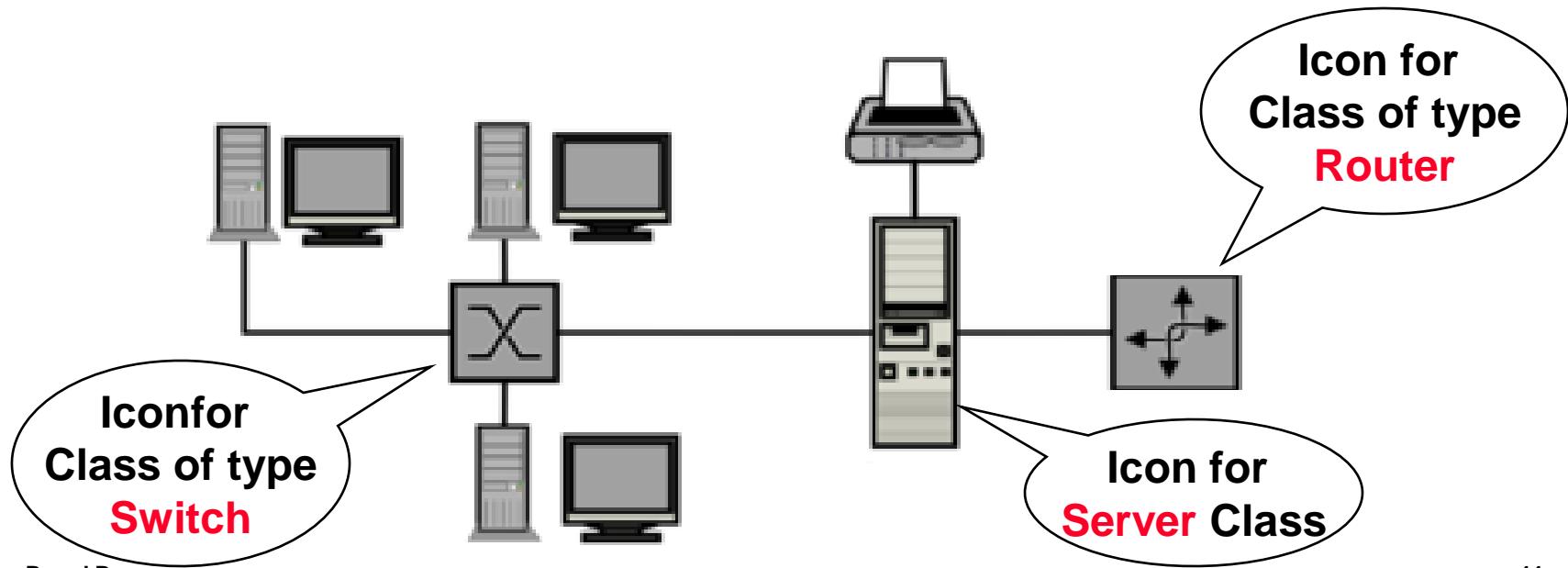
- *The attributes of a stereotype are called **tagged values***
- *Applying a stereotype to an instance of a meta class means instantiating the stereotype*
  - *If the stereotype has attributes, you have to provide values for them*
- *Confused? See following slide...*

# Applying a Stereotype (ctd)



# Icons and Symbols for Stereotypes

- One can also use **icons** or **graphical symbols** to identify a stereotype
  - When the stereotype is applied to a UML model element, the graphic replaces the standard notation for the model element.
- Example: When modeling a network, you can use icons for representing classes of type **Switch**, **Server**, **Router**, **Printer**, etc.



# Pros and Cons of Stereotype Graphics

- Advantages:
  - UML diagrams may be easier to understand if they contain graphics and icons for stereotypes
    - This can increase the readability of the diagram, especially if the client is not trained in UML
    - And they are still UML diagrams!
- Disadvantages:
  - If developers are unfamiliar with the symbols being used, it can become much harder to understand what is going on
  - Additional symbols add to the burden of learning to read the diagrams
- If you end up applying stereotypes all over your model, you should think of defining a profile.

# Stereotypes vs. Keywords

- *Not to be mixed up with keywords*
  - *Same notation (String enclosed in guillemets)*
  - *«interface» is no stereotype!*
  - *«extend» is no stereotype!*
- *See Annexes B and C of the UML Superstructure specification.*

# UML Profiles

- A *lightweight extension mechanism for UML*
- *Concepts partially present in earlier versions*
  - *Stereotypes*
  - *Tagged Values*
- *Established as a specific meta-modeling technique in UML 2.0*
  - *Contains mechanisms that allow meta classes from existing meta models to be extended*
  - *ability to tailor the UML meta model for different platforms or domains*
  - *consistent with the OMG Meta Object Facility (MOF)*
    - *MOF will be covered in the next lecture on metamodeling.*

# Additional Readings

- *UML 2.2 specification*
  - *The specification consists of two complementary parts which constitute the complete specification of UML 2:*
    - *Infrastructure and Superstructure.*
- *The infrastructure specification defines the foundational language constructs*
  - <http://www.omg.org/spec/UML/2.2/Infrastructure/PDF>
- *The superstructure defines the user level constructs.*
  - <http://www.omg.org/spec/UML/2.2/Superstructure/PDF>
- *For a complete list of all UML specifications, see*
  - <http://www.omg.org/spec/UML/>