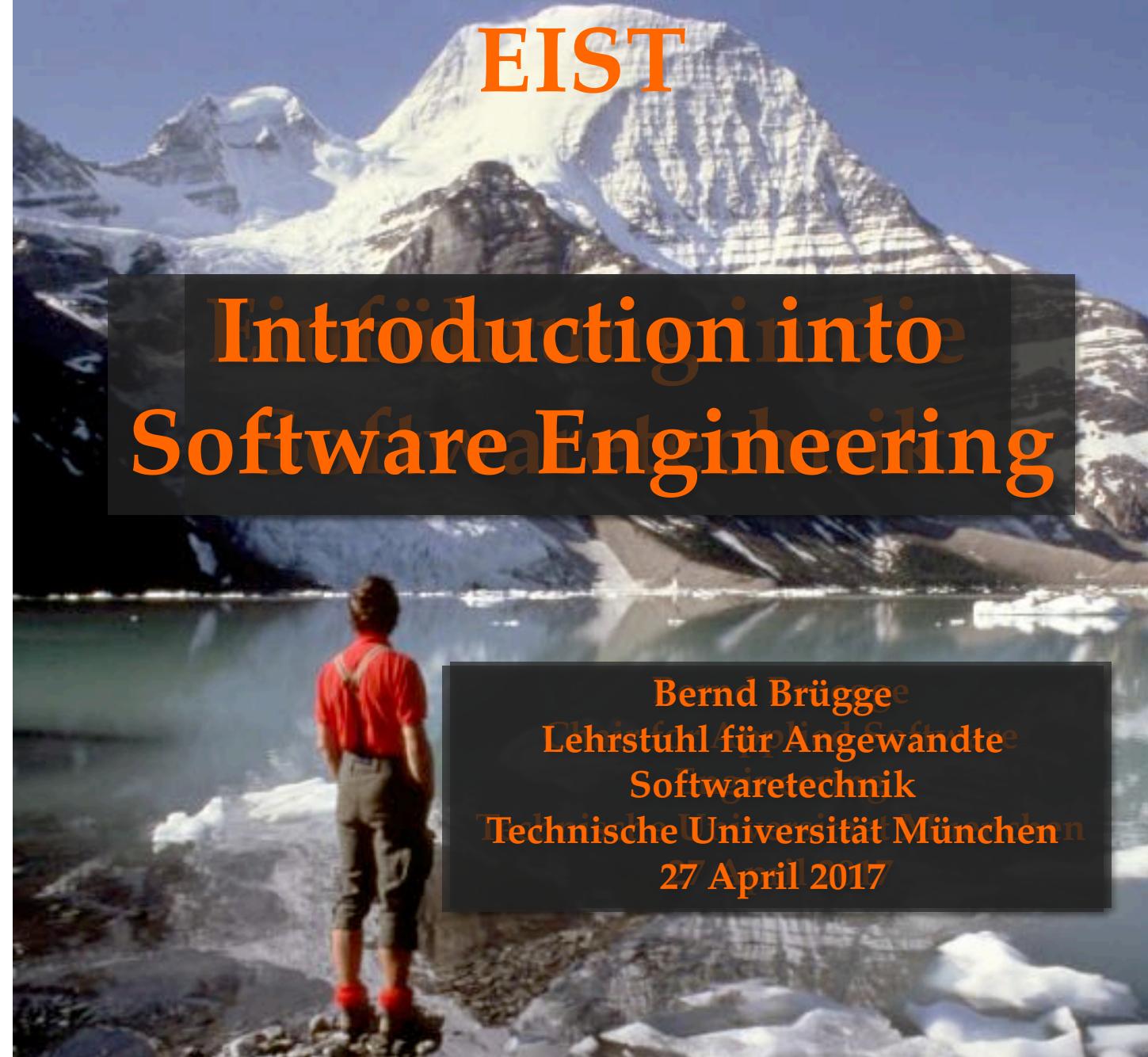
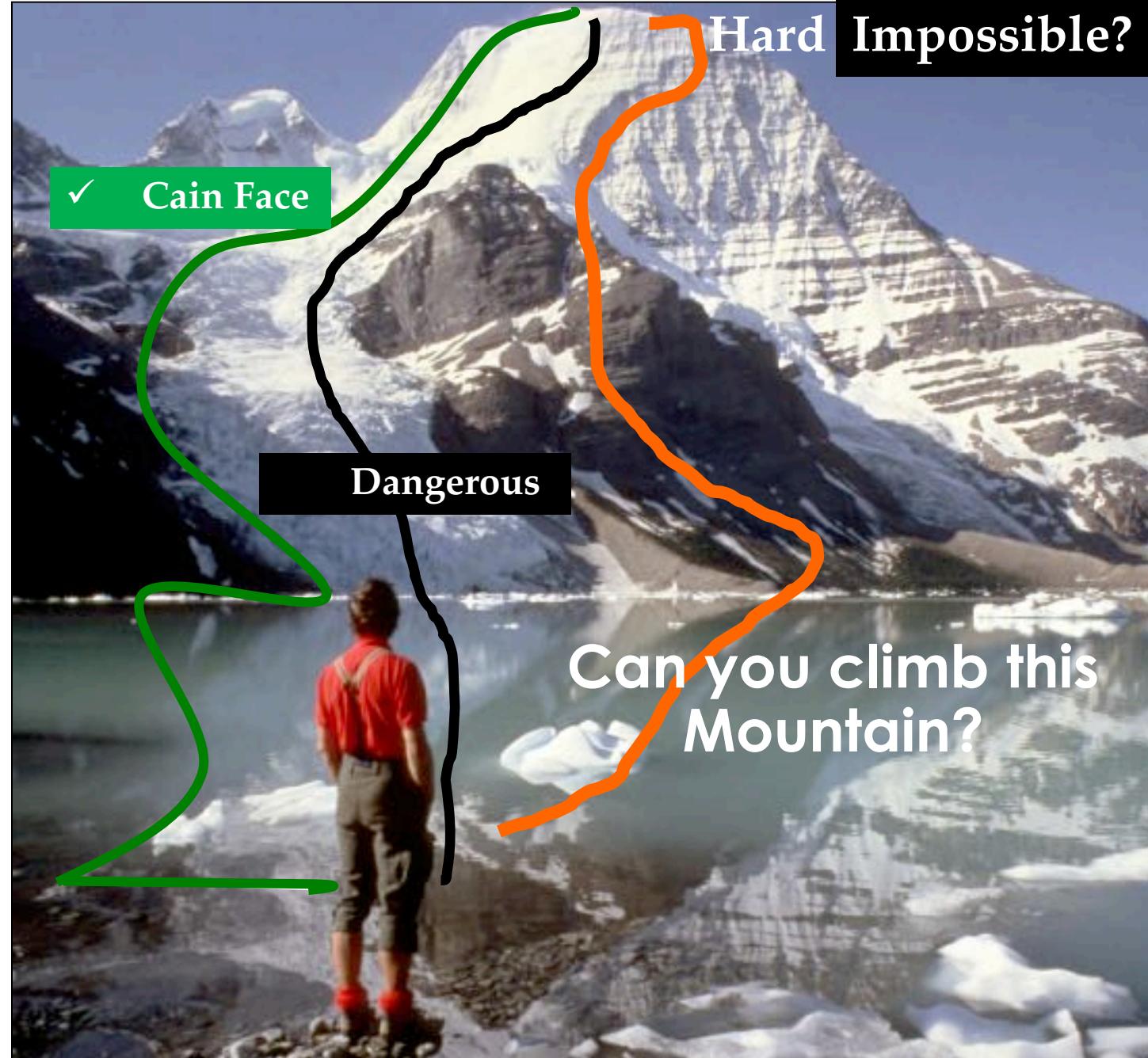


Einführung in die Softwaretechnik
Introduction into Software Engineering



Einführung in die Softwaretechnik

Introduction into Software Engineering



Roadmap for the Lecture

- **Context and Assumptions**
 - You are a bachelor student of informatics, BWL or CSE
- **Content of this lecture**
 - We will introduce some basic terminology: Problem Solving, Software Engineering, Modeling, UML
 - We will give you an overview of the organization of the lecture
 - We will give you a first pass on modeling
- **Objective:** At the end of this lecture you are able to
 - Understand the organization of the lectures and exercises
 - Interact with us during the lecture using AMATI
 - Understand the difference between models and notations.

Overview

1

Problem Solving

2

Software Engineering Definition

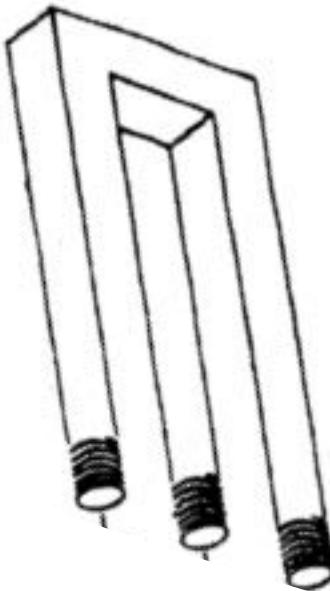
3

EIST Organization

4

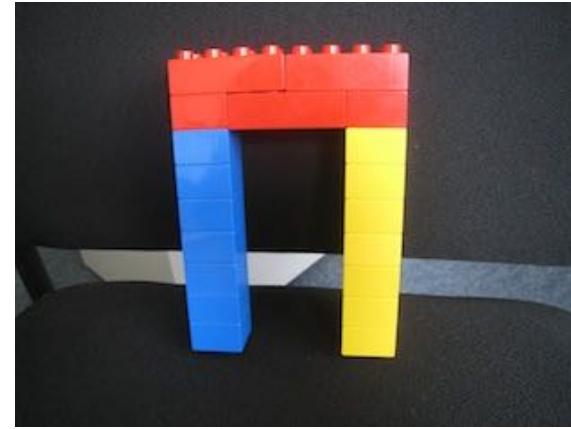
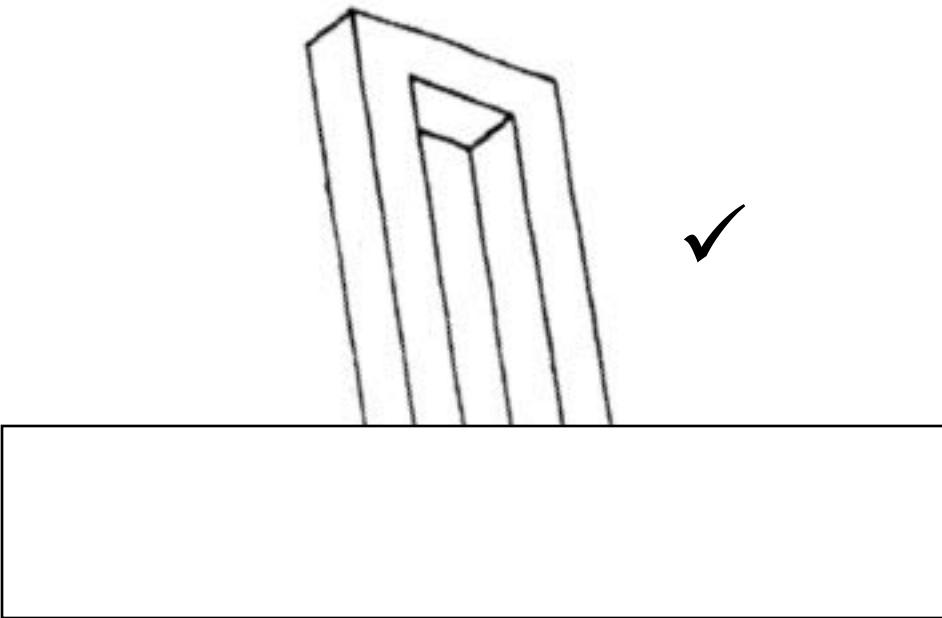
Systems, Models and Views

Can you develop this System?

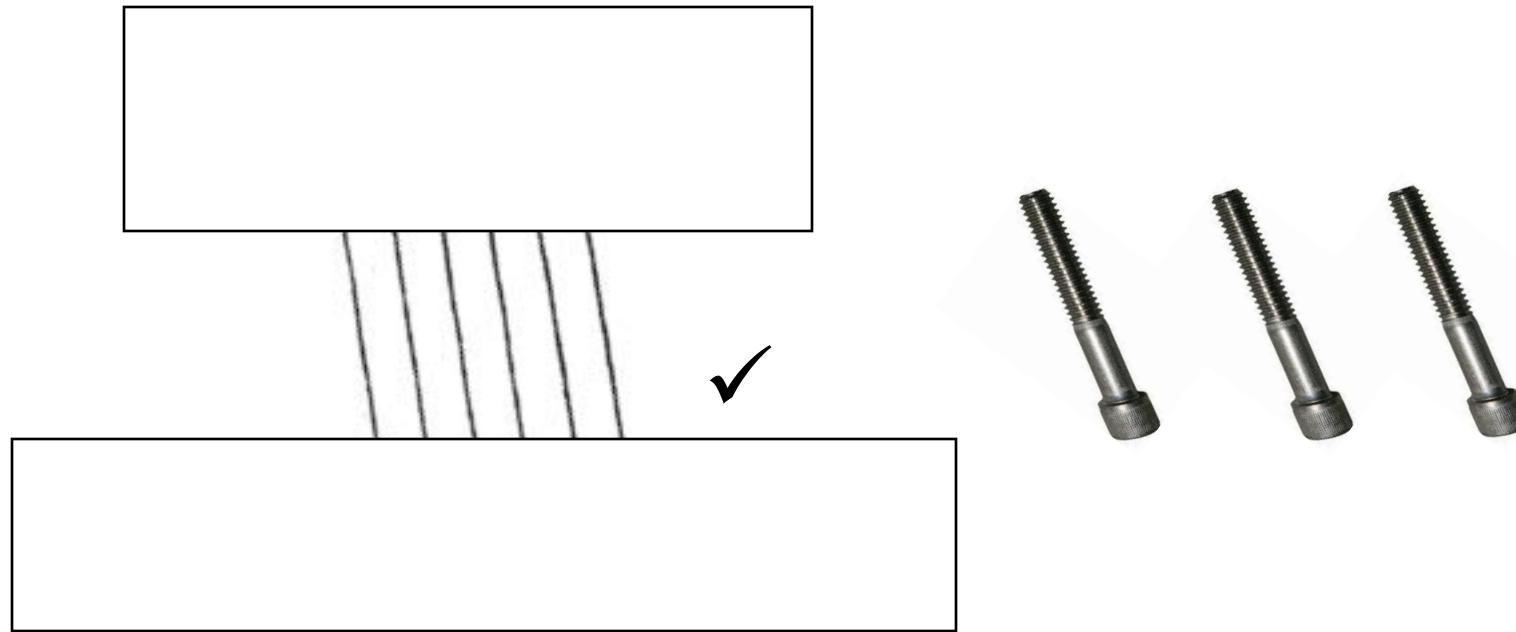


**Impossible?
Or only hard?**

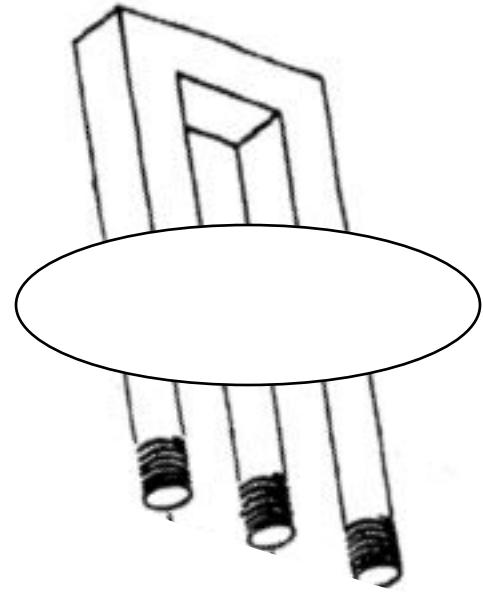
How about this System?



Can you develop this System?



The impossible Fork



The impossibility shows up,
when we physically try to merge the
subsystems from the previous slides

"System integration problem"

Also called Impossible Trident

System Integration Problems are Common



Possible reasons:

- Algorithmic fault
- Bad communication between teams
- Wrong usage of compass
- Bad interface specification
- Bad subsystem decomposition
- Physical impossibility

First Insights

1. You need to talk to the customer about the problem scope and the requirements
2. We solve a **complex** problem by dividing it into smaller pieces ("divide and conquer")
3. We then try to put the pieces back into a larger system that solves the problem
4. Problem descriptions that don't have unique objects cannot be built without reformulating (**changing**) the problem description.

Physical Model of the impossible Fork



Shigeo Fukuda, 1932-2009

Impossible Column

<http://im-possible.info/english/articles/trident/trident.html>

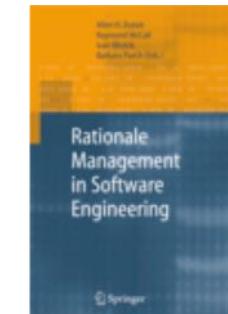
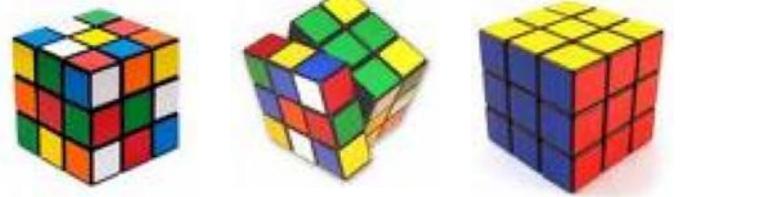
Impossible Trident in Perth, Australia:



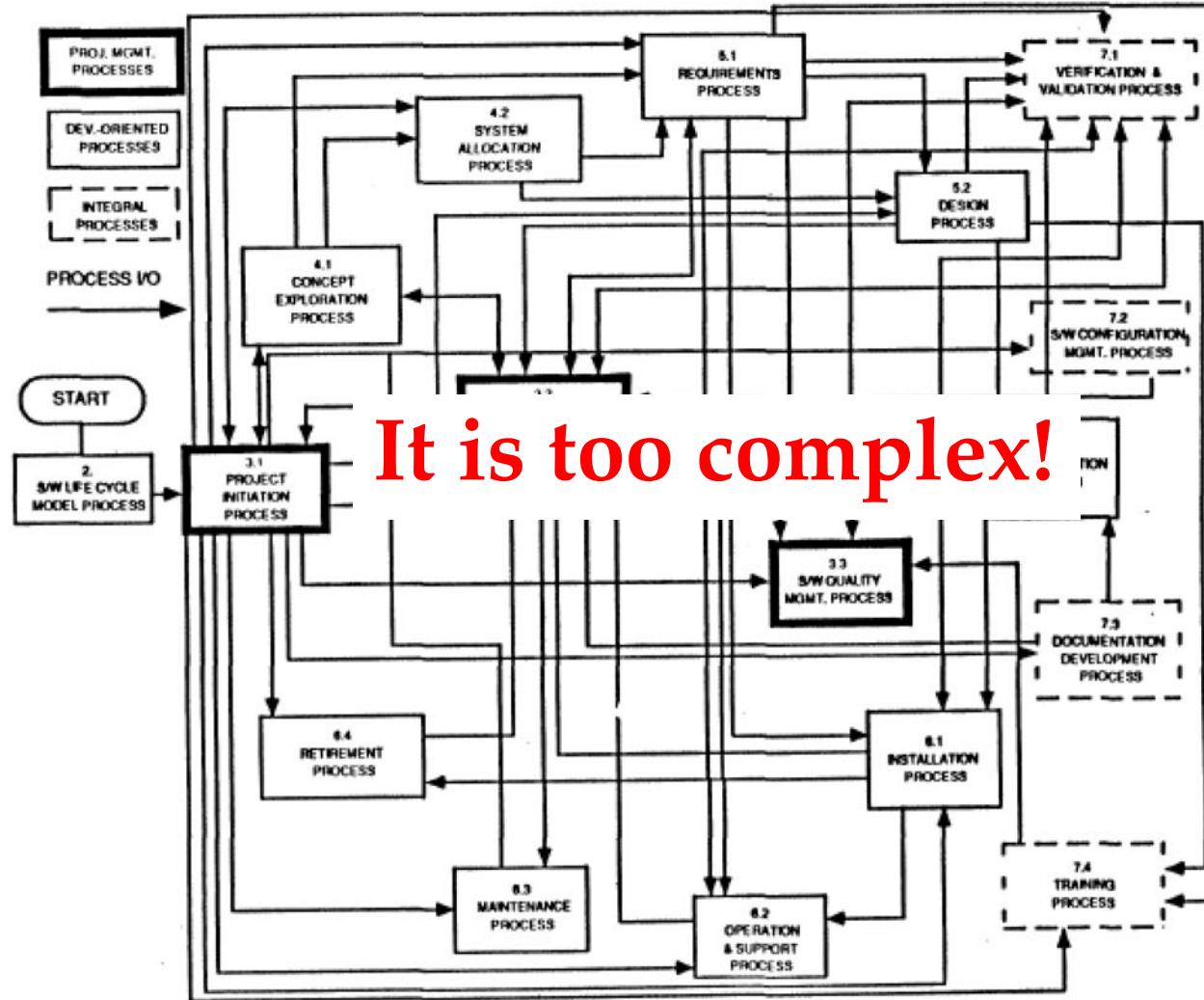
<http://im-possible.info/english/articles/real/index.html>

Software Development is more than just Writing Code

- It is **problem solving**
 1. Understanding the problem
 2. Proposing a solution and a plan
 - Often called Prototype
 3. Engineering a system based on the proposed solution using a *good* design
- It is about **dealing with complexity**
 - Creating abstractions and models
 - Notations for abstractions
- It is about **dealing with change**
 - Requirements elicitation, analysis, design, implementation, validation of the system, delivery and maintenance.



What is the Problem with this Drawing?



Solution:
Abstraction!

Abstraction

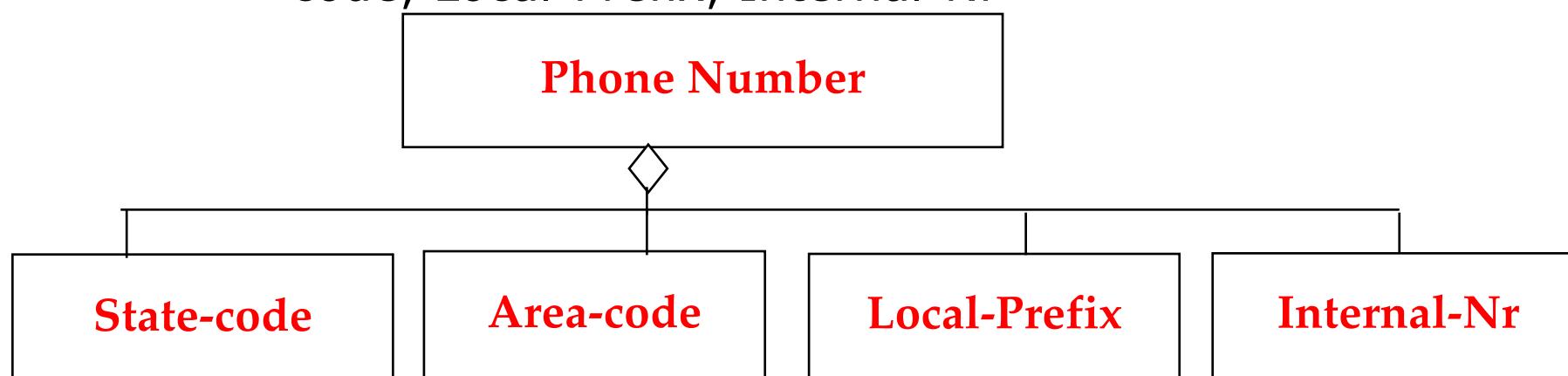
- Complex systems are hard to understand
- The 7 +- 2 phenomena
 - Our short term memory cannot store more than 7+-2 pieces at the same time -> **limitation of the brain**
 - Example: Phone numbers with more than 9 digits

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
 - Example: Phone numbers with more than 9 digits
- Chunking:
 - Group collection of objects to reduce complexity
 - Example: A phone number consists of a State-code, Area-code, Local-Prefix, Internal-Nr

Abstraction

- Complex systems are hard to understand
- The 7 +- 2 phenomena
 - Our short term memory cannot store more than 7+-2 pieces at the same time -> limitation of the brain
 - Example: Phone numbers with more than 9 digits
- Chunking:
 - Grouping into a collection of objects to reduce complexity
 - Example: A phone number **consists of** a State-code, Area-code, Local-Prefix, Internal-Nr



Abstraction

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



Models

- 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 for Software Development

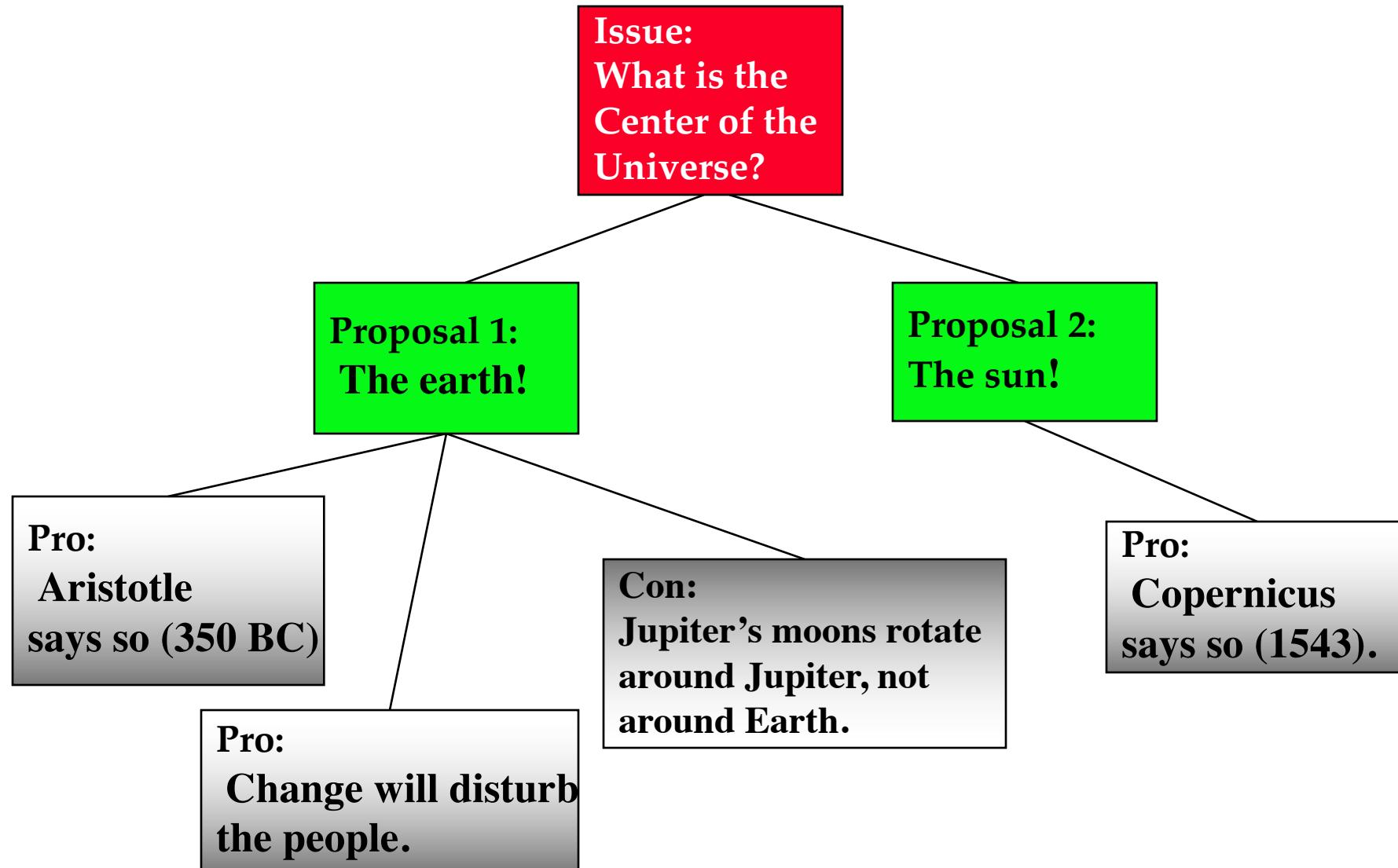
- **Object model:** What is the structure of the system?
- **Functional model:** What are the functions of the system?
- **Dynamic model:** How does the system react to external events?

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

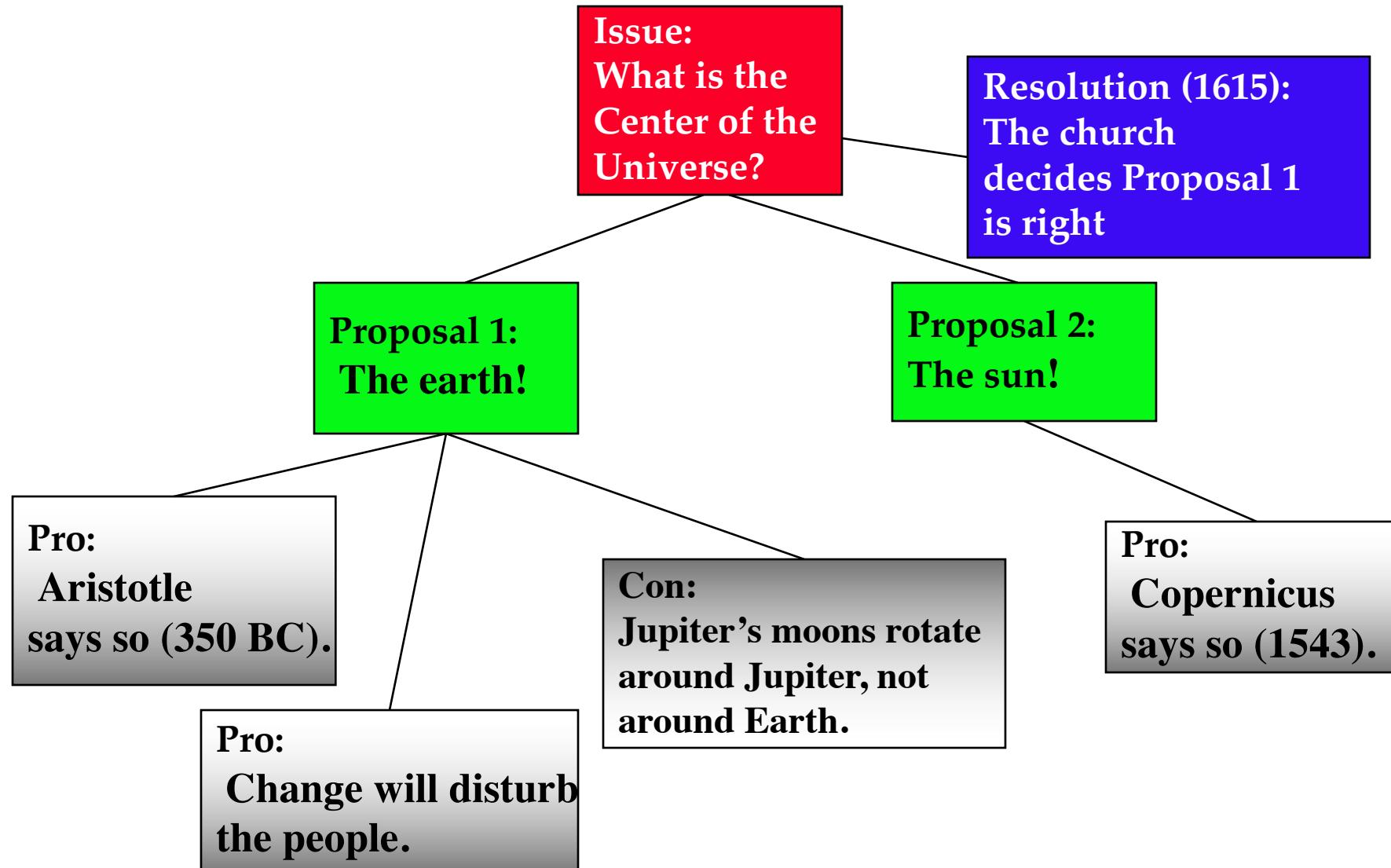
Other Models are used to manage Software System Development

- Task Model:
 - PERT Chart: Describes the dependencies between development activities
 - Schedule: Describes how the activities can be accomplished to finish a project before a deadline
 - Organization Chart: Describes the roles of the participants in the project
- Issue Model:
 - What are the open issues?
 - What proposals do we have to close the issues?
 - What resolutions can be made?

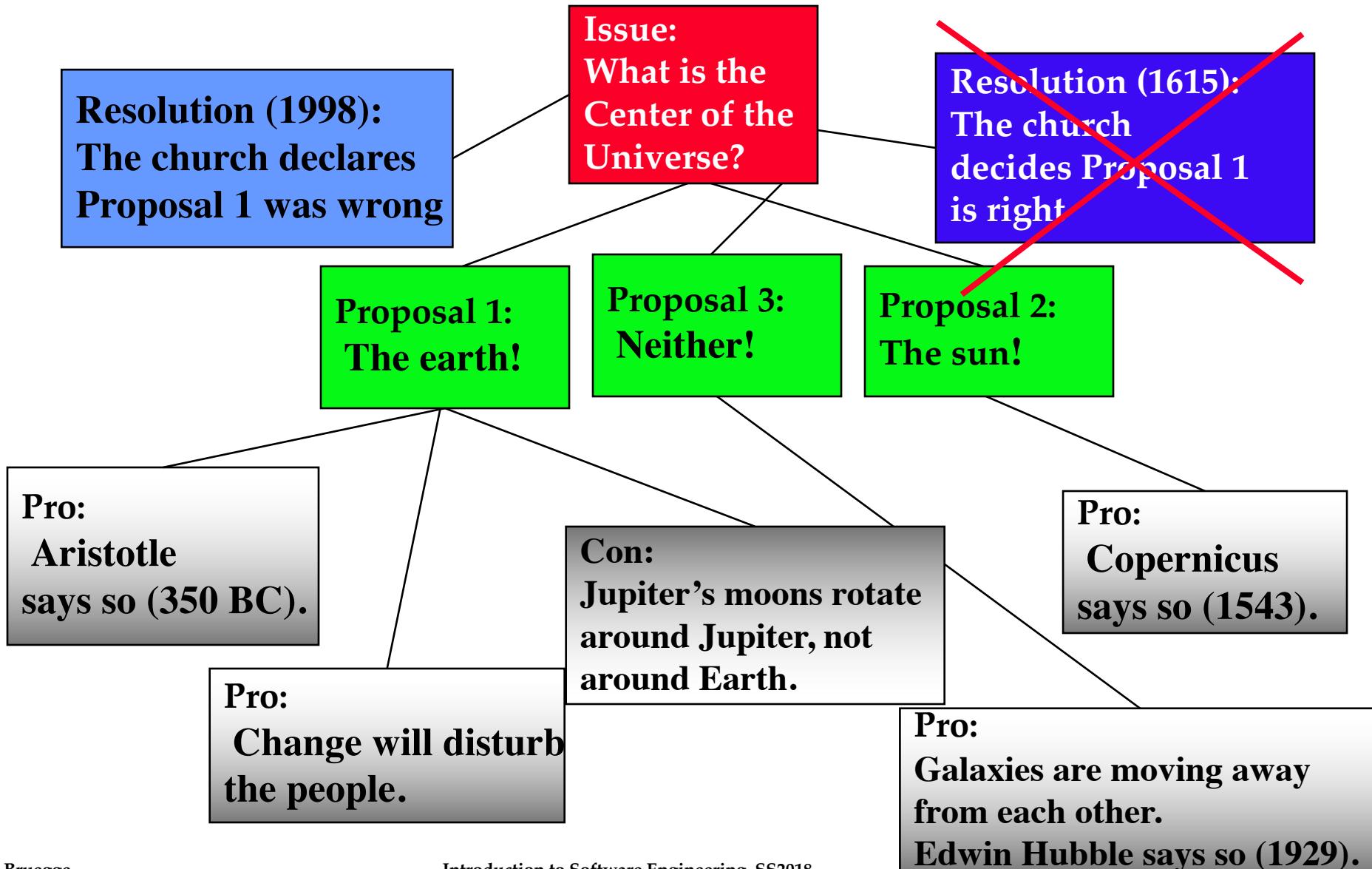
Issue-Model Example



Issue-Model Example



Issue-Model Example



Why is Software Development difficult?

- The problem is usually ambiguous (“impossible trident”)
- The requirements are usually unclear and change when they become clearer
- The problem domain (also called application domain) is complex, and so is the solution domain
- 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 - an early pioneer in software engineering who formulated in 1972 the concepts of **modularity** and **information hiding** which are the foundation of object oriented methodologies.



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 larger structure

For problem solving we use techniques, methodologies and tools.

Overview

1

Problem Solving

2

Software Engineering Definition

3

EIST Organization

4

Systems, Models and Views

Techniques, Methodologies and Tools

- **Techniques:**
 - Formal procedures for producing results using some well-defined notation
 - Example: Quicksort Algorithm
- **Methodologies:**
 - Collection of techniques applied across software development and unified by a philosophical approach
 - Example: Object-oriented Analysis and Design
- **Tools:**
 - Instruments or automated systems to accomplish a technique
 - Examples:
 - Compiler, Editor, Debugger
 - Integrated Development Environment (IDE)
 - Computer Aided Software Engineering (CASE)

Computer Science vs. Engineering

- The computer scientist
 - Assumes techniques and tools have to be developed
 - Proves theorems about algorithms
 - Designs languages and grammars
 - Has infinite time (in principle ☺)
- The engineer
 - Develops a solution for a problem formulated by a client
 - Uses computers & languages, techniques and tools
- The software engineer
 - Works in multiple application domains
 - Has only limited time ...
 - ...while changes occurs in a complex problem formulation and often also in the available technology.

Software Engineering 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

Dealing with Complexity

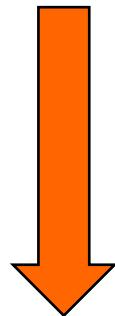
- Modeling
- Notations (UML, OCL)
- Requirements Elicitation
- Analysis and Design
 - OOSE, scenario-based design, formal specifications
- Implementation
- Testing

Dealing with Change

- Release Management
 - Configuration Management
 - Continuous Integration
- Delivery
 - Continuous Delivery
- Software Life Cycle Modeling
- Rationale Management
- Project Management

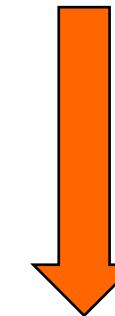
Dealing with Complexity

- Modeling
- Notations (UML, OCL)
- Requirements Elicitation
- Analysis and Design
 - OOSE, scenario-based design, formal specifications
- Implementation
- Testing



Dealing with Change

- Release Management
 - Configuration Management
 - Continuous Integration
- Delivery
 - Continuous Delivery
- Software Life Cycle Modeling
- Rationale Management
- Project Management



Application of these Topics in Exercises

Overview

1

Problem Solving

2

Software Engineering Definition

3

EIST Organization

4

Systems, Models and Views

EIST Organization

1. Exercises in EIST
2. Registration for Tutor Groups using the Matching Tool
3. Homework
4. Livestream and Lecture Recordings
5. Registration for the Slack Channel EIST 2018
6. The EIST Wall: Questions & Answers with Context
7. Using AMATI to answer Polls
8. First In-Class-Exercise with ArTEMiS: Quiz
9. Exercise Organization

1. Exercises in EIST

- In traditional lectures, exercises come in two flavors:
 - **Central Exercises** are done in a central exercise session (Zentralübung) or in tutor classes. The goal is to work on a set of problems and solve them, possibly with the help of the tutor
 - **Homework** to be solved within a week at home
- In EIST, we use the following exercises
 - **Morning Quiz:**
 - Done at the beginning of each lecture about material from the previous lecture (or all of the previous lectures ☺)
 - **In-Class Exercises:**
 - Intertwined with each lecture, usually after some content has been presented, or to clarify some concepts (e.g. modeling, programming, quiz, etc.)
 - **Homework:** Published every Monday: To be solved within a week at home
 - There are 74 EIST tutor groups. World record!!

2. Registration for Tutor Groups using the Matching Tool

- Read the manual for students:

<http://docmatching.in.tum.de/index.php/manual-students-bundle-matching>

- Login to the matching tool on and select your preferences:

<https://matching.in.tum.de/b/38p1b81-tutorials-eist-ss18>

Timeline:

- You can enter your tutor group preferences starting today 9:30 am
- Deadline is **tomorrow, Friday, 8:00 am (April 12)**
- You receive your tutor group assignment on Friday afternoon
- The first tutor group meetings take place on Monday (April 16)

EIST on Moodle

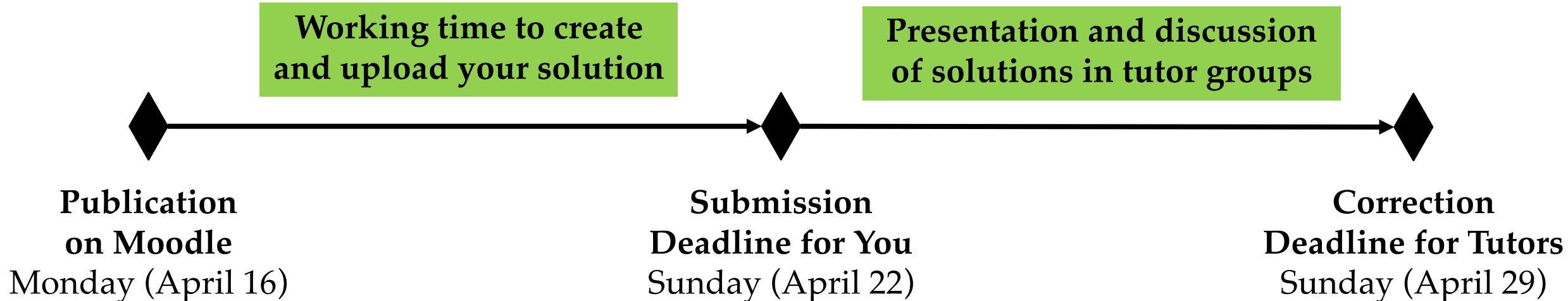
<https://www.moodle.tum.de/course/view.php?id=39072>

- Moodle contains the following information:
 - Announcements
 - Lecture slides
 - Lecture recordings
 - Homework sheets
 - Your homework solutions
 - Feedback to your homework by tutors

3. Homework

- We publish homework every **Monday** on Moodle
 - You have 1 week to solve the homework and upload it to Moodle (until Sunday)
- You can present your solution in your tutor group and discuss the solution with other students
 - **Please note:** Every student has to solve and submit the homework individually (group work is **not** allowed)
- The tutors will correct your solution and provide points and feedback within 1 week on Moodle.

Timeline for Homeworks



Timeline for the 2nd Homework



Access Homework Sheets on Moodle

<https://www.moodle.tum.de/course/view.php?id=39072>

The screenshot shows a Moodle course page for "Einführung in die Softwaretechnik (IN0006)". The page includes a navigation bar with links for HELP, de | en, notifications, and user profile (Stephan Krusche, Student). On the left, a sidebar lists "My courses" (selected), "EIST18", "Participants", "Grades", "Dashboard", and "All course categories". The main content area displays the course title "Einführung in die Softwaretechnik (IN0006)" and the breadcrumb navigation "Dashboard > My courses > EIST18". Below the title, there are two activity items: "Announcements" (with a blue icon) and "Homework Sheet 01" (with a red icon). A yellow callout box with the text "Click on the PDF file to download it" points to the "Homework Sheet 01" link. To the right, a sidebar titled "Activities" shows "Forums" (with a blue icon) and "Latest announcements" (with the note "(No announcements have been posted yet.)". The TUM logo is visible in the top right corner.

Submit Your Homework on Moodle

Lernplattform Moodle
Technische Universität München

Einführung in die Softwaretechnik (IN0006)

Dashboard > My courses > EIST18

Announcements

Homework Submission 01

Activities

Assignments

Forums

Calendar

Upcoming events

Homework Submission 01 is due Sunday, 22 April, 23:55

Go to calendar...

Open the Homework Submission

Mon Tue Wed Thu Fri Sat Sun

1

2 3 4 5 6 7 8

9 10 11 12 13 14 15

16 17 18 19 20 21 22

23 24 25 26 27 28 29

30

Submit Your Homework on Moodle (ctd)

Lernplattform Moodle
Technische Universität München

Einführung in die Softwaretechnik (IN0006)

Dashboard > My courses > EIST18 > General > Homework Submission 01

Homework Submission 01

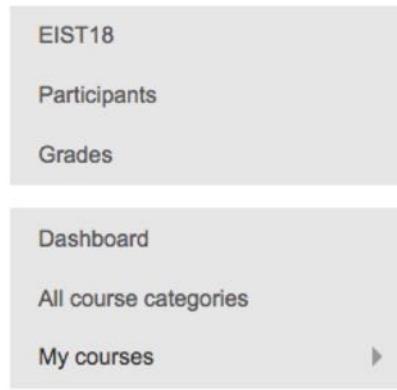
Submission status

Submission status	No attempt
Grading status	Not graded
Due date	Sunday, 22 April 2018, 11:55 PM
Time remaining	11 days 2 hours
Last modified	-
Submission comments	Comments (0)

Make sure to upload your submission within the Due Date

Click Add Submission

Submit Your Homework on Moodle (ctd)



Lernplattform Moodle
Technische Universität München



Einführung in die Softwaretechnik (IN0006)

Dashboard > My courses > EIST18 > General > Homework Submission 01 > Edit submission

Homework Submission 01

File submissions

Maximum size for new files: 100MB, maximum attachments: 1

A screenshot of the Moodle file submission interface. It shows a dashed rectangular area where files can be dropped. A red box highlights this area, and a blue arrow points down to it from above. Below the dashed area, the text "You can drag and drop files here to add them." is displayed. To the left of the dashed area, there are icons for a file and a folder, and the word "Dateien". To the right, there are icons for a grid, a list, and a folder. At the bottom of the dashed area, there is a yellow button with a blue arrow pointing to it, containing the text "Drag and drop your PDF file".

Dateien

You can drag and drop files here to add them.

Accepted file types:

PDF document .pdf

Save changes

Cancel

Drag and drop your PDF file

Submit Your Homework on Moodle (ctd)

EIST18
Participants
Grades

Dashboard
All course categories
My courses

Lernplattform Moodle
Technische Universität München

Einführung in die Softwaretechnik (IN0006)

Dashboard > My courses > EIST18 > General > Homework Submission 01 > Edit submission

Homework Submission 01

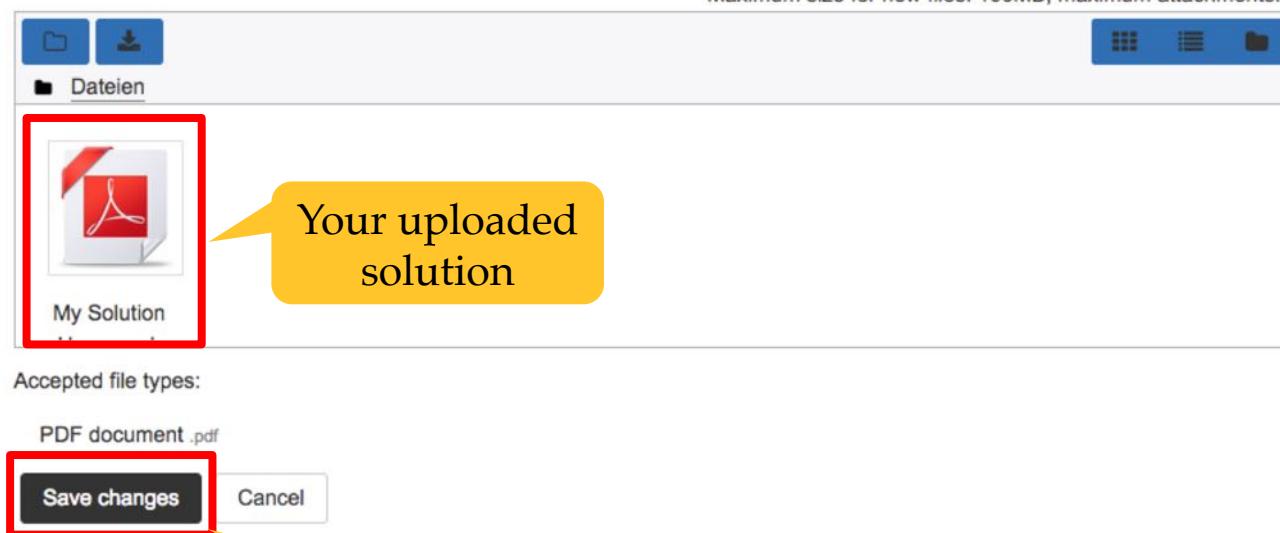
File submissions Maximum size for new files: 100MB, maximum attachments: 1

Dateien

My Solution

Accepted file types:
PDF document .pdf

Save changes Cancel



Submitted Homework on Moodle

The screenshot shows a Moodle course page for the course "Einführung in die Softwaretechnik (IN0006)". The left sidebar shows the course navigation: EIST18, Participants, Grades, Dashboard, All course categories, and My courses (with a dropdown arrow). The main content area displays the course title and navigation path: Dashboard > My courses > EIST18 > General > Homework Submission 01. The submission details are listed under "Homework Submission 01". The "Submission status" section shows "Submitted for grading" (highlighted with a red box) and "Not graded". A yellow callout bubble points to this status with the text "Status after submission". Other details include: Due date: Sunday, 22 April 2018, 11:55 PM; Time remaining: 11 days 2 hours; Last modified: Wednesday, 11 April 2018, 9:30 PM; File submissions: My Solution Homework 01.pdf; Submission comments: 0 comments. At the bottom, there is a blue "Edit submission" button and a link to "Make changes to your submission".

Lernplattform Moodle
Technische Universität München

Einführung in die Softwaretechnik (IN0006)

Dashboard > My courses > EIST18 > General > Homework Submission 01

Homework Submission 01

Submission status

Submission status	Submitted for grading
Grading status	Not graded

Grading status

Due date: Sunday, 22 April 2018, 11:55 PM

Time remaining: 11 days 2 hours

Last modified: Wednesday, 11 April 2018, 9:30 PM

File submissions: My Solution Homework 01.pdf

Submission comments: + Comments (0)

Edit submission

Make changes to your submission

Status after submission

One Week Later: Correction and Feedback

The screenshot shows a Moodle course interface for "Einführung in die Softwaretechnik (IN0006)". The left sidebar includes links for EIST18, Participants, Grades, Dashboard, All course categories, and My courses. The main content area displays the "Homework Submission 01" page. The submission status table shows:

Submission status	Submitted for grading
Grading status	Graded
Due date	Sunday, 22 April 2018, 11:55 PM
Time remaining	11 days 2 hours
Last modified	Wednesday, 11 April 2018, 9:30 PM
File submissions	My Solution Homework 01.pdf

The "Feedback" section shows a grade of "5 / 10". A yellow callout points to this grade with the text "Grade points for the whole homework sheet". Below the grade, it says "Graded on Wednesday, 11 April 2018, 9:34 PM" and "Graded by Krusche Stephan". A feedback file named "Feedback Homework 01 Stephan Krusche.pdf" is listed under "Feedback files", with a red box around it. A yellow callout points to this file with the text "PDF with feedback comments by your tutor".

Your Homework Points on Moodle

The screenshot shows the Moodle grade administration interface for the course "Einführung in die Softwaretechnik (IN0006)". A yellow callout box points to the "Grades" button in the left sidebar, which is highlighted with a red border. Another yellow callout box points to the "User report" dropdown menu, which is currently selected. A third yellow callout box points to the "Course total" row, which shows a grade of 5 and a range of 0-10, labeled as contributing 50% to the course total. A fourth yellow callout box points to the "Your personal course total" text, which is displayed as 5. A fifth yellow callout box points to the "Current max points for the whole course" text, which is also displayed as 5.

EIST18

Participants

Grades

Click on Grades

Lernplattform Moodle
Technische Universität München

Einführung in die Softwaretechnik (IN0006): View: User report

Dashboard > My courses > EIST18 > Grade administration > User report

User report - Krusche Stephan

Exercise points for one sheet

User report

Grade item	Grade	Range	Feedback	Contribution to course total
Einführung in die Softwaretechnik (IN0006)				
Homework Submission 01	5	0–10		50 %
Course total	5	0–10		-
Include empty grades.				

Your personal course total

Current max points for the whole course

4. Livestream and Lecture Recordings

There will be a live stream for each lecture:

- <https://livestream.com/ls1intum/eist2018>
- (or just search for “EIST” @ livestream.com)

All lectures will also be recorded. They will be accessible via Moodle:

- <https://www.moodle.tum.de/course/view.php?id=39072>

5. Registration for the Slack Channel EIST 2018

Signup with your @in.tum.de, or @tum.de or @mytum e-mail address:

<https://eist18.slack.com/signup>

Overview of channels:

- **#general-questions** Ask any organizational questions
- **#off-topic** Not monitored (for your happiness ☺)
- **#polls** Used for in-class polls
- **#slide-questions** Ask any questions regarding the lecture

Please use these channels for their dedicated purposes.

AMATI

The #general-questions and #slide-questions channels are supported by AMATI

AMATI: Another Massive Audience Teaching Instrument

- When you ask question make sure to finish it with "?"
 - Example: „What is EIST?”
- AMATI automatically opens a *thread* and posts the slide currently visible on the beamer next to your question.
- Please keep the discussion relevant to the question in this thread.



20 Minute Break

Good time to enter your tutor group preferences

Starting today 9:30 am

No need to rush: Deadline is
tomorrow, Friday, 8:00 am (April 12)



Question Answering Process

- Once an approved answer is provided, the question will be marked with the checkmark emoji 
- You can use this emoji as well, but AMATI ignores it.
 - Therefore, do not use the checkmark emoji in the question channel
- We will upload a logfile as a PDF of all questions & answers for each lecture to Moodle.

6. The EIST Wall: Questions & Answers with Context

 **Jan Knobloch** Today at 1:59 PM
in #slide-questions

The Slack Logo is a little big isn't it?

2 replies

 **AMATI APP** 6 minutes ago
Question ID: 1522929599.000180

Context

Context

Change the slide by replying with "L:5 S:25" (E: for exercises). (58 kB) ▾

Introducing Slack / AMATI



- We use slack to communicate with you and answer questions
- Sign up with your XXX @.tum.de / @in.tum.de / @mytum.de email address
- Sign up link: eist18.slack.com/signup

 **Jan Knobloch** 1 minute ago
But it is pretty isn't it.



Reply...

Answer marked
as correct

Context

Live Question Feed

Introducing Slack / AMATI

• We use slack to communicate with you and answer questions

• Sign up with your XXX @.tum.de / @in.tum.de / @mytum.de email address

• Sign up link: eist18.slack.com/signup

Question:

The Slack Logo is a little big isn't it?

Answer:

But it is pretty isn't it.

Show discussion

The EIST Wall – Example

Your task:

- Open EIST Slack on your computer or phone and sign up eist18.slack.com
- You are automatically added to the #slide-questions, #general-questions, and #polls channel
- If you have a question, post it in the corresponding channel
- Time: 5 Minutes

The EIST Wall - Let's take a look at your answers

eist18.interactive.ase.in.tum.de

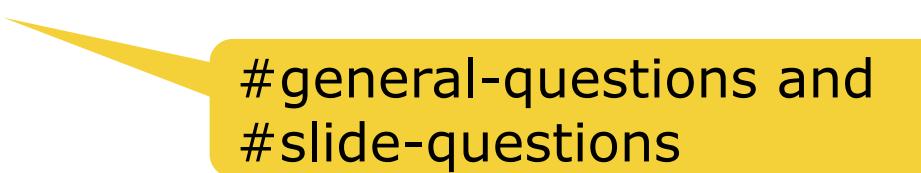
The EIST Wall allows us teachers to create a summary of the communication threads, which will be uploaded to Moodle.

The more relevant questions you ask, the more information you will have to prepare for your exam.

AMATI - Useful Commands

Command	Description
!help	Show the help menu, i.e. http://bit.ly/2pkSPqn
!lectures	Show all lectures for reference
L:3	Change the lecture context, e.g. this question addresses the third lecture
S:15	Change the slide context to slide number n
!ref S:3	Add a referencing slide, e.g. if the question addresses multiple slides
!feedback	Provide feedback concerning AMATI to us

These commands only work when posted in the public channels!



#general-questions and
#slide-questions

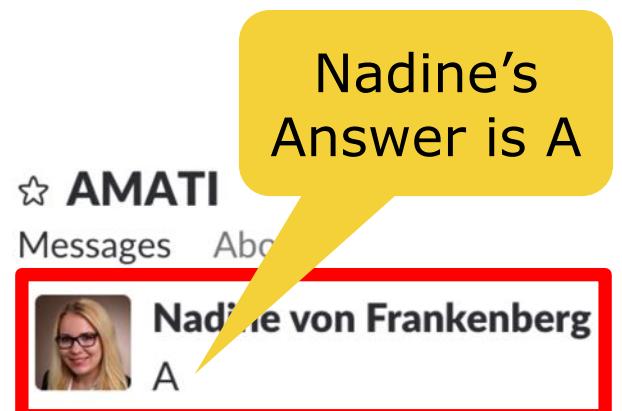
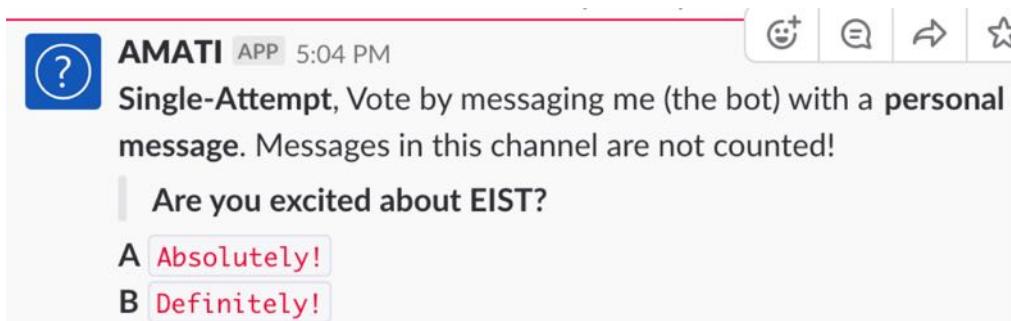
7. Using AMATI to answer Polls

Polls are conducted in the **#polls** channel

Polls allow you to answer questions from the instructor

- **Single-Attempt Poll** - you are only able to vote once
- **Multi-Attempt Poll** - you can change your vote until the poll ends
- **Do NOT write your answer in the #polls channel**
- **Instead write your answer as direct private message to AMATI**

Example of a Single-Attempt Question



#polls Example

Question: What does **EIST** mean?

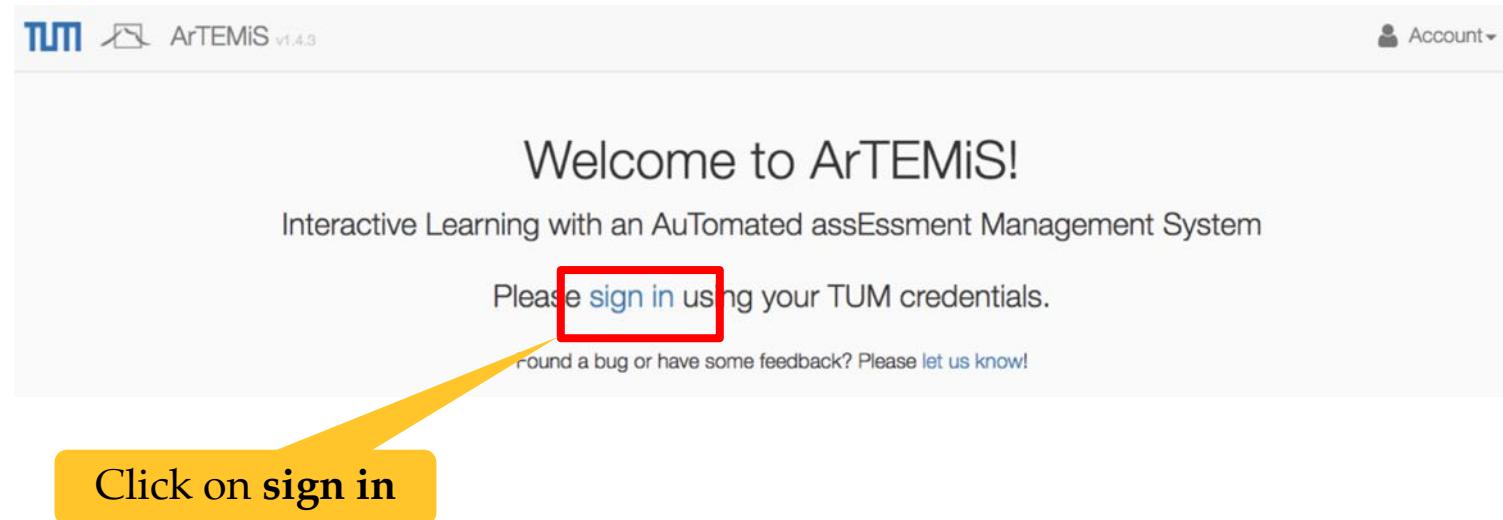
- Possible answers (you can see these choices in the #polls channel)
 - A. Egg in the Software Technology
 - B. Engineering is a Software Transition
 - C. Einführung in die Softwaretechnik
- Your task
 - Open EIST Slack on your computer or phone > eist18.slack.com
 - Open the #polls channel
 - Select your answer (A or B) and write a **direct message** to the **AMATI** bot
- Time: 5 Minutes

8. First In-Class-Exercise with ArTEMiS: Quiz

- Question: What are the Components of a System Model?
- Possible answers
 - A. Object Model, Functional Model and Dynamic Model
 - B. Object Model, Task Model and Dynamic Model
 - C. Object Model, Functional Model and Issue Model
- Your tasks:
 1. Open ArTEMiS on <https://artemis.ase.in.tum.de>
 2. Login with your TUM account (e.g. "ne23kow") and TUM password
 3. Open **Quiz 01**
 4. Wait for the quiz to start
 5. Select the correct answer and submit
 6. Wait for the results
- Time: 2 Minutes

Task 1: Open ArTEMiS on <https://artemis.ase.in.tum.de>

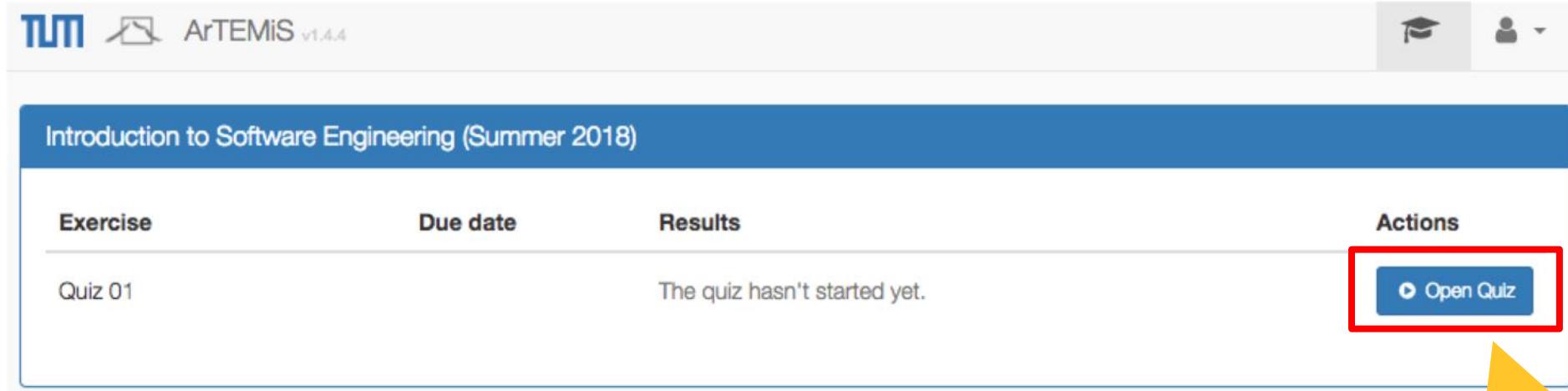
- **Please note:** Make sure to use **https** and **not** http



Task 2: Login to ArTEMiS with your TUM account



Task 3: Open Quiz 01 on ArTEMiS



The screenshot shows the ArTEMiS v1.4.4 interface for the course "Introduction to Software Engineering (Summer 2018)". The table lists one exercise, "Quiz 01", which has not started yet. The "Actions" column contains a blue button labeled "Open Quiz". A red box highlights this button, and a yellow callout with the text "Click on Open Quiz" points to it.

Exercise	Due date	Results	Actions
Quiz 01		The quiz hasn't started yet.	Open Quiz

Click on Open Quiz

Task 4: Wait for the Quiz to Start

- The quiz starts automatically
- **No need to reload the page**

Introduction to Software Engineering (Summer 2018) - Quiz 01

Please wait.

This page will refresh automatically, when the quiz starts.

Questions: 0
Total Points: 0

Waiting for Start
● Connected

Submit

Task 5: Select the Correct Answer and Submit

Introduction to Software Engineering (Summer 2018) - Quiz 01

Mark the correct answers for each question and click the "Submit" button, when you're done.

1) What are the Components of a System Model?

Points: 1

Select the correct answer

Object Model, Functional Model and Issue Model

Object Model, Functional Model and Dynamic Model

Object Model, Task Model, and Dynamic Model

Select one
correct answer

Questions: 1
Total Points: 1

Remaining Time: 1 min 43 s
Saved: never
Connected

Click on
Submit

Task 6: Wait for the results

- The results will be displayed automatically shortly after the quiz has finished
- **No need to reload the page**

Introduction to Software Engineering (Summer 2018) - Quiz 01

Please wait until the quiz has ended.

Your answers have been successfully submitted!

The results will appear when the quiz has ended.

1) What are the main phases of the Waterfall Model?

Select the correct answer

Points: 1

Questions: 1
Total Points: 1

Remaining Time: 44 s
Submitted: a few seconds ago
● Connected

Submitted

Teaching Methodology

- We intermix lectures and exercises
- You have 2 time slots:
 - Thursday: Main lecture
 - Throughout the week: Your own tutor group
- In each time slot ...
 - ... we teach you one or more concepts
 - ... we ask you to apply these concepts in in-class exercises
- Different names for this type of teaching:
 - Experiential learning
 - Blended learning
 - Just in time learning
 - Continuous learning
 - Interactive learning
 - Chaordic learning

➤ **Make sure to come with your laptop (and your smartphone) to class.**

9. Exercise Organization

- 4 Exercise Instructors



Dr. Juan
Haladjian



Dr. Stephan
Krusche



Mariana
Avezum



Nadine von
Frankenberg

- 46 Tutors (see next 3 slides)

EIST 2018 Tutors



Adrian Hözl



Aleena Yunus



Alexander
Zellner



Ali Abbas Jaffri



Anna Kovaleva



Christina
Halemba



Christopher
Lass



Clemens Horn



Daniel Eler



David Bani-
Harouni



Deborah Höltje



Erasmus
Hertel



Felix
Matschilles



Gregor
Zieglturm



Haifa Haddad



Hanya
Elhashemy



Jakob
Smretschnig



Jens Klinker

EIST 2018 Tutors (ctd)



Johann
Rottenfußer



Johannes Weiß



Julian Frattini



Julian Villing



Kevin Huang



Konstantin
Karas



Laurenz
Baumgart



Ljube
Boskovski



Marie Schmidt



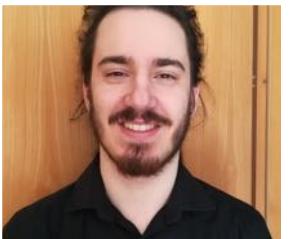
Mariyam
Fedoseeva



Marko
Stapfner



Matthias
Hamacher



Maximilian
Niedermeier

Bernd Bruegge



Moritz
Hofmann



Murat Güner



Nadia Riedl



Nadine
Angermeier



Nico Christley

EIST 2018 Tutors (ctd)



Nina-Mareike
Harders



Ninaj Nyaranga
Wambugu



Nityananda
Zibil



Philipp
Sedlmeier



Sharmin Khan



Sophie Schüle



Simon van
Endern



Thomas Eidens



Timo
Zandonella



Zaim Sari

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



Assumptions for this Course

- You have a basic knowledge of Java and know about object-oriented programming constructs
- You have solved small programming projects
- For example, you have taken:
 - **IN0001** Introduction to Informatics 1
 - **IN1501** Fundamentals of Programming
- 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 in developing a software system.

Objectives of the Class

- Appreciate the Fundamentals of Software Engineering
 - Modeling
 - Software lifecycle models
 - Requirements engineering
 - Analysis
 - System design (software architecture and detailed software design)
 - Object design (Reuse, Patterns, Interface specification)
 - Implementation
 - Testing (unit testing, integration testing, system testing)
 - Configuration management, build management and release management
 - System maintenance and evolution
 - Project organization.

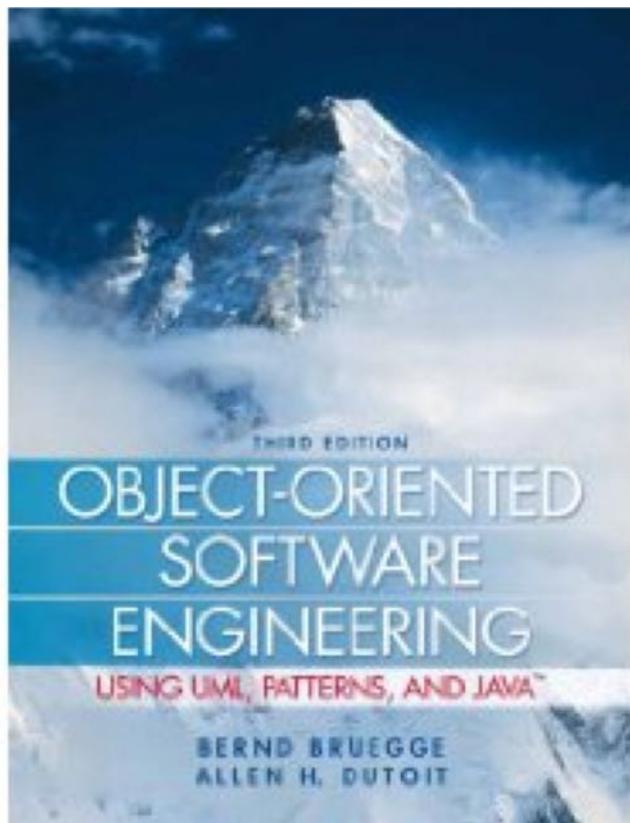
Course Schedule (Preliminary)

#	Date	Subject
1	12.04.18	Introduction
2	19.04.18	Mini Intro into Software Engineering
3	26.04.18	Requirements Elicitation and Analysis
4	03.05.18	System Design I
X	10.05.18	No class
5	17.05.18	System Design II
6	24.05.18	Object Design
X	31.05.18	No class
7	07.06.18	Model Transformations and Refactoring
8	14.06.18	Design Patterns
9	21.06.18	Lifecycle Modeling
10	28.06.18	Configuration Management
11	05.07.18	Testing
12	12.07.18	Project Management
13	19.07.18	Wrap-up and Exam Preparation

Times and Locations

- Main lecture: Thursday 8:00 – 11:00
 - For the exercises you need register via TUM Online
 - Why? We are going to create accounts for you at our department's infrastructure for submission
- We are filming in the MI HS1 Lecture Hall and transmit the lecture via video conference into the MW2001 lecture hall (Thanks to the RBG)
- We are also providing a live stream for each lecture (Thanks to Ruth Demmel and the RBG)
- Each recorded lecture will be stored on Moodle
- Tutors will be in each lecture hall to help you with the in-class exercises

Textbook



Bernd Bruegge, Allen H. Dutoit

Object-Oriented Software Engineering: Using UML, Patterns
and Java, 3rd Edition

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

ISBN-10: 0136061257

ISBN-13: 978-0136061250

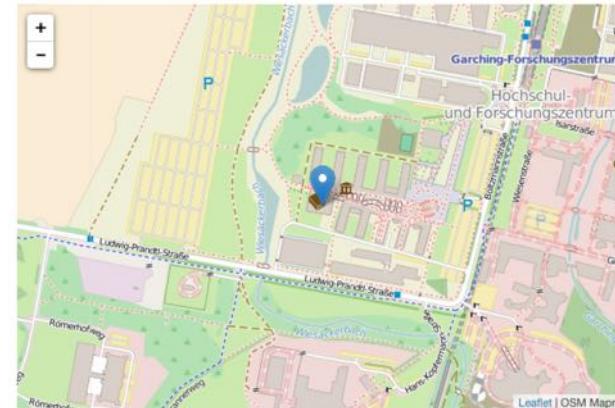
- Additional readings will be added during each lecture.

Book Availability On Campus

- Copies in the MI library
 - <https://www.ub.tum.de/en/branch-library-mathematics-informatics>
- Additional copies in the mechanical engineering library
 - <https://www.ub.tum.de/en/branch-library-mechanical-engineering>

Branch Library Mathematics & Informatics

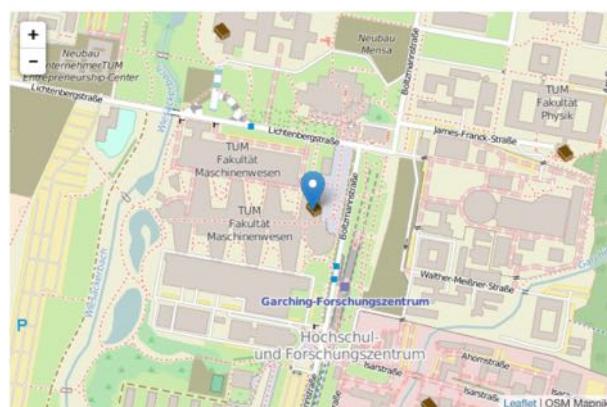
Boltzmannstraße 3
85748 Garching



Email: information@ub.tum.de
Telephone: 089-289-16900

Branch Library Mechanical Engineering

Boltzmannstraße 15
85748 Garching
Fakultätsgebäude Maschinenwesen
Gebäu deteil 0, 1. OG



Email: information@ub.tum.de
Telephone: 089-289-16368

International Edition and eBook Versions available

The screenshot shows a web browser displaying the Pearson website at pearsoned.co.uk. The page is for the book "Object-Oriented Software Engineering Using UML, Patterns, and Java: Pearson New International Edition". The book cover is visible on the left, showing a blue abstract design. A red circular badge in the top right corner of the cover area says "Save: £5.40". The main title and subtitle are displayed prominently. Below the title, it says "3rd Edition", "Bernd Bruegge, Allen Dutoit", "Jul 2013, Paperback, 728 pages", and "ISBN: 9781292024011". A note in red text says "For orders to USA, Canada, Australia, New Zealand or Japan visit your local Pearson website". There is a "Buy" button. A table below lists the book's availability in different formats: Paperback (RRP £53.99, Your Price £48.59) and PDF eBook (£26.99, £24.29). At the bottom, there are links for "Print page", "Email page", and "Share". Navigation tabs include "Description", "Table of Contents", "Features", and "Reviews". The left sidebar shows a navigation menu under "Higher Education" with categories like Accounting & Finance, Business & Management, Computing and Computer Science, Economics, Education & Teacher Training, Engineering, English, Foreign Languages, Geography, History, Hospitality, Leisure & Tourism, Humanities, Law & Criminology, Marketing, Mathematics & Statistics, and Media, Journalism & Cultural Studies. The URL in the address bar is pearsoned.co.uk/bookshop/object-oriented-software-engineering-using-uml-patterns-and-java-pearson-new-international-edition-9781292024011.

pearsoned.co.uk

PEARSON

Higher Education Professional Advanced Search Higher Education Search by keyword, author, title, ISBN

What's New Top Textbooks Offers Tip of the Week My Account

Bookshop > Object-Oriented Software Engineering Using UML, Patterns, and Java: Pearson New International E...

Higher Education

Accounting & Finance
Business & Management
Computing and Computer Science
Economics
Education & Teacher Training
Engineering
English
Foreign Languages
Geography
History
Hospitality, Leisure & Tourism
Humanities
Law & Criminology
Marketing
Mathematics & Statistics
Media, Journalism & Cultural Studies

**Object-Oriented Software Engineering Using UML, Patterns, and Java:
Pearson New International Edition**

3rd Edition
Bernd Bruegge, Allen Dutoit
Jul 2013, Paperback, 728 pages
ISBN: 9781292024011

For orders to USA, Canada, Australia, New Zealand or Japan visit your local Pearson website

Be the first to review this product >
Special online offer - Save 10%
Was £53.99, Now £48.59

Buy

This title is available in the following formats:

Format	RRP	Your Price
Paperback	£53.99	£48.59
PDF eBook	£26.99	£24.29

Print page Email page Share

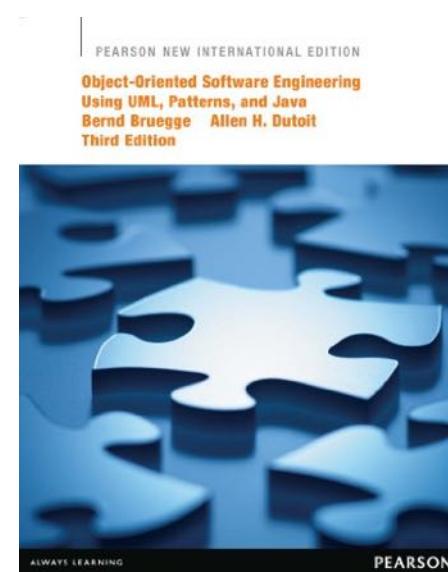
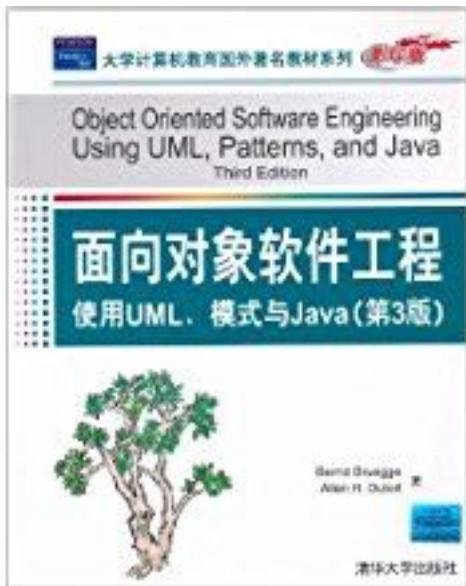
Description Table of Contents Features Reviews

For courses in Software Engineering, Software Development, or Object-Oriented Design and Analysis at the Junior/Senior or Graduate level. This text can also be utilized in short technical courses or in short, intensive management courses.

Bernd Bruegge

Introduction to Software Engineering 552010

Other OOSE Editions

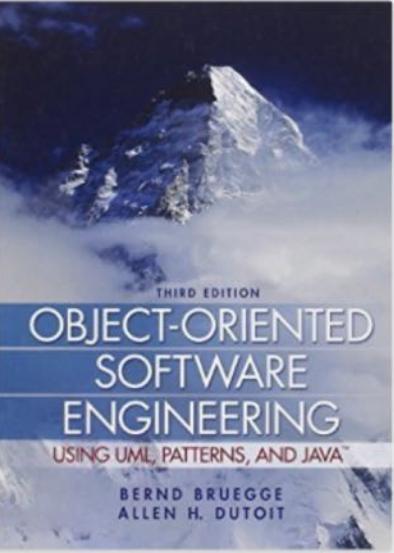


Used copies are also available online...

...but be aware of ridiculous offers

Bücher Erweiterte Suche Stöbern Bestseller Neuheiten Hörbücher Fremdsprachige Bücher Taschenbücher Fachbücher Schulbücher Angebote

« Zurück zu den Suchergebnissen für "brügge dutoit"



Object-Oriented Software Engineering Using UML, Patterns, and Java (3rd Edition) by Bernd Bruegge (2009-08-08)

Gebundene Ausgabe – 1656
von Bernd Bruegge; Allen H. Dutoit (Autor)
Geben Sie die erste Bewertung für diesen Artikel ab

Alle Formate und Ausgaben anzeigen

**Gebundene Ausgabe
ab EUR 512,06**

1 neu ab EUR 512,06

Hinweis: Dieser Artikel ist nur bei Drittanbietern erhältlich ([alle Angebote anzeigen](#)).

Die Spiegel-Bestseller
Entdecken Sie die Bestseller des SPIEGEL-Magazins aus unterschiedlichen Bereichen. Wöchentlich aktualisiert. [Hier klicken](#)

Empfehlen    

1 neu ab EUR 512,06

Alle Angebote

Auf die Liste

Möchten Sie verkaufen? Bei Amazon verkaufen?

Dieses Bild anzeigen

Grading Criteria

- To pass this course, your exam grade must be 4.0 or better
- If your grade is 4.0 or better you can improve your exam grade with a bonus 0.3 - 1.0:

How do I get the lecture bonus?

- Up to 20% regular participation in the morning quiz
- Up to 20% in-class exercises during the lecture
- Up to 60% homework submissions plus presentations in tutor group

Intermediate Summary

- Software engineering is problem solving
- We need to talk to the customer and domain experts to identify the problem to be solved("elicit the requirements")
- Some problems are impossible to solve, others are realizable only after they are changed or reinterpreted
- Software engineering is about dealing with change and complexity while trying to solve the problem
- Software engineering uses methods such as divide&conquer and tools (CASE) to solve problems

Overview

1

Problem Solving

2

Software Engineering Definition

3

EIST Organization

4

Systems, Models and Views

Phenomenon vs Concept

- **Phenomenon**
 - An object in the world of a domain as you perceive it
 - Examples: This EIST lecture at 9:25, 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

Concept= Name

Purpose

Members (Phenomena)

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**

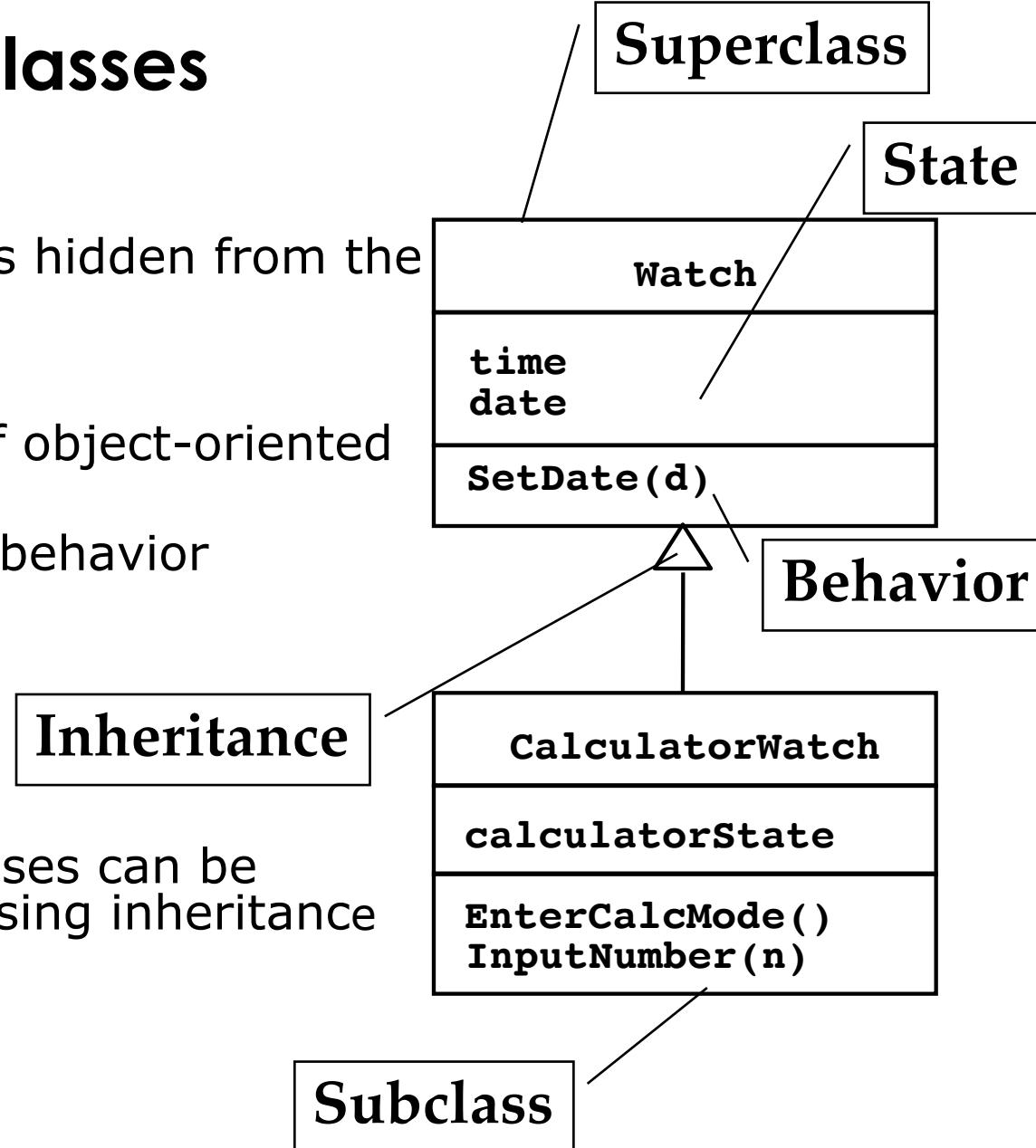
- A type whose implementation is hidden from the rest of the system

- **Class:**

- An abstraction in the context of object-oriented languages
- A class encapsulates state and behavior
 - Example: Watch

Unlike abstract data types, subclasses can be defined in terms of other classes using inheritance

- Example: CalculatorWatch



Type and Instance

- **Type:**
 - A concept in the context of programming languages
 - **Name:** int
 - **Purpose:** integral number
 - **Members:** 0, -1, 1, 2, -2, ...
 - **Name:** boolean
 - **Purpose:** logical
 - **Members:** {true, false}
- **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
- A **view** depicts selected aspects of a model
- A **notation** is a set of graphical or textual rules for depicting models and views:
 - Informal notations (“napkin design”)
 - Formal notations (UML)

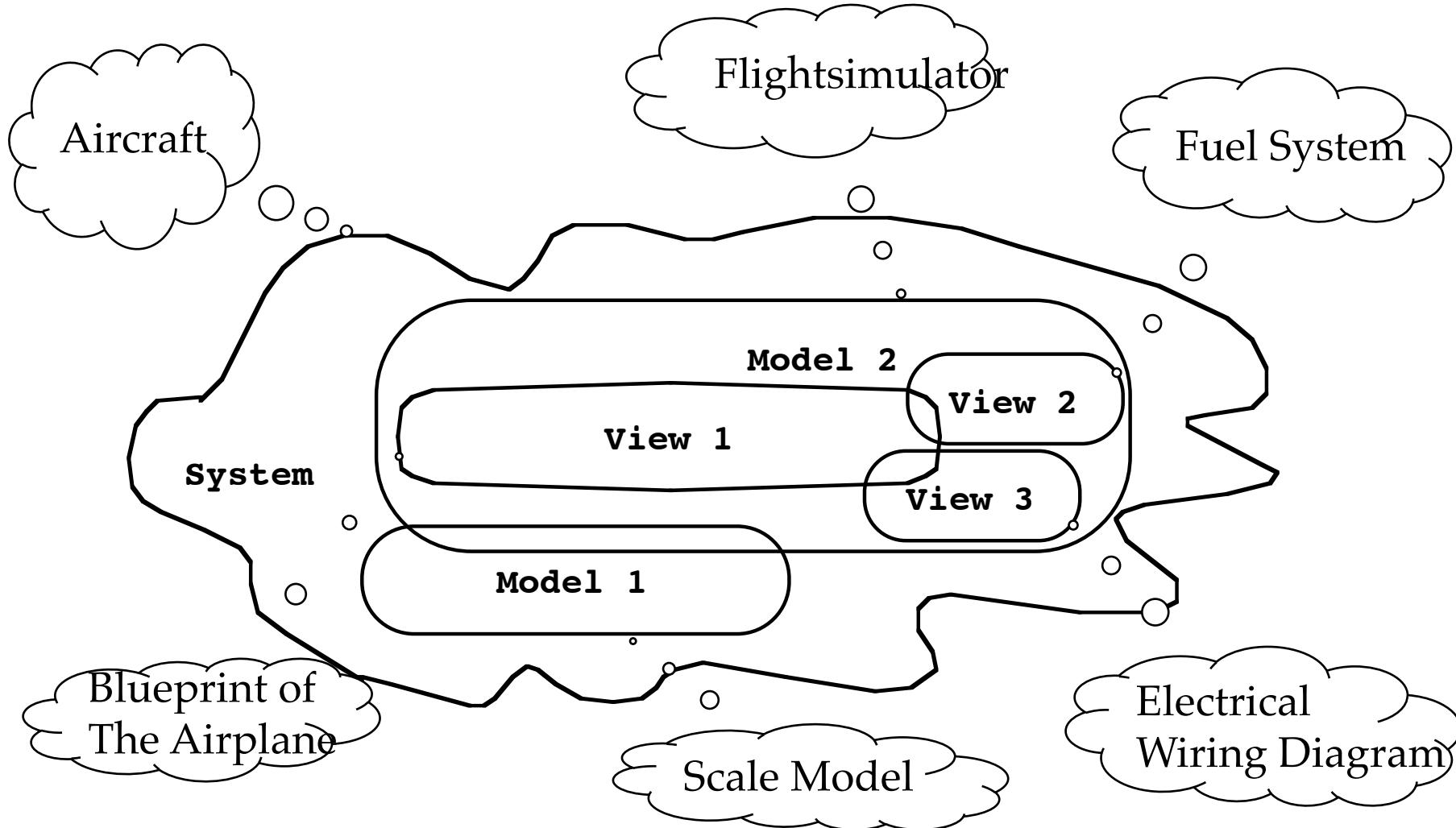
System:
Airplane

Model:
Flight simulator
Scale model

View:
Blueprint of the airplane
Electrical wiring diagram
An airplane breaking the sound barrier



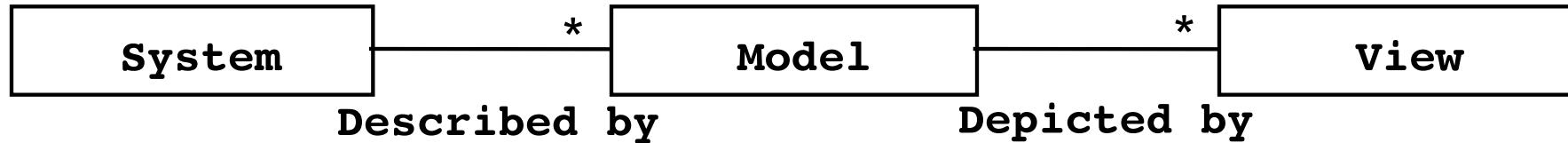
Systems, Models and Views (“Napkin” Notation)



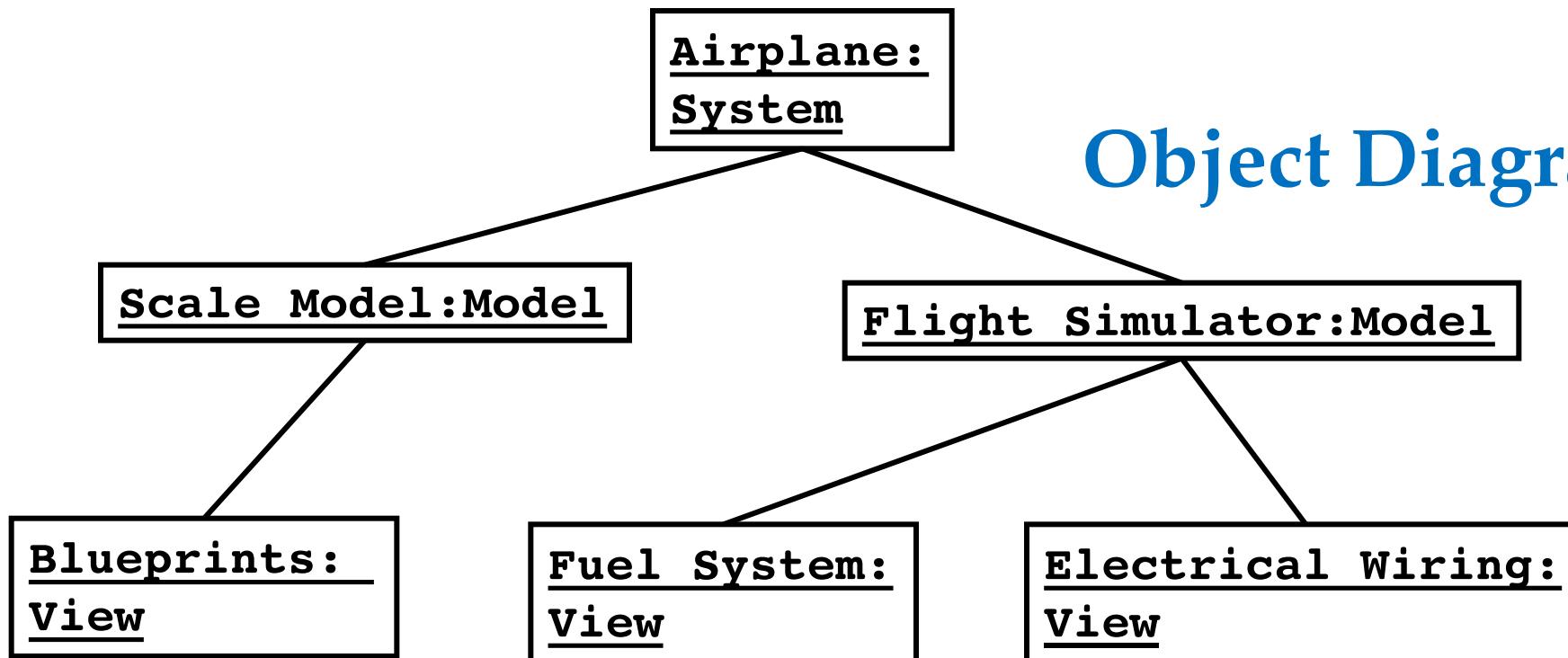
Views and models of a complex system usually overlap

Systems, Models and Views (UML Notation)

Class Diagram



Object Diagram



Summary

- Software engineering is **problem solving**
- We need to talk to the customer and domain experts to identify the problem to be solved("elicit the requirements")
- Some problems are impossible to solve, others are realizable only after they are changed or reinterpreted
- Software engineering is about dealing with **change** and **complexity** while trying to solve the problem
- Software engineering uses methods such as divide & conquer and tools (CASE, IDEs) to solve problems.
- Your task for next Thursday:
 - Read Chapter 1 and 2 from the text book
 - The morning quiz will be about material from today's slides and chapter 1 and 2.

Morning Quiz

- Quiz 2, 18 April 2018
 - Starting Time: 8:00
 - End Time: 8:10
- To do the quiz, use ArTEMiS
- The Lecture starts at 8:10

Introduction to Software Engineering (Summer 2018)			
Exercise	Due date	Results	Actions
Quiz 01	6 days ago	You have not participated in this quiz.	 Practice
Programming Exercise Tutorial	in 4 days	You have not started this exercise yet.	 Start exercise
Good Morning Quiz 02		The quiz hasn't started yet.	 Open Quiz

Model-Based Software Engineering: An Introduction

Prof. Bernd Bruegge, PhD

Chair for Applied Software Engineering

Technische Universität München

12 April 2018

EIST

Denali, Base Camp, 14400 ft

Roadmap for the Lecture

- **Context and Assumptions**

- You have attended lecture 1
- You are familiar with the content of Chapter 1 and 2 in the OOSE text book by Bruegge and Dutoit

- **Content of this lecture**

- A first iteration on software development

- **Objective:** At the end of this lecture you are able to...

- ... generate an executable program from a problem statement
- ... perform activities and create entities of a typical software lifecycle
 - Entities: use case model, object model, source code, deliverable
 - Activities: analysis, design, implementation, delivery
- ... understand use case diagrams that model the functionality of a system
- ... understand a class diagram that model the structure of a system
- ... understand a user interface as an example of a dynamic model.

Outline of this lecture

✓ Miscellaneous

- ✓ Morning Quiz
- Tutor Group Selection
- Exam Dates
- Software Lifecycle
- Problem Statement
- UML
- Analysis, Design, Implementation and Delivery of a Game

Learning Goals:

- You start with a problem statement from the customer
- You use a game to walk through the activities of a tailored software lifecycle
- You learn a notation that can be used uniformly throughout the activities: UML.

Tutor Group Selection

We had many matching issues:

- Registration, (Non) Participation, and outcome

We are sorry, but we are not able to answer your emails

1) If you have not been matched:

- Please register in TUM Online for an open tutor group (first come, first serve)
- Look first at Moodle to find a free tutor group.

Tutor Group Selection (ctd)

2) If you have been matched, but you want to change:

- You can only change to a tutor group in the same time slot
- You have to deregister from the old group in TUM Online and register for the new group (first come, first server) in TUM Online
- Make sure to notify your current and your new tutor

Please note:

- If you are not officially registered for a tutor group in TUM Online, your homework will not be corrected and you cannot obtain a bonus
- Changes in tutor groups are only possible until April 27, 2018

Final Exam

- The final exam is scheduled on Friday, July 27, 16:00 - 18:30
- The exam will be in English
- You can answer it either in English or German (do not mix languages)

Outline of this lecture

✓ Miscellaneous

- ✓ Morning Quiz
- ✓ Tutor Group Selection
- ✓ Exam Dates

→ Software Lifecycle

- Problem Statement
- UML
- Analysis, Design, Implementation and Delivery of a Game

Learning Goals:

- You start with a problem statement from the customer
- You use a game to walk through the activities of a tailored software lifecycle
- You learn a notation that can be used uniformly throughout the activities: UML.

Definitions

- **Software lifecycle:**

- Set of activities and their relationships to each other to support the development of a software system

Examples of activities:

- Requirements Elicitation, Analysis, System Design, Implementation, Testing, Configuration Management, Delivery

Examples of relationships:

- Testing must be done before Implementation
- Analysis and System Design can be done in parallel

- **Software lifecycle model:**

- An abstraction representing the development of software for the purpose of understanding, monitoring, or controlling the software.

Identifying Activities for a Software Lifecycle

What is the problem?

What is the solution?

**What are the best mechanisms
to implement the solution?**

**How is the solution
constructed?**

Is the problem solved?

Can the customer use the solution?

Are enhancements needed?

Software Development Activities – Example

Requirements Analysis

What is the problem?

Application
Domain

System Design

What is the solution?

Detailed Design

What are the best mechanisms
to implement the solution?

Program Implementation

How is the solution
constructed?

Solution
Domain

Testing

Is the problem solved?

Delivery

Can the customer use the solution?

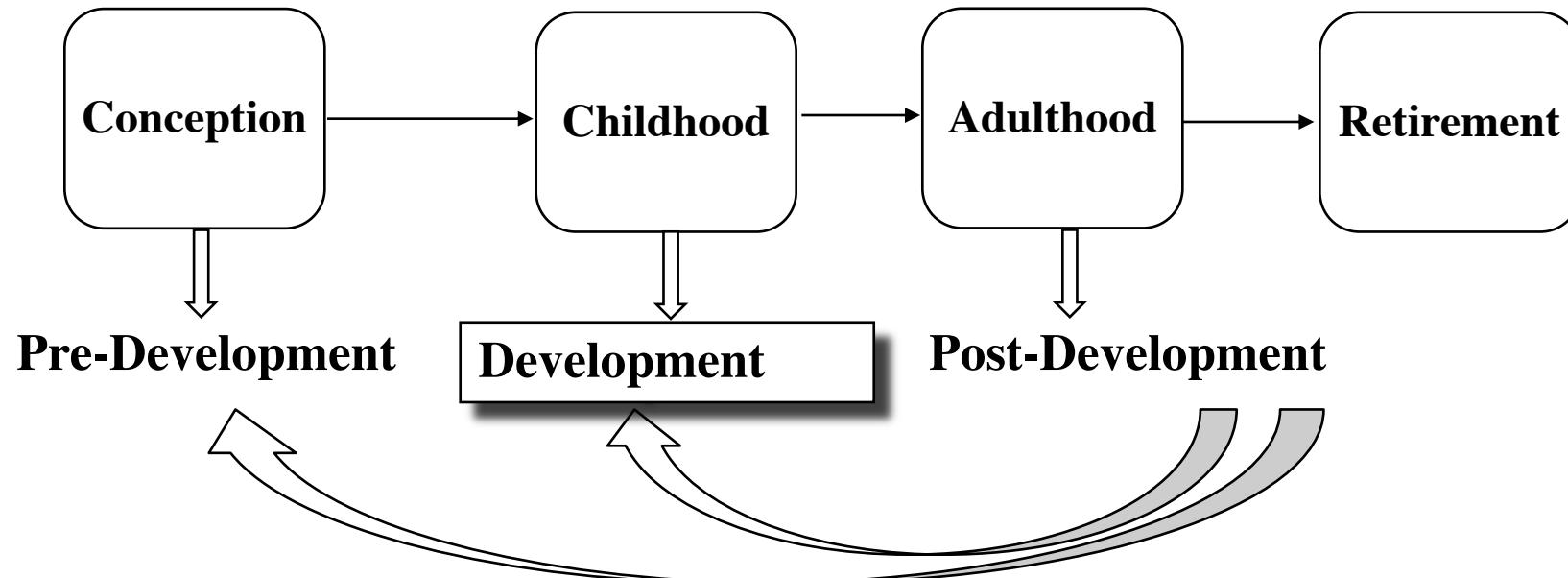
Maintenance

Are enhancements needed?

Software Lifecycle

More details in Lecture 9:
Lifecycle Modeling

The term “Lifecycle” is based on the metaphor of the life of a person:



Tailoring

- There is no “one size fits all” software lifecycle model that works for all possible software engineering projects
- Tailoring: Adjusting a lifecycle model to fit a project
 - Naming: Adjusting the naming of activities
 - Cutting: Removing activities not needed in the project
 - Ordering: Defining the order of the activities.

**Today we are using a tailored Software Process Lifecycle for the development of a Game:
Analysis, Object Design, Implementation & Delivery**

Controlling Software Development with a Process

How do we control software development? Two opinions:

- Through **organizational maturity** (Humphrey)
 - Defined process, Capability Maturity Model (CMM)
- Through **agility** (Schwaber):
 - Large parts of software development is empirical in nature; they cannot be modeled with a defined process
 - There is a difference between a defined and an empirical process
- How can software development better be described?
 - with a **defined process** control model?
 - with an **empirical process** control model?

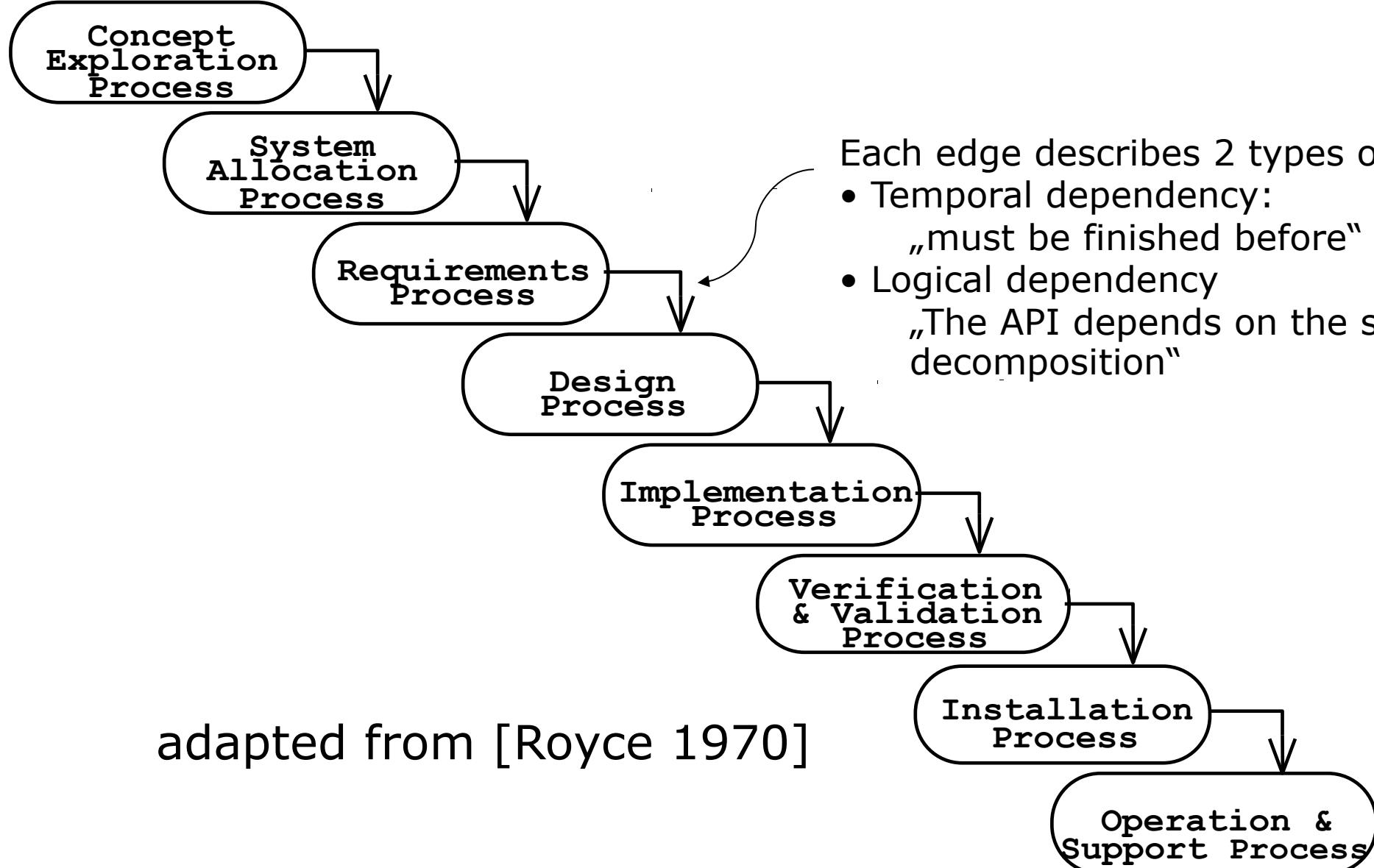
Defined Process Control Model

Defined process

- Given a well-defined set of inputs, the same outputs are generated every time
- All activities and tasks are well-defined
- Deviations are errors that need to be corrected
- Precondition to apply this model
 - Requires that every piece of work is completely understood before the process starts
- If the preconditions are not satisfied
 - Lot of surprises, loss of control, wrong work products,...
- Conditions when to apply this model:
 - Change can be ignored, the output is predictable.

Defined processes do not deal well with Interferences.

Example of a Defined Software Process: The Waterfall Model

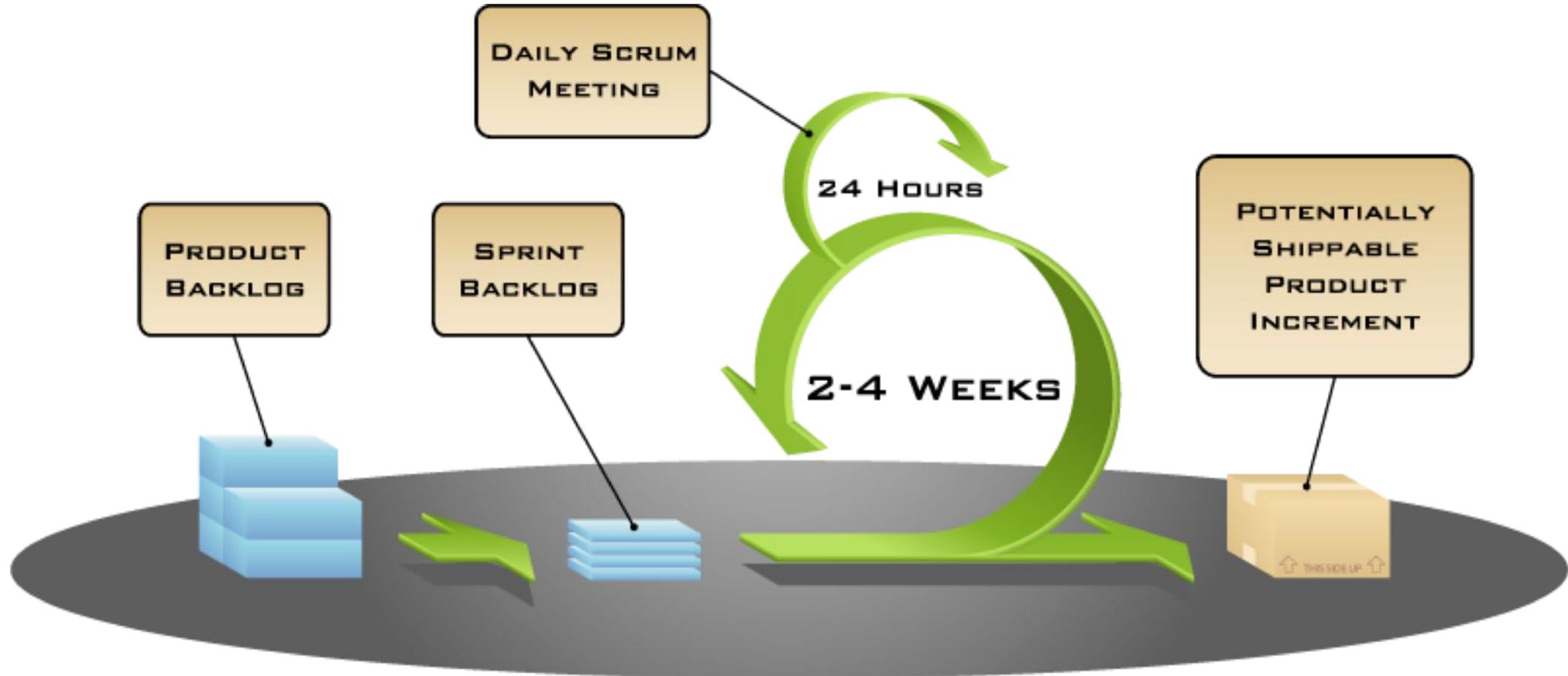


adapted from [Royce 1970]

Empirical Process Control Model

- **Empirical process**
 - An imperfectly defined process, not all pieces of work are completely understood
 - Given a well-defined set of inputs, different outputs may be generated when the process is executed
- Deviations are seen as opportunities that need to be investigated
 - The empirical process “expects the unexpected”
- Control and risk management is exercised through frequent inspection
- **Scrum is an empirical process model.**

Example of an Empirical Process Control Model: Scrum



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

Outline of this lecture

✓ Miscellaneous

- ✓ Morning Quiz
- ✓ Tutor Group Selection
- ✓ Exam Dates

✓ Software Lifecycle

→ Problem Statement

- UML
- Analysis, Design, Implementation and Delivery of a Game

Learning Goals:

- You start with a problem statement from the customer
- You use a game to walk through the activities of a tailored software lifecycle
- You learn a notation that can be used uniformly throughout the activities: UML.

Problem Statement

- A description of the **problem addressed by the system**
- Synonym: Statement of work
- A problem statement describes
 - The current situation
 - The functionality the new system should support
 - The environment in which the system will be deployed
 - Deliverables expected by the client
 - Delivery dates (milestones)
 - A set of acceptance criteria (criteria for system tests)

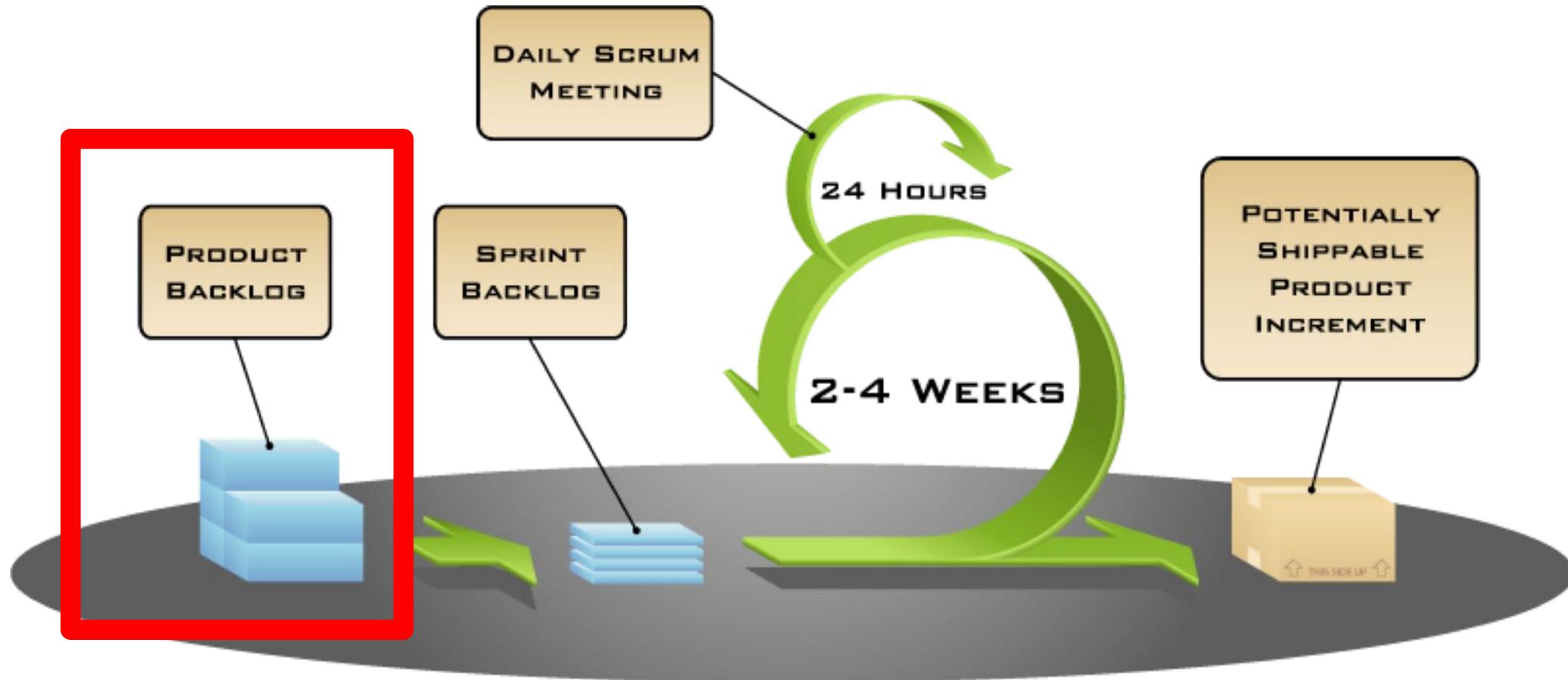
Example of a Problem Statement: Bumpers

Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change it's speed.

The game should be platform independent, and visualizes different parameters of the car, e.g. the speed, consumption and location of the car. When cars crash, there has to be a sound effect. The game should support different collisions and the determination of the collision winner should be changeable during gameplay.

To develop Bumpers we create a Product Backlog from the Problem Statement



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

Bumpers – Problem Statement

Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

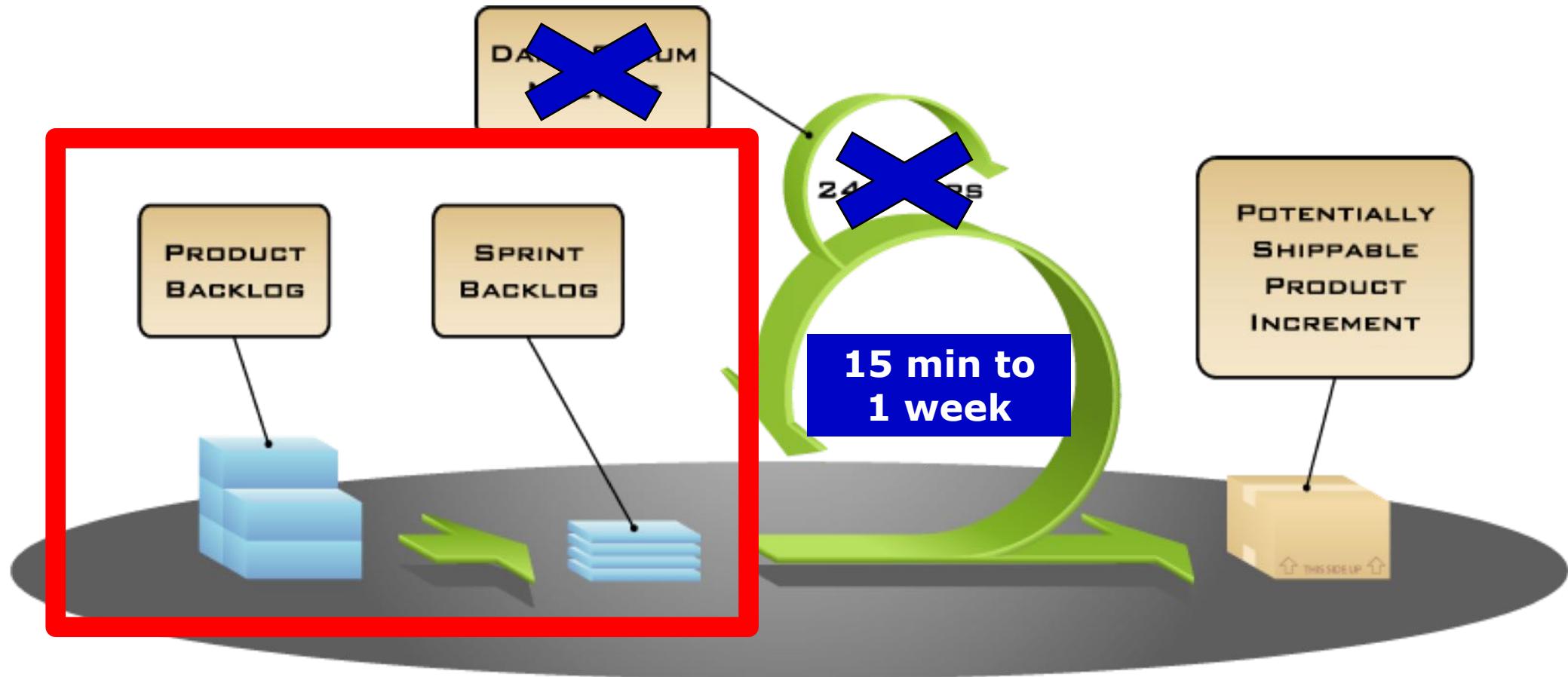
A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change its speed.

The game should be platform independent, and visualizes different parameters of the car, e.g. the speed, consumption and location of the car. When cars crash, there has to be a sound effect. The game should support different collisions and the determination of the collision winner should be changeable during gameplay.

Bumpers – Initial Product Backlog

1. User Interface design of the game board
2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. The player starts and stops the game
6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play.

Today we use a Modified Form of Scrum



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

Tailored Software Lifecycle for today's lecture



- Each sprint has a sprint backlog which is a subset of the product backlog
- Each sprint is either an in-class exercise or a homework
- After each sprint there is a review
 - Our sample solution or a solution from you
- The customer can change the product backlog after any sprint.

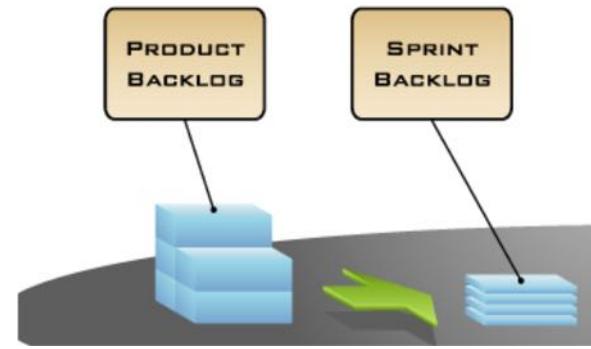
Bumpers – Initial Product Backlog

1. User Interface design of the game board
2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. The player starts and stops the game
6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the “right before left” rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play

Sprint Backlog for Sprint 1

1. User Interface design of the game board
2. **Cars drive on the game board**
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. **The player starts and stops the game**
6. **Music plays when the game begins and stops to play when the game ends**
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed

10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play.



How do we proceed?

- Before we jump into the implementation activity we need to step back up and create the system model
- The system model consists of...
 - The functional model: It represents the **functionality** of a system
 - The object model: It represents the **structure** of the system
 - The dynamic model represents the **behavior** of the system
- We will use UML as a notation to draw these models
- For Bumpers we will start with two models, the functional and the object model, and two UML notations:
 - UML use case diagrams
 - UML class diagrams.

Outline of this lecture

- ✓ Miscellaneous

- ✓ Morning Quiz

- ✓ Tutor Group Selection

- ✓ Exam Dates

- ✓ Software Lifecycle

- ✓ Problem Statement



- Analysis, Design, Implementation and Delivery of a Game

Learning Goals:

- You start with a problem statement from the customer
- You use a game to walk through the activities of a tailored software lifecycle
- You learn a notation that can be used uniformly throughout the activities: UML.

UML Overview

- UML (Unified Modeling Language)
 - A standard for modeling software systems
 - Object Management Group: <http://www.uml.org/>
 - Convergence of notations used in object-oriented methods
 - OMT (James Rumbaugh and colleagues)
 - Booch (Grady Booch)
 - OOSE (Ivar Jacobson)
- Current Version: UML 2.5.1 (December 2017)
 - Basic Information at <http://www.uml.org/what-is-uml.htm>
- Commercial tools: Visual Paradigm, Rational (IBM), Together (Borland), Visual Architect (business processes, BCD)
- Open Source tools: ArgoUML, StarUML, Umbrello, Unicase
- Commercial and Opensource: PoseidonUML (Gentleware)

History of UML

- Unified Modeling Language: Convergence of different notations for object-oriented methods, mainly
 - Booch Notation
 - OMT
 - OOSE



James Rumbaugh focused on **object-oriented analysis**. OMT was the first notation combining data modeling with inheritance



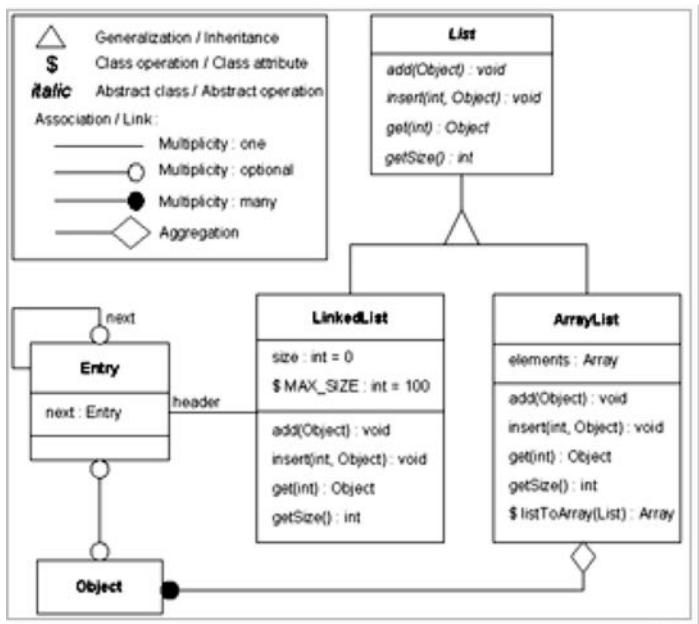
Ivar Jacobson stressed the importance of **functional modeling** as part of object-oriented methods (OOSE)
www.ivarjacobson.com



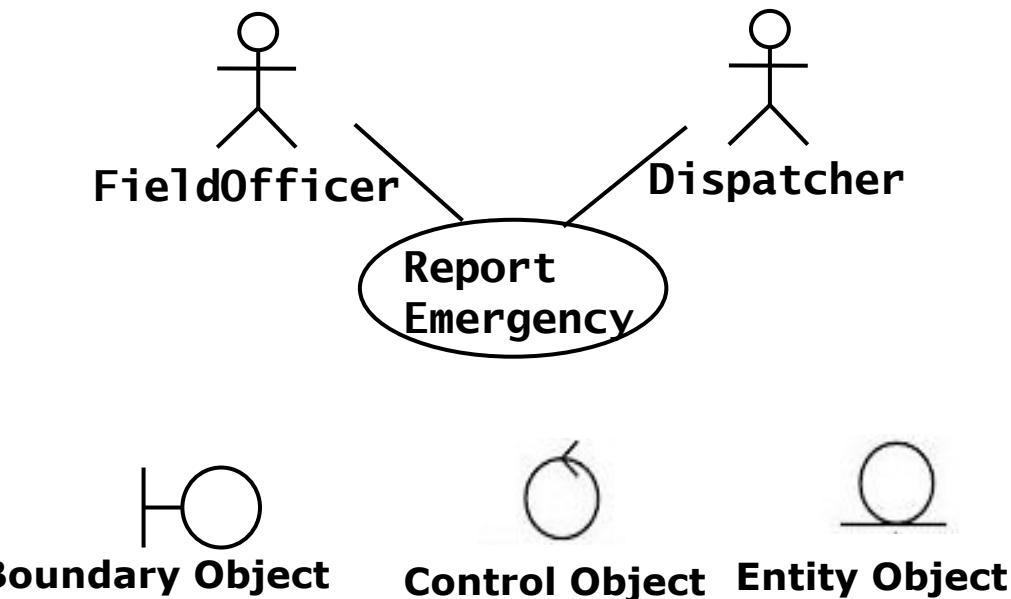
Grady Booch developed the "Booch clouds" focusing on **object-oriented design**
www.booch.com/

OMT, OOSE and Booch Notation

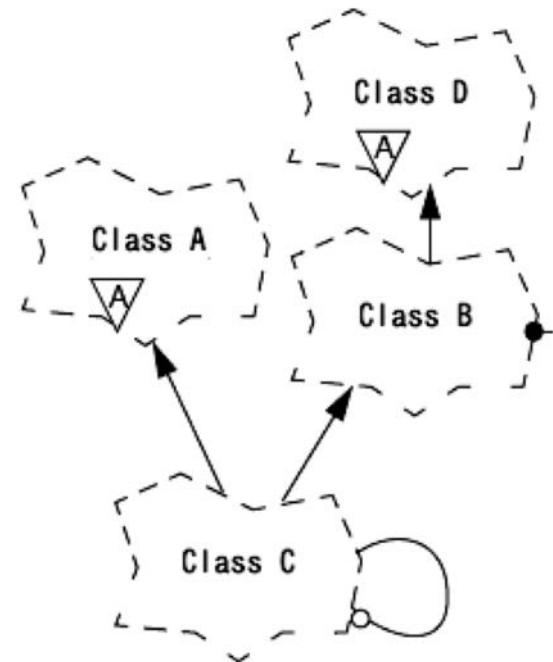
OMT Notation



OOSE Notation



Booch Notation



Why Do We use UML?

- It reduces complexity by **focusing** on abstractions



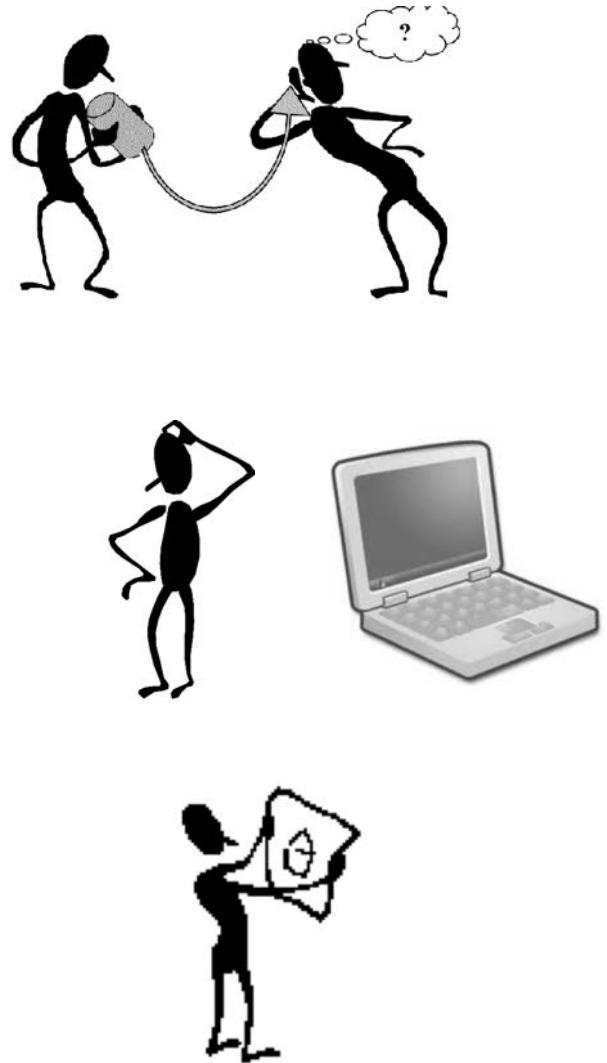
- It can be seen as a high level “Programming Language” enabling the **generation** of source **code**

- It is a mean for **communication** between people involved in a software project.



There are 3 Ways to use UML Models

- **Communication:** UML provides a common vocabulary for informal communication (“informal model”, “conceptual model”)
 - Target is a human being (developer, end user)
- **Analysis and Design:** UML Models enable developers to specify a future system
 - Target is a tool (CASE tool, compiler)
- **Archival:** UML Models provide a way for storing the design and rationale of an existing system
 - Target is a human (analyst, project manager).



**More details in Lecture 12:
Project Management**

Application vs Solution Domain

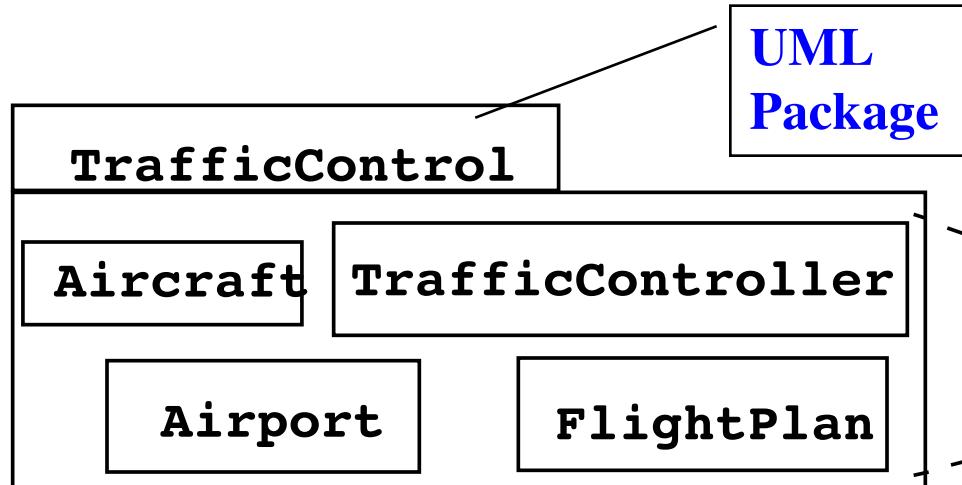
- Application Domain (Analysis):
 - The environment in which the system is operating
- Solution Domain (Design, Implementation):
 - The technologies used to build the system
- Both domains contain abstractions that we can use for the construction of the system model.

UML: One notation for Analysis and Design

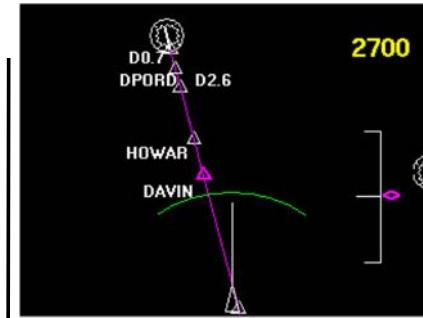


Application Domain
(Phenomena)

System Model (Concepts) *Analysis*



Reality



DE/FCTR	Pax/Passenger	Ind	W	H	L	U	Common	Details
1/01	14.04	*	15.8100	10	14.14	15.8100	1	
1/02	14.04	*	15.8100	10	14.14	15.8100	1	
1/03	14.04	*	15.8100	10	14.14	15.8100	1	
1/04	14.04	*	15.8100	10	14.14	15.8100	1	
1/05	14.04	*	15.8100	10	14.14	15.8100	1	
1/06	14.04	*	15.8100	10	14.14	15.8100	1	
1/07	14.04	*	15.8100	10	14.14	15.8100	1	
1/08	14.04	*	15.8100	10	14.14	15.8100	1	
1/09	14.04	*	15.8100	10	14.14	15.8100	1	
1/10	14.04	*	15.8100	10	14.14	15.8100	1	
1/11	14.04	*	15.8100	10	14.14	15.8100	1	
1/12	14.04	*	15.8100	10	14.14	15.8100	1	
1/13	14.04	*	15.8100	10	14.14	15.8100	1	
1/14	14.04	*	15.8100	10	14.14	15.8100	1	
1/15	14.04	*	15.8100	10	14.14	15.8100	1	
1/16	14.04	*	15.8100	10	14.14	15.8100	1	
1/17	14.04	*	15.8100	10	14.14	15.8100	1	
1/18	14.04	*	15.8100	10	14.14	15.8100	1	
1/19	14.04	*	15.8100	10	14.14	15.8100	1	
1/20	14.04	*	15.8100	10	14.14	15.8100	1	
1/21	14.04	*	15.8100	10	14.14	15.8100	1	
1/22	14.04	*	15.8100	10	14.14	15.8100	1	
1/23	14.04	*	15.8100	10	14.14	15.8100	1	
1/24	14.04	*	15.8100	10	14.14	15.8100	1	
1/25	14.04	*	15.8100	10	14.14	15.8100	1	
1/26	14.04	*	15.8100	10	14.14	15.8100	1	
1/27	14.04	*	15.8100	10	14.14	15.8100	1	
1/28	14.04	*	15.8100	10	14.14	15.8100	1	
1/29	14.04	*	15.8100	10	14.14	15.8100	1	
1/30	14.04	*	15.8100	10	14.14	15.8100	1	
1/31	14.04	*	15.8100	10	14.14	15.8100	1	
1/32	14.04	*	15.8100	10	14.14	15.8100	1	
1/33	14.04	*	15.8100	10	14.14	15.8100	1	
1/34	14.04	*	15.8100	10	14.14	15.8100	1	
1/35	14.04	*	15.8100	10	14.14	15.8100	1	
1/36	14.04	*	15.8100	10	14.14	15.8100	1	
1/37	14.04	*	15.8100	10	14.14	15.8100	1	
1/38	14.04	*	15.8100	10	14.14	15.8100	1	
1/39	14.04	*	15.8100	10	14.14	15.8100	1	
1/40	14.04	*	15.8100	10	14.14	15.8100	1	
1/41	14.04	*	15.8100	10	14.14	15.8100	1	
1/42	14.04	*	15.8100	10	14.14	15.8100	1	
1/43	14.04	*	15.8100	10	14.14	15.8100	1	
1/44	14.04	*	15.8100	10	14.14	15.8100	1	
1/45	14.04	*	15.8100	10	14.14	15.8100	1	
1/46	14.04	*	15.8100	10	14.14	15.8100	1	
1/47	14.04	*	15.8100	10	14.14	15.8100	1	
1/48	14.04	*	15.8100	10	14.14	15.8100	1	
1/49	14.04	*	15.8100	10	14.14	15.8100	1	
1/50	14.04	*	15.8100	10	14.14	15.8100	1	
1/51	14.04	*	15.8100	10	14.14	15.8100	1	
1/52	14.04	*	15.8100	10	14.14	15.8100	1	
1/53	14.04	*	15.8100	10	14.14	15.8100	1	
1/54	14.04	*	15.8100	10	14.14	15.8100	1	
1/55	14.04	*	15.8100	10	14.14	15.8100	1	
1/56	14.04	*	15.8100	10	14.14	15.8100	1	
1/57	14.04	*	15.8100	10	14.14	15.8100	1	
1/58	14.04	*	15.8100	10	14.14	15.8100	1	
1/59	14.04	*	15.8100	10	14.14	15.8100	1	
1/60	14.04	*	15.8100	10	14.14	15.8100	1	
1/61	14.04	*	15.8100	10	14.14	15.8100	1	
1/62	14.04	*	15.8100	10	14.14	15.8100	1	
1/63	14.04	*	15.8100	10	14.14	15.8100	1	
1/64	14.04	*	15.8100	10	14.14	15.8100	1	
1/65	14.04	*	15.8100	10	14.14	15.8100	1	
1/66	14.04	*	15.8100	10	14.14	15.8100	1	
1/67	14.04	*	15.8100	10	14.14	15.8100	1	
1/68	14.04	*	15.8100	10	14.14	15.8100	1	
1/69	14.04	*	15.8100	10	14.14	15.8100	1	
1/70	14.04	*	15.8100	10	14.14	15.8100	1	
1/71	14.04	*	15.8100	10	14.14	15.8100	1	
1/72	14.04	*	15.8100	10	14.14	15.8100	1	
1/73	14.04	*	15.8100	10	14.14	15.8100	1	
1/74	14.04	*	15.8100	10	14.14	15.8100	1	
1/75	14.04	*	15.8100	10	14.14	15.8100	1	
1/76	14.04	*	15.8100	10	14.14	15.8100	1	
1/77	14.04	*	15.8100	10	14.14	15.8100	1	
1/78	14.04	*	15.8100	10	14.14	15.8100	1	
1/79	14.04	*	15.8100	10	14.14	15.8100	1	
1/80	14.04	*	15.8100	10	14.14	15.8100	1	
1/81	14.04	*	15.8100	10	14.14	15.8100	1	
1/82	14.04	*	15.8100	10	14.14	15.8100	1	
1/83	14.04	*	15.8100	10	14.14	15.8100	1	
1/84	14.04	*	15.8100	10	14.14	15.8100	1	
1/85	14.04	*	15.8100	10	14.14	15.8100	1	
1/86	14.04	*	15.8100	10	14.14	15.8100	1	
1/87	14.04	*	15.8100	10	14.14	15.8100	1	
1/88	14.04	*	15.8100	10	14.14	15.8100	1	
1/89	14.04	*	15.8100	10	14.14	15.8100	1	
1/90	14.04	*	15.8100	10	14.14	15.8100	1	
1/91	14.04	*	15.8100	10	14.14	15.8100	1	
1/92	14.04	*	15.8100	10	14.14	15.8100	1	
1/93	14.04	*	15.8100	10	14.14	15.8100	1	
1/94	14.04	*	15.8100	10	14.14	15.8100	1	
1/95	14.04	*	15.8100	10	14.14	15.8100	1	
1/96	14.04	*	15.8100	10	14.14	15.8100	1	
1/97	14.04	*	15.8100	10	14.14	15.8100	1	
1/98	14.04	*	15.8100	10	14.14	15.8100	1	
1/99	14.04	*	15.8100	10	14.14	15.8100	1	
1/100	14.04	*	15.8100	10	14.14	15.8100	1	

System Model (Concepts) *Design*

FlightPlanTable

TrafficControl

A Mini Tutorial on Use Case and Class Diagrams

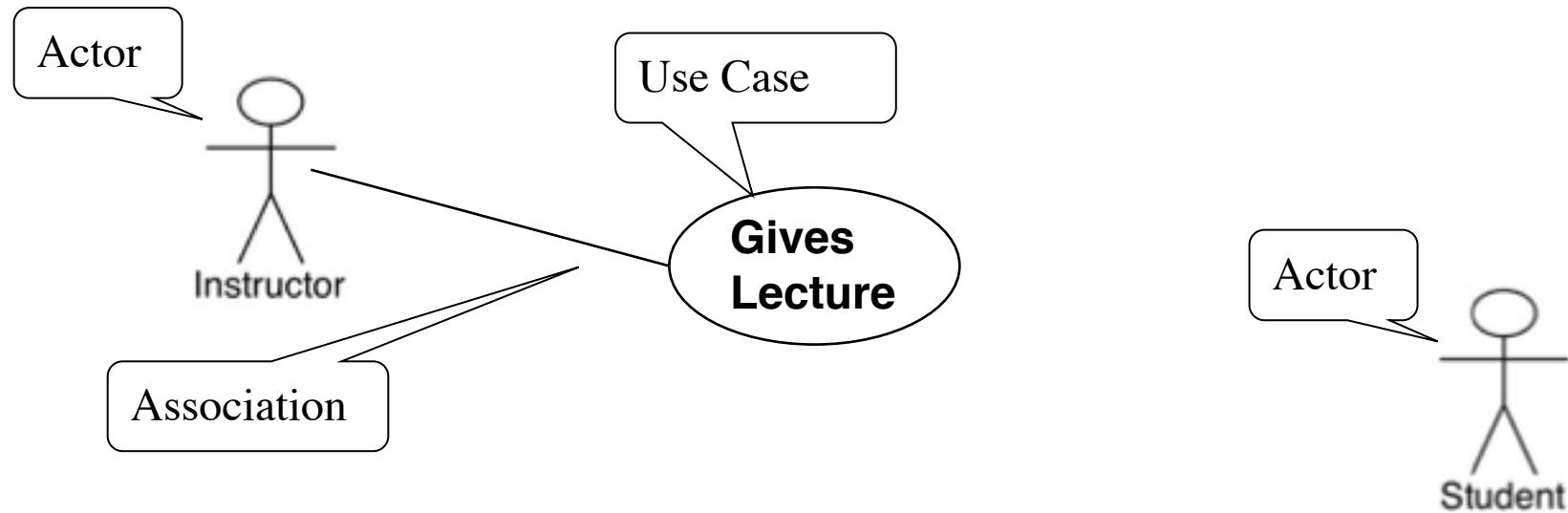
→ Use case diagrams

- Describe the functional behavior of the system as seen by the user

• Class diagrams

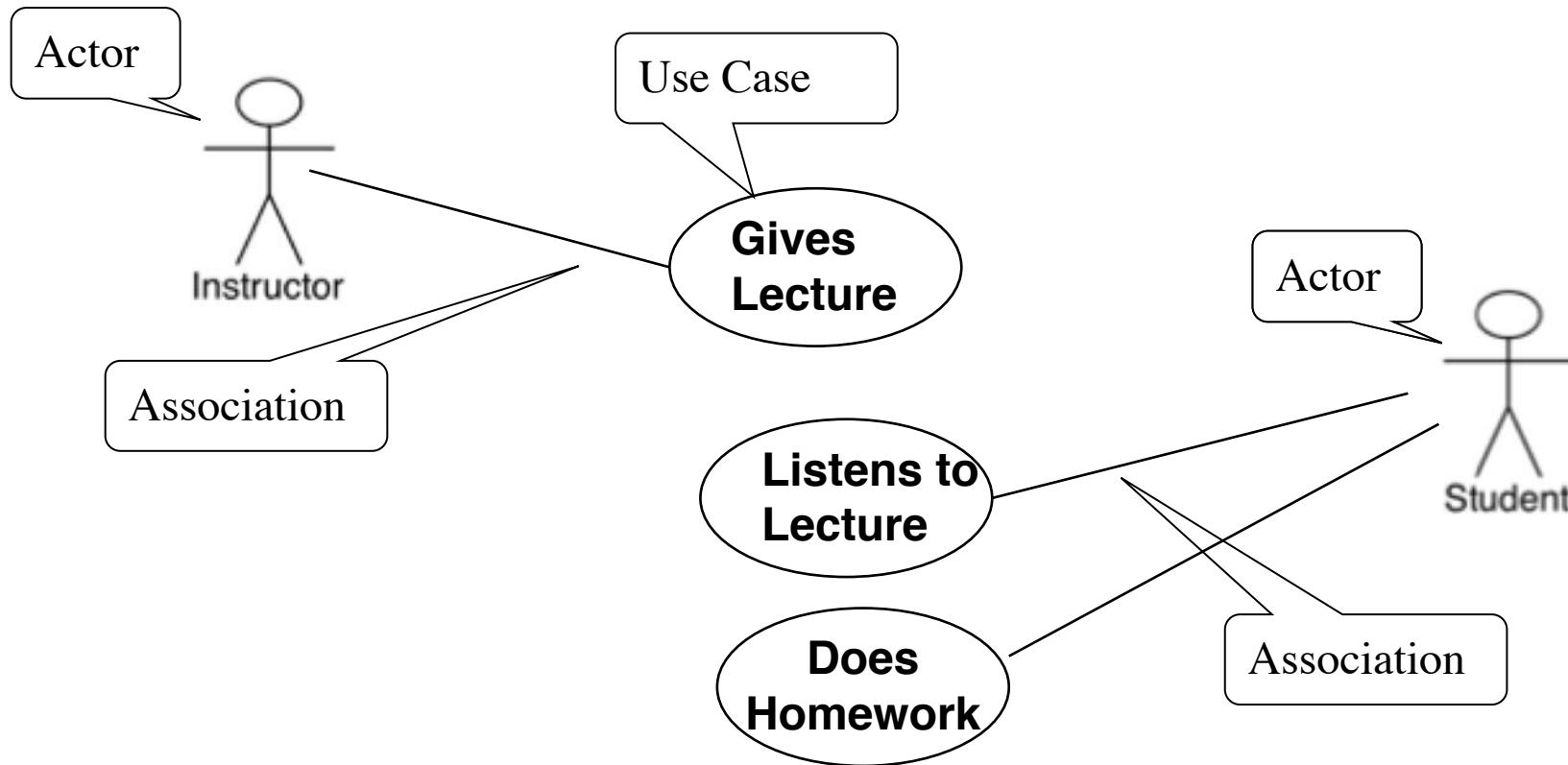
- Describe the static structure of the system: Objects, attributes, associations

Example of a Use Case Diagram



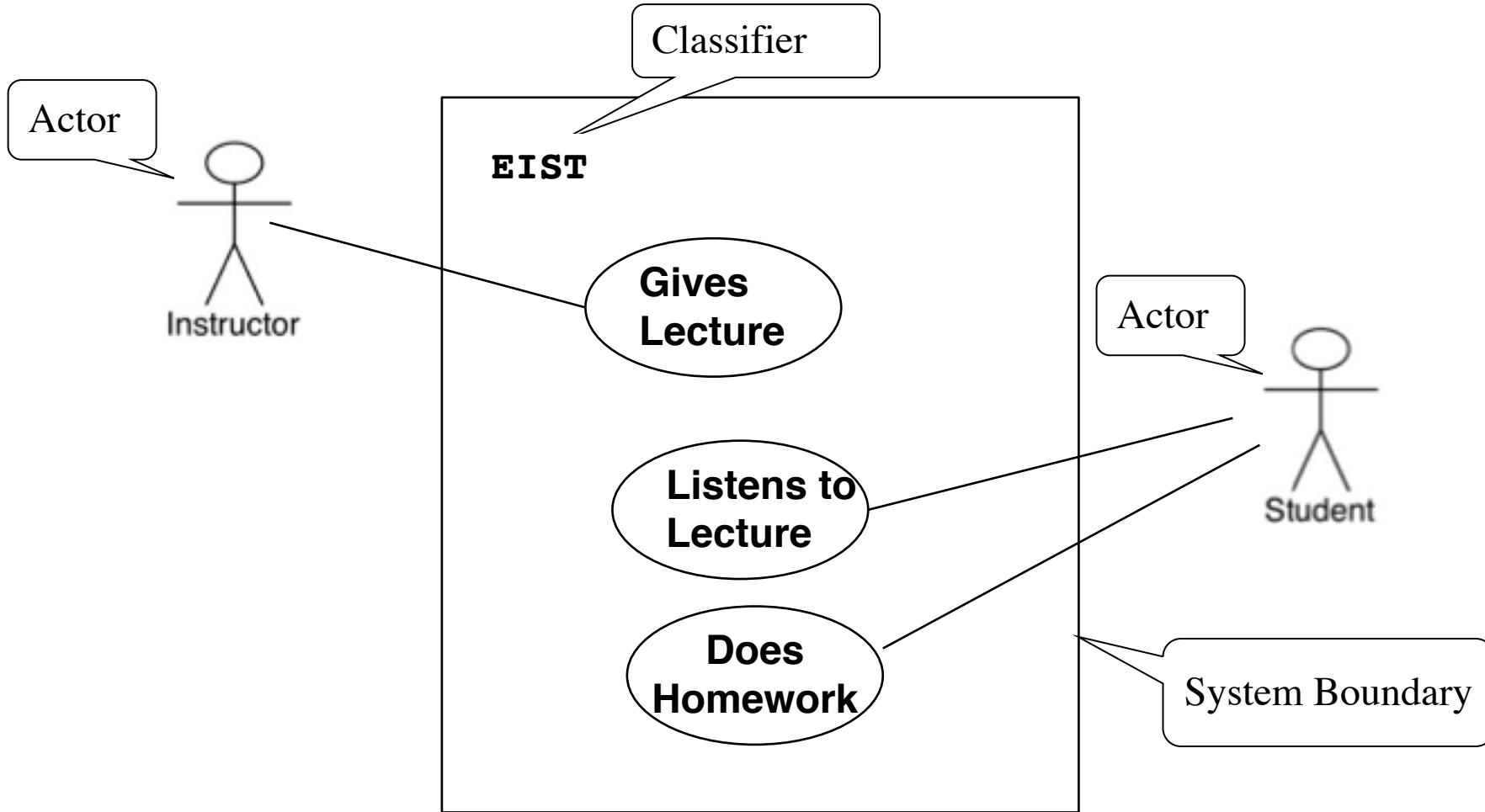
...Represents the **functionality** of a system from **user's** point of view

Example of a Use Case Diagram (2)



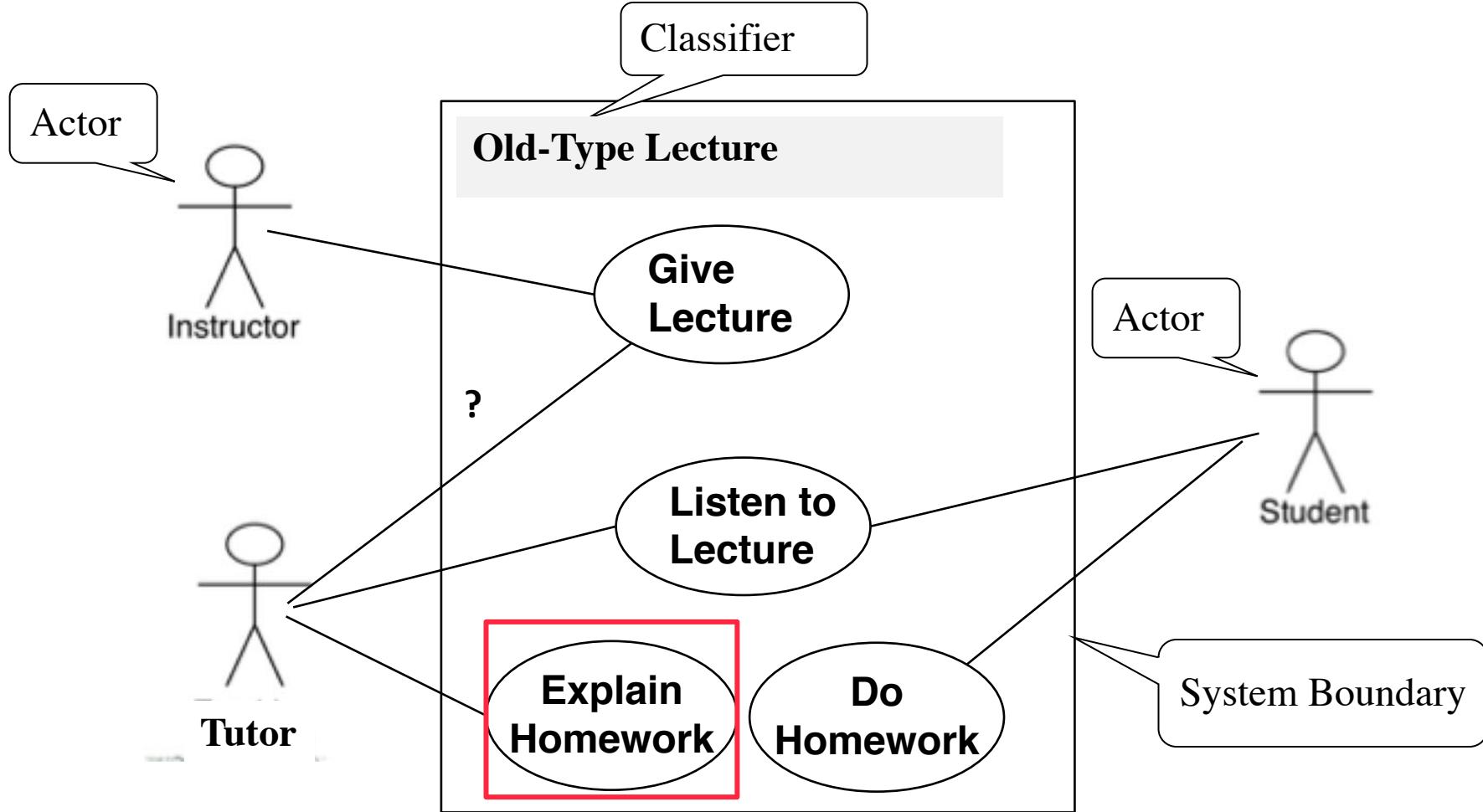
...Represents the **functionality** of a system from **user's** point of view

Example of a Use Case Diagram (3)

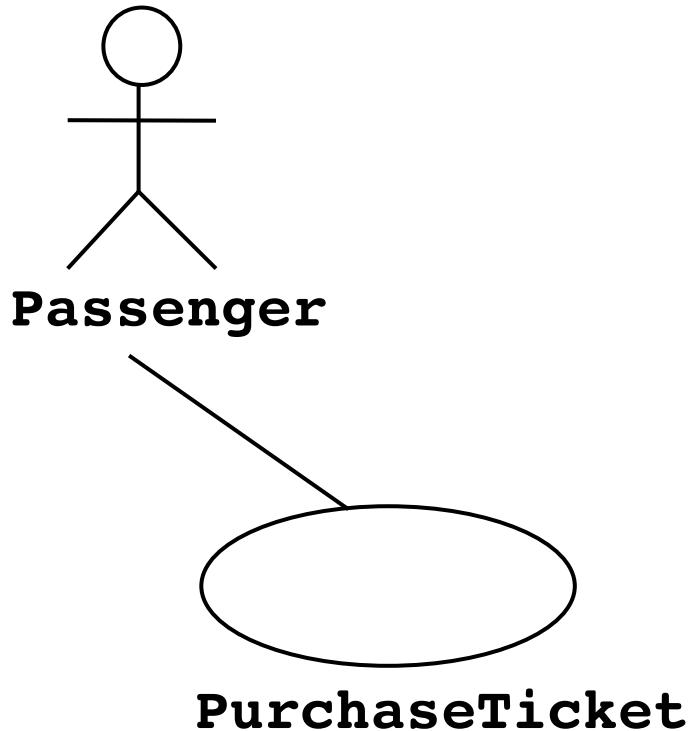


...Represents the **functionality** of a system from **user's** point of view

Modeling Means Iteration



Definition: Use Case Diagram



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

An **actor** represents a specific type of user (also called a **role**) of the system

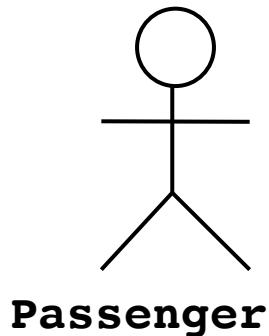
A **use case** represents a functionality provided by the system

Use cases are associated with actors

Use case model:

A graph of use cases and actors that describes the functionality of the system

Actors



- An actor is a model for an external entity which interacts or 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 a navigation system with GPS information.

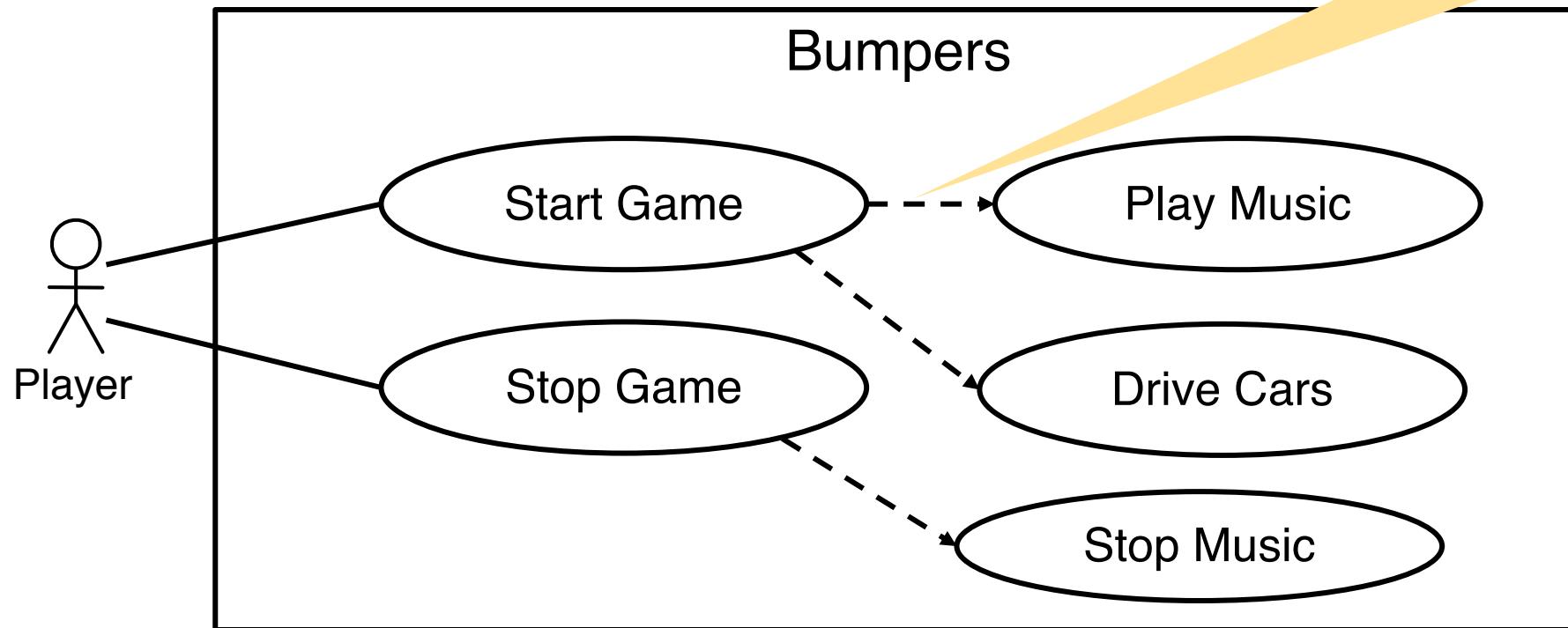
Optional
Description

Name

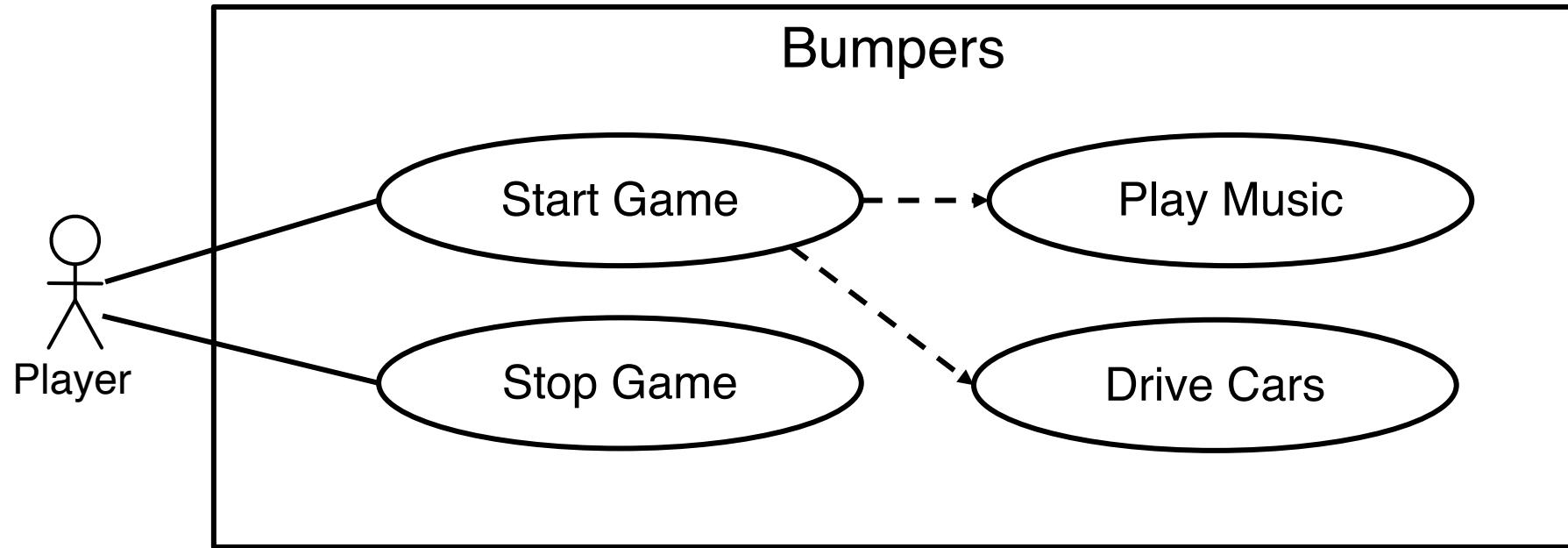
Sprint Backlog 1 - Use Cases

2. Cars drive on the game board
5. The player starts and stops the game
6. Music plays when the game begins and stops to play when the game ends

More details in Lecture 3
on Requirements Analysis



Use Case Model for Sprint 1



A Mini Tutorial on Use Case and Class Diagrams

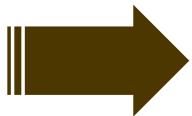
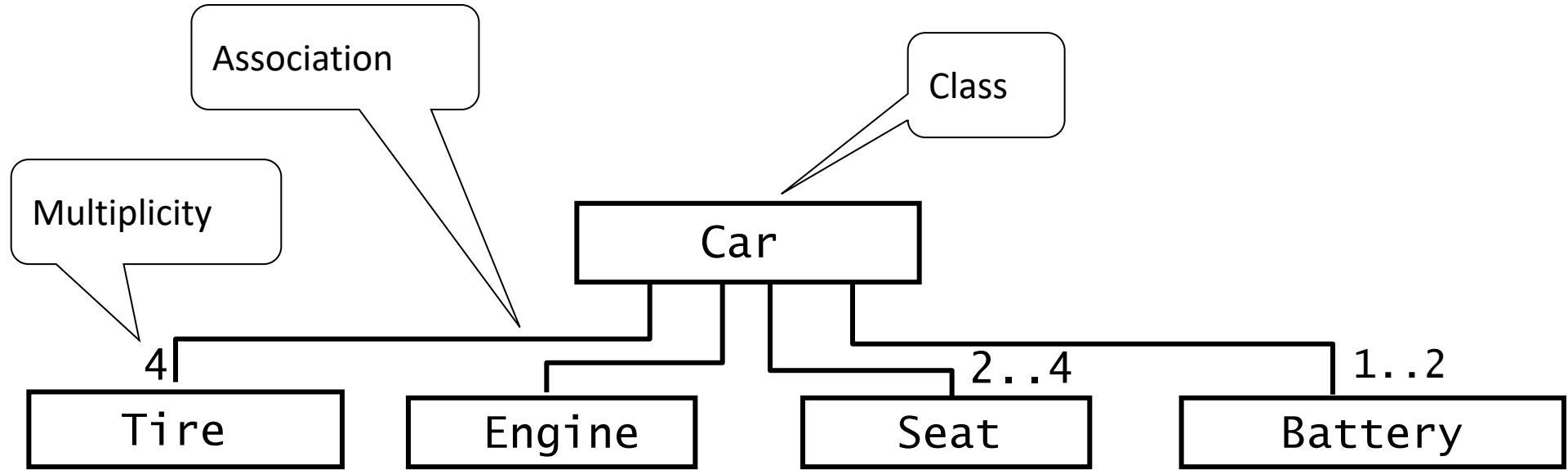
✓ Use case diagrams

- ✓ Describe the functional behavior of the system as seen by the user

→ Class diagrams

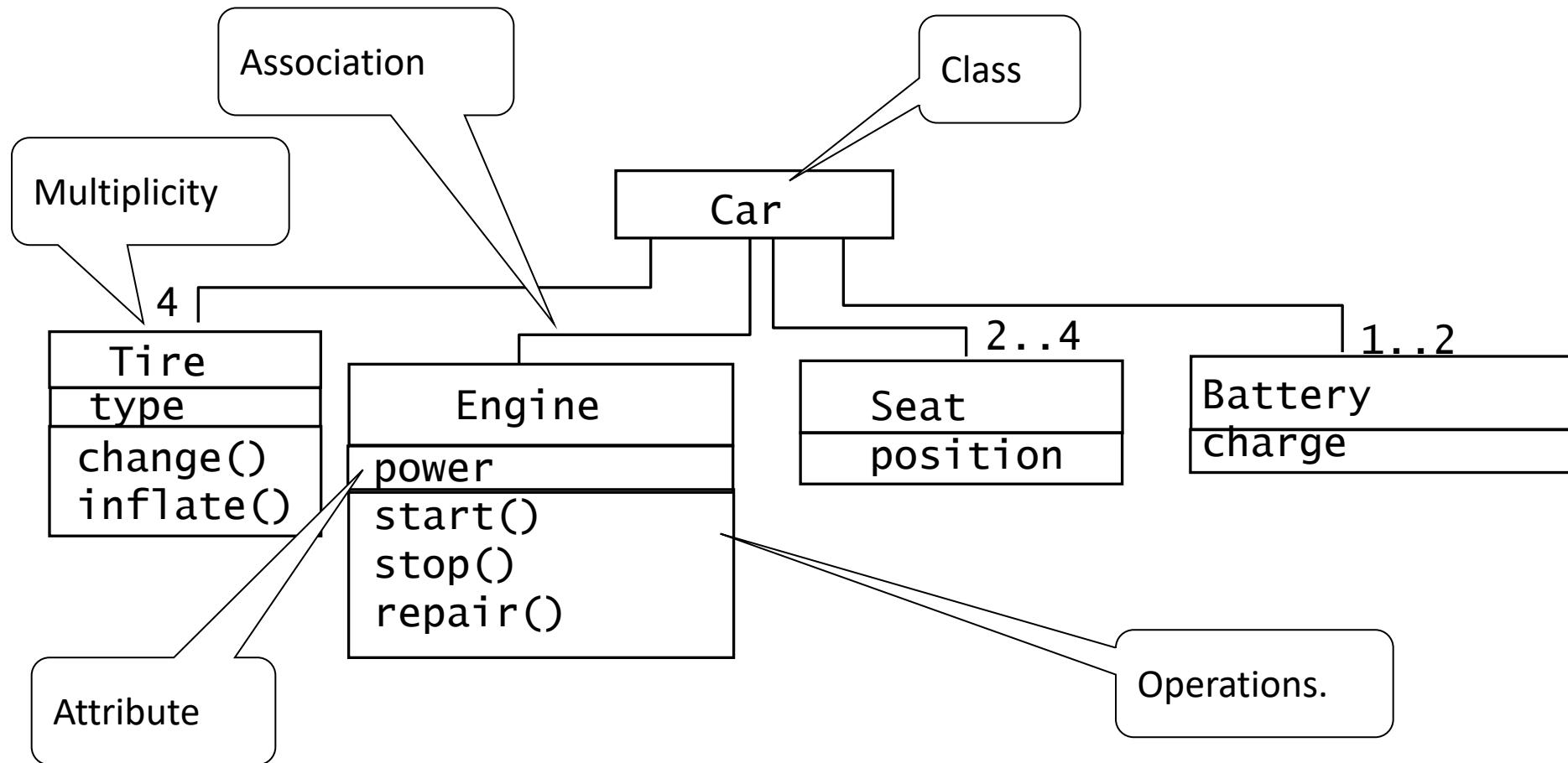
- Describe the static structure of the system: Objects, attributes, methods associations

Class Diagrams...



... represent the **structure** of the system and **relationships** of its classes

Class Diagrams



Class Diagrams

More details in Lecture 3:
Requirements Elicitation and Analysis

- Class diagrams represent the structure of the system
- We use them in many modeling situations:
 - During **requirements analysis** to model application domain objects
 - During **system design** to model solution domain objects
 - During **object design** to specify the detailed behavior and the types of attributes of classes (also called signatures)

Outline of this lecture

✓ Miscellaneous

- ✓ Morning Quiz
- ✓ Tutor Group Selection
- ✓ Exam Dates

✓ Software Lifecycle

✓ Problem Statement

✓ UML

 Analysis, Design, Implementation and Delivery of a Game

Learning Goals:

- You start with a problem statement from the customer
- You use a game to walk through the activities of a tailored software lifecycle
- You learn a notation that can be used uniformly throughout the activities: UML.

30 Minute Break



Analysis: Finding Application Domain Objects

→ Syntactical investigation with Abbot's technique:

- Flow of events in use cases
- Problem statement from the customer
- Use other knowledge sources:
 - **Application knowledge:** End users and application domain experts know the abstractions of the application domain
 - **General world knowledge:** You can also use your generic knowledge and intuition
- During system design and object design we use another knowledge source:
 - **Solution knowledge:** Solution domain experts know abstractions in the solution domain

Creating Class diagrams from Problem Statement Written in Natural Language (**Abbot's Technique**)

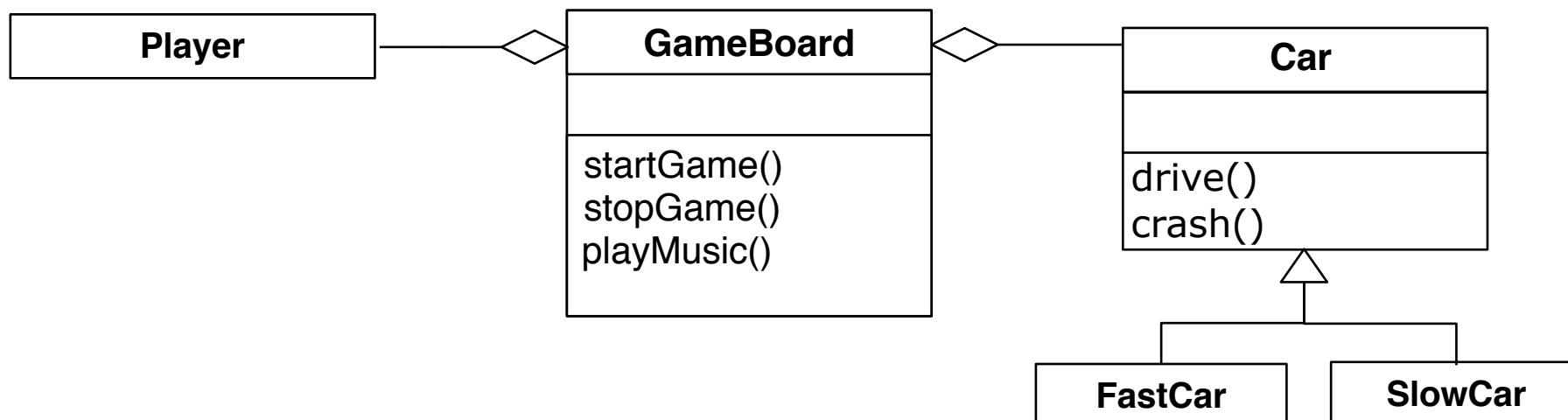
<i>Example</i>	<i>Grammatical construct</i>	<i>UML model component</i>
“Monopoly”	Proper noun	object
Toy	Improper noun	class
Buy, recommend	Doing verb	operation
Is a	being verb	inheritance
has an	having verb	aggregation
must be	modal verb	constraint
dangerous	adjective	attribute
enter	transitive verb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

Bumpers – From Problem Statement to Object Model

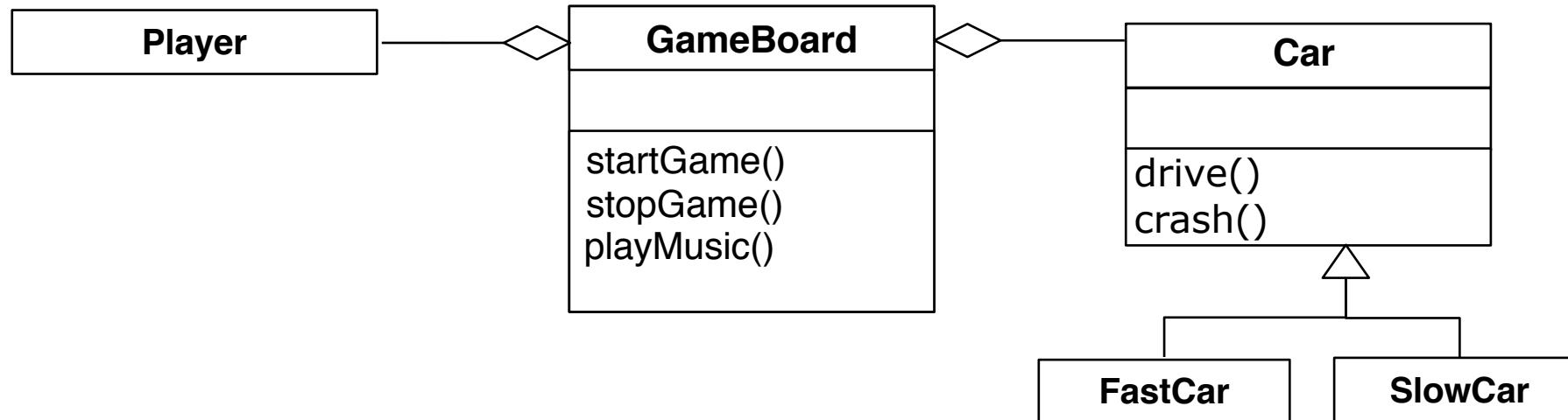
Bumpers is a game where **cars drive** on a **game board** and can **crash** each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The **player** can **start** and **stop** the **game**. When the game is started, **music is played**.

A **car** can be **either fast or slow**. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change its speed.

The game should be platform independent, and visualizes different parameters of the car, e.g. the speed, consumption and location of the car. When cars crash, there has to be a sound effect. The game should support different collisions and the determination of the collision winner should be changeable during gameplay.

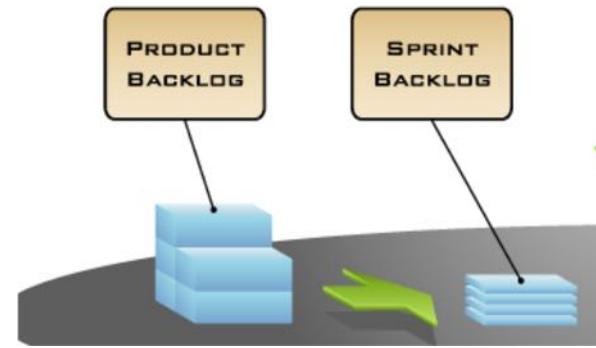


Analysis Object Model for Sprint 1



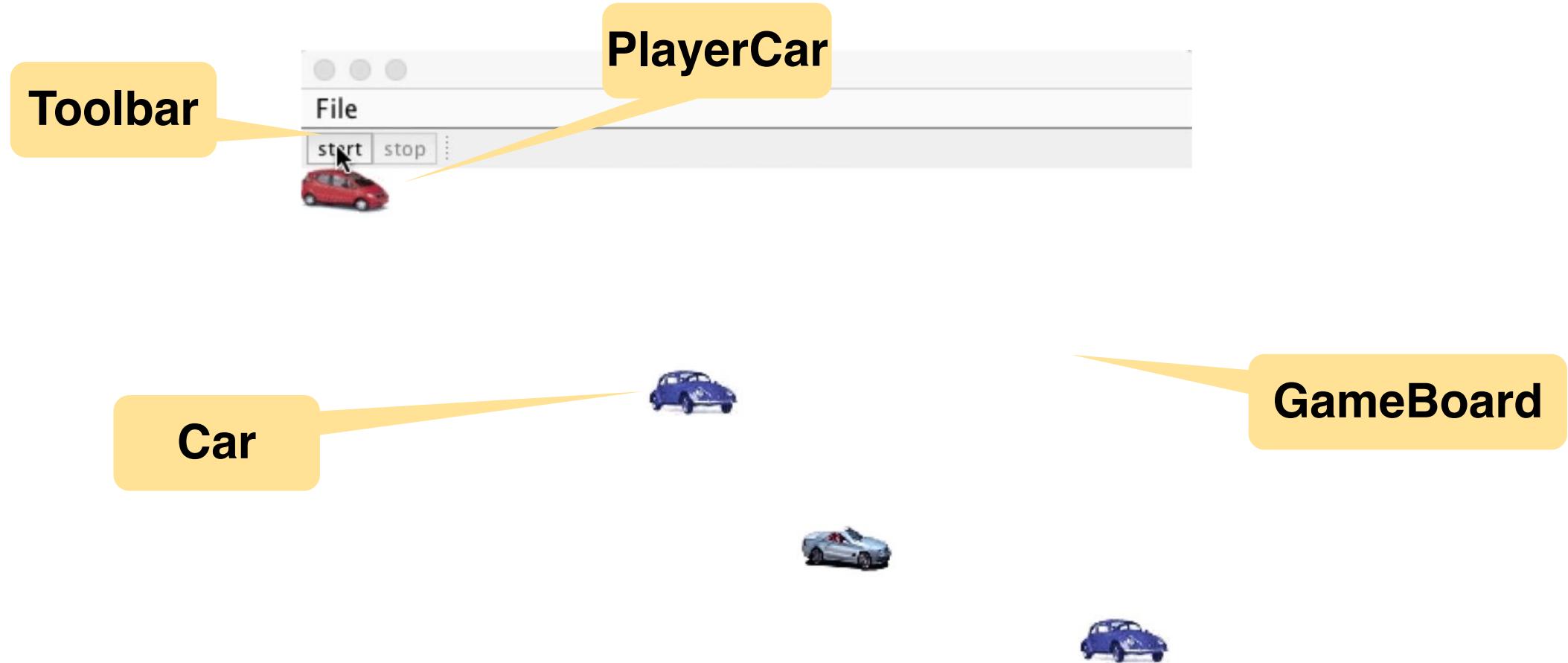
Sprint Backlog for Sprint 1

1. User Interface design of the game board
2. **Cars drive on the game board**
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. **The player starts and stops the game**
6. **Music plays when the game begins and stops to play when the game ends**
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed



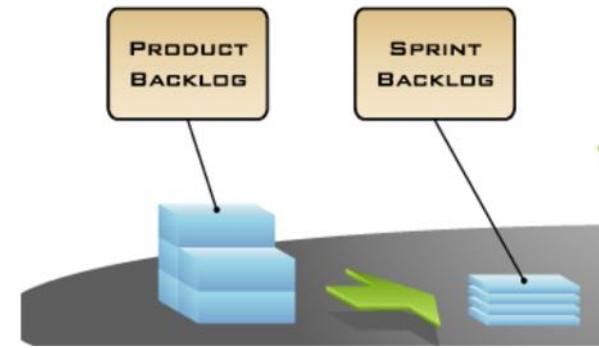
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play

User Interface Model (dynamic model)



Sprint Backlog for Sprint 1

1. User Interface design of the game board
2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. → The player starts and stops the game
6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed



10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play

2 Additional Change Requests:

15. Choose a different car image
16. Choose a different game music

In-Class Exercise: Implementation of Sprint 1



Implement and delivery the following 4 Sprint backlog items:

- Start game
- Stop game
- Play music
- Stop music

Also solve the 2 change requests:

- Choose a different car image
- Choose a different game music

Prerequisites for the exercises

- You have a TUM Online Account
- You are registered for the EIST 2018 course
- You have a stable internet connection
- You have installed the following tools on your development computer
 - **Java JDK** (1.8.162 or later)
<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
 - **Eclipse IDE** for Java Developers (4.7. or later):
<https://www.eclipse.org/downloads/eclipse-packages>
 - **SourceTree with git** (2.7.1 or later for macOS, 2.4.8 or later for Windows)
<https://www.sourcetreeapp.com>
 - Alternative 1: **GitKraken with git** (3.5.1 or later for all platforms)
<https://www.gitkraken.com/download>
 - Alternative 2 for experienced students: **git command line tools**:
<https://git-scm.com>

Task 1: Start exercise on ArTEMiS

- Open ArTEMiS on <https://artemis.ase.in.tum.de>
- Sign in with your TUM Online Account (e.g. "ne23kow")
- Open Courses
- Click on **Start Exercise** on **EIST 2018 Lecture 02 Bumpers Sprint 01**

Introduction to Software Engineering (Summer 2018)

Exercise	Due date	Results	Actions
Quiz 01	6 days ago	You have not participated in this quiz.	C Practice Statistic
EIST 2018 Lecture 02 Bumpers Sprint 01	in 3 days	You have not started this exercise yet.	Start exercise

Click



Task 2: Clone the repository

More details in Lecture 10:
Configuration Management

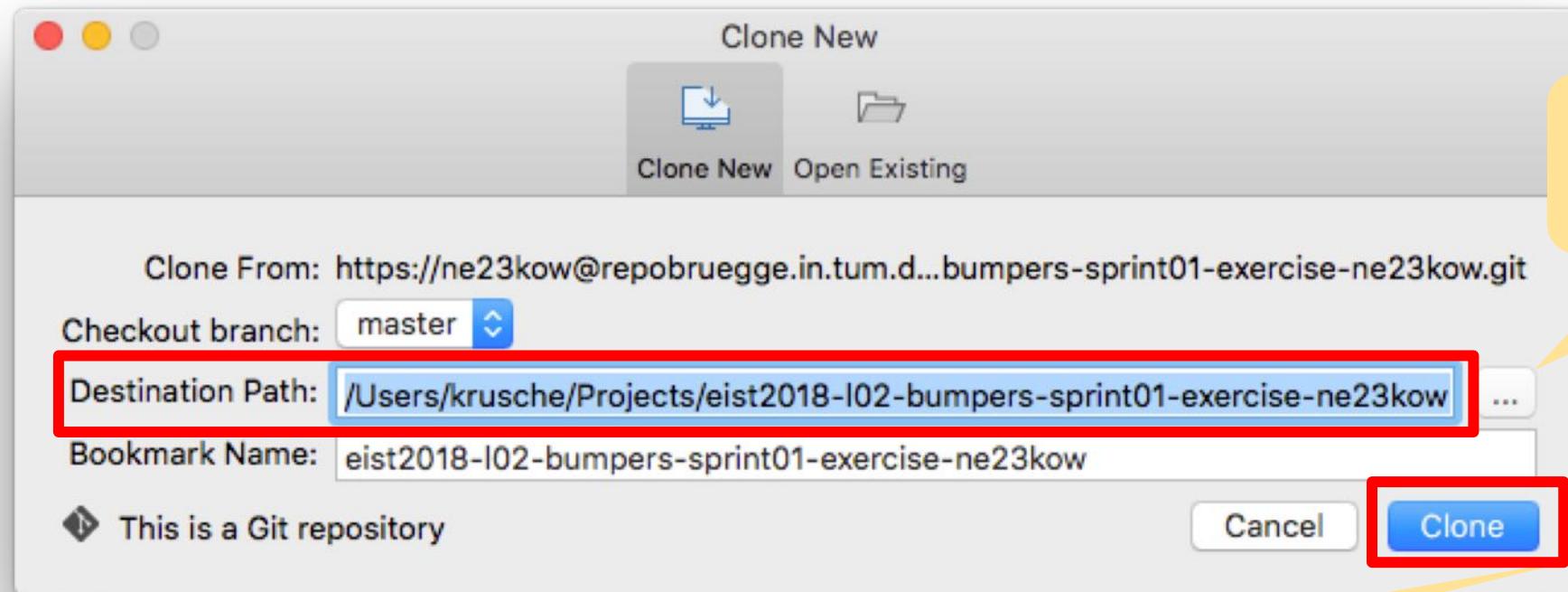
Introduction to Software Engineering (Summer 2018) ⓘ

Exercise	Due date	Results	Actions
Quiz 01 ⓘ		Clone your personal repository for this exercise: https://ne23kow@repobruegge.in.tum.de/scm/eist2018l02bumperss01/eist2018-l02-bumpers-sprint01-exercise-ne23kow.git	C Practice ...l Statistic Clone repository
EIST 2018 Lecture 02 Bu			Clone repository
Programming Exercise Tu		Clone in SourceTree Atlassian SourceTree is the free Git client for Windows or Mac.	Clone repository

Click (highlighted button)

Click (highlighted button)

Task 2: Clone the repository



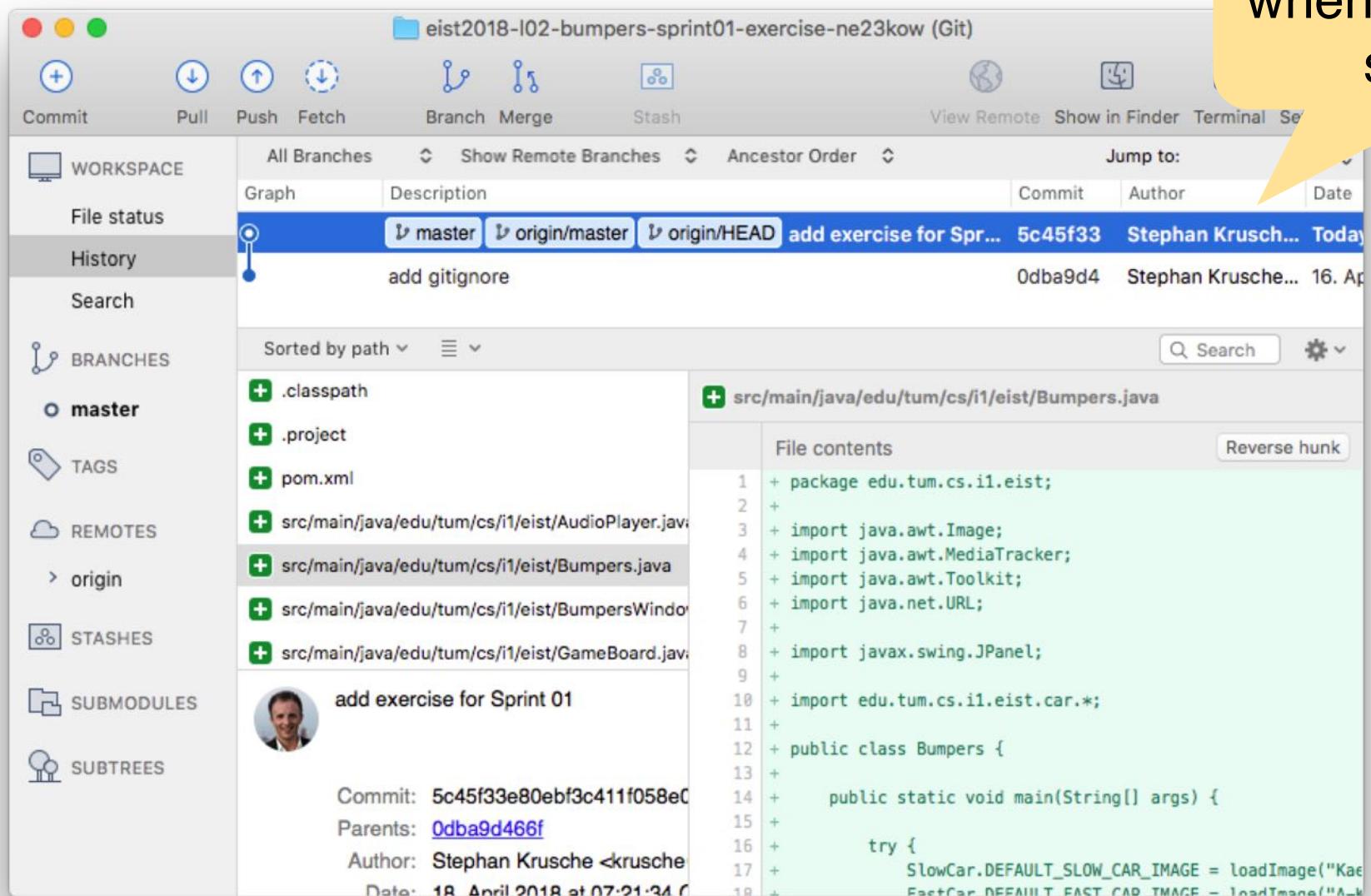
Choose a folder
in your file system

Click Clone

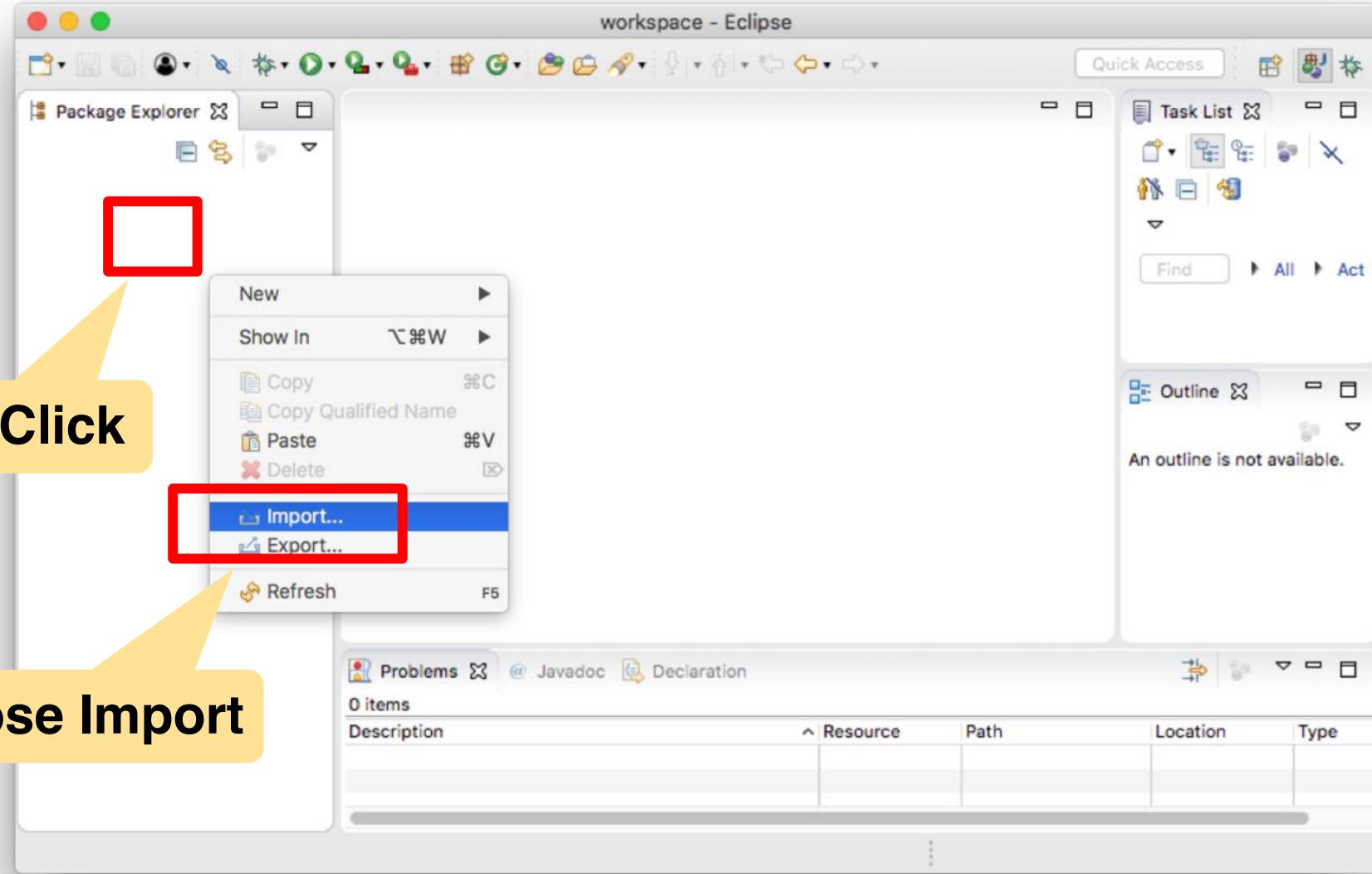
- In case you have problems:
 - Have another look into the [ArTEMiS Tutorial](#) on Moodle
 - Ask one of the tutors.

Task 2: Clone the repository

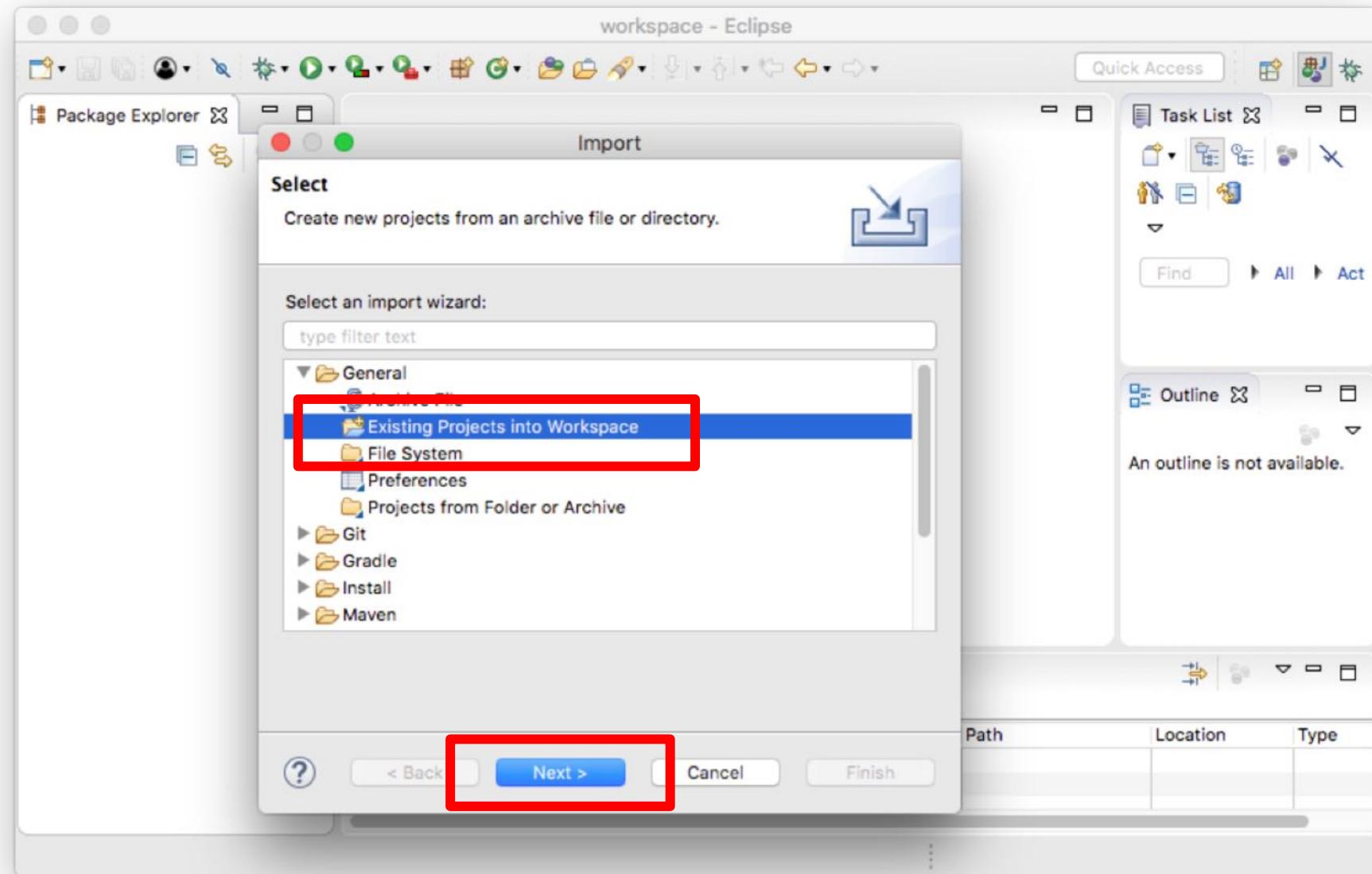
SourceTree opens
when the clone was
successful



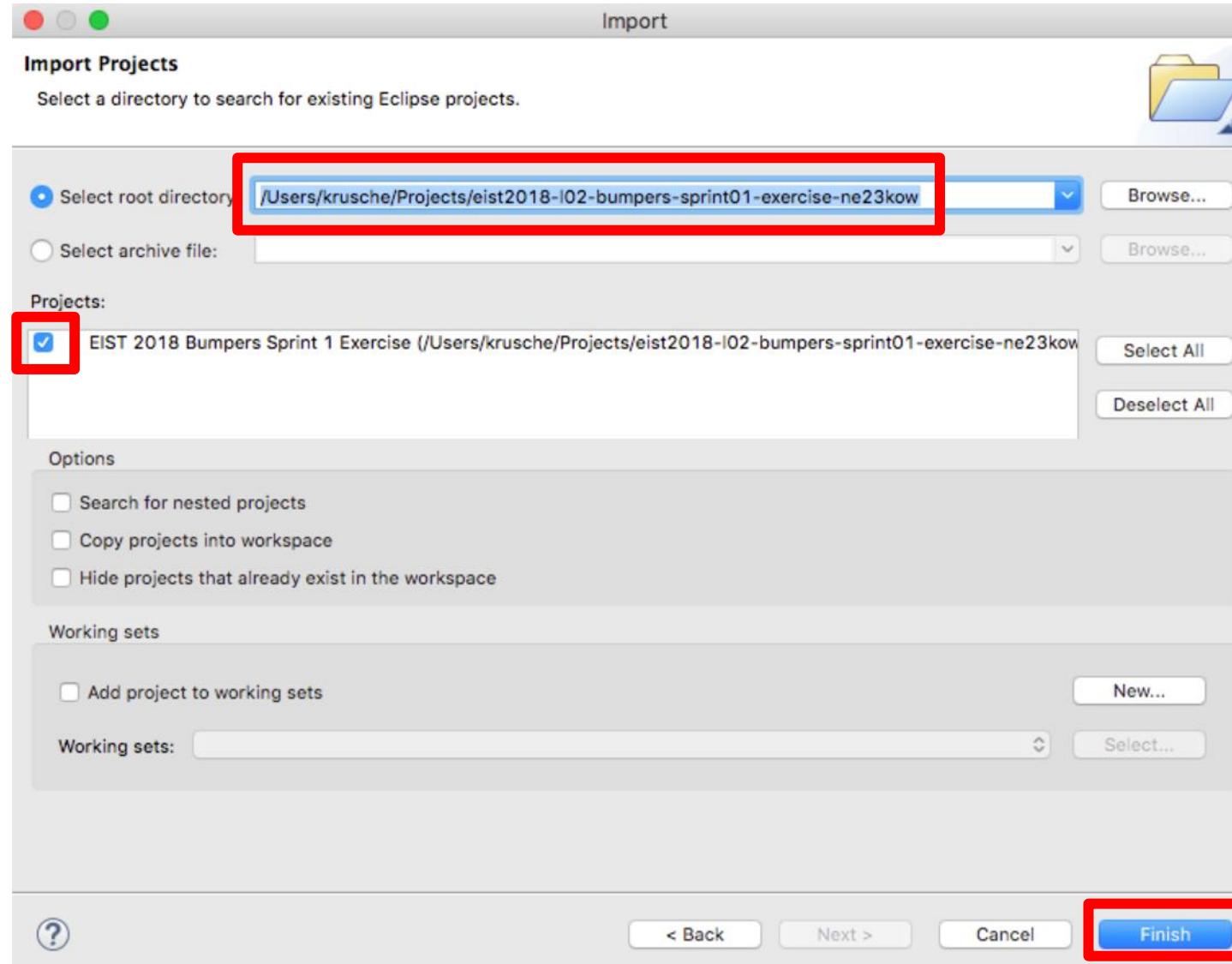
Task 3: Import into Eclipse



Task 3: Import into Eclipse



Task 3: Import into Eclipse



Task 4: Inspect the Eclipse Project

The screenshot shows the Eclipse IDE interface with the following components:

- Package Expl.**: Shows the project structure under "EIST 2018 Bumpers Sprint 1 Exercise". The "GameBoard.java" file is selected.
- Bumpers.java** and **GameBoard.java**: The Java files are open in the editor. The code for **GameBoard.java** is displayed, containing methods like startGame(), stopGame(), playMusic(), stopMusic(), and moveCars().

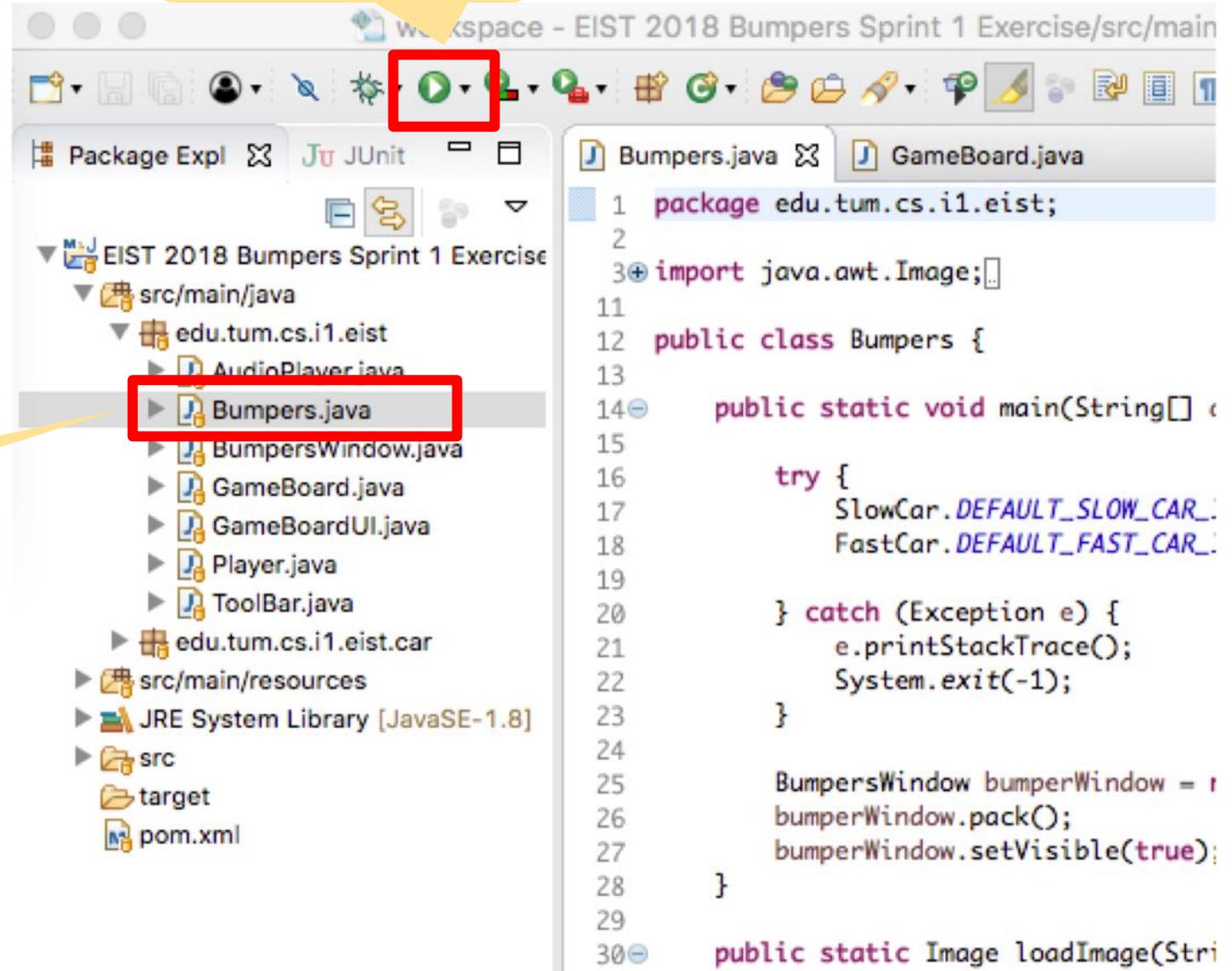
```
62 }
63
64 public void startGame() {
65     //TODO Call the method playMusic() and start the game by setting isRunn
66 }
67
68 public void stopGame() {
69     //TODO Call the method stopMusic() and stop the game by setting isRu
70 }
71
72 public void playMusic(){
73     //TODO Call the method playBackgroundMusic on audioPlayer
74 }
75
76 public void stopMusic(){
77     //TODO Call the method stopBackgroundMusic on audioPlayer
78 }
79
80 public void moveCars() {
81
82     int maxX = size.width;
83     int maxY = size.height;
84
85     for (Car car : cars) {
86         car.updatePosition(maxX, maxY);
87     }
88 }
```
- Outline**: Shows the class hierarchy and member variables and methods for **GameBoard**. It includes fields like **cars**, **player**, **audioPlayer**, **size**, and **isRunning**, along with methods like **NUMBER_OF_S**, **GameBoard(Di**, **addCars()**, **resetCars()**, **isRunning()**, **getCars()**, **getPlayerCar()**, **update()**, **getAudioPlaye**, **startGame()**, and **stopGame()**.
- Problems**, **Javadoc**, **Declaration**, **Console**, **Tasks**, and **Call Hierarchy**: These views are visible at the bottom of the interface.

Task 5: Run the project

Click on Run

- Select Bumpers.java
- Click the Run icon

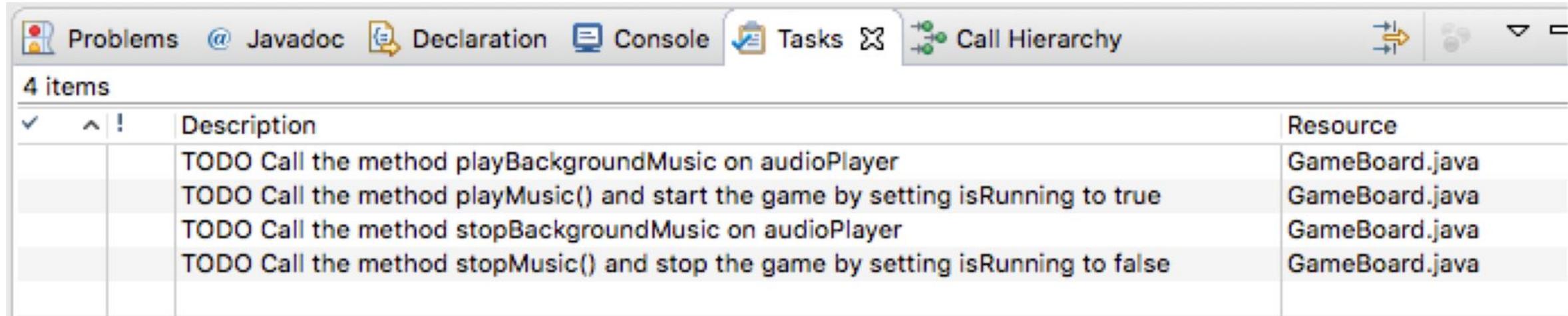
Select Bumpers.java



The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer view displays the project structure under 'EIST 2018 Bumpers Sprint 1 Exercise'. The 'src/main/java' folder contains several Java files: Bumpers.java (highlighted with a red box), BumpersWindow.java, GameBoard.java, GameBoardUI.java, Player.java, ToolBar.java, and edu.tum.cs.i1.eist.car (which itself contains SlowCar and FastCar). Other folders include src/main/resources and JRE System Library [JavaSE-1.8]. On the right, the Editor view shows the code for Bumpers.java. At the top of the editor, there is a toolbar with various icons, including a green circle with a white play symbol which is highlighted with a red box. The code itself starts with a package declaration for 'edu.tum.cs.i1.eist' and imports 'java.awt.Image'. It defines a public class 'Bumpers' with a main method that handles exceptions and prints stack traces.

```
1 package edu.tum.cs.i1.eist;
2
3 import java.awt.Image;
4
5 public class Bumpers {
6     public static void main(String[] args) {
7         try {
8             SlowCar.DEFAULT_SLOW_CAR_SIZE = 100;
9             FastCar.DEFAULT_FAST_CAR_SIZE = 50;
10        } catch (Exception e) {
11            e.printStackTrace();
12            System.exit(-1);
13        }
14        BumpersWindow bumperWindow = new BumpersWindow();
15        bumperWindow.pack();
16        bumperWindow.setVisible(true);
17    }
18
19    public static Image loadImage(String name) {
20        return ImageIO.read(new File("src/main/resources/" + name));
21    }
22
23    public static void main(String[] args) {
24        BumpersWindow bumperWindow = new BumpersWindow();
25        bumperWindow.pack();
26        bumperWindow.setVisible(true);
27    }
28
29    public static void main(String[] args) {
30        BumpersWindow bumperWindow = new BumpersWindow();
31        bumperWindow.pack();
32        bumperWindow.setVisible(true);
33    }
34}
```

Task 6: Solve the 4 tasks in GameBoard.java



The screenshot shows the Eclipse IDE interface with the 'Tasks' view open. The 'Tasks' tab is selected in the top navigation bar. Below the bar, there are four items listed:

>Description	Resource
TODO Call the method playBackgroundMusic on audioPlayer	GameBoard.java
TODO Call the method playMusic() and start the game by setting isRunning to true	GameBoard.java
TODO Call the method stopBackgroundMusic on audioPlayer	GameBoard.java
TODO Call the method stopMusic() and stop the game by setting isRunning to false	GameBoard.java

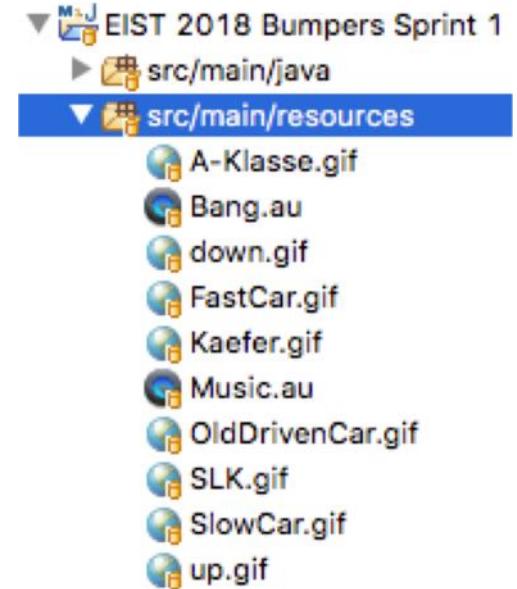
Hints for the 2 Additional Change Requests

15. Choose a different car image

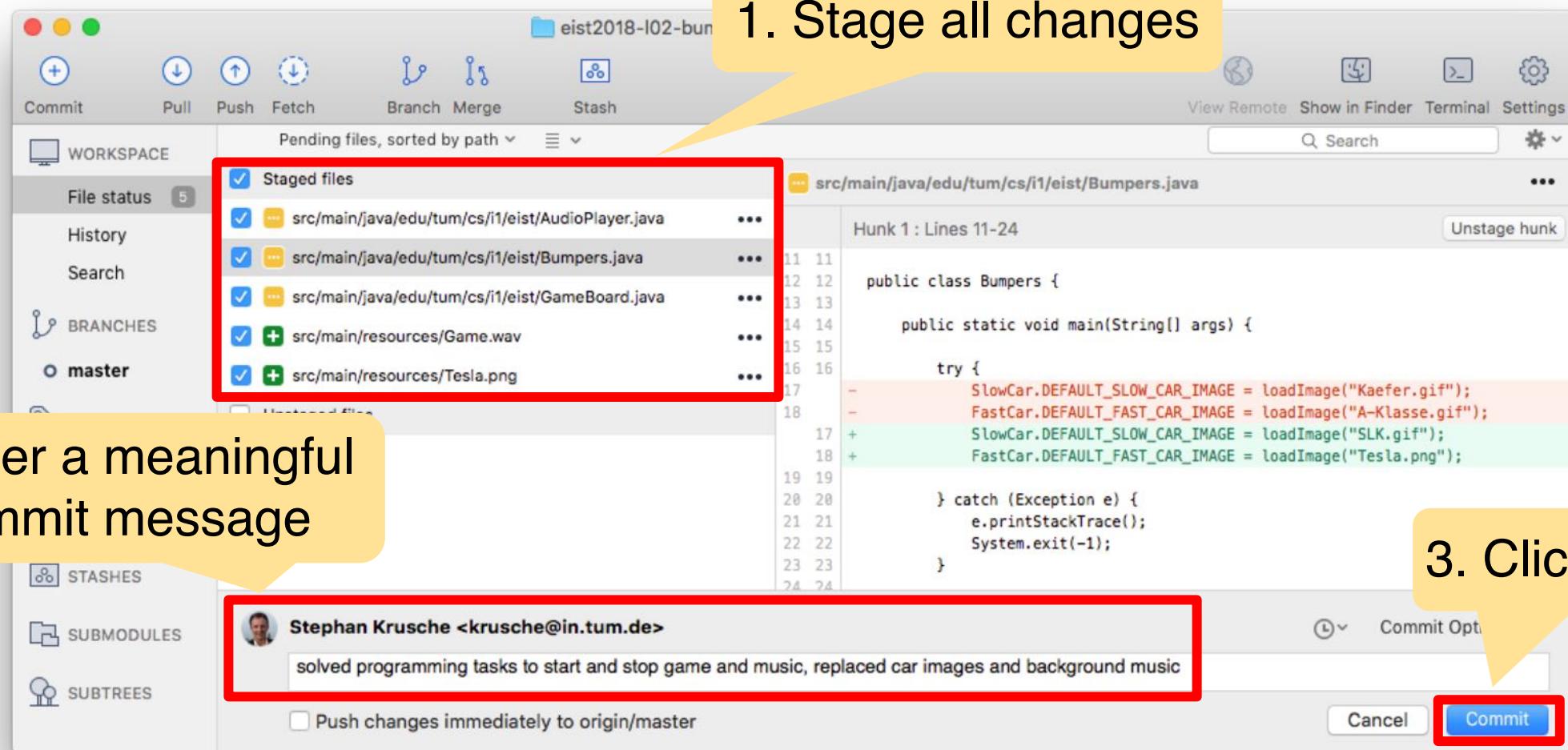
- Find a small image (gif) of a nice electrical and/or autonomous driving car and replace the player car with it
- Add the image file to the resources folder **src/main/resources** and replace the corresponding entry in **Bumpers.java**

16. Choose a different game music

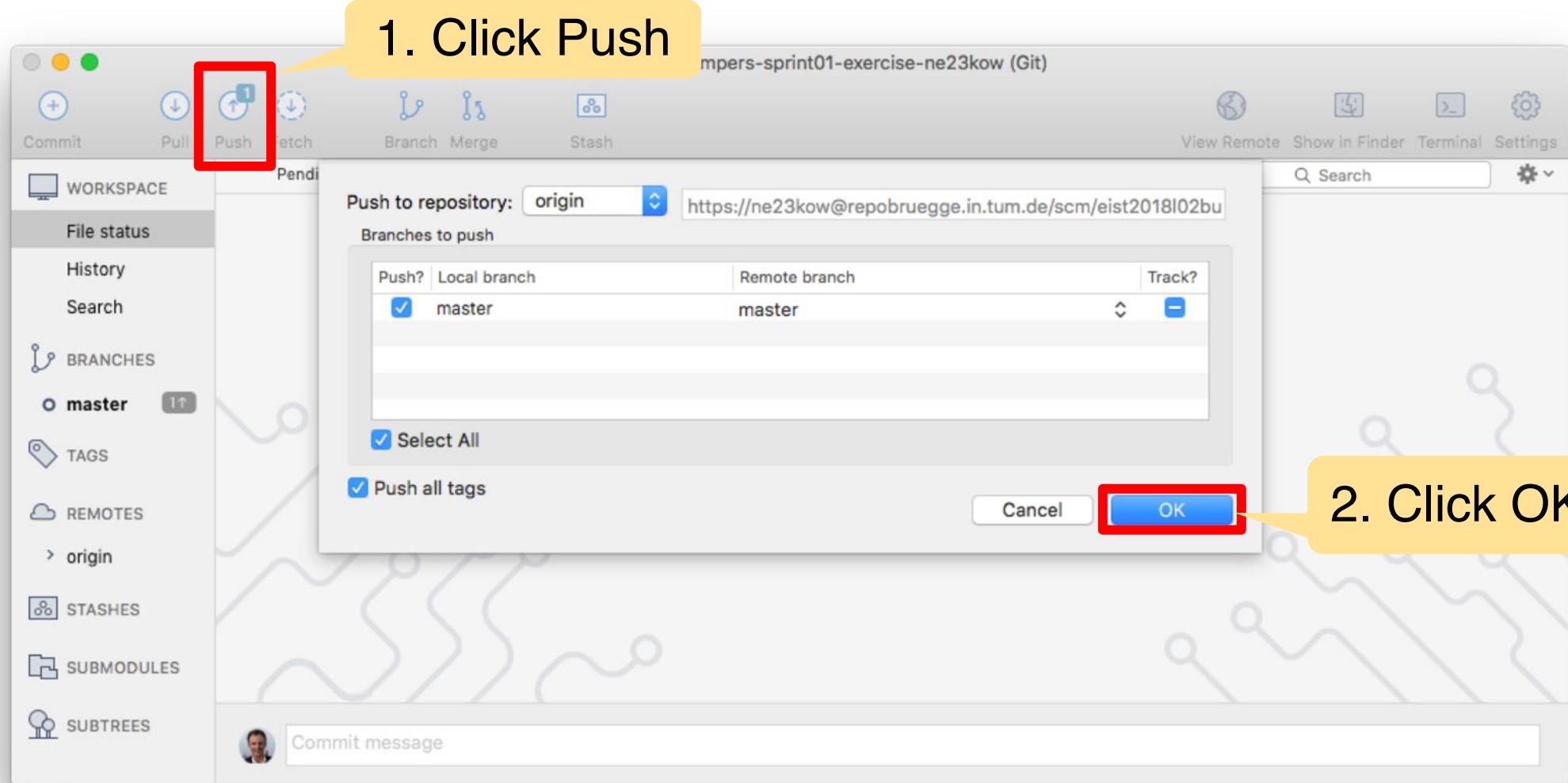
- Find better game music (*.au or *.wav) and replace it
- Add the music file to the resources folder **src/main/resources** and replace the corresponding entry in **AudioPlayer.java**



Task 7: Commit and push your solution



Task 7: Commit and push your solution



Task 8: Inspect the results on ArTEMiS

Introduction to Software Engineering (Summer 2018)			
Exercise	Due date	Results	Actions
Quiz 01	6 days ago	You have not participated in this quiz.	C Practice Statistic
EIST 2018 Lecture 02 Bumpers Sprint 01	in 2 days	 4 passed, Score: 100%, Submission: a minute ago	Clone repository

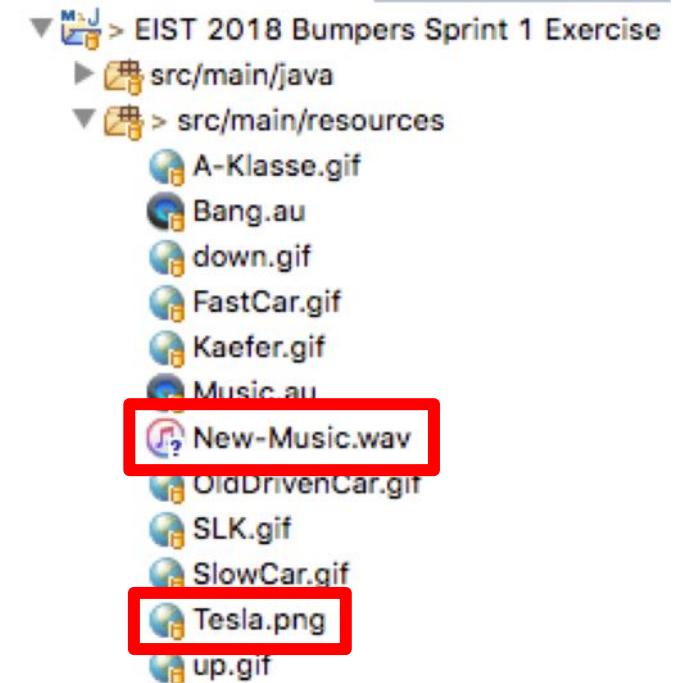
- It might take some time until your result is shown
- If test cases fail, have a look at the feedback (click e.g. on 1 of 4 failed)
- Try to understand the problem, fix it in the source code in Eclipse, then commit and push your code again

Sprint Review: Solution for Sprint 1

```
public void startGame() {  
    playMusic();  
    this.isPlaying = true;  
}  
  
public void stopGame() {  
    stopMusic();  
    this.isPlaying = false;  
}  
  
public void playMusic(){  
    this.audioPlayer.playBackgroundMusic();  
}  
  
public void stopMusic(){  
    this.audioPlayer.stopBackgroundMusic();  
}
```

Sprint Review: Solution for the Change Requests

```
public class AudioPlayer {  
    //...  
    public AudioPlayer() {  
        AudioPlayer.MUSIC = loadAudioClip("New-Music.wav");  
        AudioPlayer.BANG = loadAudioClip("Bang.au"),  
        AudioPlayer.playingBackgroundMusic = false;  
    }  
    //...  
}  
  
public class Bumpers {  
    public static void main(String[] args) {  
        try {  
            SlowCar.DEFAULT_SLOW_CAR_IMAGE = loadImage("SLK.aif");  
            FastCar.DEFAULT_FAST_CAR_IMAGE = loadImage("Tesla.png");  
        } //...  
    }  
    //...  
}
```



Status of Product Backlog after Sprint 1

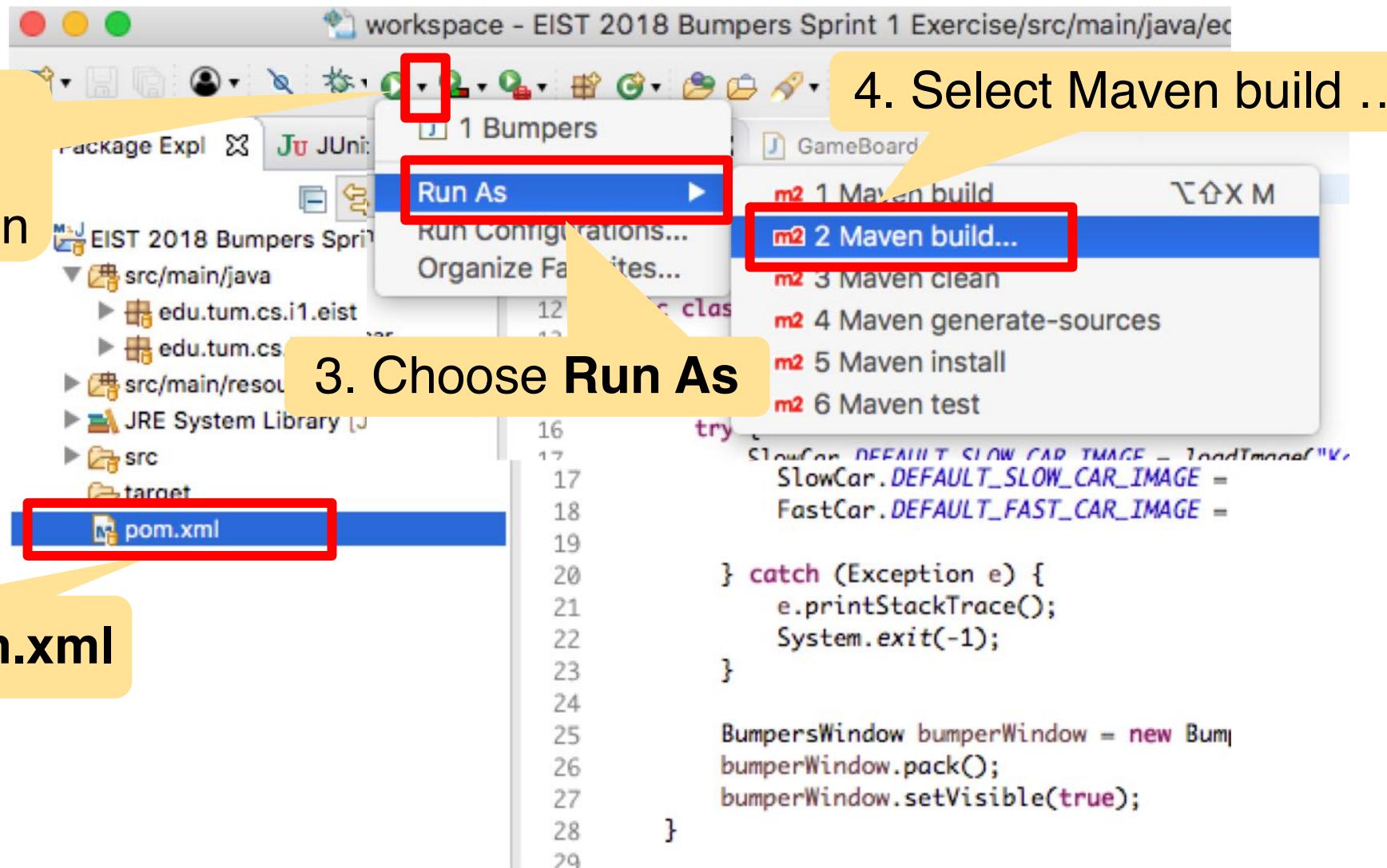
1. User Interface design of the game board
2. **Cars drive on the game board**
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. **The player starts and stops the game**
6. **Music plays when the game begins and stops to play when the game ends**
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play

Task 9: Deliver the Application

2. Click on the triangle next to the run button

3. Choose Run As

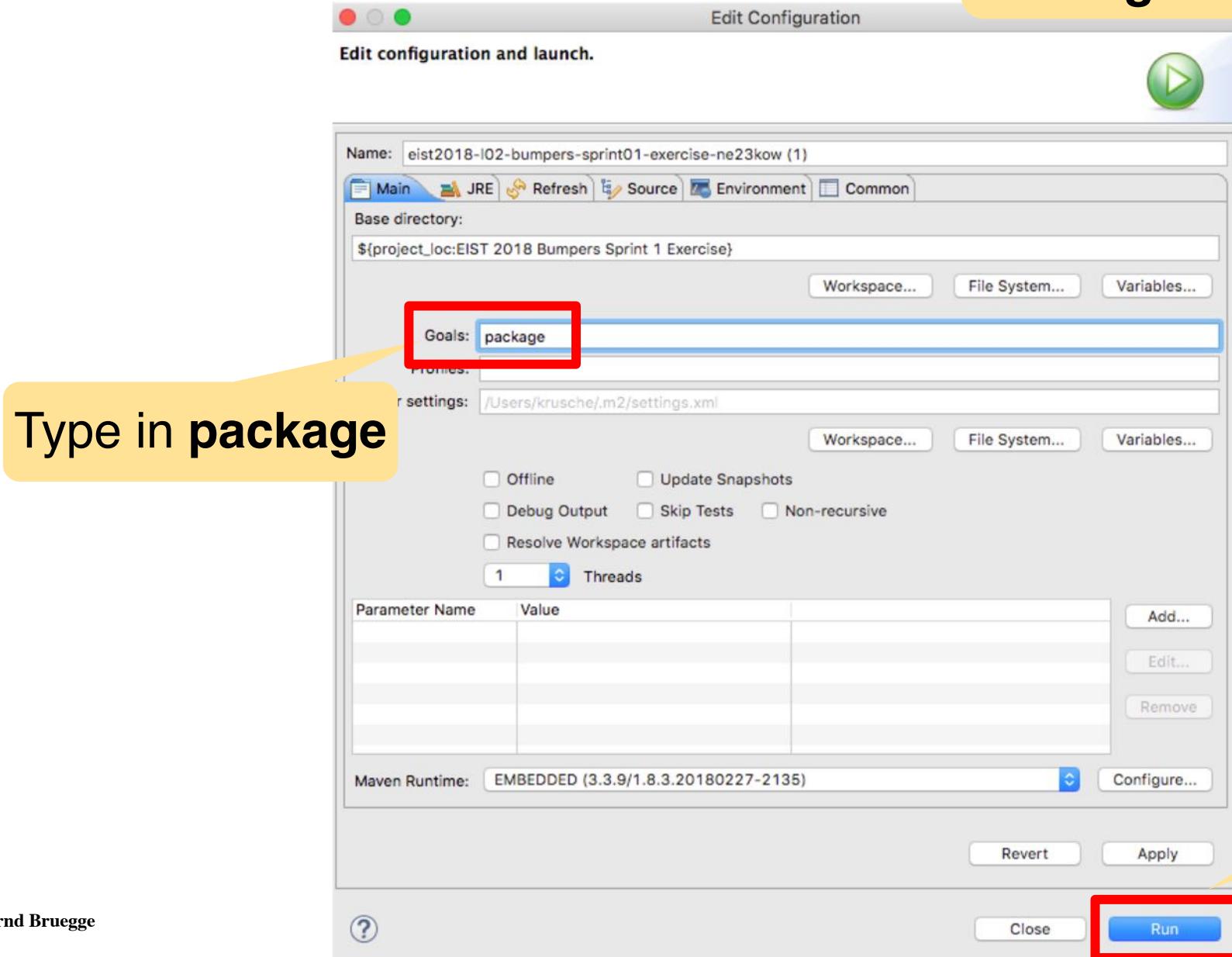
4. Select Maven build ...



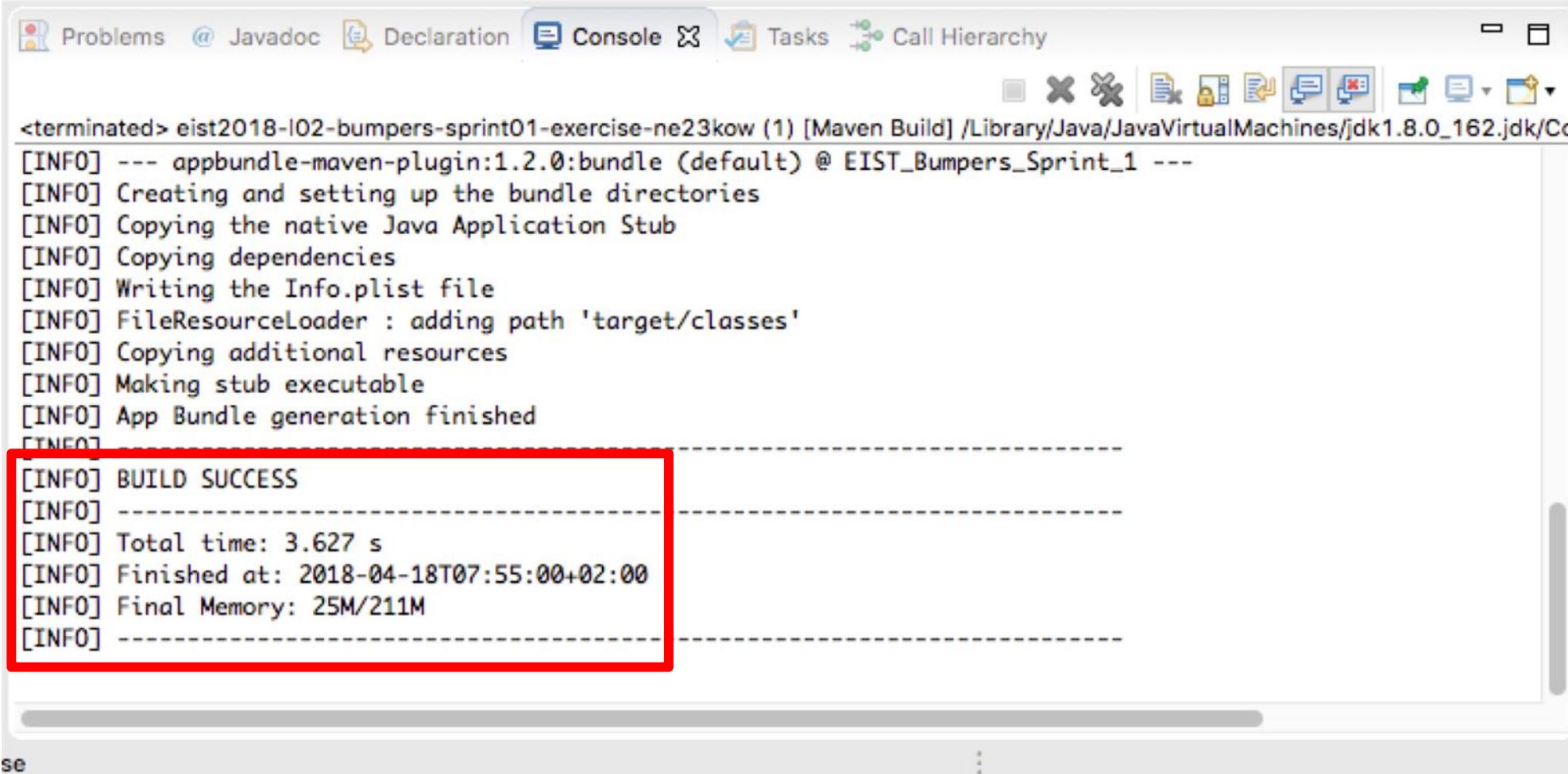
1. Select pom.xml

Task 9: Deliver the Application

More details in Lecture 10:
Configuration Management



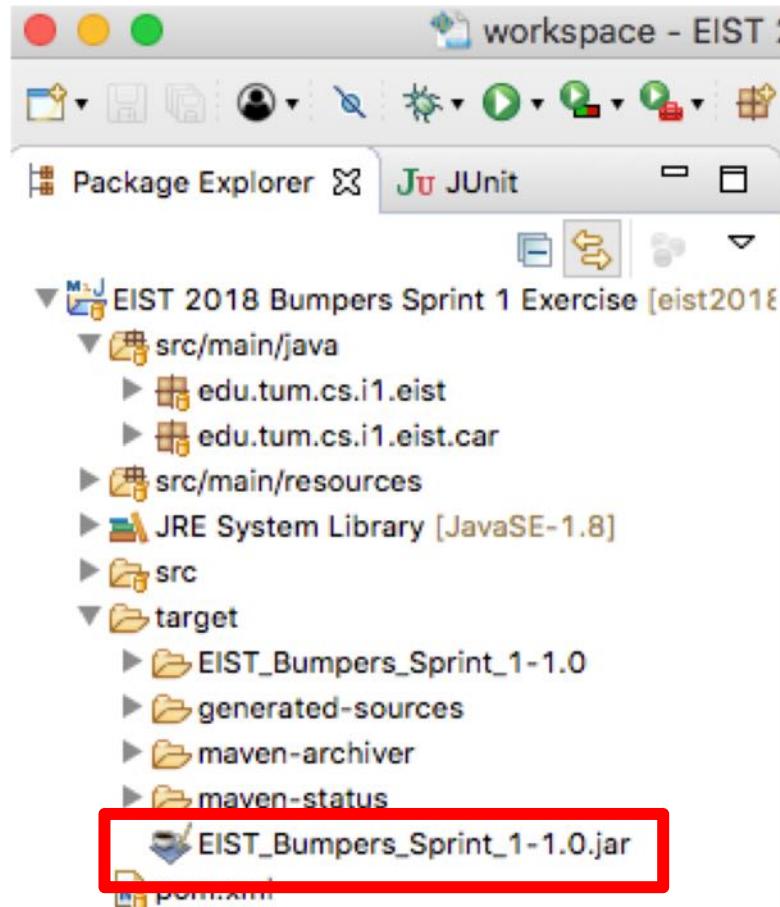
Task 9: Deliver the Application



```
<terminated> eist2018-l02-bumpers-sprint01-exercise-ne23kow (1) [Maven Build] /Library/Java/JavaVirtualMachines/jdk1.8.0_162.jdk/Co
[INFO] --- appbundle-maven-plugin:1.2.0:bundle (default) @ EIST_Bumpers_Sprint_1 ---
[INFO] Creating and setting up the bundle directories
[INFO] Copying the native Java Application Stub
[INFO] Copying dependencies
[INFO] Writing the Info.plist file
[INFO] FileResourceLoader : adding path 'target/classes'
[INFO] Copying additional resources
[INFO] Making stub executable
[INFO] App Bundle generation finished
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.627 s
[INFO] Finished at: 2018-04-18T07:55:00+02:00
[INFO] Final Memory: 25M/211M
[INFO] -----
```

Task 9: Deliver the Application

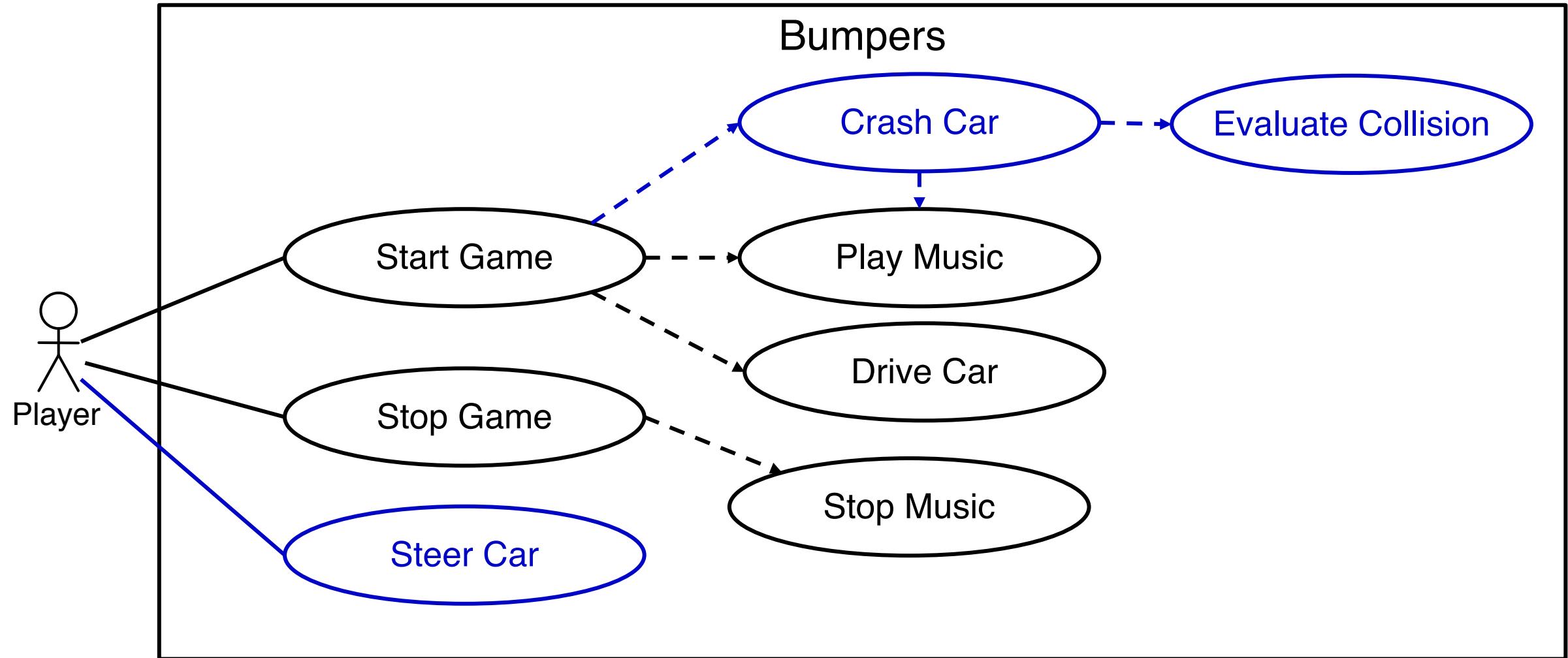
- Find the built application
EIST_Bumpers_Sprint_1-1.0.jar
in the directory **target**
- Open the application (jar file) and make sure it runs properly
- Rename the file and append your <full name>, e.g.
EIST_Bumpers_Sprint_1-Bernd_Bruegge-1.0.jar
- Send the application via Slack to your tutor



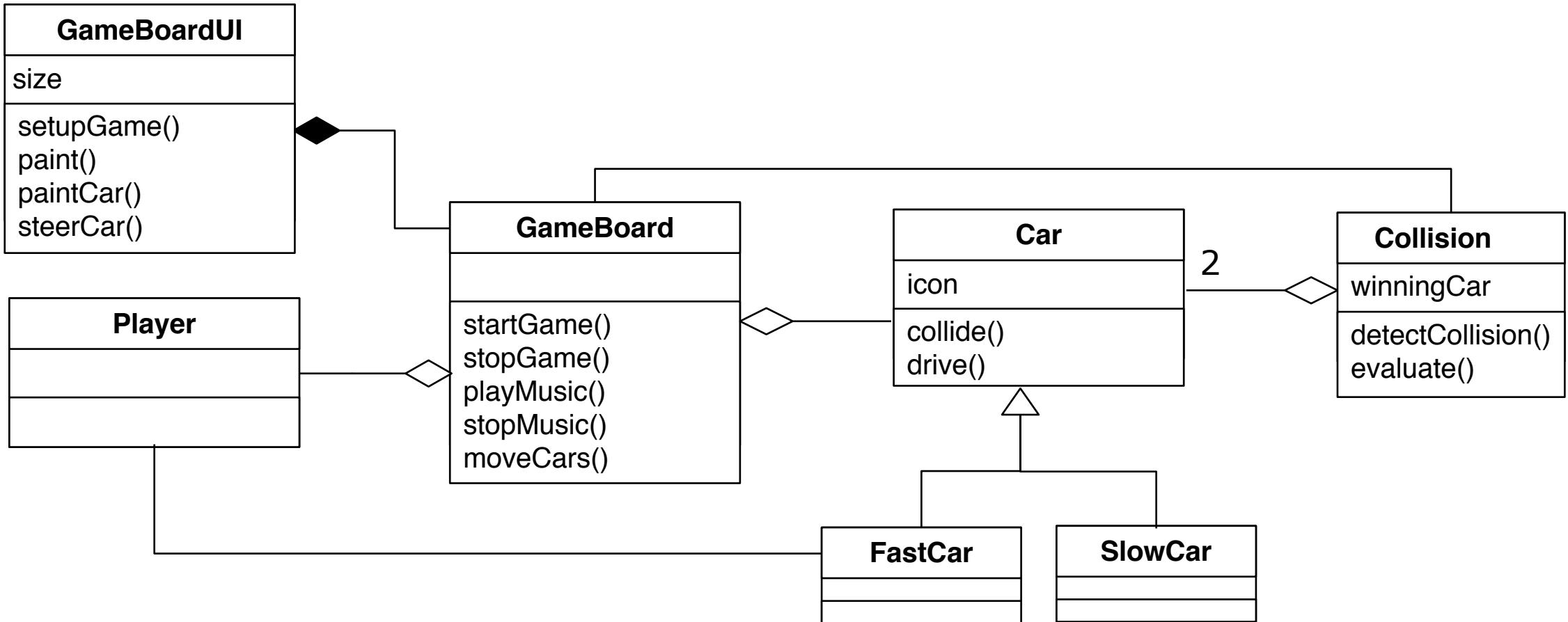
Bumpers – Sprint Backlog for Sprint 2

1. User Interface design of the game board
- ✓ 2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
- ✓ 5. The player starts and stops the game
- ✓ 6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play

Second Iteration of the Use Case Model for Bumpers



Second Iteration of the Analysis Object Model for Bumpers



Sprint 2 Backlog

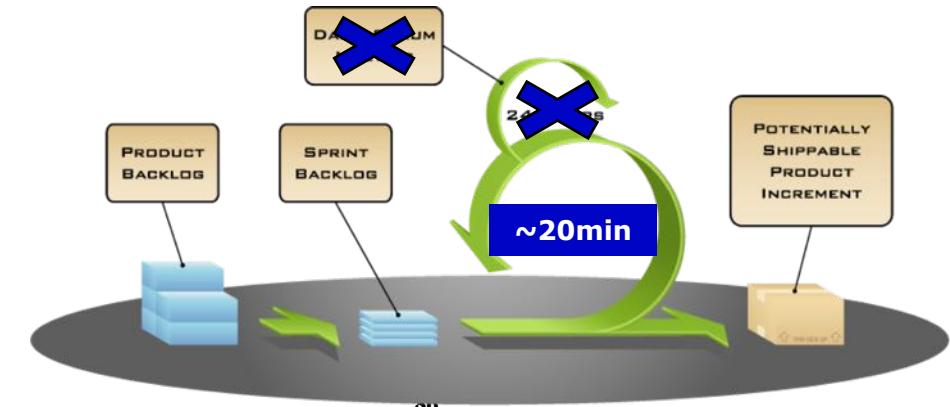
Lecture

- 3. Cars collide with each other and each collision has a winning car
- 8. The player steers the car with the mouse

In-class exercise

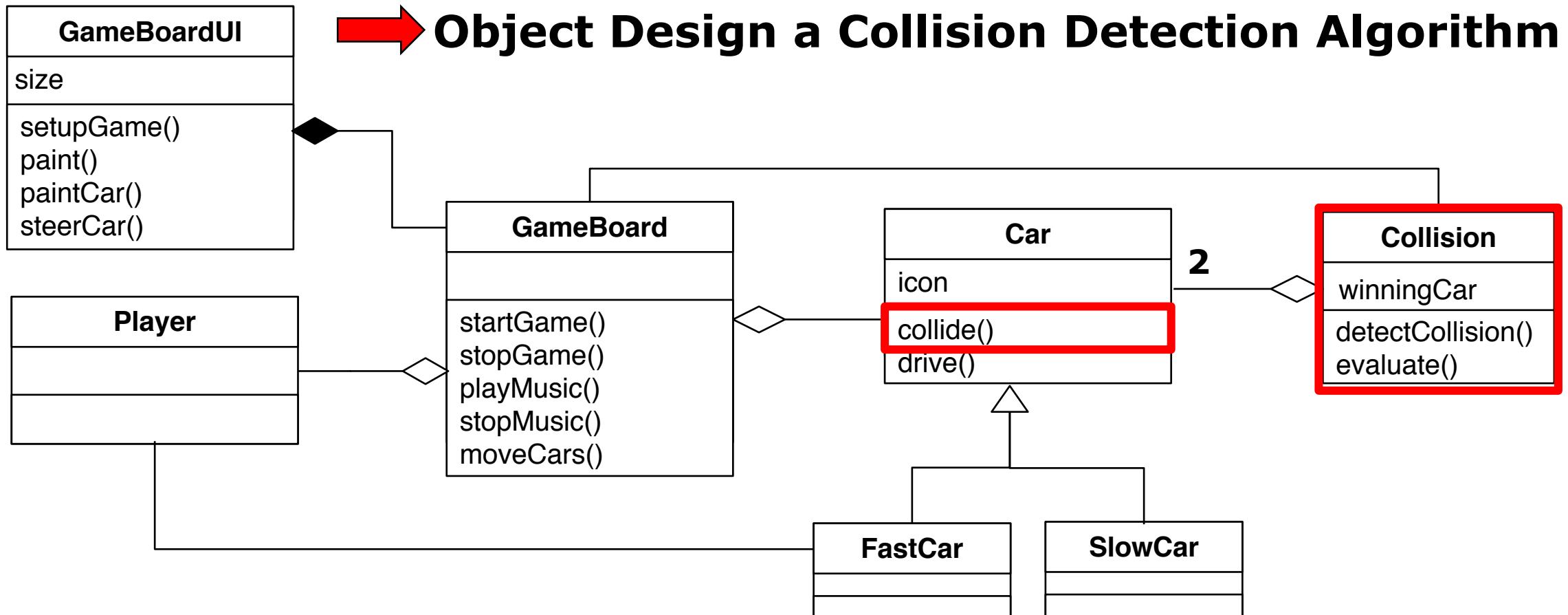
Reuse the code for the exercise Bumpers Sprint 2 on <https://arTEMIS.ase.in.tum.de>

- 13. A crash sound and/or animation plays when two cars collide



3. Cars collide with each other and each collision has a winning car

- How do we detect whether two cars collided with each other?
- How do we determine the winner of the collision?



Object Design of the Collision Detection Algorithm

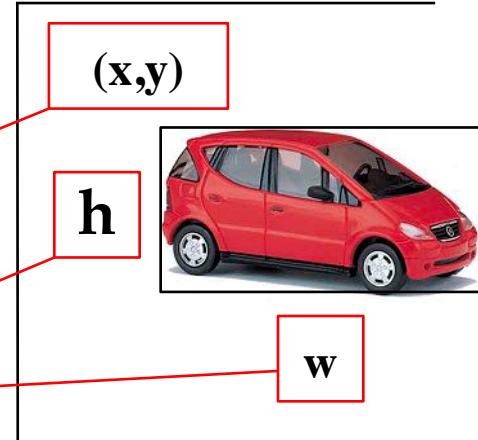
- To create realistic collisions, we have to consider a few additional details
- Each car has a bounding box, a rectangle covering all pixels of the car
- The Rectangle class is defined in package `java.awt`
- A rectangle is defined by the x and y coordinates of the upper left corner, its width, and its height
- `getSize()` computes the size of the rectangle
- `intersects()` checks if one given rectangle intersects with another given rectangle
- `grow(x, 7)` expands the size of a rectangle by a given height and width
- `translate(x, y)` moves the upper left corner point of a rectangle to the new coordinates x and y

Rectangle
+ <code>x: int</code>
+ <code>y: int</code>
+ <code>height: int</code>
+ <code>width: int</code>
+ <code>Dimension getSize()</code>
+ <code>intersects(Rectangle r): boolean</code>
+ <code>grow(int h, int v)</code>
+ <code>translate(int x, int y)</code>

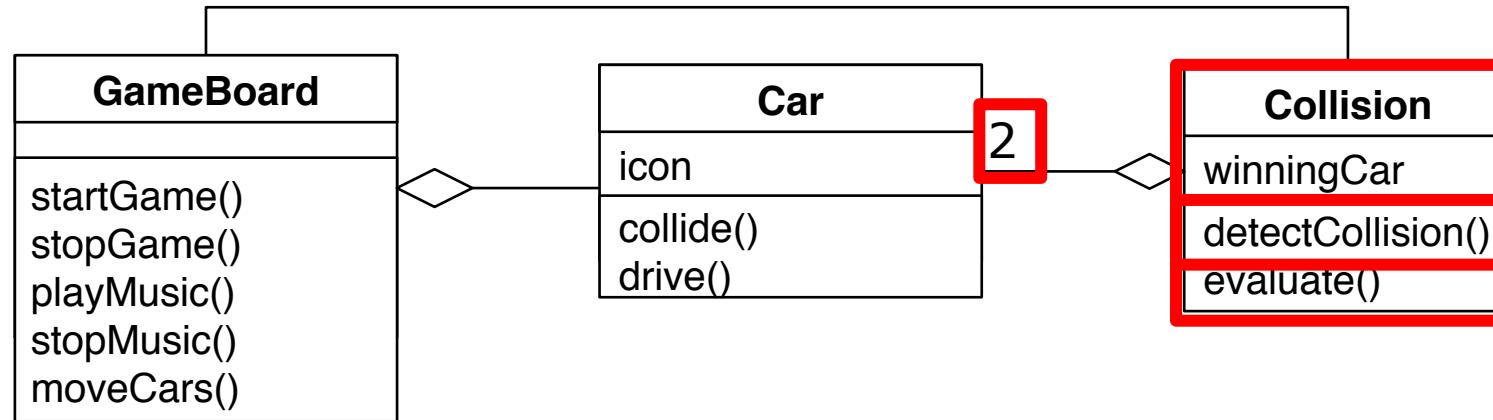
More details on Object Design in Lecture 6

Object Design of the Collision Detection Algorithm

- Each car is represented by its bounding box
 - This is the smallest rectangle, that contains all pixels of the picture of the car
- We get bounding box of a picture by calling `getSize()` followed by calling `getPosition()`
- A collision occurs, if two given bounding boxes have at least one pixel in common (`intersects()`)
 - Small problem: The picture of a car rarely resembles a rectangle, the bounding box often contains pixels that don't belong to the picture of the car.
- To compensate for the "empty space", we reduce the bounding box to 3/4 of its original size (with `growth()`), and place its center slightly to the lower left (with `translate()`)
- Remaining Problem: Who is the winner of the collision?

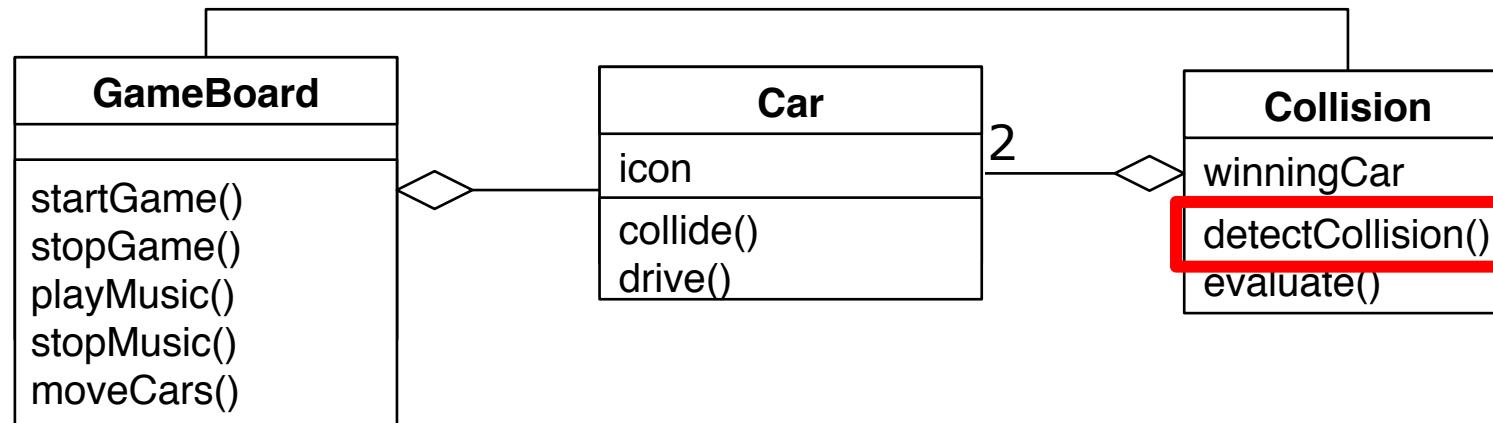


Implementation of the Collision Detection Algorithm



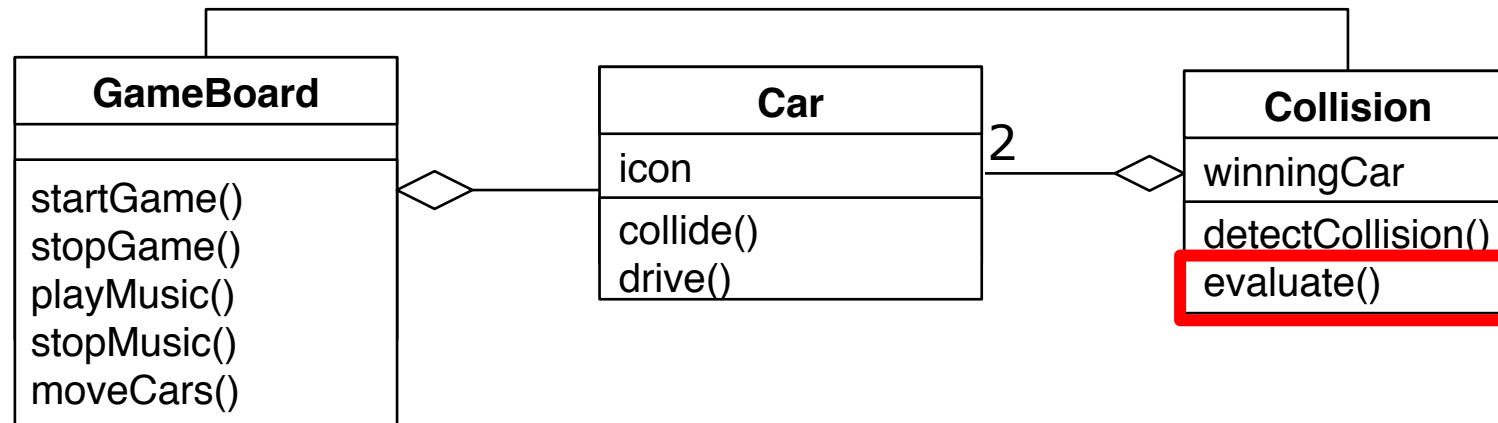
```
public class Collision {  
  
    private Car car1;  
    private Car car2;  
    public boolean isCollision;  
  
    public Collision(Car car1, Car car2) {  
        this.car1 = car1;  
        this.car2 = car2;  
        this.isCollision = detectCollision();  
    }  
}
```

Implementation of the Collision Detection Algorithm



```
private boolean detectCollision() {  
    Dimension d1 = car1.getSize( );  
    Point p1 = car1.getPosition( );  
    Rectangle r1 = new Rectangle(p1, d1);  
    r1.translate(p1.x / 8, p1.y / 8);  
    r1.grow(-1 * d1.width / 4, -1 * d1.height / 4);  
  
    Dimension d2 = car2.getSize( );  
    Point p2 = car2.getPosition( );  
    Rectangle r2 = new Rectangle(p2, d2);  
    r2.translate(p2.x / 8, p2.y / 8);  
    r2.grow(-1 * d2.width / 4, -1 * d2.height / 4);  
  
    return r1.intersects(r2);  
}
```

Implementation of the Collision Detection Algorithm



```
public Car evaluate() {
    Point p1 = car1.getPosition();
    Point p2 = car2.getPosition();

    Car winnerCar = null;
    if (p1.x > p2.x){
        winnerCar = car2;
    } else {
        winnerCar = car1;
    }
    return winnerCar;
}
```

Implementation of the Collision Detection Algorithm

```
public void moveCars() {  
    List<Car> cars = getCars();  
    int maxX = size.width;  
    int maxY = size.height;  
    for (Car car : cars) {  
        car.updatePosition(maxX, maxY);  
    }  
    player.getCar().updatePosition(maxX, maxY);  
    for (Car car : cars) {  
        if (car.isCrunched()) {  
            continue;  
        }  
        Collision collision = new Collision(player.getCar(), car);  
        if (collision.isCollision()) {  
            Car winner = collision.evaluate();  
            System.out.println(winner);  
        }  
    }  
}
```



Sprint 2 Backlog

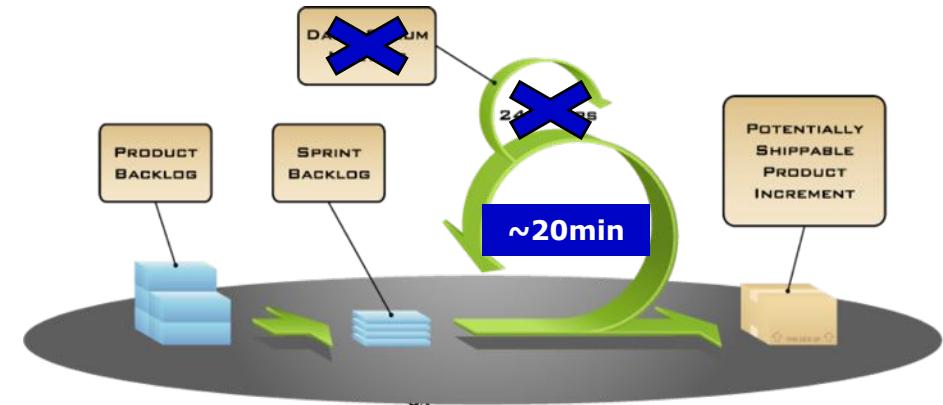
Lecture

- 3. Cars collide with each other and each collision has a winning car
- 8. The player steers the car with the mouse

In-class exercise

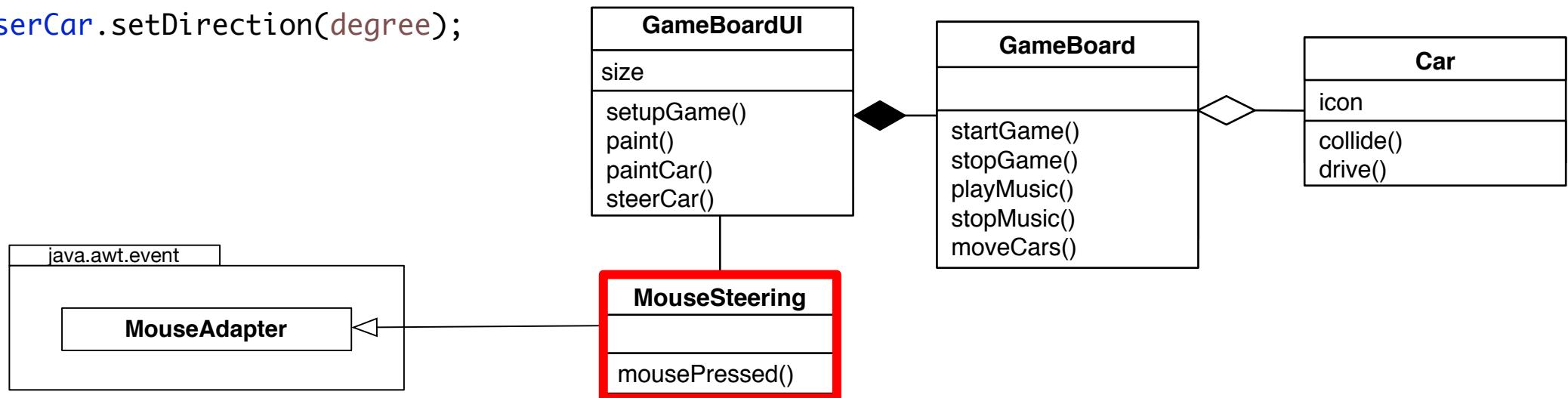
Reuse the code for the exercise Bumpers Sprint 2 on <https://arTEMIS.ase.in.tum.de>

- 13. A crash sound and/or animation plays when two cars collide



8. A player steers their car with the mouse

```
public class MouseSteering extends MouseAdapter {  
    private Car userCar;  
    private GameBoardUI gameBoard;  
  
    public MouseSteering(GameBoardUI playingField, Car userCar) {  
        this.userCar = userCar;  
        this.gameBoard = playingField;  
        this.gameBoard.addMouseListener(this);  
    }  
  
    public void mousePressed(MouseEvent e) {  
        . . .  
        userCar.setDirection(degree);  
    }  
}
```



Sprint 2 Backlog

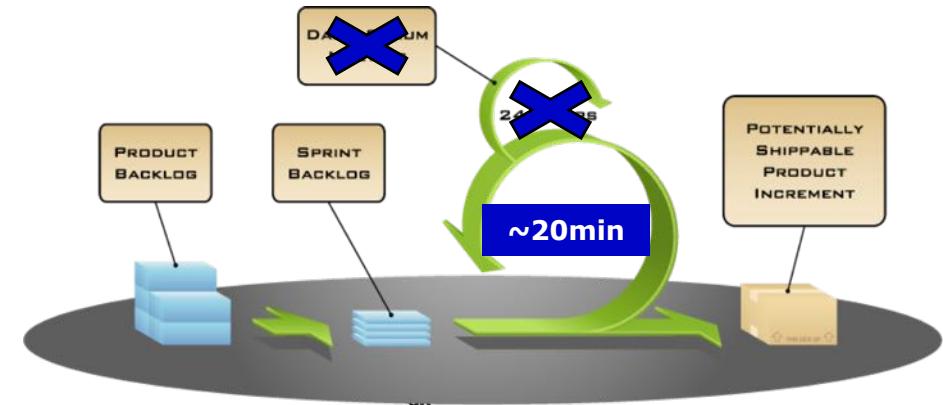
Lecture

- 3. Cars collide with each other and each collision has a winning car
- 8. The player steers the car with the mouse

In-class exercise

Reuse the code for the exercise Bumpers Sprint 2 on <https://arTEMIS.ase.in.tum.de>

→ 13. A crash sound and/or animation plays when two cars collide



In-Class Exercise: Issue a sound when two cars collide



Play a bang sound

```
public void moveCars() {  
    ...  
    if(collision.isCollision) {  
        Car winner = collision.evaluate();  
        System.out.println(winner);  
  
        // TODO Play bang sound  
        // Hint: take a look at AudioPlayer.java  
    }  
}  
  
public static AudioClip MUSIC;  
public static AudioClip BANG;  
  
public AudioPlayer() {  
    //...  
    AudioPlayer.BANG = loadAudioClip("Bang.au");  
    //...  
}
```

GameBoard
startGame()
stopGame()
playMusic()
stopMusic()
moveCars()

Challenge: Configure your own crash sound

Start exercise on ArTEMiS

- Open ArTEMiS on <https://artemis.ase.in.tum.de>
- Sign in with your TUM Online Account (e.g. "ne23kow") and open Courses
- Click on **Start Exercise** on **EIST 2018 Lecture 02 Bumpers Sprint 02**
- Clone your exercise repository and import the project into Eclipse
- Solve the tasks marked with TODO in the code
- Commit and push your solution and review the result on ArTEMiS

Introduction to Software Engineering (Summer 2018)			
Exercise	Due date	Results	Actions
Quiz 01	6 days ago	You have not participated in this quiz.	<button>C Practice</button> <button>Statistic</button>
EIST 2018 Lecture 02 Bumpers Sprint 01	in 2 days	✔ 4 passed, Score: 100% , Submission: 42 minutes ago	<button>Clone repository</button>
EIST 2018 Lecture 02 Bumpers Sprint 02	in 2 days	You have not started this exercise yet.	<button>Start exercise</button>

Click

Solution: Issue a sound when two cars collide

Play a bang sound

```
public void moveCars() {  
    ...  
    if(collision.isCollision){  
        Car winner = collision.evaluate();  
        System.out.println(winner);  
        this.audioPlayer.playBangAudio();  
    }  
}
```



Solution for the Challenge: Configure your own sound

2. Configure the sound file in AudioPlayer.java and create your own sound

- Drag and drop your Audio file to the resources folder



- Make sure to use an .au or .wav file
- mp3 files do not work!

- Rename the BANG sound

```
public static AudioClip MUSIC;
public static AudioClip BANG;

public AudioPlayer() {
    AudioPlayer.BANG = LoadAudioClip("MyBang.au");
    ...
}
```

Sprint 3 - Homework

1. User Interface design of the game board
- ✓ 2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
- ✓ 5. The player starts and stops the game
- ✓ 6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
- ✓ 8. The player steers the car with the mouse
9. The player can change their car's speed

10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
- ✓ 13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play
15. Implement a new type of collision

The customer can change the product backlog after any sprint

Suggestions for types of collisions

- Treat cars like a wave
- The winner has to hit the car twice
- The winner is the car on top
- Find your own solution
- All working solutions will be entered into a lottery
- The most creative solution will be awarded a prize in the next lecture.

Summary

- We gave you a first introduction Model-Based Software Engineering
- We introduced the notion of a Software Lifecycle
- We distinguished between defined and empirical process control models
- We introduced Abbot's Technique to extract the abstractions for the system model
- We did a first pass on UML: Use case and class diagrams
- We used two in-class exercise to do the analysis, object design, implementation and delivery of a game called Bumpers.

Morning Quiz

- Quiz 3a
 - Starting Time: 8:00
 - End Time: 8:10
- To do the quiz, use ArTEMiS
- The Lecture starts at 8:10

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	<input type="button"/> Start exercise
Good Morning Quiz 03a		<input type="button"/> Open Quiz



Only click on Submit when you have finished all answers!

Object-Oriented Software Engineering

Using UML, Patterns, and Java

Requirements Elicitation And Analysis

Bernd Bruegge
Chair for Applied Software Engineering
Technische Universität München
26 April 2018

Roadmap for Today's Lecture

- **Context and Assumptions**

- We have completed an initial explanation of use cases and class diagrams
- You are familiar with the content of Chapter 2 in the OOSE text book by Bruegge and Dutoit

- **Content of this lecture**

- A deeper look into class diagrams and use cases
- UML is an extensible language: Predefined types and Stereotypes
- Requirements Elicitation and Analysis

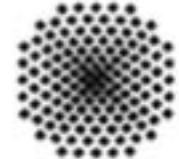
- **Objective:** At the end of this lecture you are able to...

- ... create different types of objects with stereotypes
- ... explain the difference between different types of requirements
- ... use different techniques to describe requirements
- ... understand requirements reviews and requirements documents.

Outline of the Lecture

- Odds and Ends
 - Ferienakademie
 - Availability of the Book in German
- A deeper look into use case diagrams
- A deeper look into class diagrams
- Extending UML: Stereotypes
- Dynamic Modeling
- Requirements Engineering
- Types of Requirements
- Techniques to describe Requirements
- Requirements Review

Ferienakademie 2018



Universität Erlangen-Nürnberg - Technische Universität München - Universität Stuttgart

- Duration: Sunday 23. September - Friday 5. October 2018
- Free travel, free lodging, free food
- Other activities: Hiking, table tennis, chess, grass ski
- Exiting Projects.





8

1

EO
Program 2

23.09. – 5.10. 2018

Sarntal (Südtirol)

1	Der Sarntaler: eine digitale Wirkwelt für die Ferienakademie	B. Schröder, Erlangen H. Seidl, München	Informatik, Wirtschaftsinformatik, Mathematik (Bachelor ab 1. oder 2. Studienjahr)
2	SarntalX: Agile Aircraft Design	B. Brügge, München S. Wagner, Stuttgart	Informatik, Software Engineering, Elektrotechnik, Luft- und Raumfahrttechnik, Maschinenwesen, Industrial Design, Human Factors Engineering (Bachelor ab 2. Studienjahr oder Master)
3	Physik und Elektronik im Alltag	G. Denninger, Stuttgart R. Gross, München V. Krstic, Erlangen (GD)	Physik, Elektro- und Informationstechnik (Bachelor im 1. oder 2. Studienjahr)
4	Multiscale Problems in Mechanics: Models – Simulation – Application	E. Rank, München H. Steinb, Stuttgart	Ingenieurwissenschaften, Mathematik, Informatik (Master)
5	Simulation of Fluids and Wave Phenomena – High Order, High Performance, High Productivity	M. Bader, München C.-D. Munz, Stuttgart D. Fey, Erlangen (GD) H. Kestner, Erlangen (GD)	Ingenieurwissenschaften, Mathematik, Informatik, Physik (Bachelor ab 3. Studienjahr oder Master)
6	Accelerating Physics Simulations with Deep Learning	M. Engel, Erlangen N. Thuerey, München M. Mehrl, Stuttgart (GD)	Informatik, Mathematik, Physik, Verfahrenstechnik, Chemie, Ingenieurwissenschaften (Bachelor ab 3. Studienjahr oder Master)
7	Energiewende: Faktencheck mit (Big) Data Analytics	K. Diepold, München J. Karl, Erlangen	Ingenieurwissenschaften, Naturwissenschaften, Wirtschaftswissenschaften, Mathematik, Informatik (alle Fachsemester)
8	Simulation Technology: From Models to Software	B. Flemisch, Stuttgart B. Wohlmuth, München G. Leugering, Erlangen (GD)	Mathematik, Informatik, Physik, Ingenieurwissenschaften, Simulation Technology (Bachelor ab 2. Studienjahr oder Master)
9	Redundancy and Irrelevance in Source and Channel Coding	B. Edler, Erlangen G. Kramer, München A. Kavc, Erlangen (GD)	Elektrotechnik, Informations- und Kommunikationstechnik, Informatik, Mathematik, Physik (Bachelor ab 3. Studienjahr oder Master)
10	Computational Medical Imaging	T. Lasser, München A. Maier, Erlangen	Medizintechnik, Informatik, Elektrotechnik, Mathematik, Physik (Bachelor ab 3. Studienjahr oder Master)

Course Example Course 2: Agile Aircraft Design

- Rugby: Modified Scrum-based Approach
- Design and implementation of drone applications
 - Design of the airplane, programming, customized chips, Arduino, 3-D Printing, ...
- Students and professors from TUM, Alexander Universität Erlangen-Nürnberg, Universität Stuttgart and a guest professor from Carnegie Mellon University, USA
- Apply now:
<https://www.ferienakademie.de/en/application/>
- Deadline: 2 May 2018

Agile Development



Location: Sarntal, Dolomites, Italy



Outline of the Lecture

- Odds and Ends
 - Ferienakademie
 - Availability of the Book in German
- A deeper look into use case diagrams
- A deeper look into class diagrams
- Extending UML: Stereotypes
- Advanced Modeling
- Requirements Engineering
- Types of Requirements
- Techniques to describe Requirements
- Requirements Review

Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java

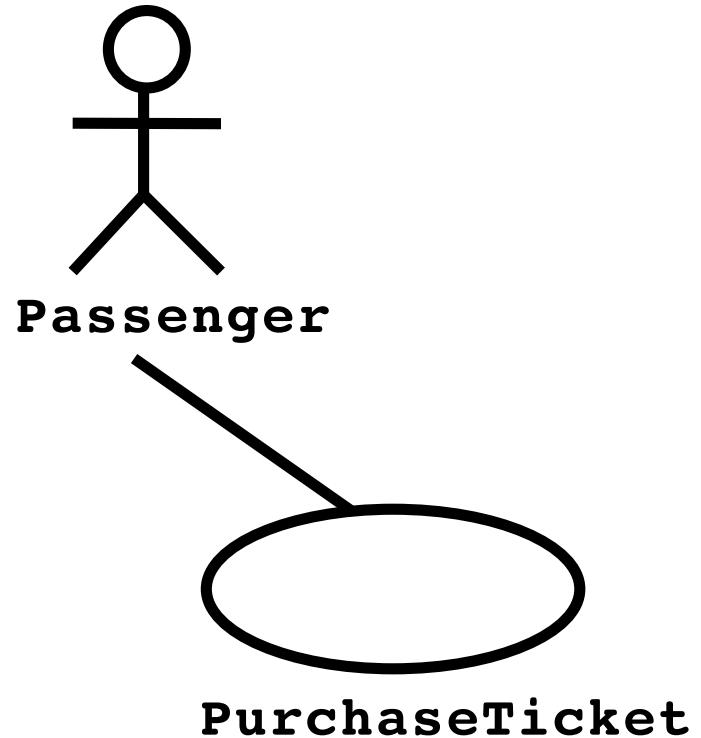
- eBook licensed only for TUM students
- Pearson Education Studium
- Available on Moodle



Outline of the Lecture

- Odds and Ends
 - Ferienakademie
 - Availability of the Book in German
- A deeper look into use case diagrams
- A deeper look into class diagrams
 - Extending UML: Stereotypes
 - Advanced Modeling
 - Requirements Engineering
 - Types of Requirements
 - Techniques to describe Requirements
 - Requirements Review

Use Case Diagrams



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

An **actor** represents a specific type of user of the system (also called a **role**)

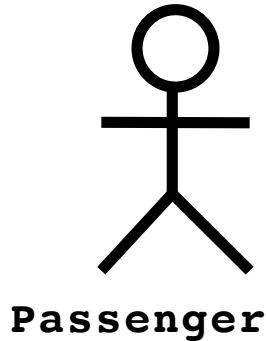
A **use case** represents a functionality provided by the system

Use cases are associated with actors

Use case model:

A graph of use cases, actors and their relationships that describes the functionality of the system

Actors

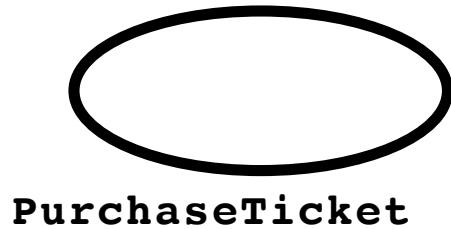


Name

- An actor is a model for an external entity which interacts or 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 a navigation system with GPS information.

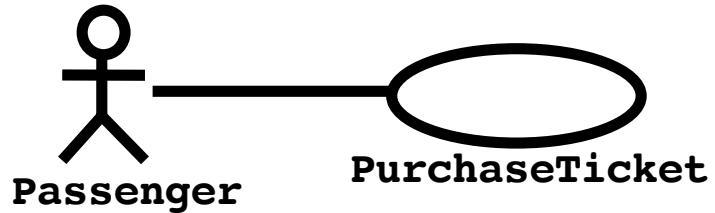
Optional
Description

Use Cases



- 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 the actor and the system
- A textual 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 actors:** Passenger
- 3. Entry conditions:**
 - The Passenger stands in front of the ticket distributor
 - The Passenger has sufficient money to purchase a ticket
- 4. Exit conditions:**
 - The Passenger has the ticket

- 1. Name**
- 2. Participating actors**
- 3. Entry conditions**
- 4. Exit conditions**
- 5. Flow of events**
- 6. Special requirements**

5. Flow of events:

1. The passenger selects the number of zones to be traveled
2. The ticket distributor displays the amount due
3. The passenger inserts at least the amount due
4. The ticket distributor returns change
5. The ticket distributor issues the ticket

6. Special requirements:

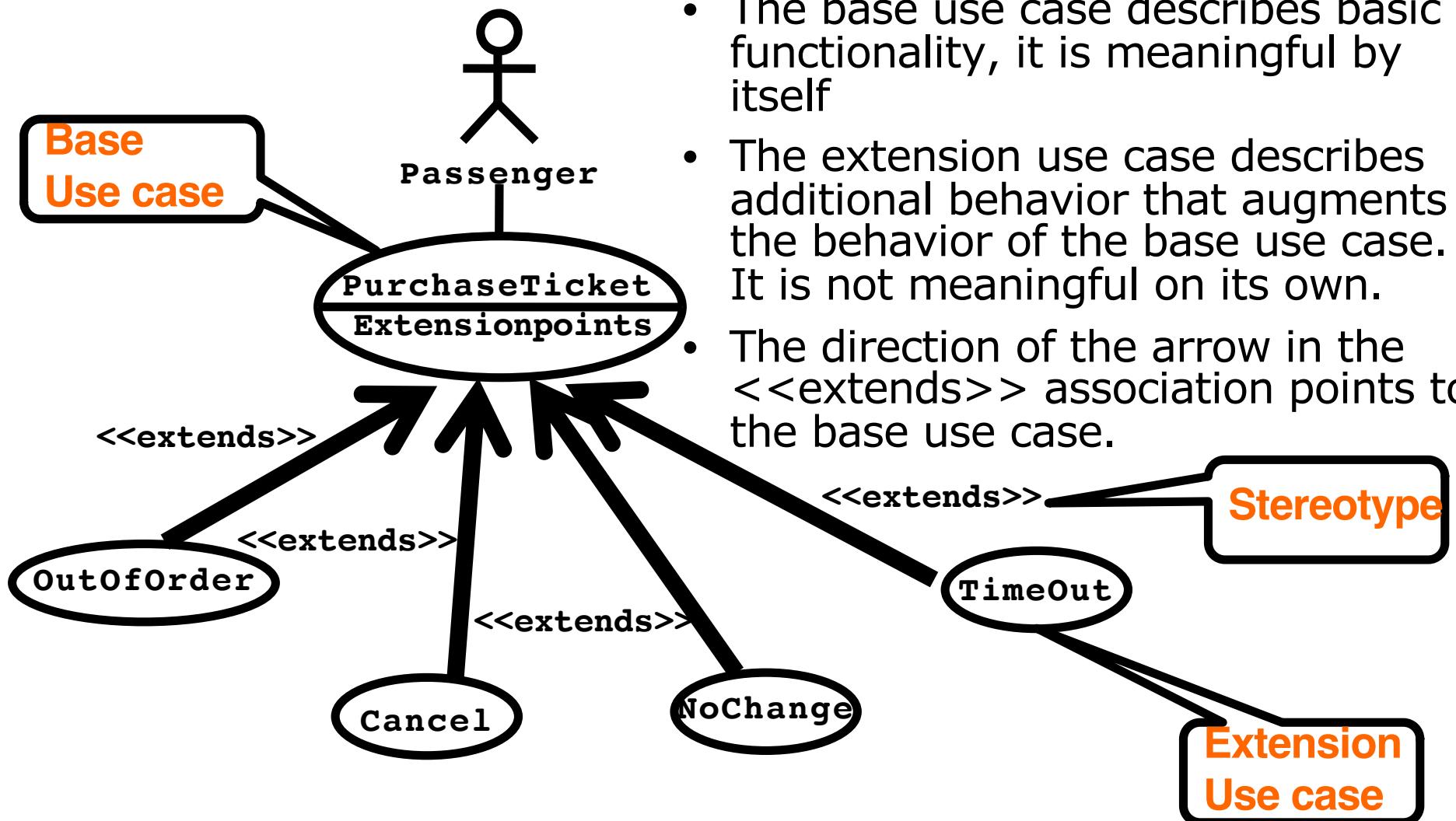
- The ticket distributor is connected to a power source.

Uses Cases Can be Related to Each Other

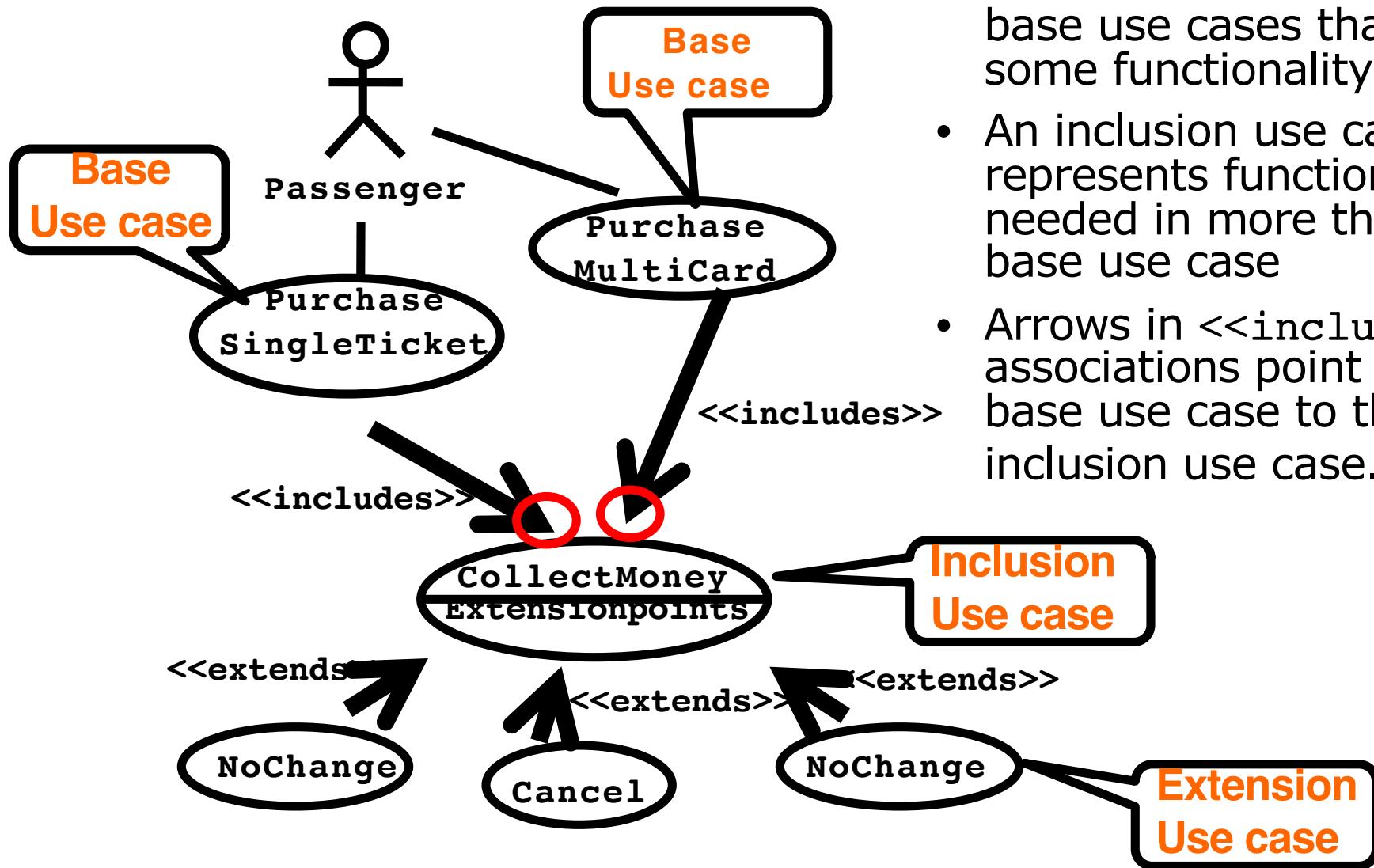
We distinguish two types of relationships

- **<<extends>> Relationship**
 - To model rarely invoked use cases or exceptional functionality
- **<<includes>> Relationship**
 - To model functional behavior that is common to more than one use case.

The <<Extends>> Relationship



The Includes Relationship



- Assume we have several base use cases that share some functionality
- An inclusion use case represents functionality needed in more than one base use case
- Arrows in `<<includes>>` associations point from the base use case to the inclusion use case.

Conclusion: There are 3 Different types of Use Cases

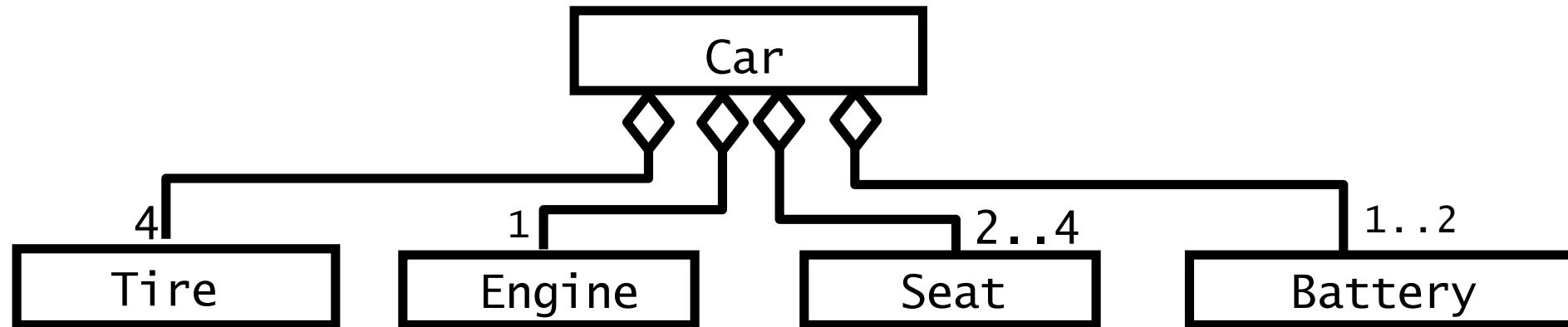
- **Base use case**
 - A use case that describes a function of the system ("core functionality")
- **Extension use case**
 - A use case that describes additional behavior that augments the behavior of a use case
 - Contains Extension Points
- **Inclusion use case**
 - A use case that describes functionality which is needed in more than one use case.

Outline of the Lecture

- Odds and Ends
 - Ferienakademie
 - Availability of the Book in German
- A deeper look into use case diagrams
- A deeper look into class diagrams
 - Extending UML: Stereotypes
 - Dynamic Modeling
 - Requirements Engineering
 - Types of Requirements
 - Techniques to describe Requirements
 - Requirements Review

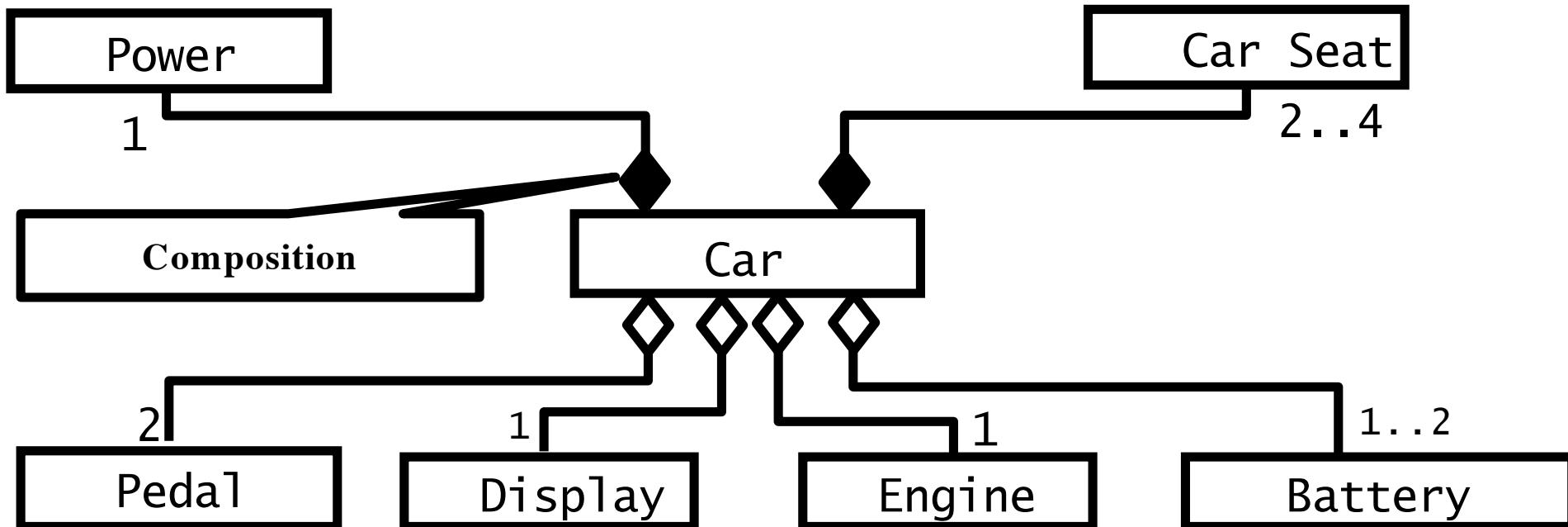
Aggregation

- An **aggregation** is a special case of an association denoting a “Part-of” hierarchy
- The *aggregate* is the parent class, the components are the children classes
- Example: Modeling a car with aggregation (Iteration on the previous car model)

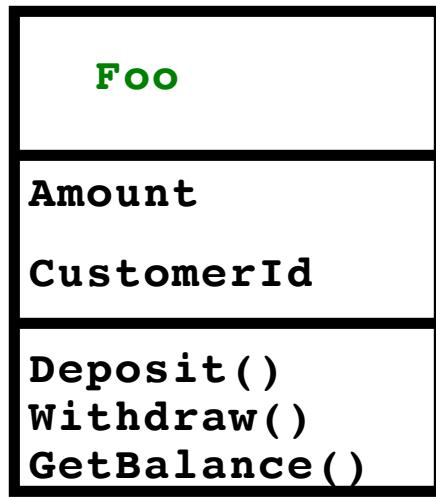


Composition

- **Composition** is a special form of aggregation
 - In UML diagrams it is drawn as a solid diamond
- It is used when the life time of the component instances are controlled by the aggregate
 - The components (parts) do not exist on their own
 - The aggregate controls/destroys the components



Let's Practice Object Modeling with Class Diagrams

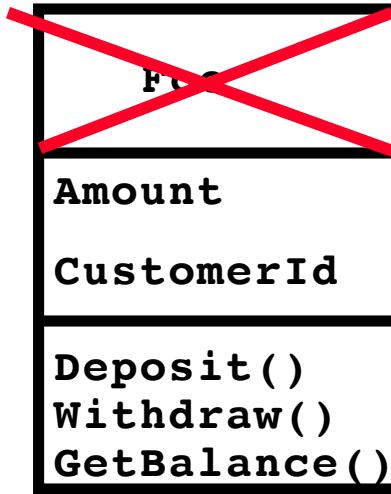
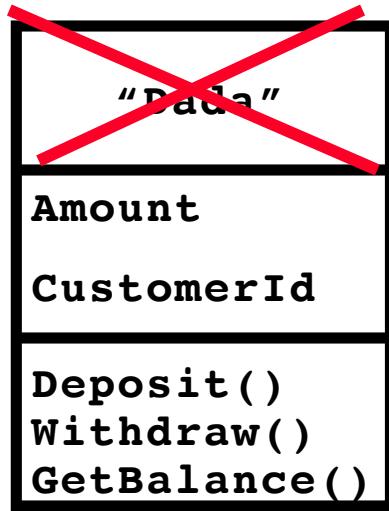


First Step: Class Identification

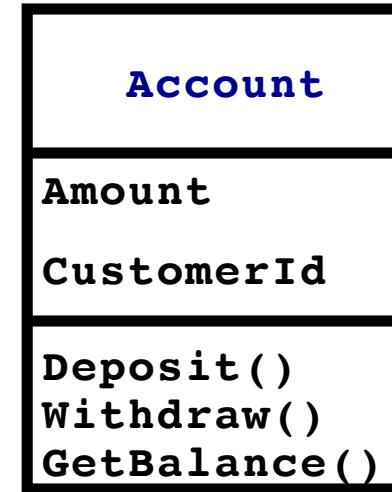
Identify name of Class, find attributes and operations

Is **Foo** the right name?

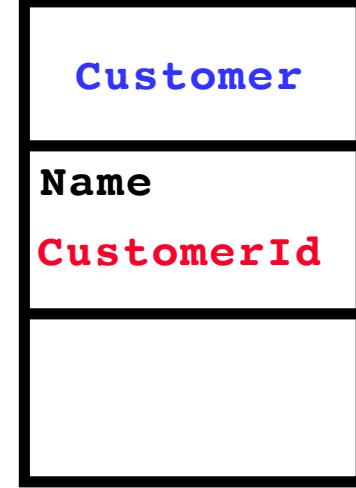
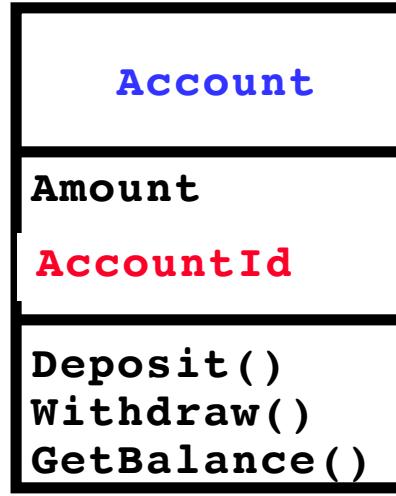
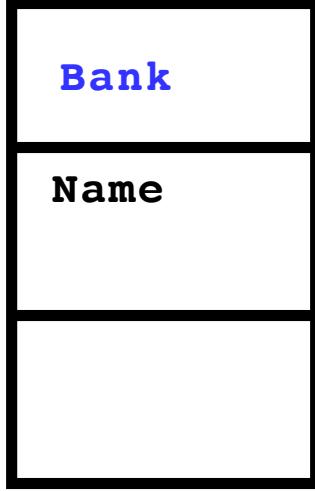
Object Modeling in Practice: Brainstorming



Is Foo the right name?

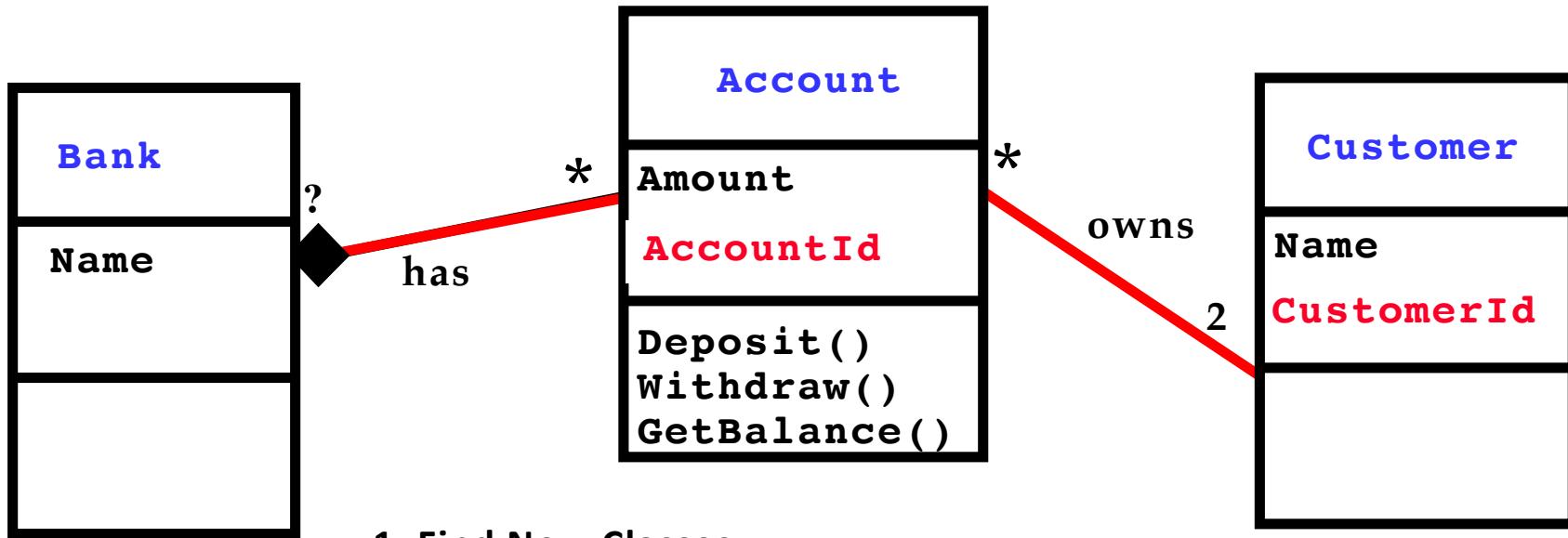


Object Modeling in Practice: More Classes



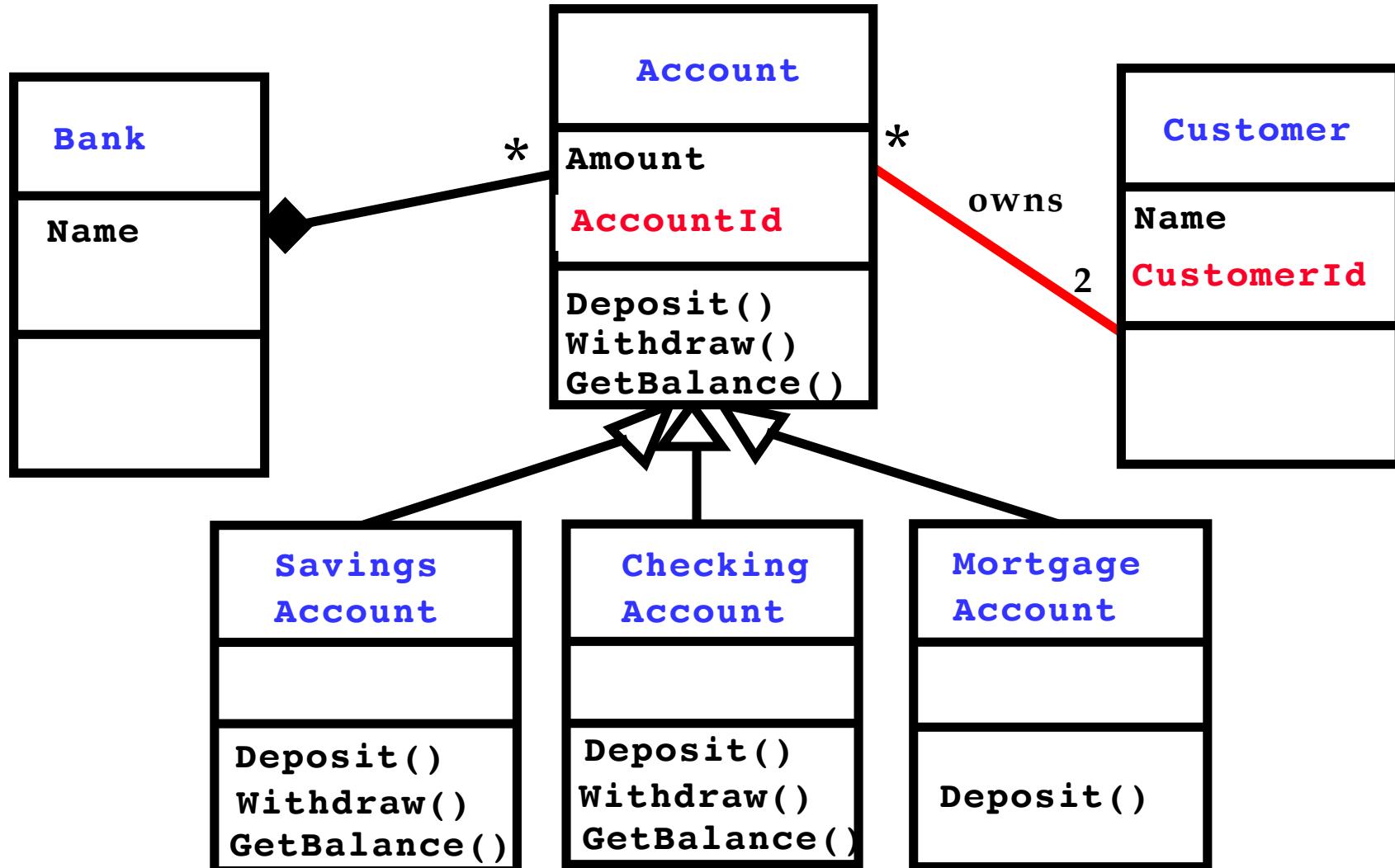
1. Find New Classes
2. Review Names, Attributes and Methods

Object Modeling in Practice: Associations

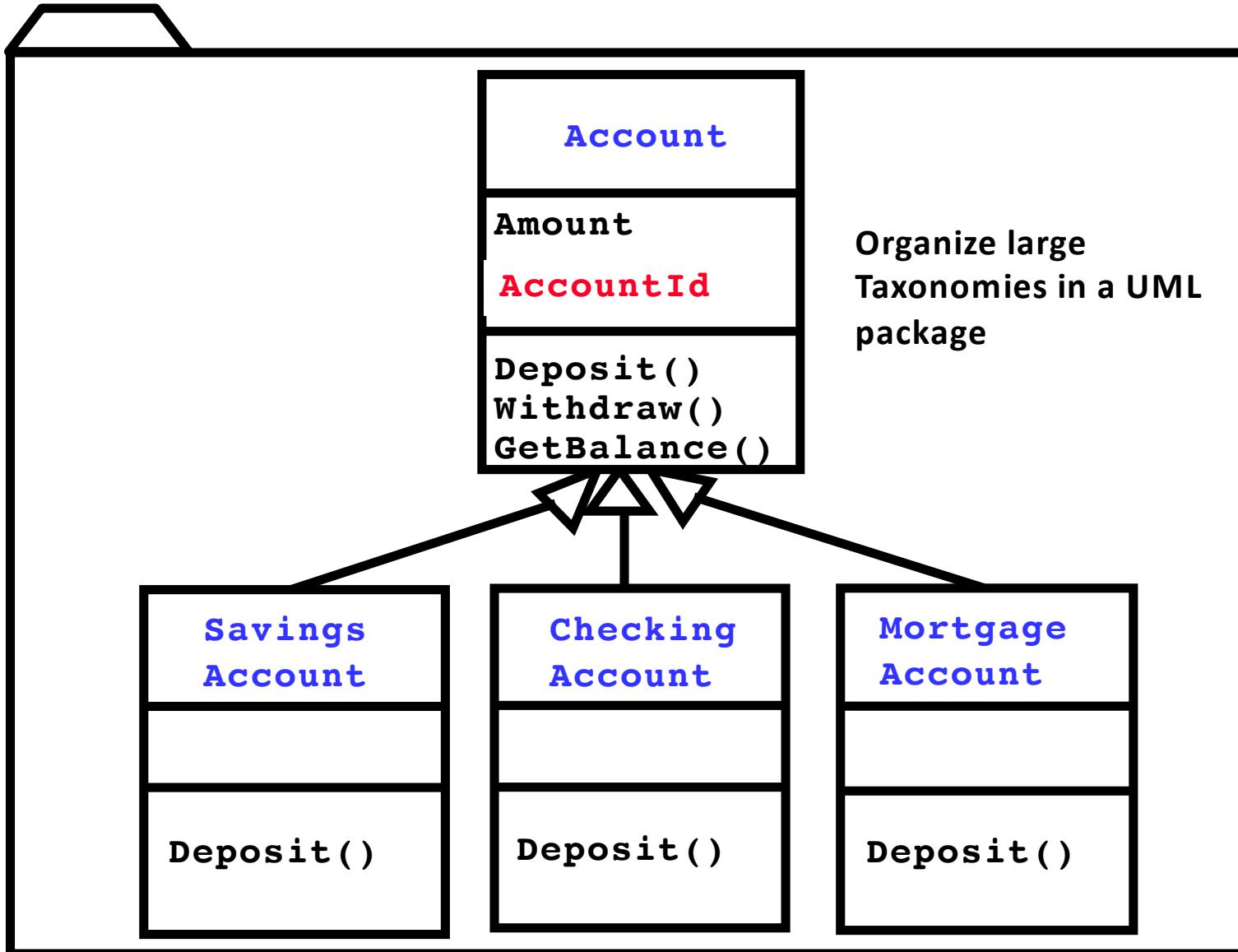


1. Find New Classes
2. Review Names, Attributes and Methods
3. Find Associations between Classes
4. Label the generic associations
5. Determine the multiplicity of the associations
6. Review associations

Object Modeling in Practice: Find Taxonomies

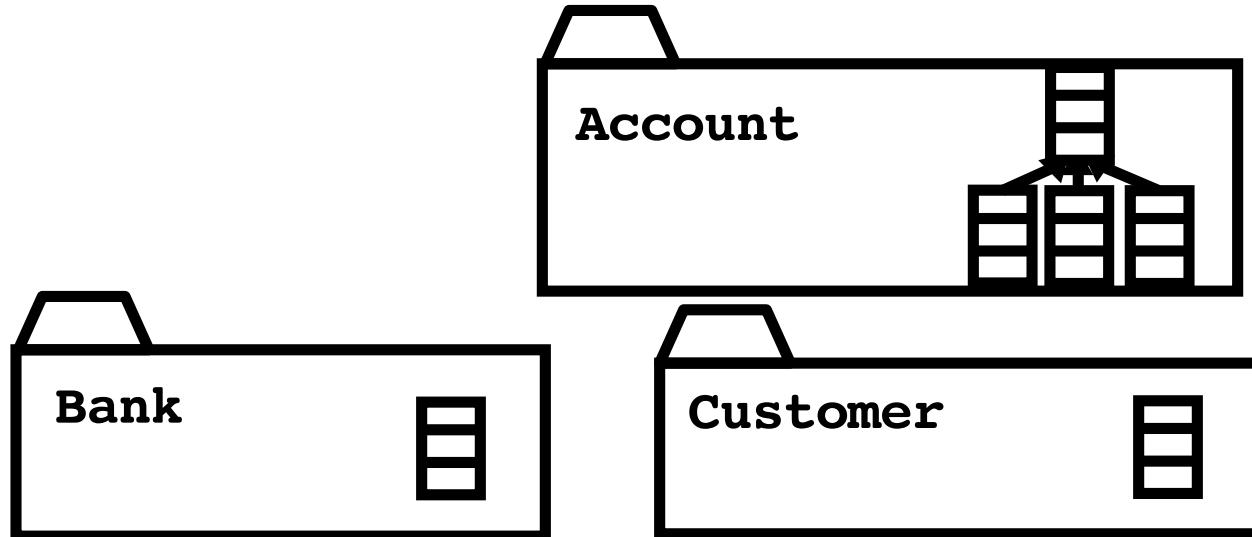


Object Modeling in Practice: Simplify, Organize



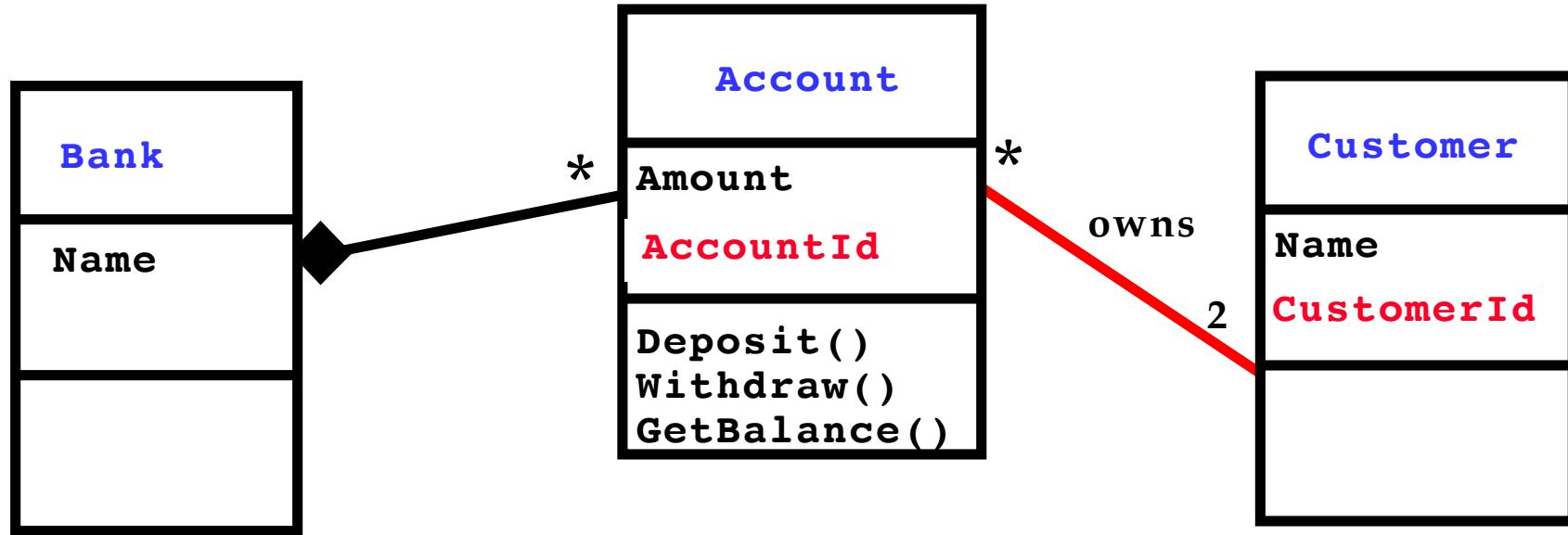
UML 2 Package Notation

- 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: Use the 7 ± 2 Heuristic



or better 5+-2!

Outline of the Lecture

- Odds and Ends
 - Ferienakademie
- A deeper look into use case diagrams
- A deeper look into class diagrams

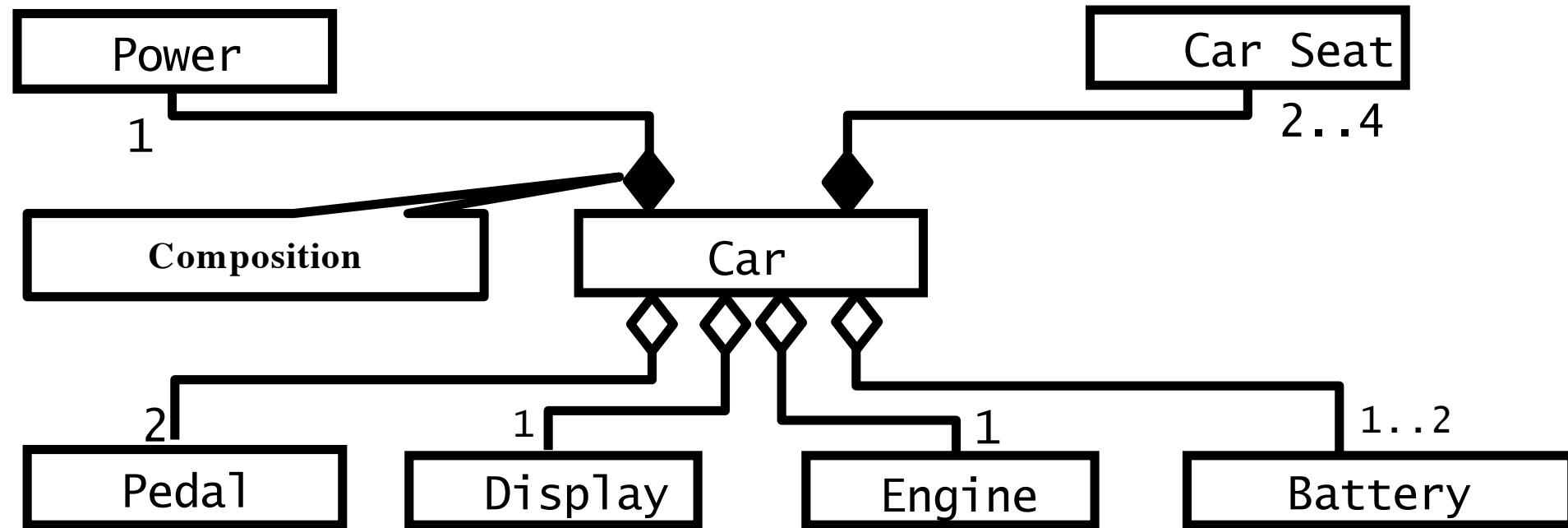
→ Extending UML: Stereotypes

- Dynamic Modeling
- Requirements Engineering
- Types of Requirements
- Techniques to describe Requirements
- Requirements Review

UML Supports 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 to be performed by the system.

Review: Modeling A Car



Example: Objects of a Car

To distinguish different object types
in a class diagram we use stereotypes

Pedal

Engine

Battery

Display

Seat

Power

Boundary Object

Control Object

Entity Object

Example: Objects of a Car

To distinguish different object types
in a class diagram we use stereotypes

<<Boundary>>
Pedal

<<Control>>
Engine

<<Entity>>
Battery

<<Boundary>>
Display

<<Entity>>
Seat

<<Entity>>
Power

Boundary Object

Control Object

Entity Object

Graphical Icons for Object Types

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

<<Boundary>>
Pedal

<<Control>>
Engine

<<Entity>>
Battery

<<Boundary>>
Display

<<Entity>>
Seat

<<Entity>>
Power

Boundary Object

Control Object

Entity Object

Graphical Icons for Object Types

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



Pedal

<<Control>>
Engine

<<Entity>>
Battery

<<Boundary>>
Display

<< Entity >>
Seat

<<Entity>>
Power

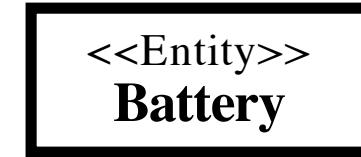
Boundary Object

Control Object

Entity Object

Graphical Icons for Object Types

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



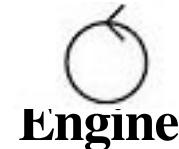
Boundary Object

Control Object

Entity Object

Graphical Icons for Object Types

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



Boundary Object

Control Object

Entity Object

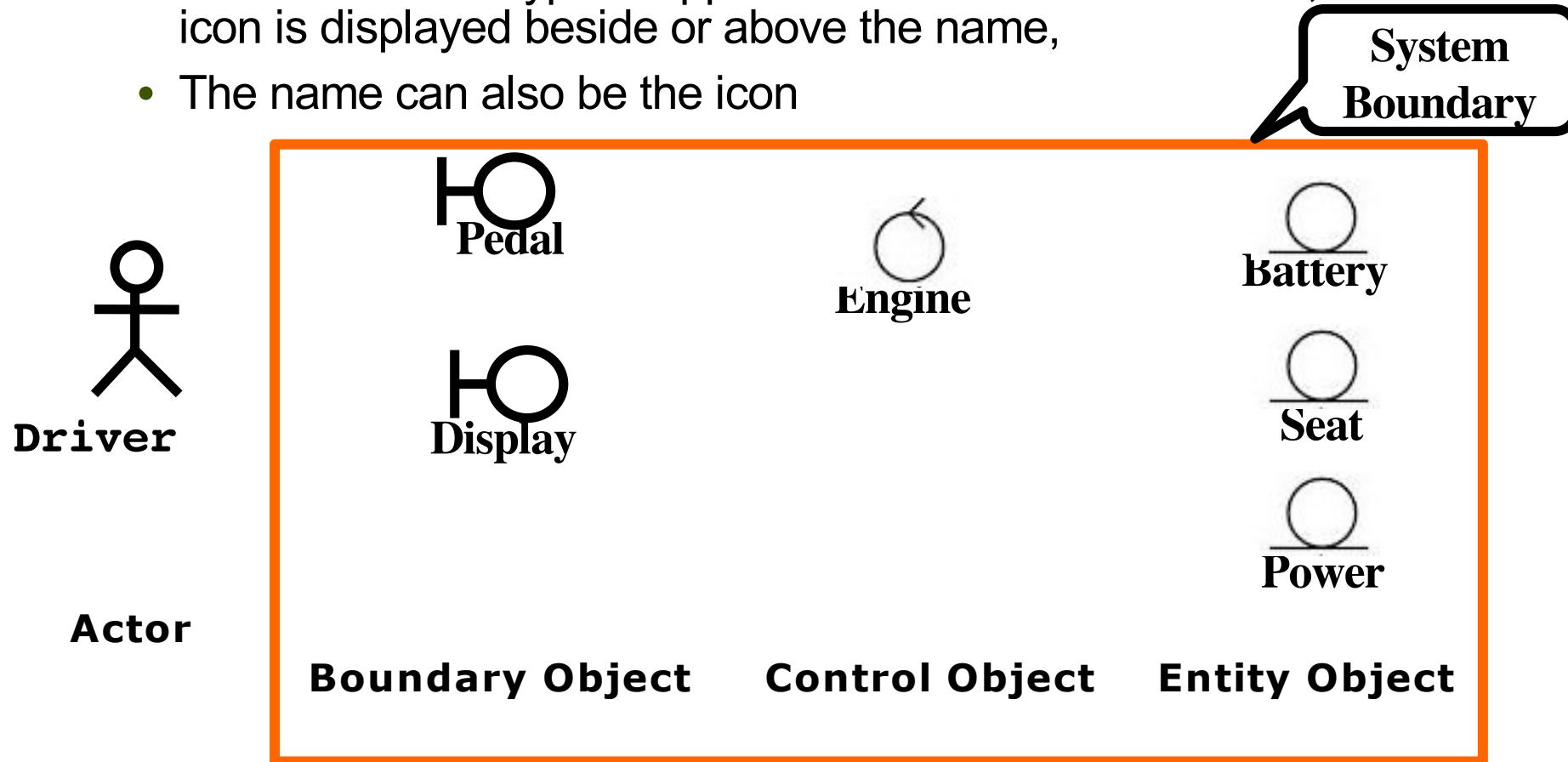
<<Entity>>
Battery

<<Entity>>
Seat

<<Entity>>
Power

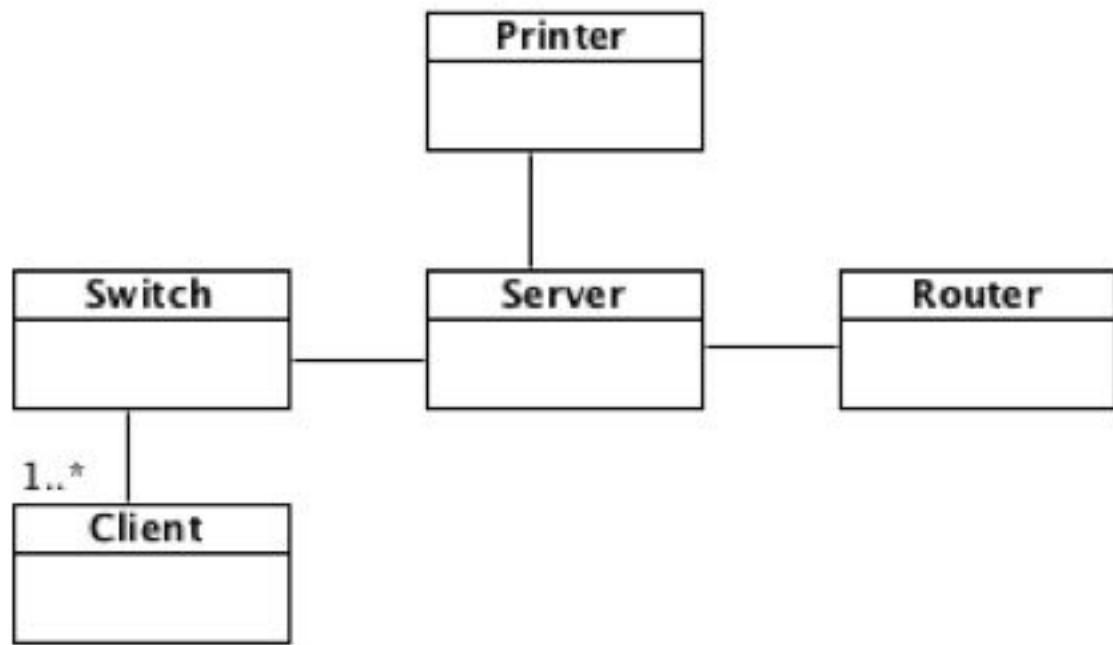
Graphical Icons for Object Types

- We can also use **graphical icons** to identify a stereotype
 - When the stereotype is applied to a UML model element, the icon is displayed beside or above the name,
 - The name can also be the icon



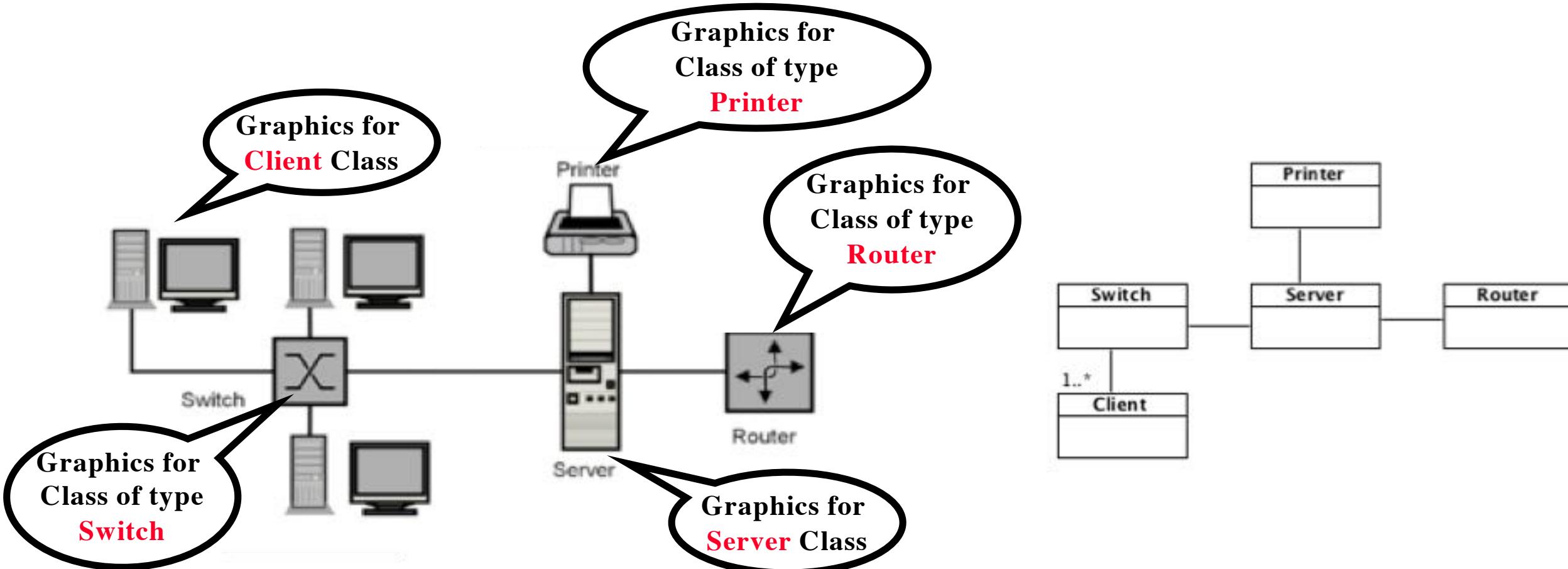
Another Example for Graphical Icons: Model of a Network

- When modeling a network, we use a class diagram for the components of the network: **Client**, **Switch**, **Server**, **Printer** and **Router**



Another Example for Graphical Icons: Model of a Network

- When modeling a network, we use a class diagram for the components of the network: **Client**, **Switch**, **Server**, **Printer** and **Router**

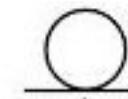


Pros and Cons of Stereotype Graphics

- Advantages:
 - UML diagrams are often easier to understand when they contain graphics and icons for stereotypes, especially if the graphics are standardized in the application domain
 - Graphics 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 is much harder for them to understand what is going on
 - Additional symbols add to the burden of learning to read the diagrams.

Stereotypes

- 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
- Stereotypes can be represented *textually* as well as with *graphical icons*
 - ✓ Class diagrams: <>boundary>>, <>control>>, <>entity>>



- Other stereotypes:
 - Use case relationships: <>extend>>, <>include>> (**Today: Lecture 3**)
 - Subsystem interface: <>interface>> (**Lecture 4 System Design**)
 - Stereotypes for classifying method behavior:
<>constructor>>, <>getter>>, <>setter>> (**Lecture 6 Object Design**)

Important Distinction: Actor vs Class vs Object

- **Actor**

- Any entity outside the system, interacting with the system ("Passenger")



Passenger

- **Class**

- A concept from the application domain or in the solution domain
- Classes are part of the system model ("User", "Ticket distributor", "Server")

User

- **Object**

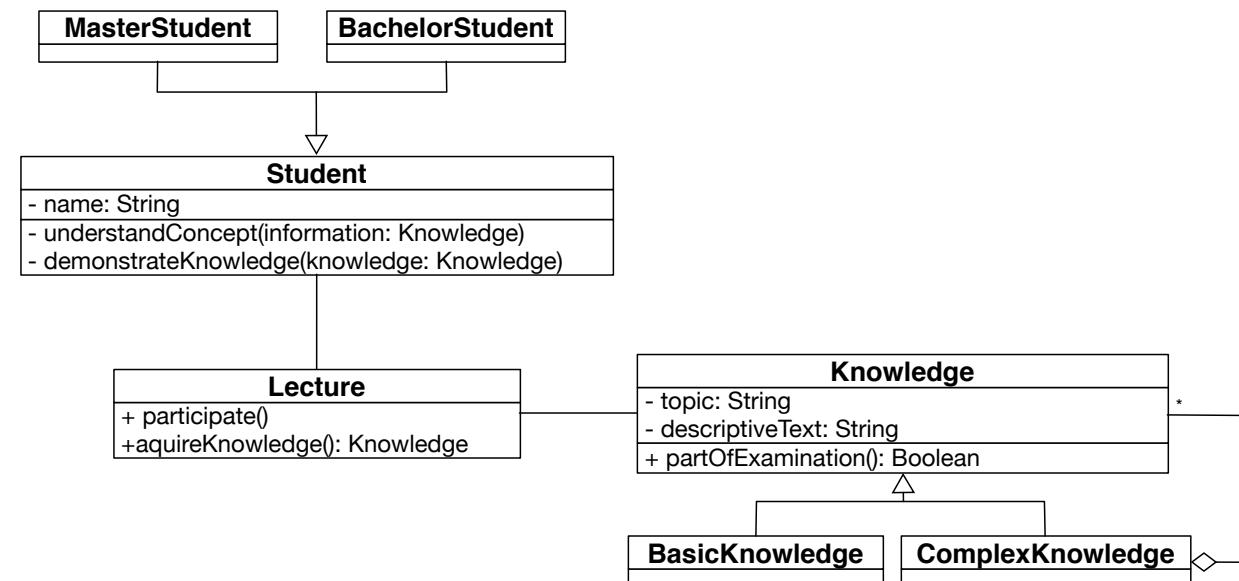
- A specific instance of a class ("Joe, the passenger who is purchasing a ticket from the ticket distributor")

Joe:User

In-Class Quiz 03b

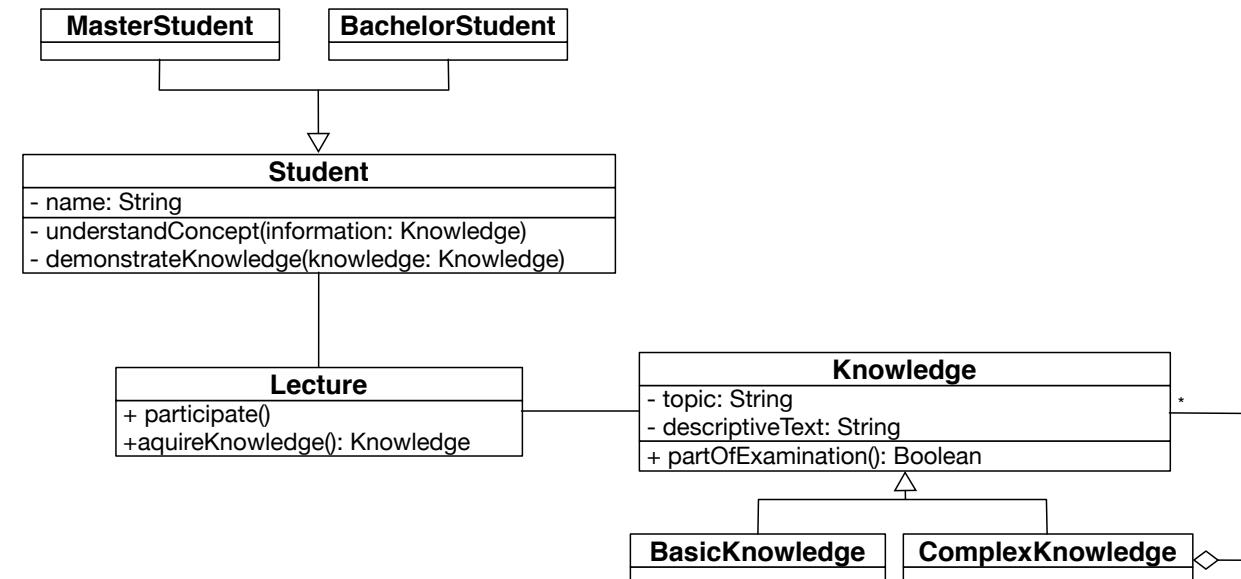
Open ArTEMiS and answer the following question:

- How many objects exist, when this class diagram is instantiated for a Master Student taking a basic knowledge Lecture?

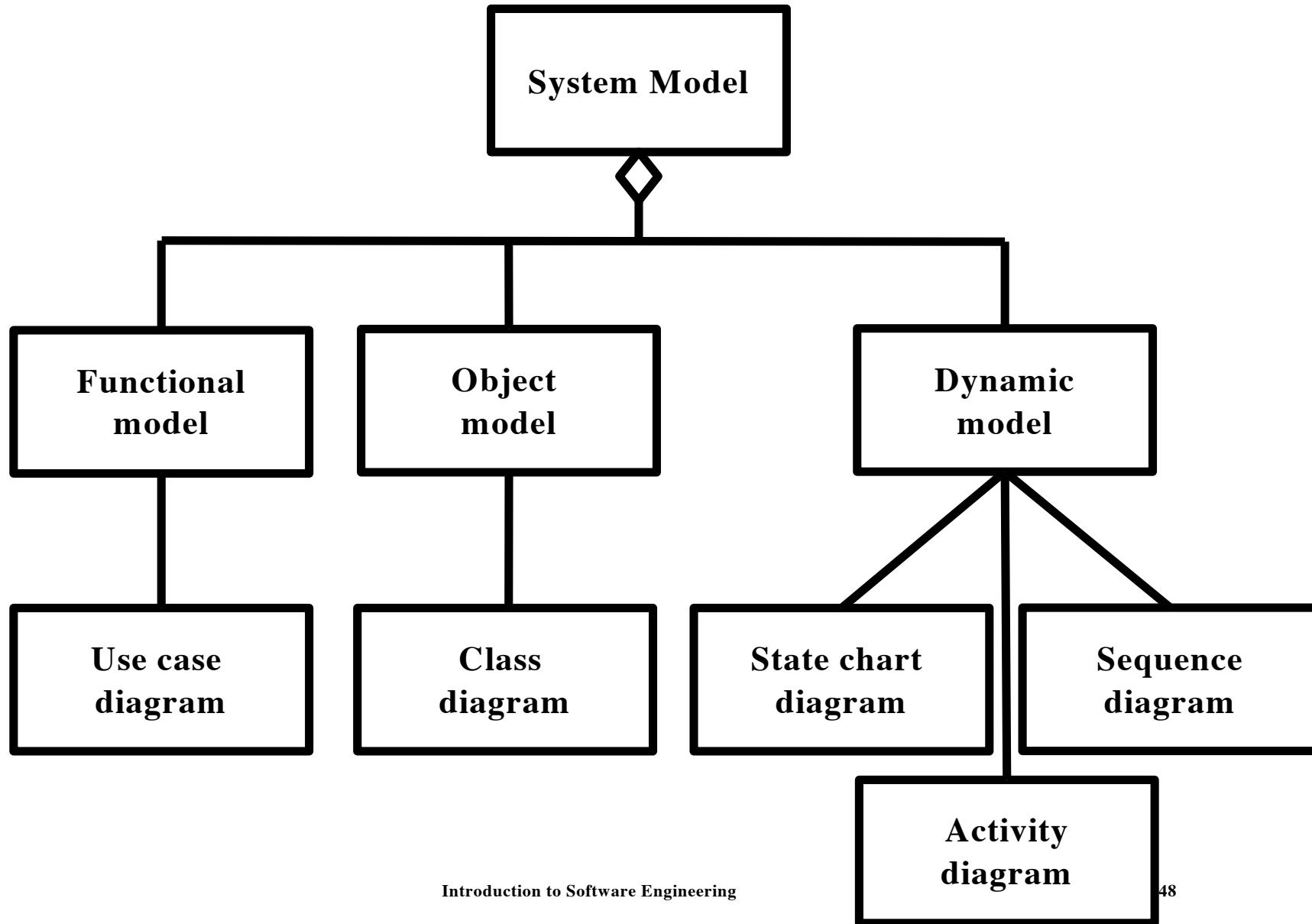


In-Class Quiz Answer

- Answer: 3
- Inheritance Hierarchies are collapsed into a single Instance!



Modeling a System Model with UML



Purpose of Modeling: Why all these models?

- Functional model:
 - Describes the functionality of the system
- Object model:
 - Describes the structure of the system
- Dynamic model:
 - Describes the dynamic behavior of the system
- The center of the modeling universe is the object model
- We have two supplier industries:
 - functional model and dynamic model.

Purpose of Object Modeling

- Purpose:
 - Identify the structure of the system
 - This is the center of the modeling activities
 - Without knowledge of the systems functions and behavior, we usually identify the wrong object (Example: Chief, Chef, Eskimo, Face)

Purpose of Functional Modeling

- Review:
 - Definition of a **functional model**: Describes the functionality of the system.
 - The functional model is described with scenarios, use cases
- Purpose:
 - Identification of functional requirements
 - Delivery of new operations (methods) for the object model.

Outline of the Lecture

- Odds and Ends
 - Ferienakademie
- A deeper look into use case diagrams
- A deeper look into class diagrams
- Extending UML: Stereotypes

→ Dynamic Modeling

- Requirements Engineering
- Types of Requirements
- Techniques to describe Requirements
- Requirements Review

Purpose of Dynamic Modeling

- Review:
 - 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
 - **Sequence diagrams**: For the interaction between classes
- Purpose:
 - *Identification of new classes* for the object model and identification of supply operations for the classes.

Identification of Classes and Operations from the Dynamic Model

- 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)
- Two additional sources for identifying classes from dynamic models:
 - Activities in state chart diagrams are candidates for public operations in classes
 - Activity lines in sequence diagrams are candidates for objects.

How do we detect Operations?

- We look for objects, which are interacting and extract their “protocol”
- 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*.

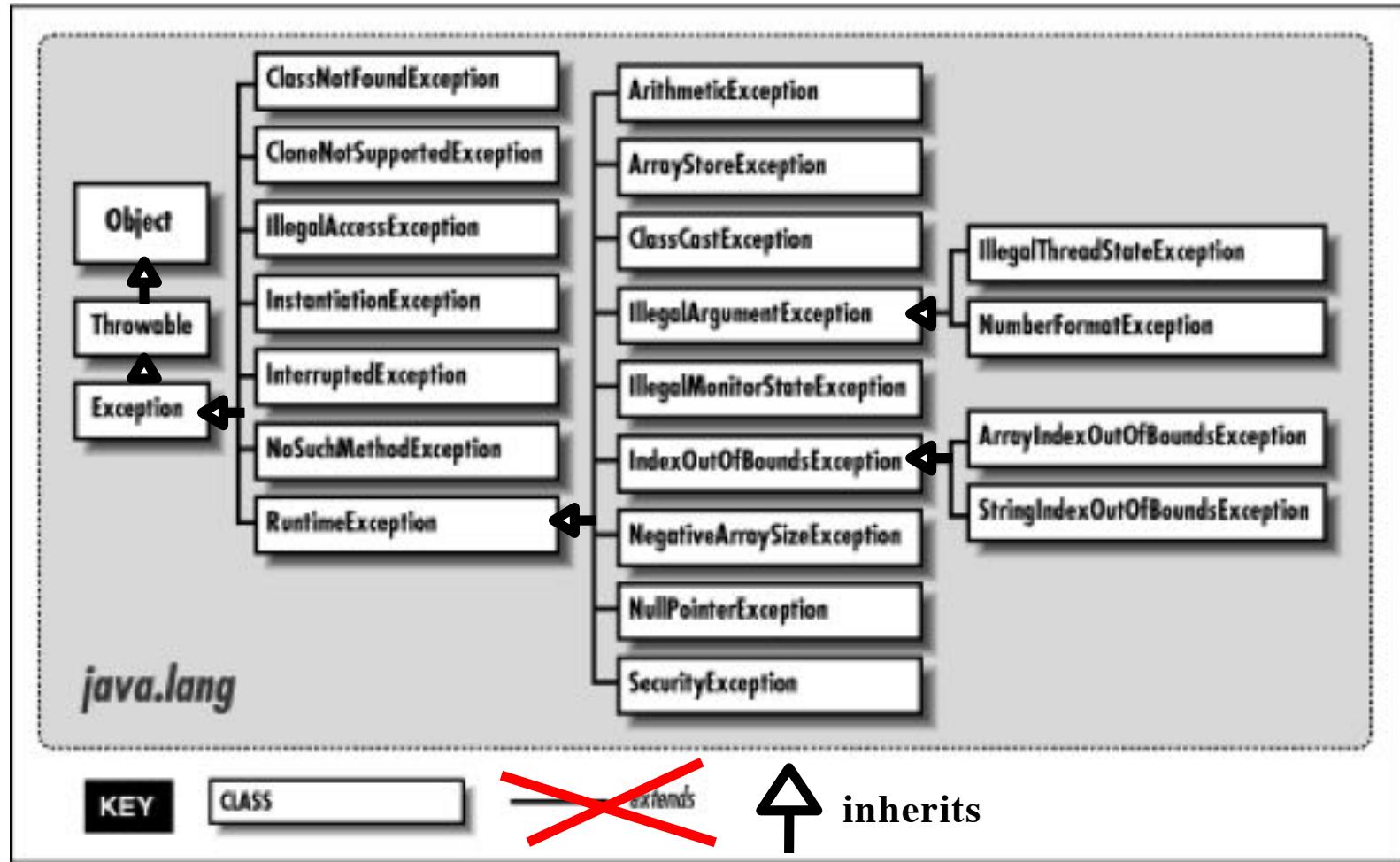
How do we detect Operations?

- We look for objects, who are interacting and extract their “protocol”
- 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?

What is an Event?

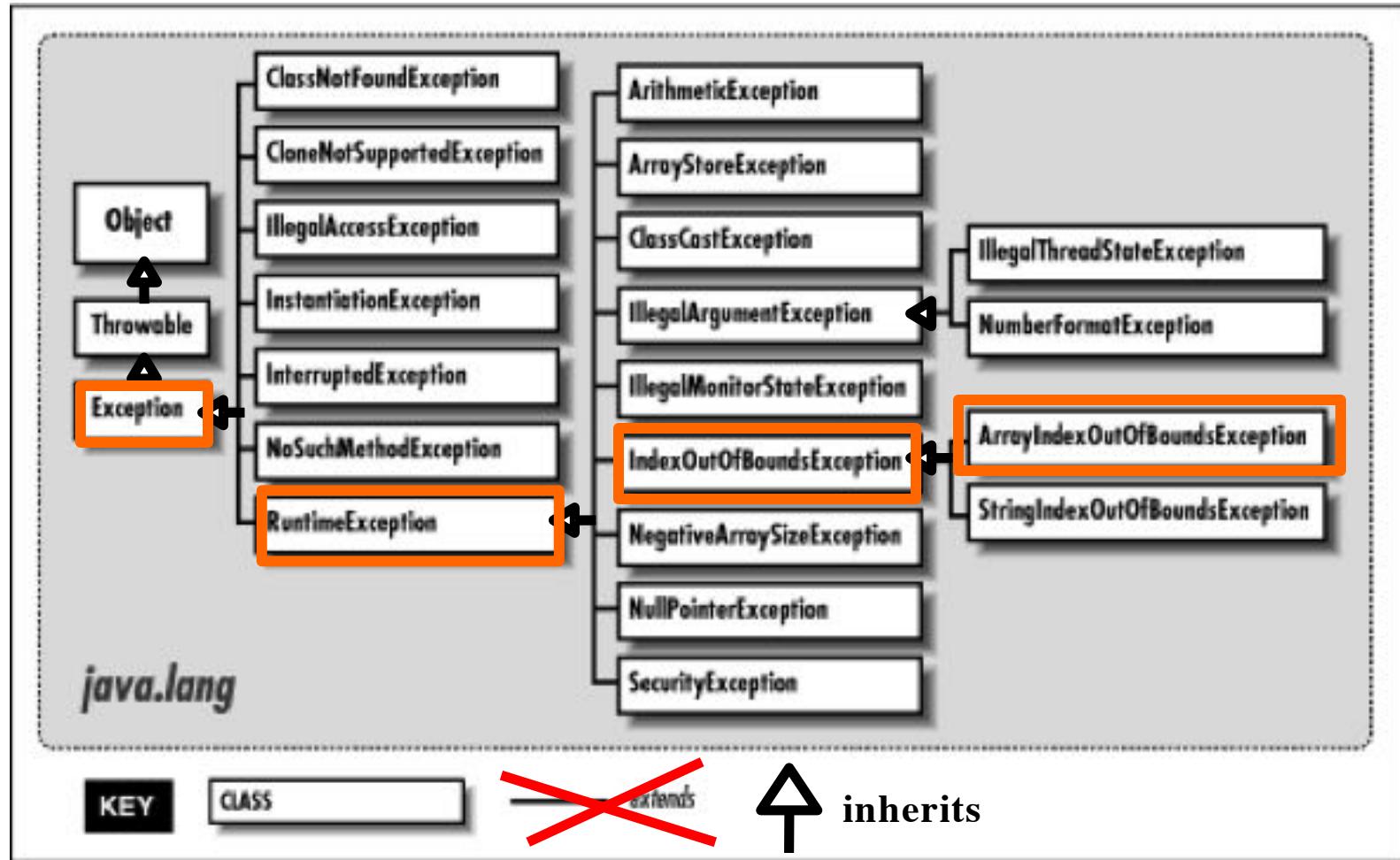
- Something that happens at a point in time
- An event sends information from one object to another
 - Example: A message in a sequence diagram
- Events can have associations with each other:
 - Causally related:
 - An event happens always before another event
 - An event happens always after another event
 - Causally unrelated:
 - Events that happen concurrently
- Events can also be grouped in event classes with a hierarchical structure => Event taxonomy
 - Example: Java's exception hierarchy.

Event Taxonomy for Java's Exceptions



Source: Mark Grand, Java Language Reference, 2nd edition, O'Reilly, 1997

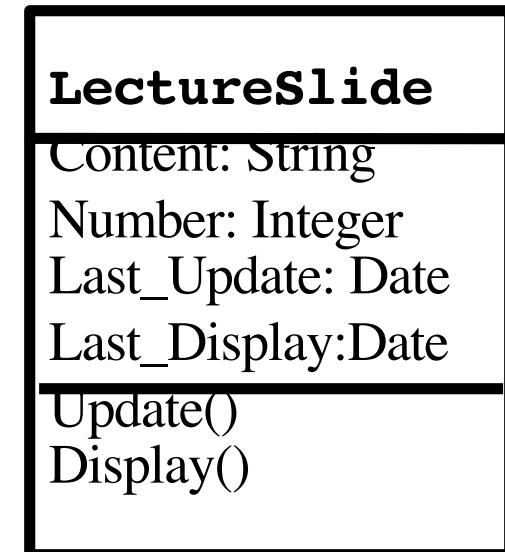
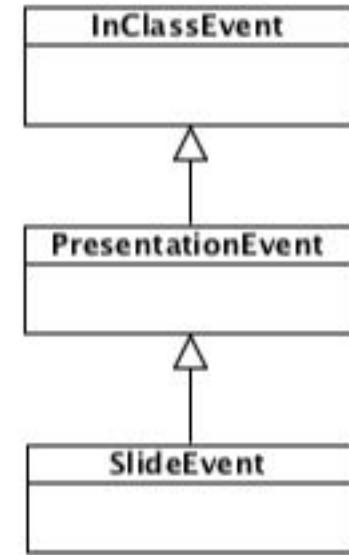
Event Taxonomy for Java's Exceptions



Source: Mark Grand, Java Language Reference, 2nd edition, O'Reilly, 1997

The Term ‘Event’ is often used in two Ways

1. The instance of an event class
 - “Update Slide”
An instance of the class **SlideEvent**, which is a subclass of **PresentationEvent**, which is a subclass of **InClassEvent**
2. The values of the attributes of an event class
 - “Lecture slide update at 8:00 AM”
 - “Lecture slide 69 display at 9:15 PM”



How do we detect Operations?

- We look for objects, who are interacting and extract their “protocol”
- 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.

How do we detect Operations? (ctd)

- We look for objects, who are interacting and extract their “protocol”
- 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**.

Finding Participating Objects

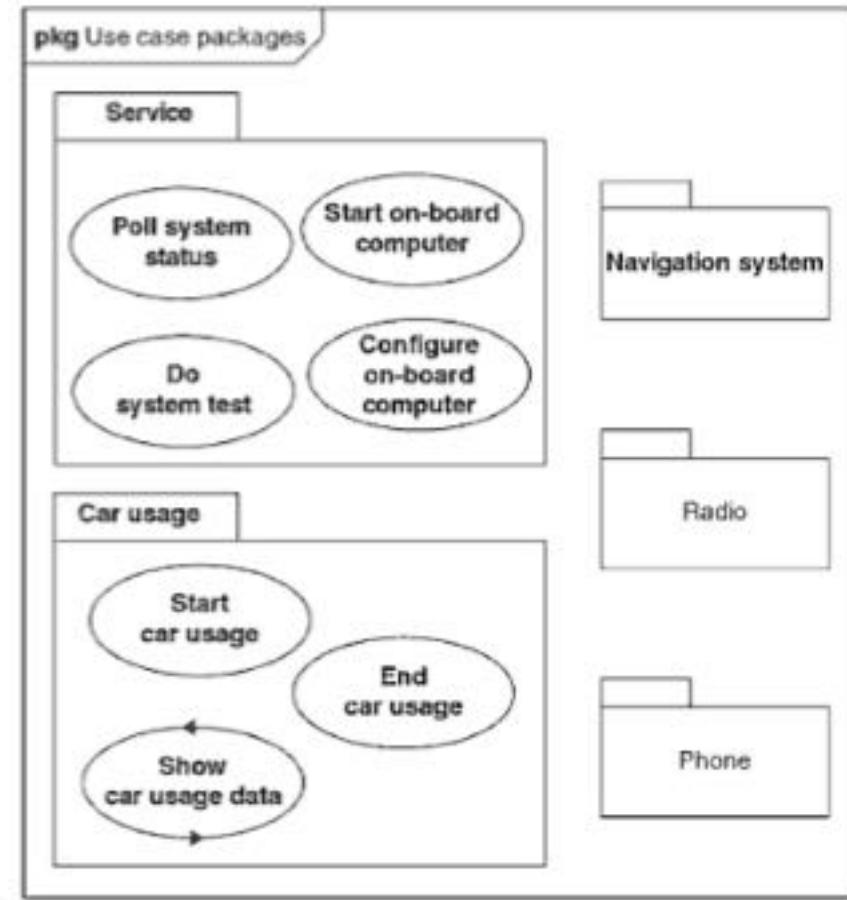
- Start with the flow of events
- An event always has a sender and a receiver
 - That is, we can think of it as a message between two objects
- A good heuristic for finding participating objects:
 - Find the sender and receiver for each event
 - ⇒These are often participating objects in the use case

This an iterative and incremental activity.

Organization of Models

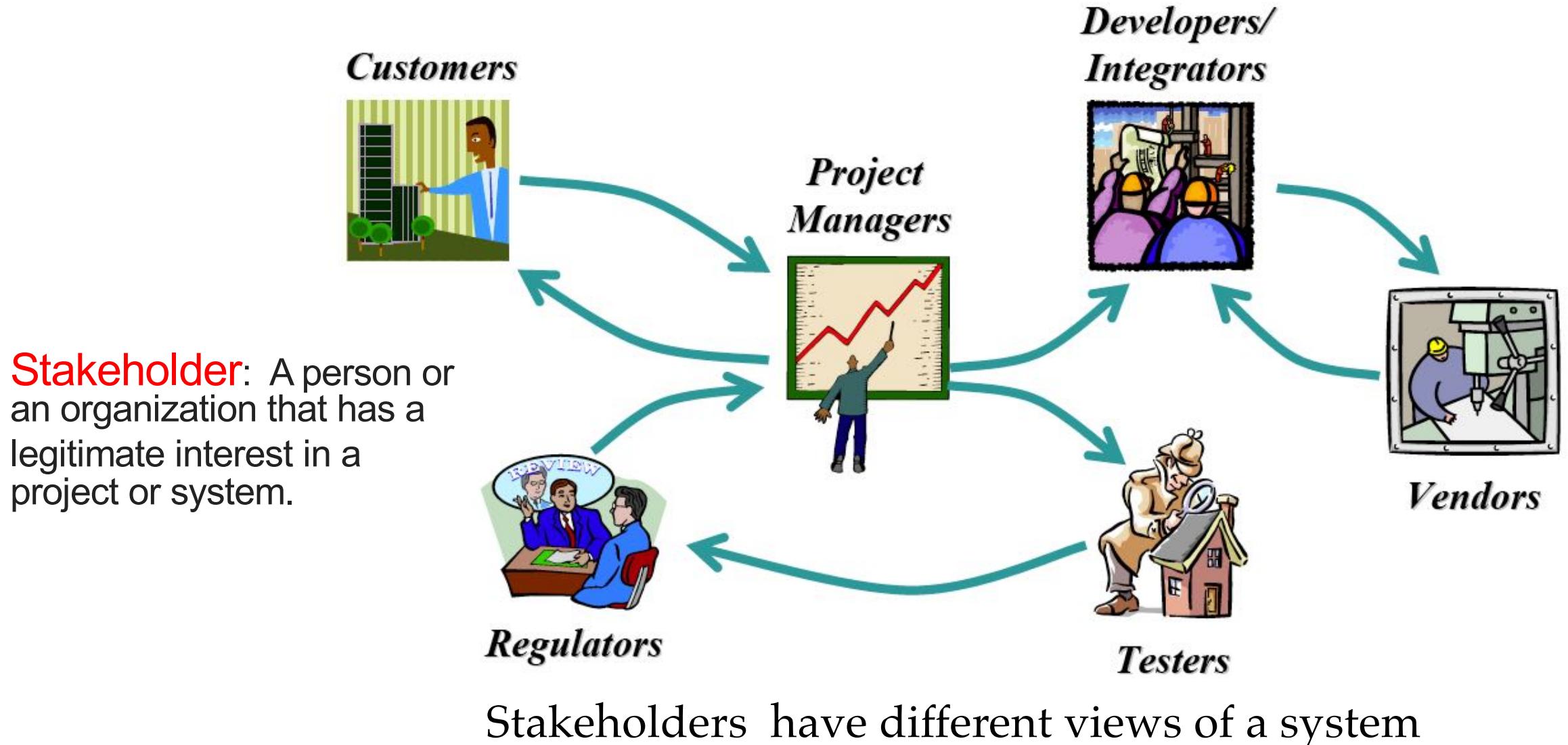
- The model elements can be organized in multiple ways
 - By system hierarchy (Enterprise, Systems, components, classes)
 - By system model
 - Functional model: Use cases
 - Object model: Classes, subsystems
 - Dynamic model: Events
 - By domain (application domain, solution domain)
 - By stakeholders
 - By view points
- UML Packages can be used for the organization
 - They allow to group model elements into a name space

Organizing a Model by Use Cases



Tim Weilkiens, Systems engineering with SysML/UML: modeling, analysis, design

Organizing a Model by Stakeholders



Outline of the Lecture

- Odds and Ends
 - Ferienakademie
- A deeper look into use case diagrams
- A deeper look into class diagrams
- Extending UML: Stereotypes
- Dynamic Modeling

→ Requirements Engineering

- Types of Requirements
- Techniques to describe Requirements
- Requirements Review

30 Minute Break



Dialog between a Requirements Engineer and his wife

She:

“Honey, we’re out of bread, could you please go to the store and get a loaf of bread. And if they have eggs, bring six.”

He:

“Sure, I will.”

He goes and comes back with six loaves of bread.

She:

“Why on earth did you buy six loaves of bread?!?”

He:

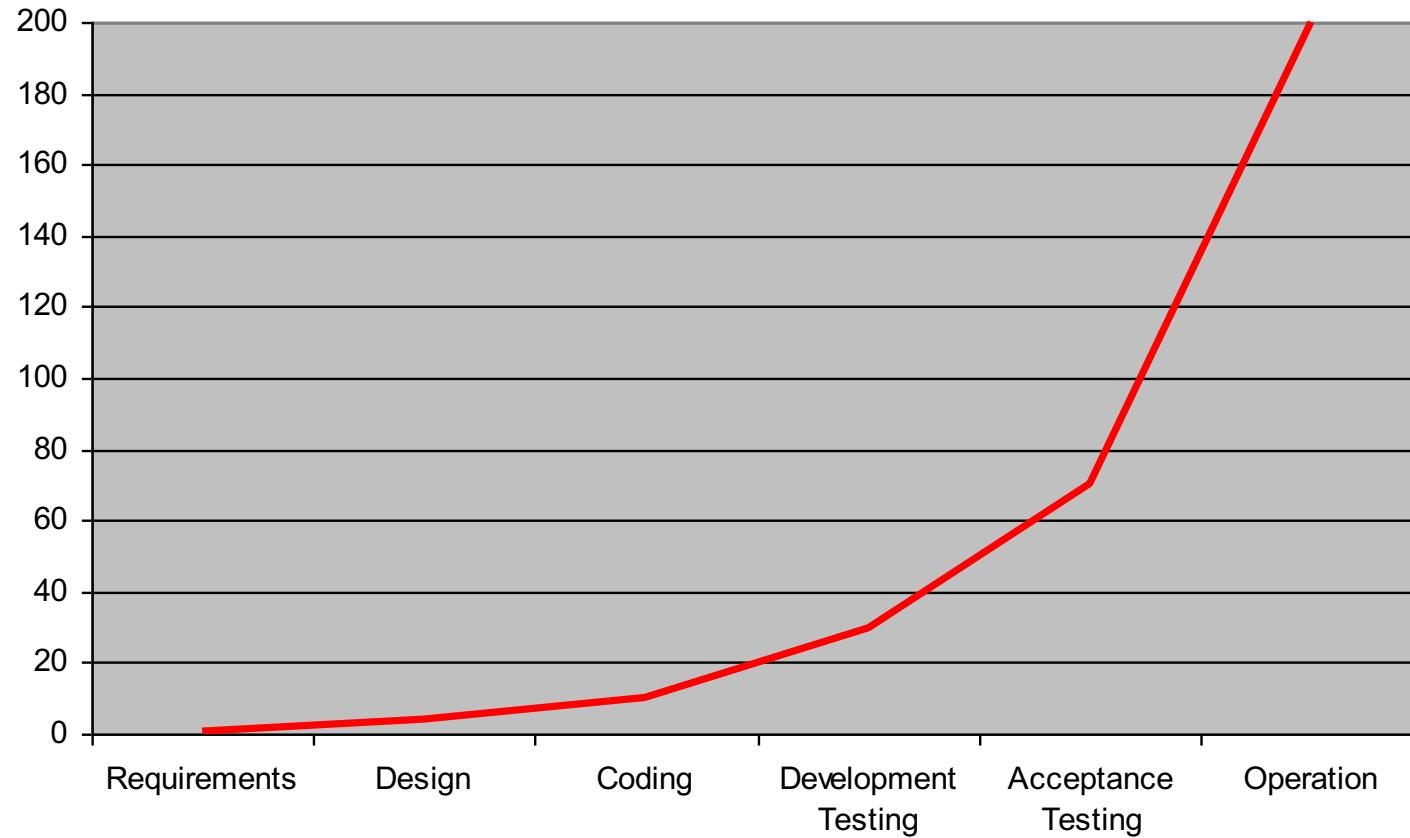
“They had eggs...”

Requirements Engineering

- Requirements Elicitation
 - Definition of the system in terms understood by a customer or user
 - Result: “Requirements specification”
- Analysis
 - Definition of the system in terms understood by a developer
 - Result: “Analysis model” (also called “Technical specification” (German: “Lastenheft”))
- Requirements Engineering
 - Combination of the two activities, requirements elicitation and analysis. *An activity that defines the requirements of the system under construction.*

Relative Cost of Requirements Fixes

The earlier a defect is found, the cheaper it is to fix

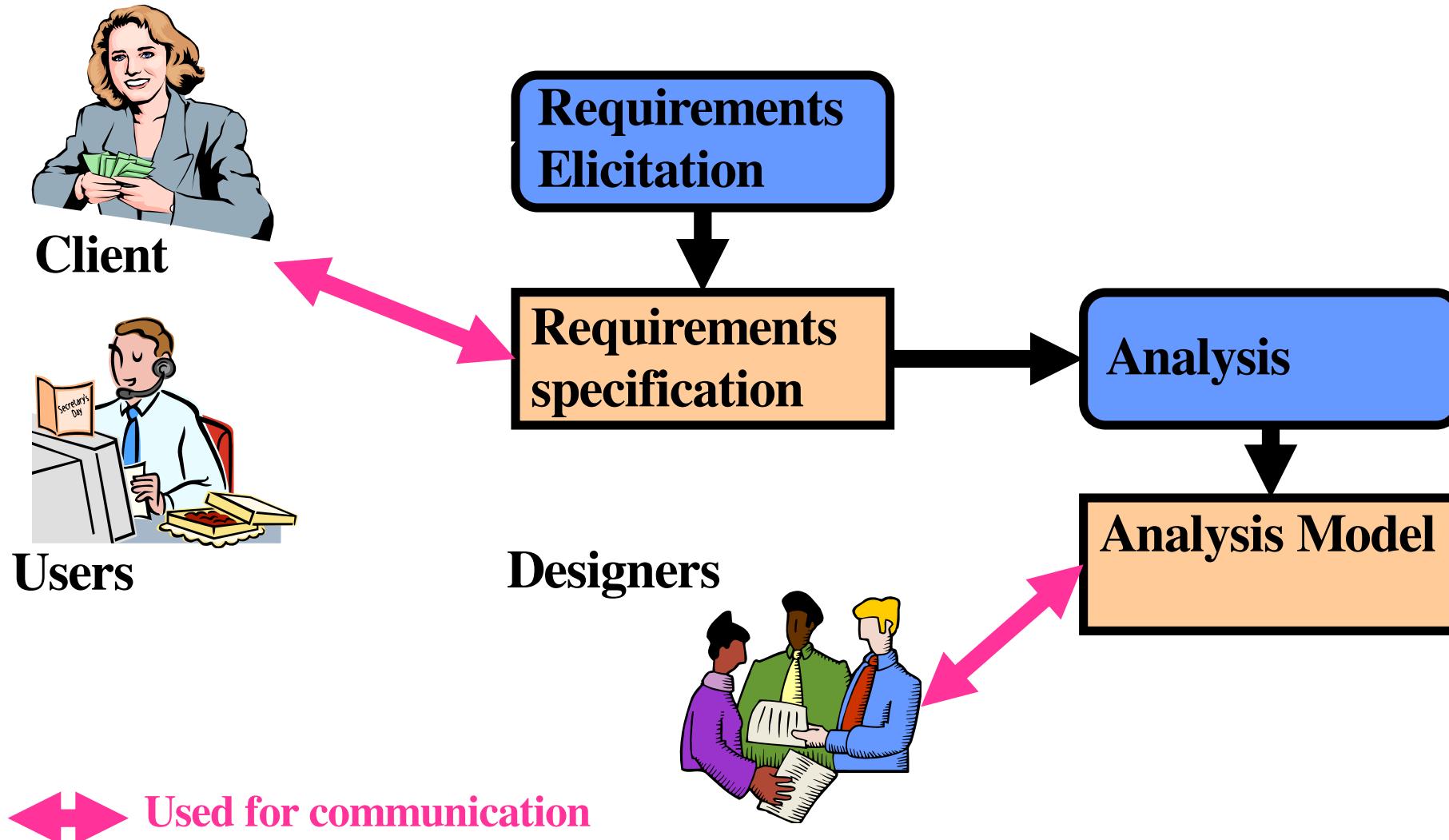


[Boehm 1981]

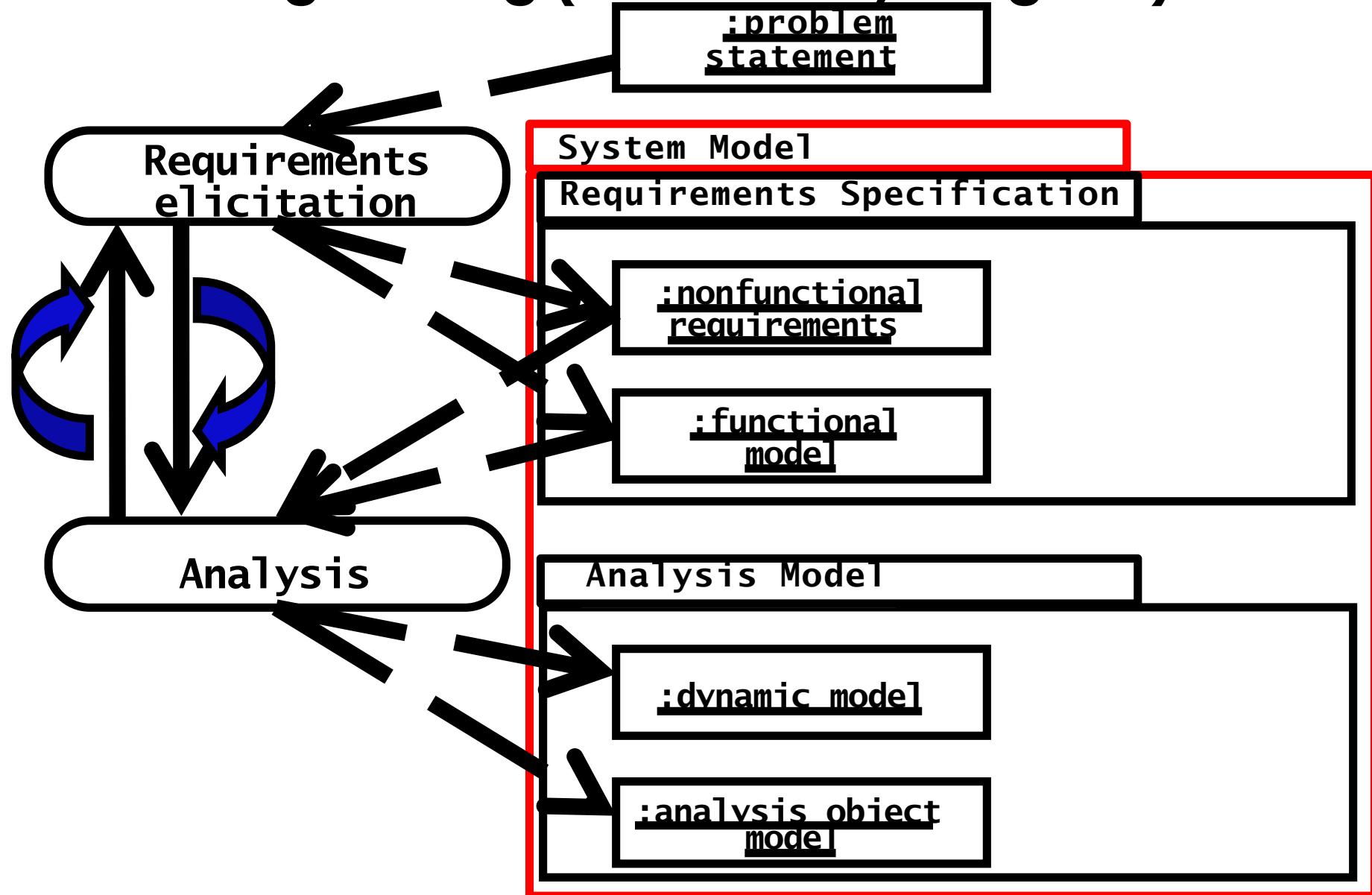
Requirements Elicitation is a Development Activity

- Determining the requirements of the system specified by the customer and/or user
- Another formulation:
 - “From the problem statement to the requirements specification”
- Currently a very informal process full with problems.

Requirements Engineering: Informal Model



Requirements Engineering (UML Activity Diagram)



Requirements Specification vs Analysis Model

Both are models focusing on the requirements from the user's view of the system

- The **requirements specification** uses natural language (derived from the problem statement)
 - Verb-Noun Analysis (Abbott's Technique)
- The **analysis model** uses a (semi-)formal language
 - Examples of semiformal and formal languages
 - Graphical Languages: SA/SD, OMT, **UML**, SysML
 - Mathematical Specification Languages
 - VDM (Vienna Definition Method)
 - Z (based on Zermelo–Fraenkel set theory).

Requirements

- **Requirement:** *A feature that the system must have or a constraint it must satisfy to be accepted by the client*
 - Requirements describe the **user's view** of the system
 - Identify the **what** of the system, not the **how**

- **Part of Requirements**
 - **Functionality**
 - **User interaction**
 - **Error handling**
 - **Environmental conditions (interfaces)**

- **Not part of Requirements**
 - **System Design**
 - **Implementation technology**
 - **Development methodology**

Requirements Elicitation and Analysis are Examples of Software Development Activities

Typical Software Development Activities

Requirements Elicitation

What is the problem?

Application Domain

Analysis

System Design

What is the solution?

Object Design

What are the best data structures and algorithms for the solution?

Implementation

How is the solution constructed?

Solution Domain

Testing

Is the problem solved?

Delivery

Can the customer use the solution?

Maintenance/Evolution

Are enhancements needed?

Application Domain

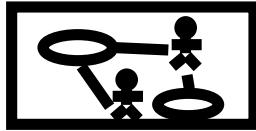
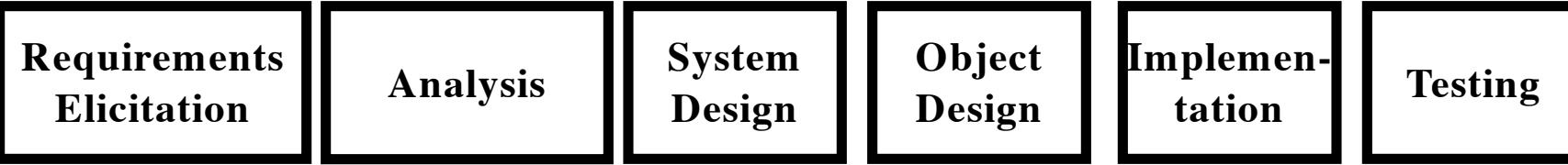
Software Development Activities in EIST

...and their Models

Software Development Activities in EIST

...and their Models

Chapter 4



Use Case
Model

Software Development Activities in EIST ...and their Models

Chapter 4

Chapter 5

Requirements
Elicitation

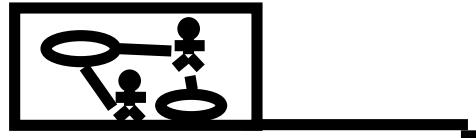
Analysis

System
Design

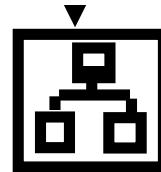
Object
Design

Implemen-
tation

Testing



Expressed in
terms of



Use Case
Model

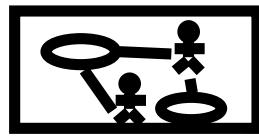
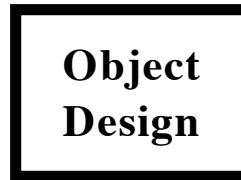
Application
Domain
Objects

Software Development Activities in EIST ...and their Models

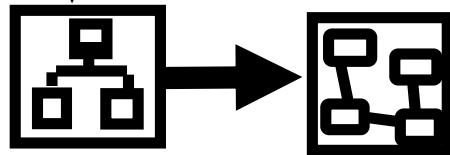
Chapter 4

Chapter 5

Chapter 6 to 9



Expressed in terms of Structured by



Use Case Model

Application Domain Objects

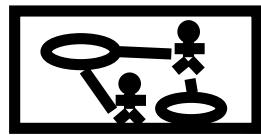
Sub-systems

Software Development Activities in EIST ...and their Models

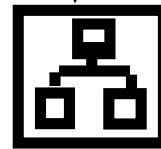
Chapter 4

Chapter 5

Chapter 6 to 9

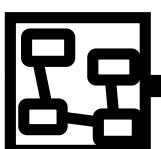


Expressed in terms of



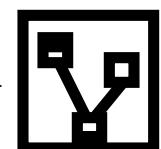
Use Case Model

Structured by



Application Domain Objects

Realized by



Sub-systems

Solution Domain Objects

Software Development Activities in EIST ...and their Models

Chapter 4

Chapter 5

Chapter 6 to 9

Chapter 10

Requirements
Elicitation

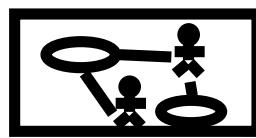
Analysis

System
Design

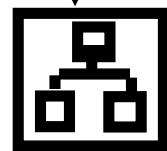
Object
Design

Implemen-
tation

Testing

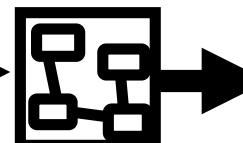


Expressed in
terms of



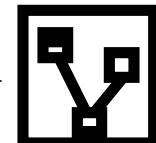
Use Case
Model

Structured
by



Application
Domain
Objects

Realized by



Sub-
systems

Implemented by



Solution
Domain
Objects

Source
Code

Software Development Activities in EIST ...and their Models

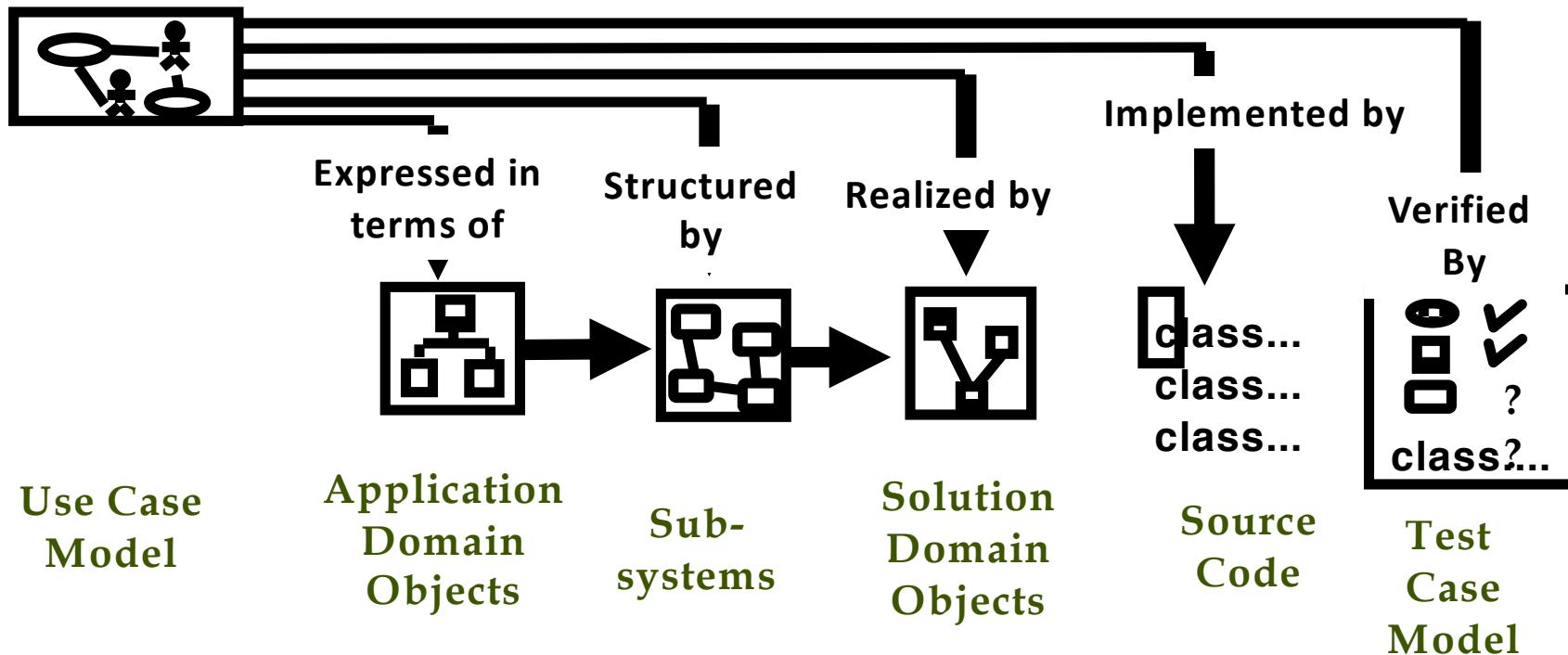
Chapter 4

Chapter 5

Chapter 6 to 9

Chapter 10

Chapter 11



Requirements Elicitation: Difficulties

Questions that need to be answered:

1. How can we identify the **purpose** of a system?
 - What are the requirements, what are the constraints?
2. How can we identify the system boundary?
 - What is **inside**, what is **outside** the system?
 - Defining the system boundary is often difficult.

Outline of the Lecture

- Odds and Ends
 - Ferienakademie
 - A deeper look into use case diagrams
 - A deeper look into class diagrams
 - Extending UML: Stereotypes
 - Dynamic Modeling
 - Requirements Engineering
- Types of Requirements
- Techniques to describe Requirements
 - Requirements Review

Types of Requirements Elicitation

- **Greenfield Engineering**
 - Development from scratch, no prior system exists
 - Requirements are extracted from client and user
 - Triggered by user needs
- **Re-engineering**
 - Re-design or re-implementation of an existing system
 - Requirements triggered by new technology
- **Interface Engineering**
 - Provide services of existing system in new environment
 - Requirements triggered by technology or new market needs
- Each of these requirements elicitation types should start with the Problem Statement.

Types of Requirements

- Functionality
 - What is the software supposed to do?
- External interfaces (->Actors)
 - Interaction with people, hardware, other software

**Functional Requirements:
Use case model**

Functionality

- Includes
 - Relationship of **outputs** to **inputs**
 - Response to **abnormal situations**
 - **Exact sequence** of operations
 - **Validity checks** on the inputs
- Should be phrased as an action or a verb
 - Withdraw money
 - Deposit money
 - Transfer money
 - Load cash card
 -

Types of Requirements

- Functionality
 - What is the software supposed to do?
- External interfaces (->Actors)
 - Interaction with people, hardware, other software

✓ Functional Requirements:
Use case model

- Usability
- Reliability
- Performance
- Supportability
- Constraints (Pseudo Requirements)
 - Required standards, operating environment, etc.

Nonfunctional Requirements
(also called Quality Requirements)

Non functional Requirements

Also called Quality Requirements

- **Usability**
- **Reliability**
 - Robustness
 - Safety

→ Performance

- Response time
- Throughput
- Availability
- Accuracy
- Supportability
 - Adaptability
 - Maintainability, Portability

Nonfunctional requirements mnemonic: URPS

Performance requirements

- Number of simultaneous users supported
- Amount of information handled
- Number of transactions processed within certain time periods (average and peak workload)
 - Example: 95% of the transactions shall be processed in less than 1 second

Nonfunctional Requirements

- Usability
- Reliability
 - Robustness
 - Safety
- Performance
 - Response time
 - Throughput
 - Availability
 - Accuracy
- Supportability
 - Adaptability
 - Maintainability, Portability

Examples

Nonfunctional Requirements Definitions (1)

- **Usability**

- The ease with which actors can perform a system function
- Usability is one of the most frequently misused requirement terms.
 - **Example:** “The system is easy to use”
 - Usability must be *measurable*, otherwise it is *marketing*
 - **Example:** We measure the time and number of steps needed to purchase a product with a web browser



- **Robustness:** The ability of a system to maintain a function...

- ...when the user enters a wrong input
- ...when there are changes in the environment
 - **Example:** The system can tolerate temperatures up to 90C.



Nonfunctional Requirements Definitions (2)

- **Availability:**

- The ratio of the expected uptime of a system to the sum of the expected up and down time
- **Example:** The availability of the system is 99.7%
 - That means the downtime is at most 30 minutes per week (10050/10080)



- **Adaptability:**

- The ability of a system to adapt *itself* to changed circumstances
- An adaptive system is an open system that is able to change its behavior when the environment changes



- **Maintainability:**

- The ease with which a system can be modified *by a developer* to correct defects, deal with new requirements or cope with a changed environment.



Nonfunctional Requirements Definitions (3)

- **Safety:** Protection against unwanted incidents
 - Random incidents are usually unwanted, happening as a result of one or more coincidences.
- **Security:** Protection against intended incidents
 - **Examples:** Attack at the target, intrusion, deliberate and planned inclusion of viruses and trojan horses.



Example: Nonfunctional Requirements for an ATM Machine

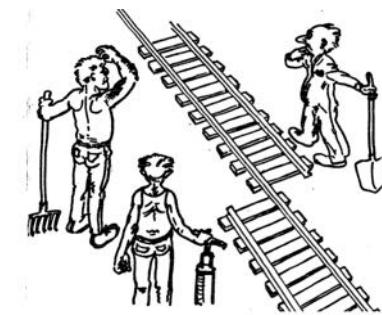
- **Usability**
 - User interaction shall be done via a touch-screen
 - The touch-screen shall be non-reflective
 - Text shall appear in letters at least 1cm high
- **Safety**
 - System shall under no circumstances leak PIN numbers or account information to unauthorized users
- **Performance**
 - Each individual transaction shall take less than 10s.

Constraints (Pseudo Requirements)

- Compliance to Standards
 - Report format, audit tracing
- Implementation requirements
 - Tools, programming languages
 - The development technology and methodology should not be constrained by the client. Fight for it!
- Operations requirements
 - Administration and management of the system
 - Example for the ATM machine: Remote system updates must be possible
- Legal requirements
 - Licensing, regulation, certification

Constraint Definitions (Pseudo Requirements)

- **Packaging requirements**
 - Constraints on the actual delivery of the system
 - Example: The software must be delivered on floppy disks
- **Interface requirements**
 - Constraints imposed by external systems
- **Legal requirements**
 - The software system must comply with federal law
 - Example: Government software must comply with Section 508 of the Rehabilitation Act of 1973, to make it accessible for people with disabilities.



Model Correctness: Model Validation vs Model Verification

- **Verification** is an equivalence check between two models, one of them generated from the other one
- **Validation** is the comparison of the model with reality (with the client)
 - Validation is a critical step in any software development process.

Example of a Pseudo Requirement

- Bumpers should support older versions than Java 9
- What is the reason?
- By the way: Do not use the word “pseudo requirement” when talking to a customer, use the word “constraint”.

Outline of the Lecture

- Odds and Ends
 - Ferienakademie
- A deeper look into use case diagrams
- A deeper look into class diagrams
- Extending UML: Stereotypes
- Dynamic Modeling
- Requirements Engineering
- Types of Requirements
- Techniques to describe Requirements
- Requirements Review

Techniques to Describe Requirements

- Goal: Bridging the conceptual gap between end users and developers
- Techniques:
 - **Scenario:** Describes the use of the system as a series of interactions between a specific end user and the system
 - **Use case:** A concept that describes a set of scenarios of a generic end user, called actor, interacting with the system
 - A scenario describes a single instance of a use case
 - **User Story:** Describes a functional requirement from the perspective of an end user.

Scenarios

- **Scenario:** A concrete, focused, informal description of a single feature of the system used by a single actor
 - Central is the textual description of the usage of a system. The description is written from an end user's point of view
 - A scenario can also include video, pictures (story boards)
 - It may also contain details about the work place, social situations and resource constraints
- **Scenario-based Design:** Scenarios are used as the basis for the design of the hypothetical interaction of the end user with a new system.

Use of Scenarios in Development Activities 4 26 2018

Scenarios can be used in many activities during the software lifecycle

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

Types of Scenarios (1)

- **As-is scenario:**
 - Describes a current situation. The scenario describes the usage of an existing system
 - **Example:** Bernd and Yang play correspondence chess via post cards
 - Commonly used in re-engineering projects
- **Visionary scenario:**
 - Describes a future system
 - **Example:** Jan and Enrico play chess using a brainwave recognition machine
 - Used in all types of projects (greenfield & interface engineering and reengineering)
 - Visionary scenarios are usually not formulated by the user or developer alone.



Types of Scenarios (2)

- **Evaluation scenario:**
 - Description of a user task against which the system is to be evaluated
 - **Example:** The system must be demonstrated with two users (one novice, one expert) playing in a wrestling tournament
- **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 with augmented reality glasses.



Types of Scenarios

As-is scenario

- Used in describing a current situation
- Reengineering projects
- The user describes the system

Visionary scenario

- Describes a future system
- Greenfield engineering & reengineering projects
- Can often not be done by the customer/user or developer alone

Evaluation scenario

- User tasks against which the system will be evaluated
- Demos & Acceptance Tests

Training scenario

- Guide a novice user through a system
- System Delivery

How do you find scenarios?

- Don't expect the client to be verbose if the system does not exist
 - Clients understand the application domain (problem domain), not the solution domain
- Don't wait for information if the system exists
 - Don't think: "What is obvious, does not need to be said"
- Engage in a dialectic approach
 - Help the client to formulate the requirements
 - The client then helps you to understand the requirements
 - The requirements often evolve while these scenarios are being formulated
 - Usually the problem statement is a good start.

Problem Statement: ATM Machine

Problem Statement

- A bank customer specifies a account and provides credentials to the ATM machine proving that he is authorized to access the bank account
- The bank customer can specify the amount of money he wishes to withdraw or to deposit
- The bank checks if the amount is consistent with the rules of the bank and the state of the bank customer's account.
- If that is the case, the bank takes the deposit or the bank customer receives the money in cash.

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

- Joe logs into the ATM machine
- Then he accesses his checking account
- Joe selects withdrawal from checking account
- He selects 50.00 Dollars
- He receives \$ 50.00 in five 10-Dollar bills.

Observations about the Withdrawal

- It is a concrete scenario with a linear event flow:
- It describes a single instance of withdrawing 50 Dollar
- It does not describe all possible situations in which the ATM machine can be accessed.

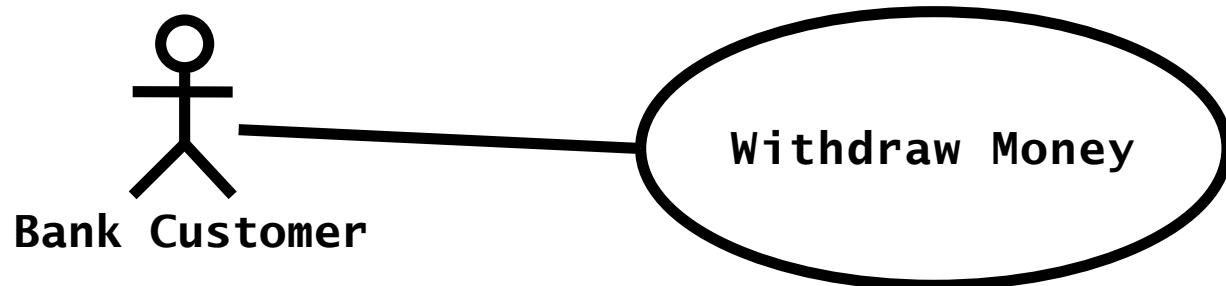
After the scenario is formulated

- Find functions in the scenario where an actor interacts with the system
 - Example from the Withdrawal scenario:
 - Joe selects withdrawal from checking account
 - Here “withdrawal from checking account” is a candidate for a use case
 - Another candidate: “selects withdrawal”
- 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 constraints.

Use Case Modeling Review (See Lecture 2)

- A use case is a flow of events in the system, including interaction with actors
- A textual use case description has 7 parts (we have added Exceptions here)
- Graphical notation: An oval with the name of the use case

1. Name
2. Participating actors
3. Entry conditions
4. Flow of events
5. Exit conditions
6. Exceptions
7. Special requirements



Use Case Model: The set of all actors and use cases specifying the complete functionality of the system

Requirements Review

- After we are done with requirements elicitation and with analysis, what is the next step?
 - **We can write a requirements analysis document**
 - We can validate the requirements with the client and the user

Requirements Analysis Document (*)

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 Class diagrams
 - 3.5.4 Dynamic models
 - 3.5.5 User interface
4. Glossary

(*(Requirements Analysis Template from the Text book (page 200)

Requirements Review

- After we are done with requirements elicitation and with analysis, what is the next step?
 - We write the requirements analysis document
 - **We can validate the requirements with the client and the user**

Requirements Validation: 6 Criteria

- **Correctness:**
 - The requirements represent the client's view
- **Clarity:**
 - The requirements can only be interpreted in one way
- **Completeness:**
 - All possible ways of using the system are described
- **Consistency:**
 - There are no requirements that contradict each other
- **Realism:**
 - The requirements can be implemented and delivered
- **Traceability:**
 - System components and behavior can be traced to the functional requirements.

Checklist for a Requirements Review

→ Is the model **correct**?

- The model represents the client's view of the system
 - The model should not contain unintended features
 - Difference between verification and validation
- Is the model **clear (unambiguous)**?
 - The model describes one system, not many
- Is the model **complete**?
 - Every scenario is described in the model
 - Are classes or associations missing?
- Is the model **consistent**?
 - Dynamic, functional and object model use consistent naming / no spelling errors
- Is the model **realistic**?
 - The model can be implemented

Example of an Incorrect 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. Why?**
 - The driver door was not treated as a door in the source code!



What should not be in the Requirements?

- A description of the system structure, use of a specific implementation technology
 - The development methodology
 - A rational design process: How and why to fake it (Parnas, 1986)
 - A description of the development environment
 - A specific implementation language
 - Reusability
-
- It is desirable that none of these above are constrained by the client.

Tools for Requirements Management

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

Outline of the Lecture

- Odds and Ends
 - Ferienakademie
- A deeper look into use case diagrams
- A deeper look into class diagrams
- Extending UML: Stereotypes
- Dynamic Modeling
- Requirements Engineering
- Types of Requirements
- Techniques to describe Requirements
- Requirements Review

Summary

- We gave you a deeper look into class diagrams and use cases
 - Aggregation vs. composition (classes)
 - Extends vs. includes (use cases)
- UML is an extensible language: predefined types and Stereotypes
- We introduced requirements elicitation and analysis as software development activities
- We distinguish between functional requirements and nonfunctional requirements (URPS and constraints)
- We introduced scenario-based design
- We defined 6 criteria for requirements validation: correctness, clarity, completeness, consistency, realism, traceability

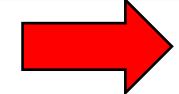
Additional Readings

- Barry Boehm
 - Software Engineering Economics. Englewood Cliffs, Prentice-Hall, 1981
- David Parnas
 - A rational design process: How and why to fake it, IEEE Trans. on Software Engineering, Vol 12(2), February 1986
- 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
- User Stories
 - Mike Cohn, User Stories Applied: For Agile Software Development, Addison-Wesley, 2004.

Morning Quiz

- Quiz 04
 - Starting Time: 8:00
 - End Time: 8:10
- To participate in the quiz, use ArTEMiS
- The Lecture starts at 8:10

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	
Good Morning Quiz 04		 

Waiting for Start

● Connected

Submit

Only click on Submit when you have finished all answers!

Object-Oriented Software Engineering

Using UML, Patterns, and Java



System Design I: System Decomposition

Bernd Bruegge

Chair for Applied Software Engineering
Technische Universität München

3 May 2018

Roadmap for Today's Lecture

- **Context and Assumptions**

- We completed Chapter 1-2 and 4-5 in the text book (We skipped Chapter 3)
- We did not quite finish Lecture 3 Requirements Elicitation

- **Content of this lecture**

- We finish Lecture 3 Requirements Elicitation (UML notations for dynamic modeling)
- We start with system design (Chapter 6 in the text book)

- **Objective:** At the end of this lecture you should be able to understand

- how a **UML sequence diagram** models the dynamic behavior of an event flow
- how a **UML state chart diagram** models a *single* object with interesting dynamic behavior
- how an **UML activity diagram** models the dynamic behavior *between* objects
- how a **UML communication diagram** visualizes the event flow in a class diagram
- the 8 most important issues in system design
- the difference between non functional requirements and design goals
- typical design goal tradeoffs
- how to do a subsystem decomposition.

Overview of Today's Lecture

UML Notations for dynamic Modeling

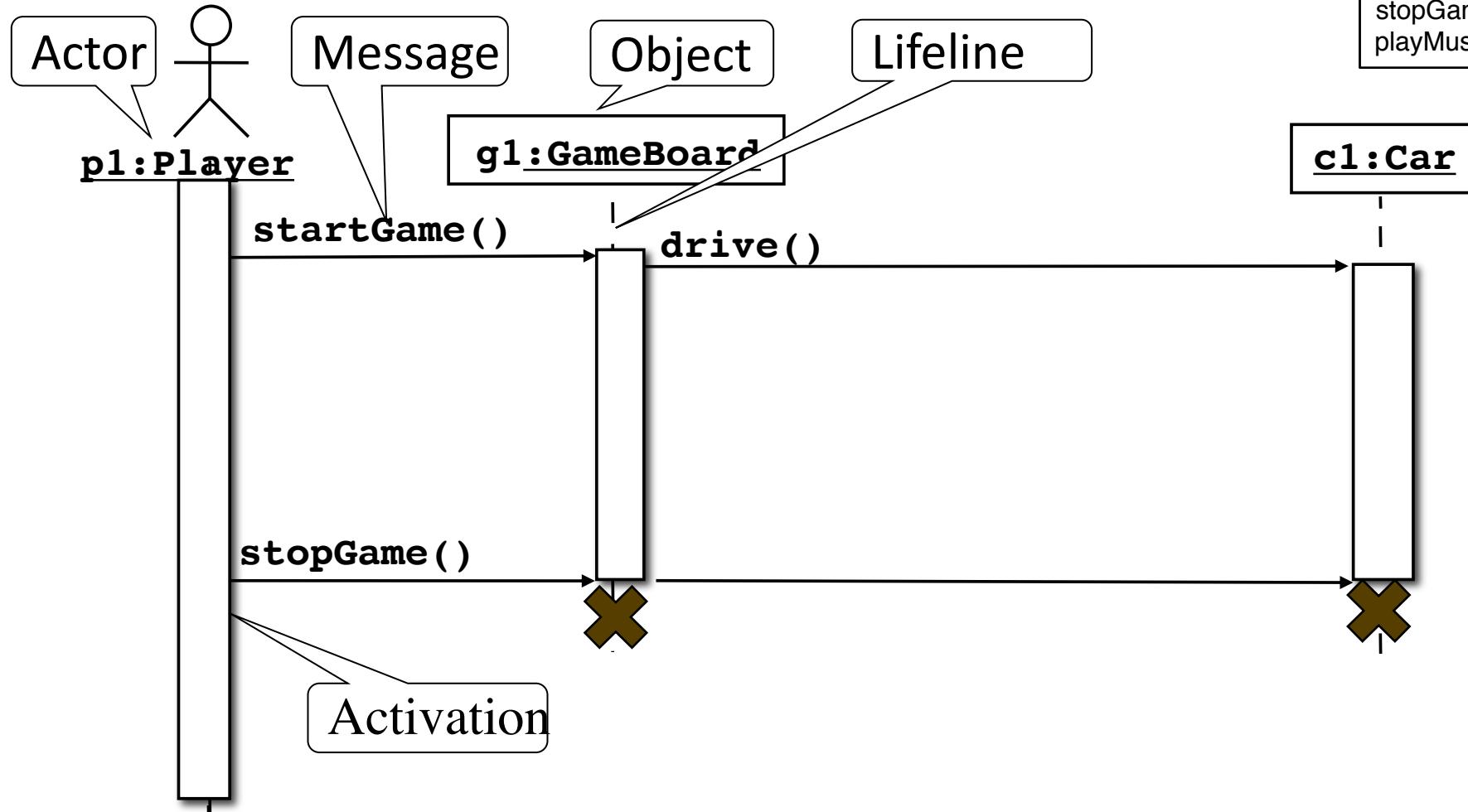
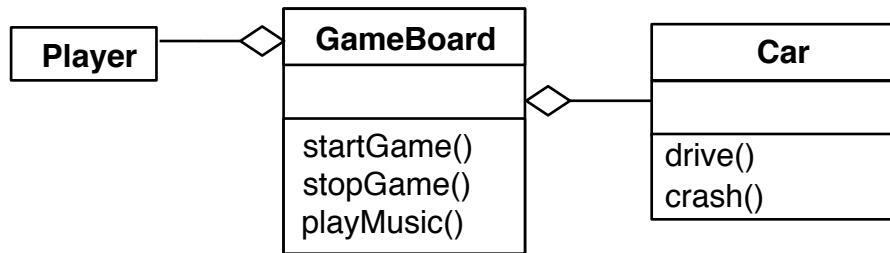
- Sequence Diagrams
- 2. State Chart Diagrams
- 3. Activity Diagrams
- 4. Communication Diagrams

System Design (Today and 17 May 2018)

The Scope of System Design

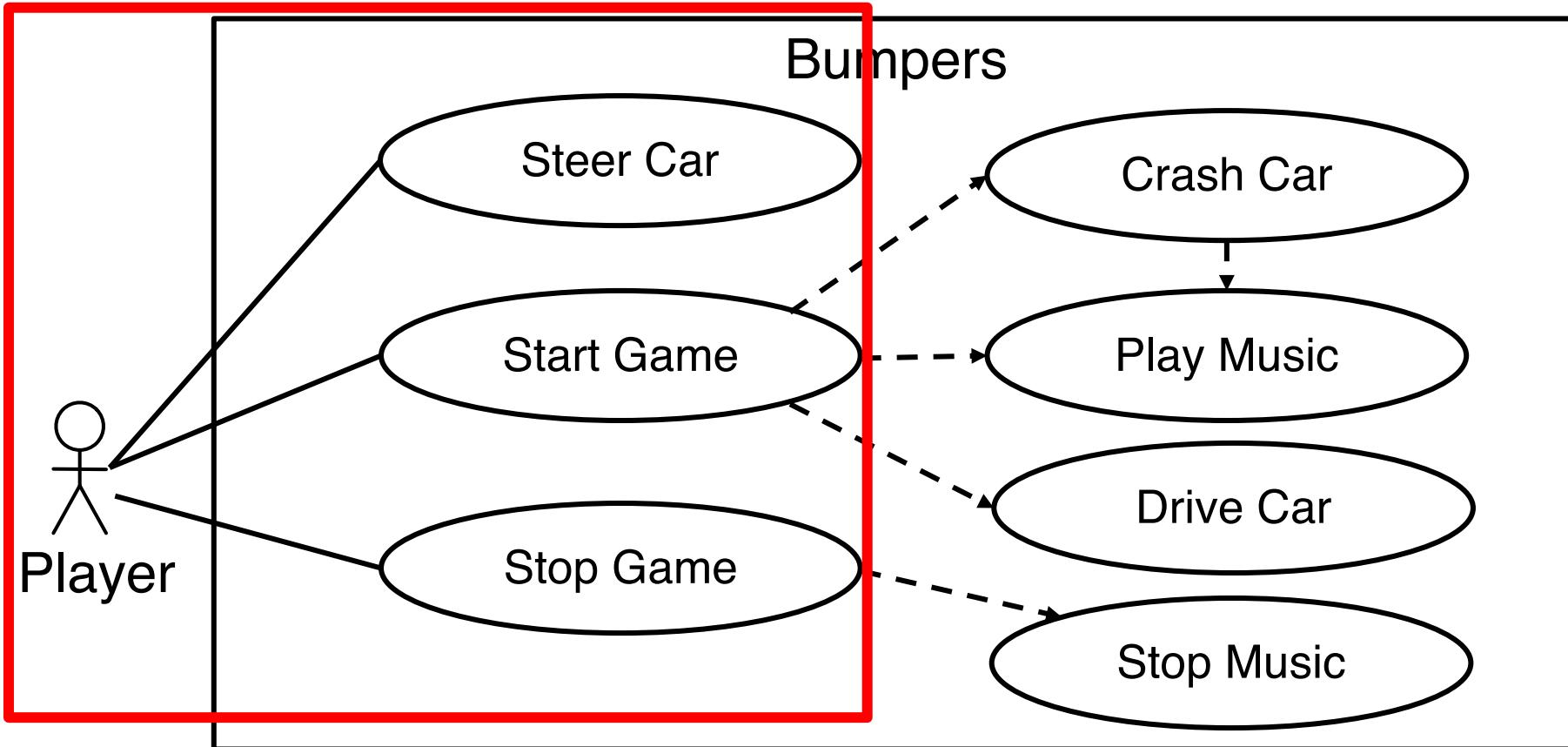
- 1. Design Goals and Trade-Offs
- 2. Subsystem Decomposition, Architectural Styles
- 3. Concurrency: Identification of parallelism
- 4. Hardware/Software Mapping: Mapping subsystems to processors
- 5. Persistent Data Management: Storage for entity objects
- 6. Global Resource Handling & Access Control: Who can access what?
- 7. Software Control: Who is in control?
- 8. Boundary Conditions: Administrative use cases (System startup & termination).

Sequence Diagrams...

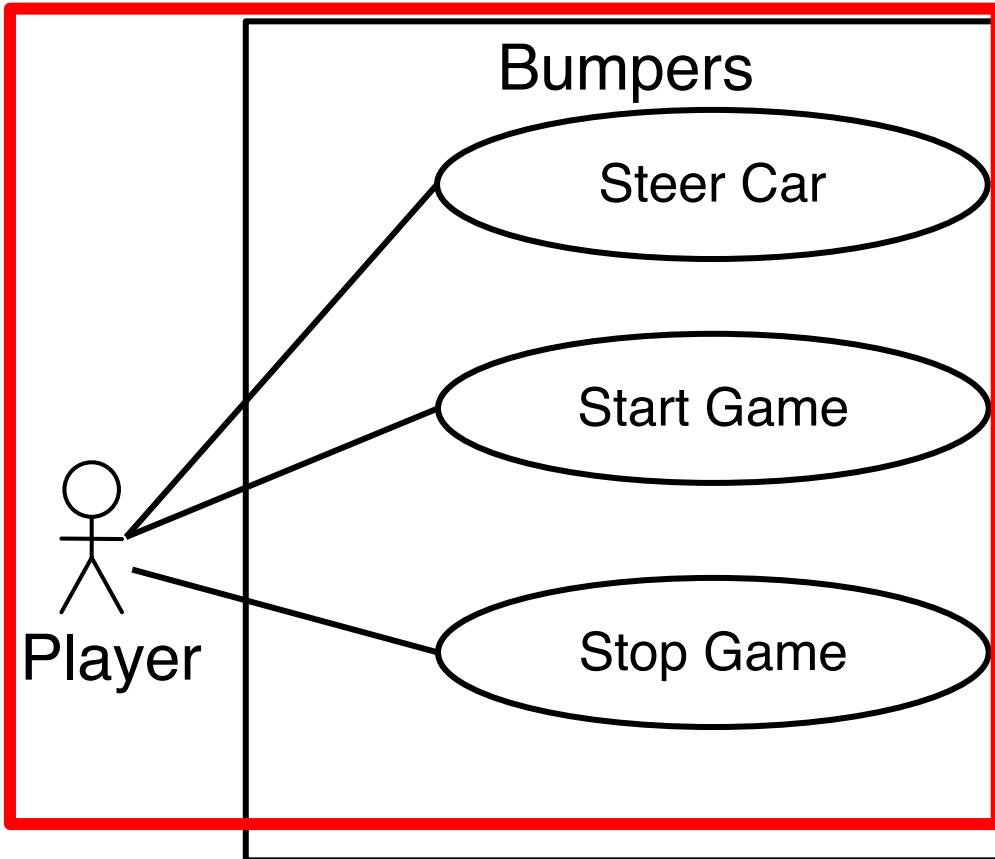


...represent the behavior of a system as messages between *different objects*.

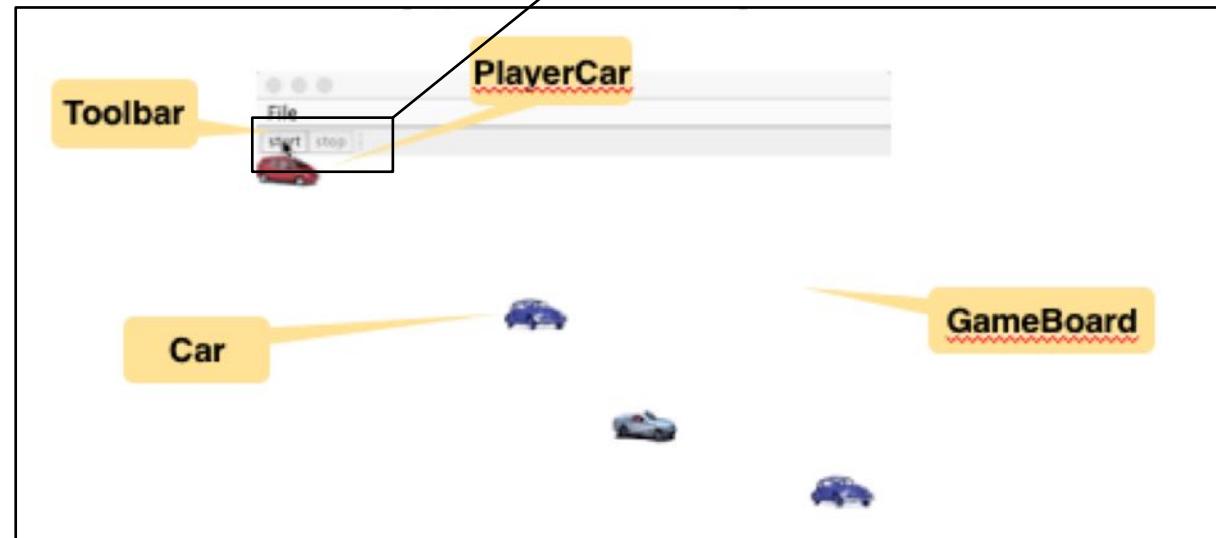
Sequence Diagrams can be combined with Use Case Diagrams for an initial user interface design



Sequence Diagrams can be combined with Use Case Diagrams for an initial user interface design

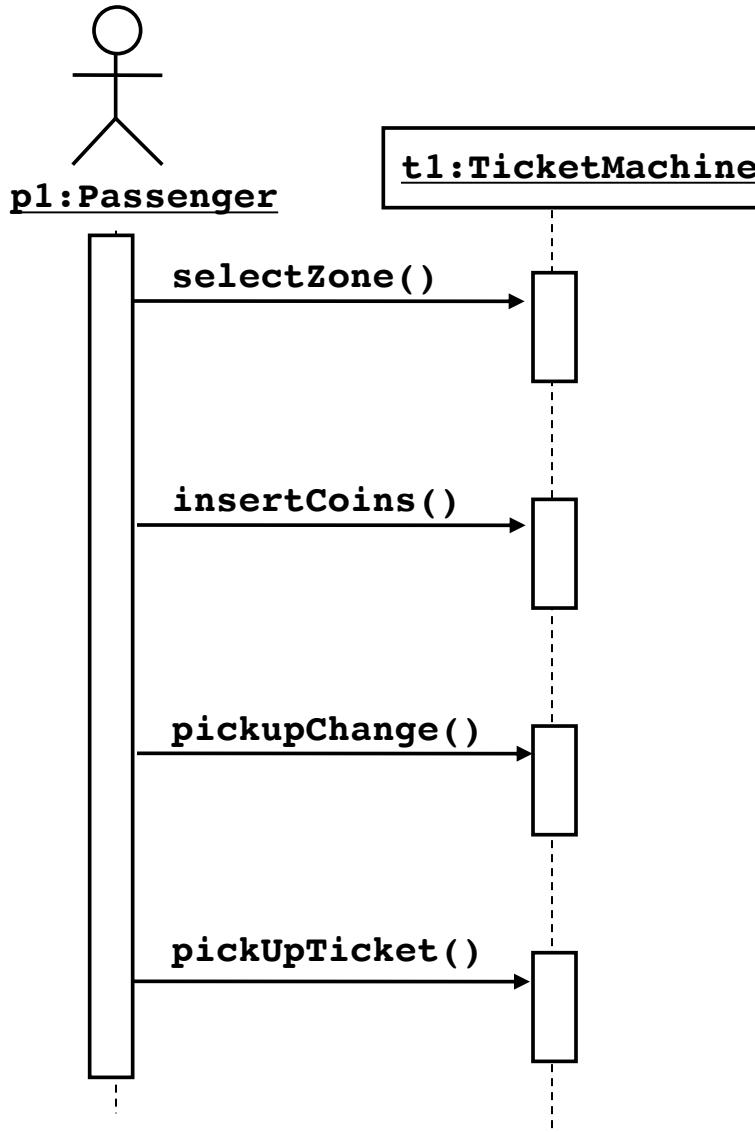


- Check their event flows for participating objects
- These are candidates for boundary objects
- Start and Stop Game are Buttons
- The mouse is the participating object in Steer Car



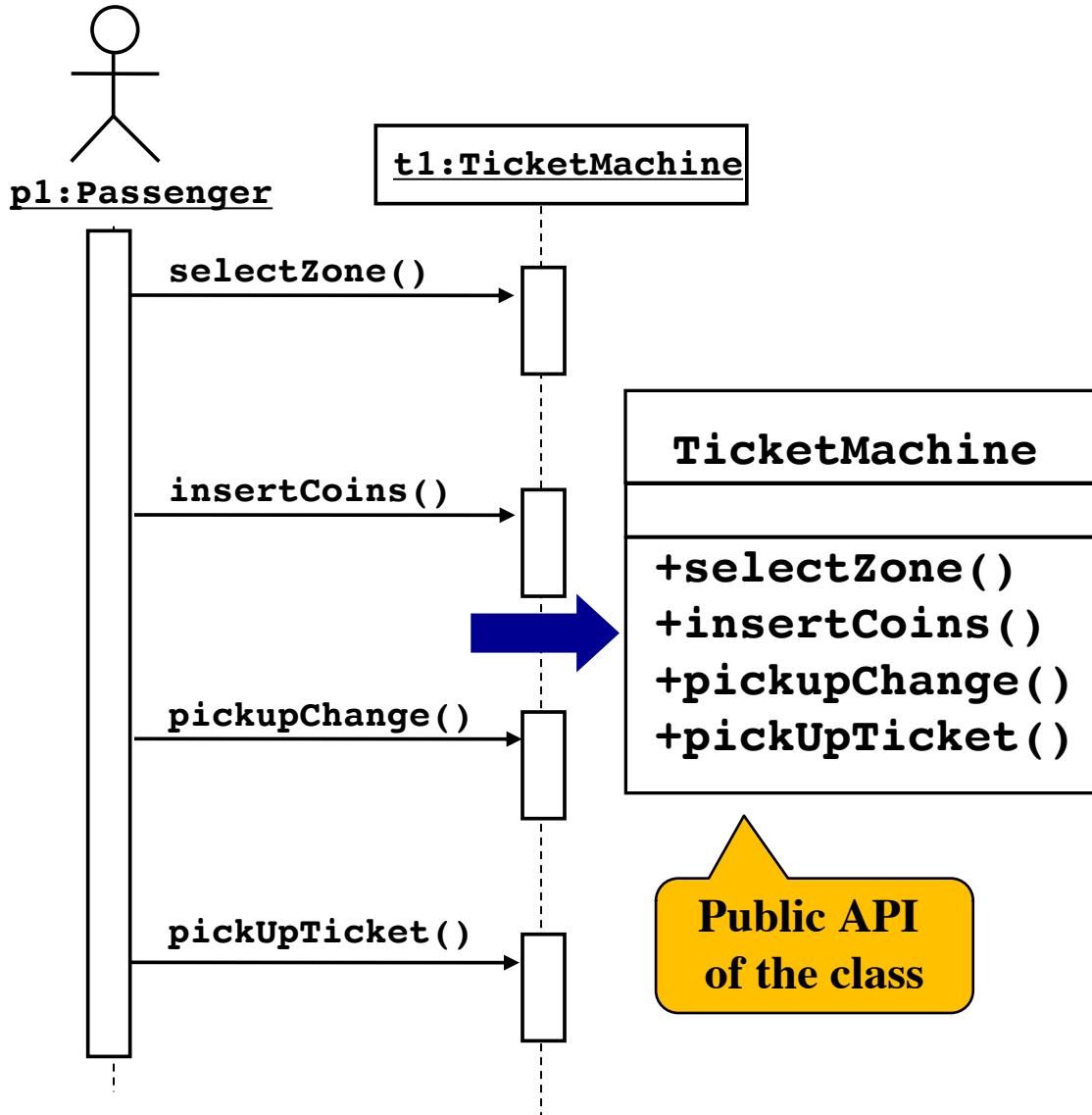
Initial User Interface Design
(See Lecture 2, Slide 58)

Sequence Diagrams Represent Control Flow



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

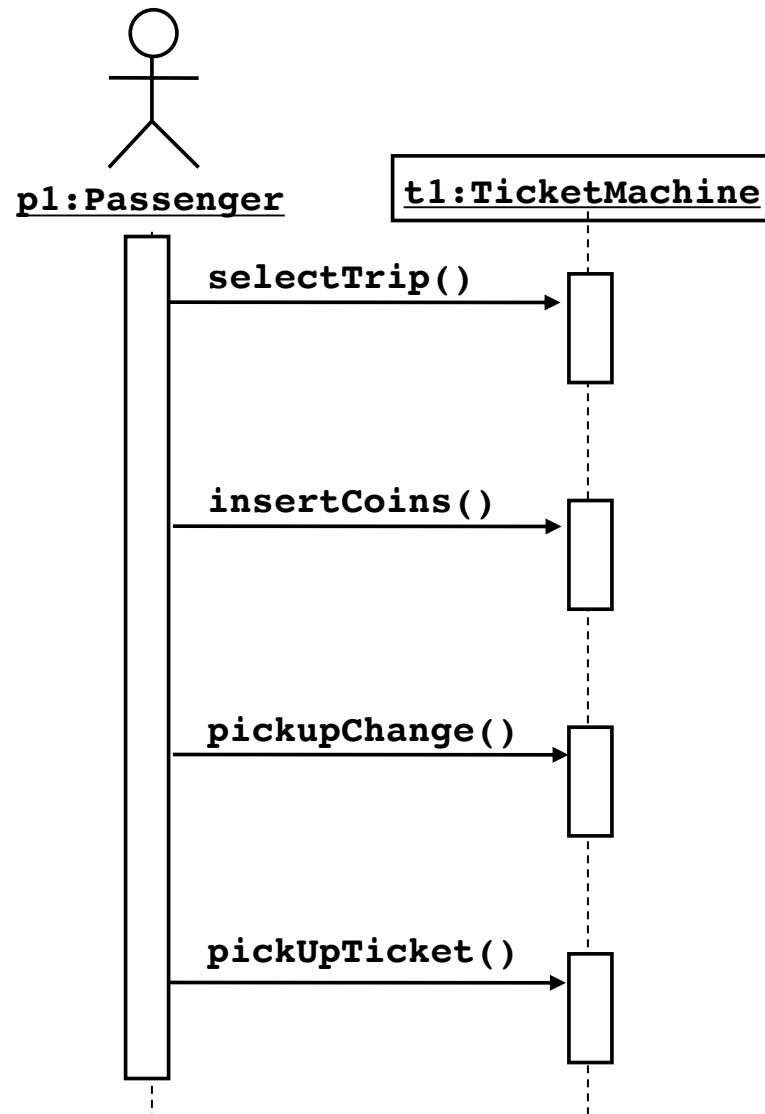
Sequence Diagrams Represent Control Flow



- All message to an object must be public methods in the corresponding class
- They are good candidates for the public interface of a class.

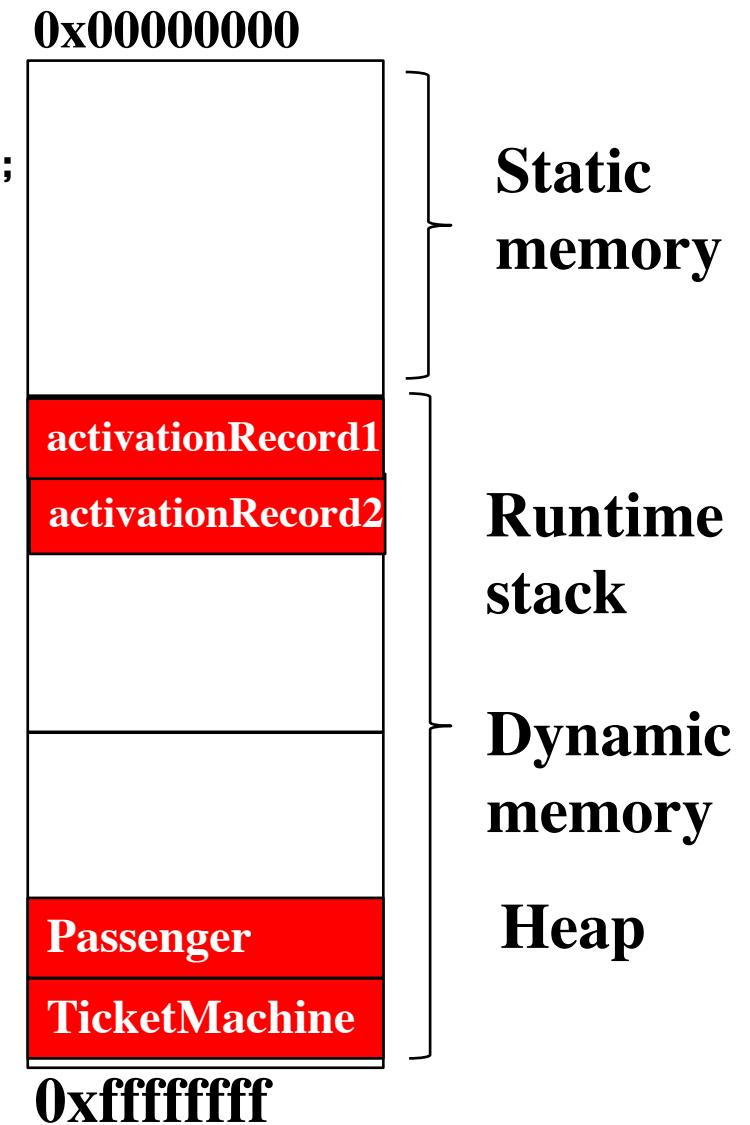
More in the Lecture
about Object Design

Sequence Diagrams, Java Code and Virtual Address Space

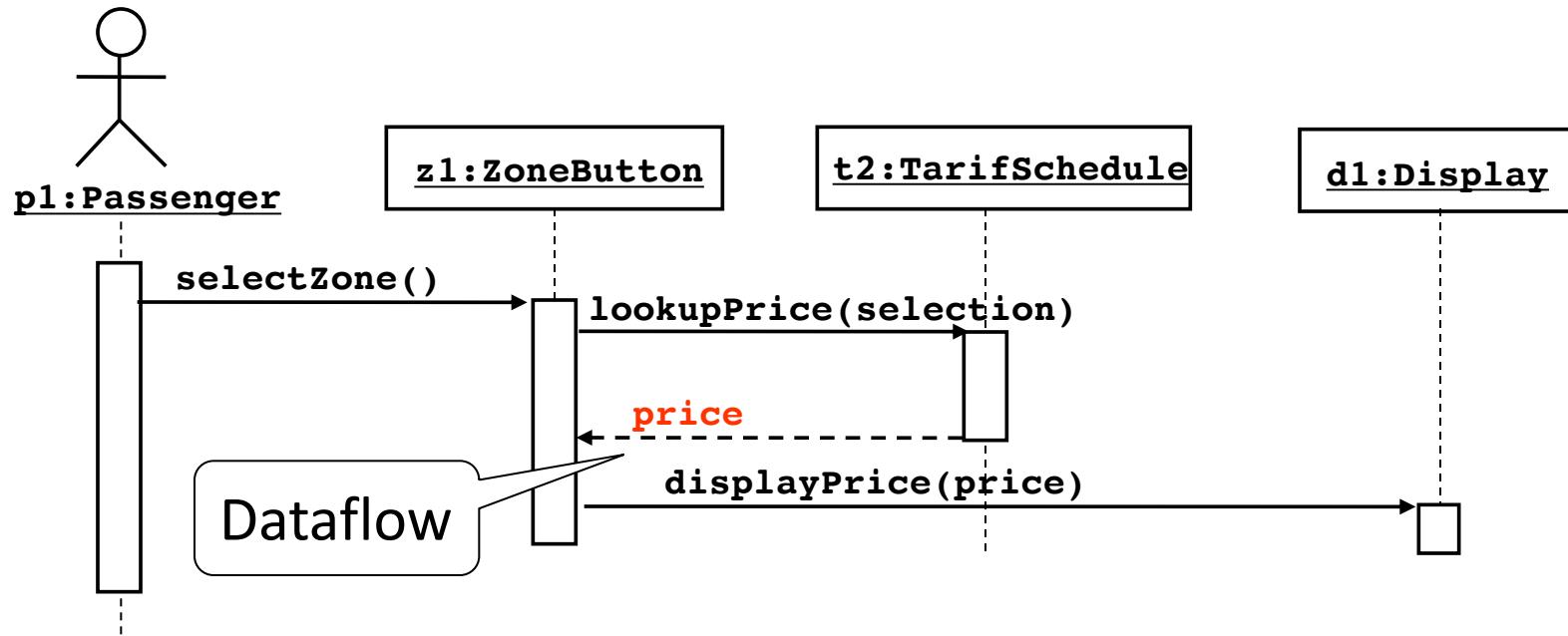


```
t1 = new TicketMachine();
p1 = new Passenger();

t1.selectTrip();
...
t1.insertCoins();
...
t1.pickupChange();
...
t1.pickupTicket();
...
```

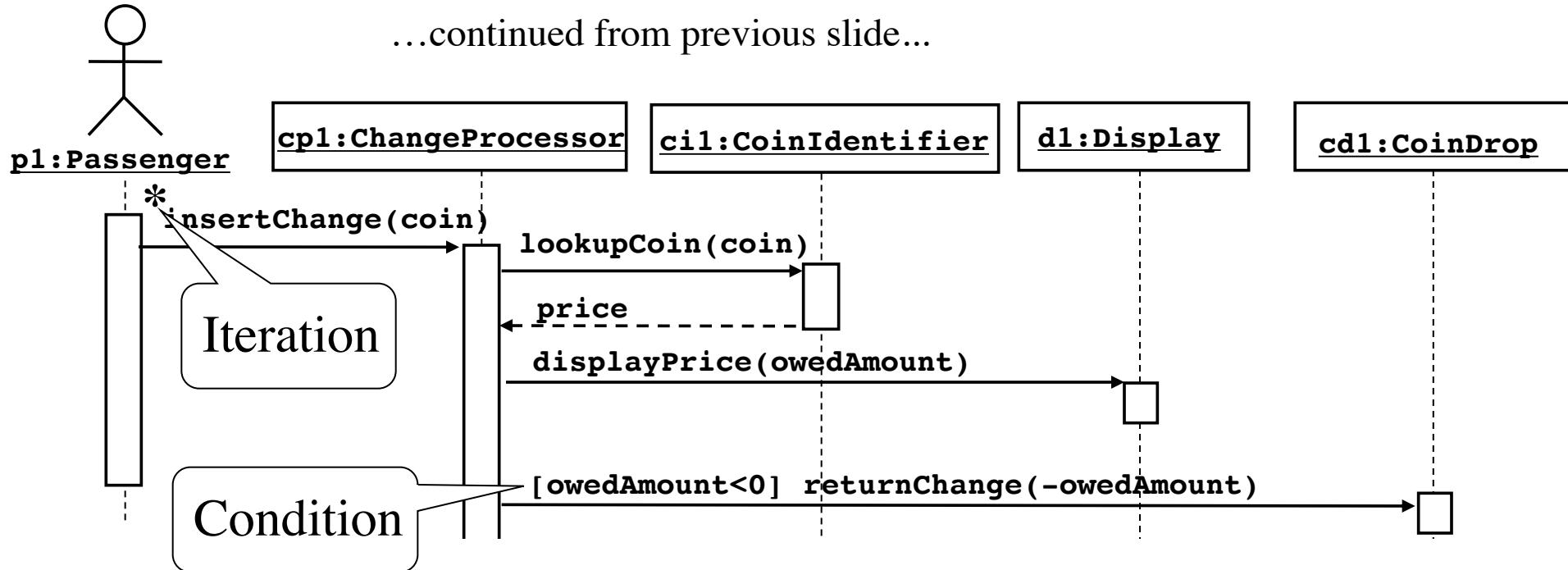


Sequence Diagrams can Represent Data Flow



- 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

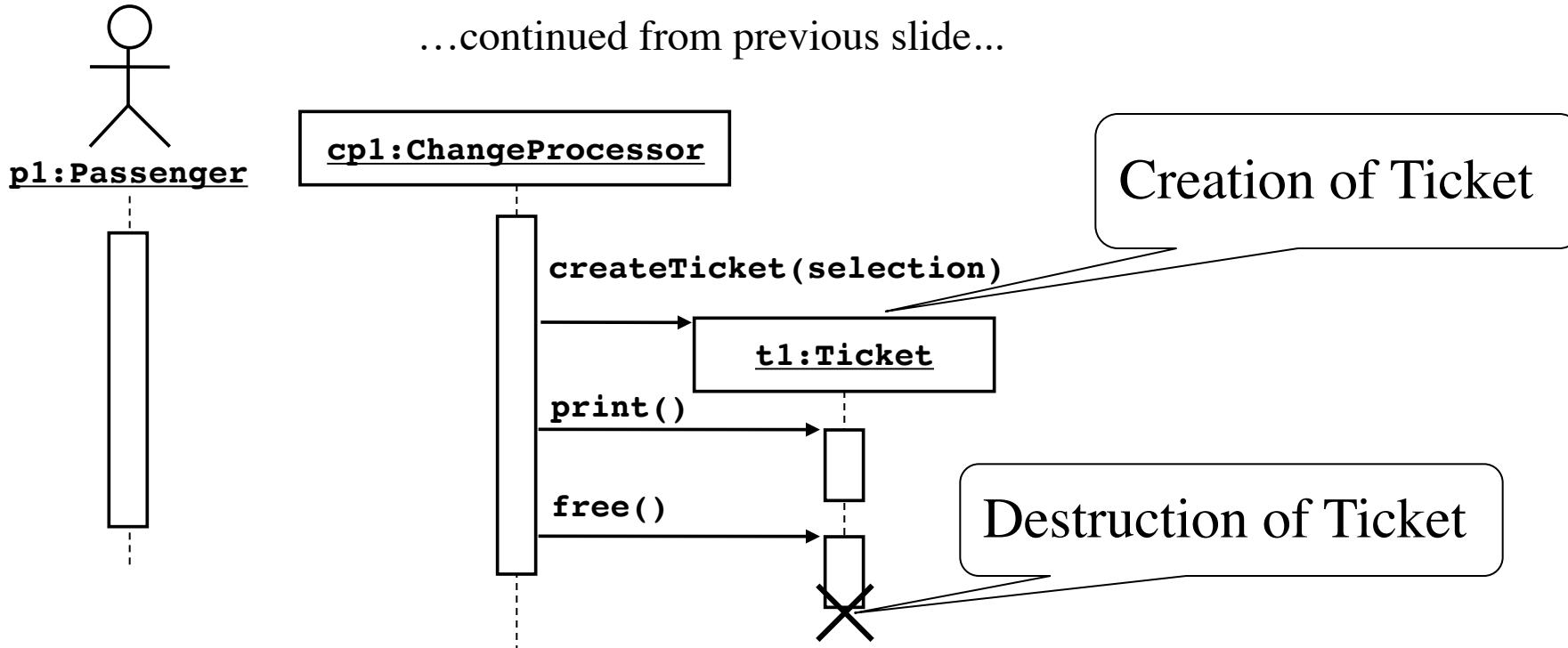
Sequence Diagrams: Iteration & Condition



...continued on next slide...

- Iteration is denoted by a * preceding the message name
- Condition is denoted by boolean expression in [] before the message name

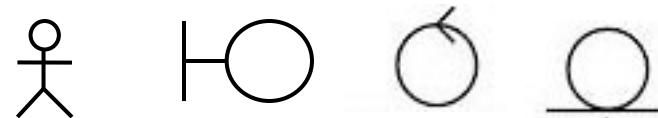
Creation and destruction



- 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.

Heuristics for Drawing Sequence Diagrams

- Column Layout (from left to right)
 - 1st column: Should be the **actor** of the use case
 - Next columns: The **boundary objects**
 - Next columns: The **control objects** that manage the rest of the use case
 - Remaining columns: The **entity objects**



- Order in the creation of objects
 - Create control objects at beginning of the flow of events
 - The control objects then create the boundary objects
- Access of objects (Who can access whom)
 - Control and boundary objects can access entity objects
 - Entity objects should not access boundary or control objects.

Properties of Sequence Diagrams

- UML sequence diagrams represent ***behavior in terms of interactions***
- Useful to identify missing objects
- Time consuming to build
- Useful during modeling when we have to identify the public methods of a class or a subsystem (API)
- Complement the class diagrams (which represent structure).

Overview of Today's Lecture

UML Notations for dynamic Modeling

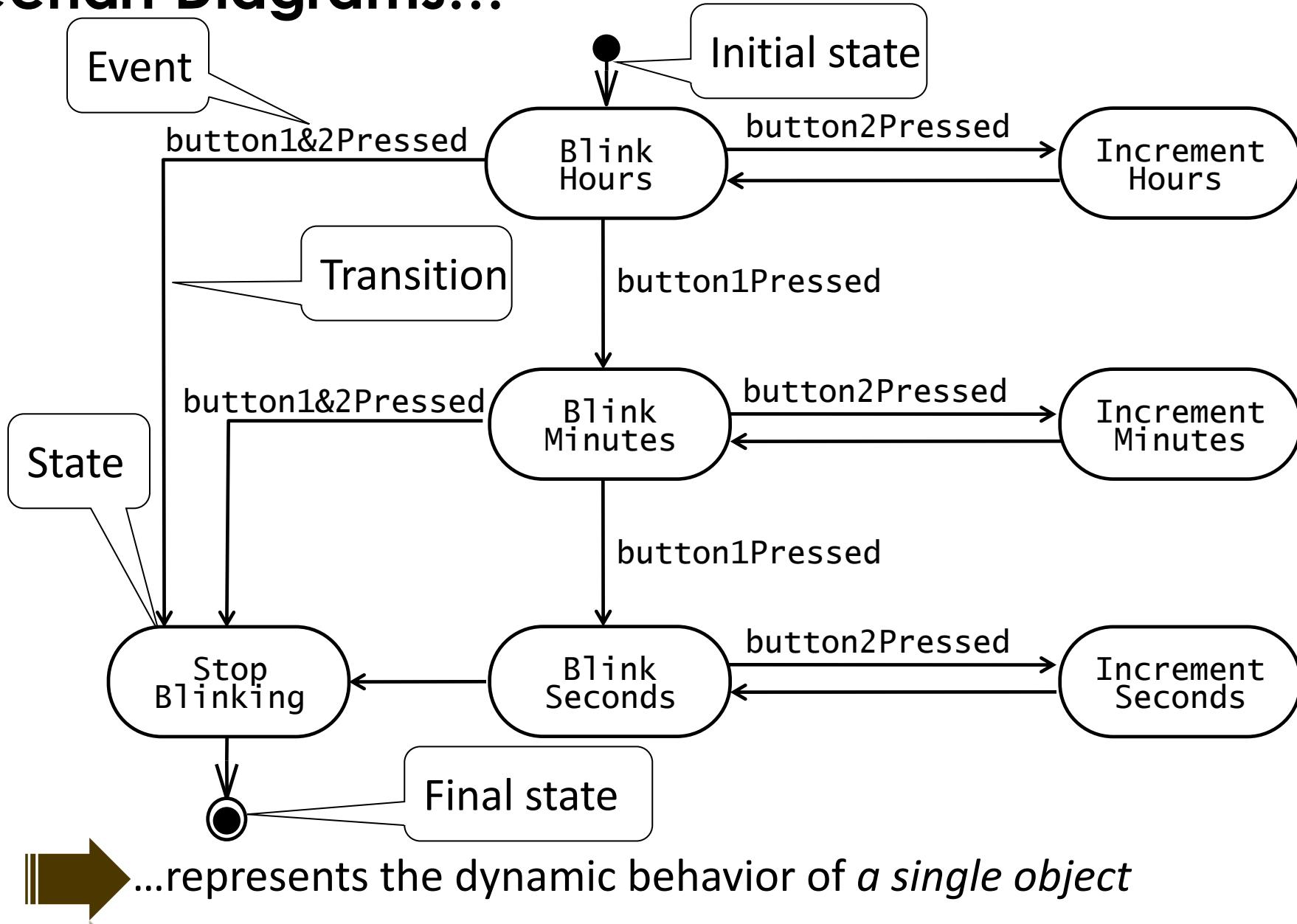
1. Sequence Diagrams
2.  State Chart Diagrams
3. Activity Diagrams
4. Communication Diagrams

System Design (Today and 17 May 2018)

The Scope of System Design

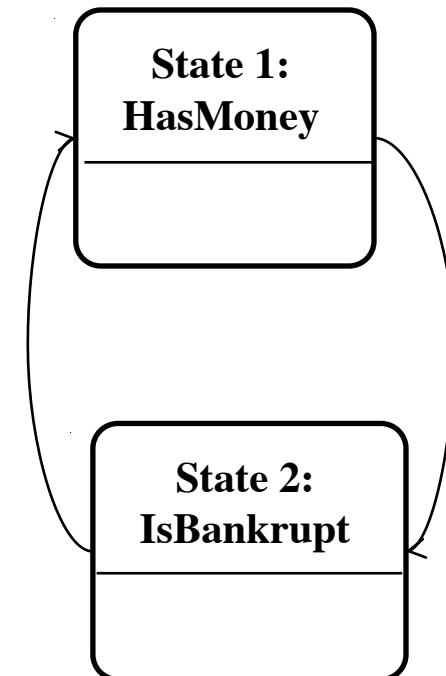
1. Design Goals and Trade-Offs
2. Subsystem Decomposition, Architectural Styles
3. Concurrency: Identification of parallelism
4. Hardware/Software Mapping: Mapping subsystems to processors
5. Persistent Data Management: Storage for entity objects
6. Global Resource Handling & Access Control: Who can access what?
7. Software Control: Who is in control?
8. Boundary Conditions: Administrative use cases (System startup & termination).

Statechart Diagrams...

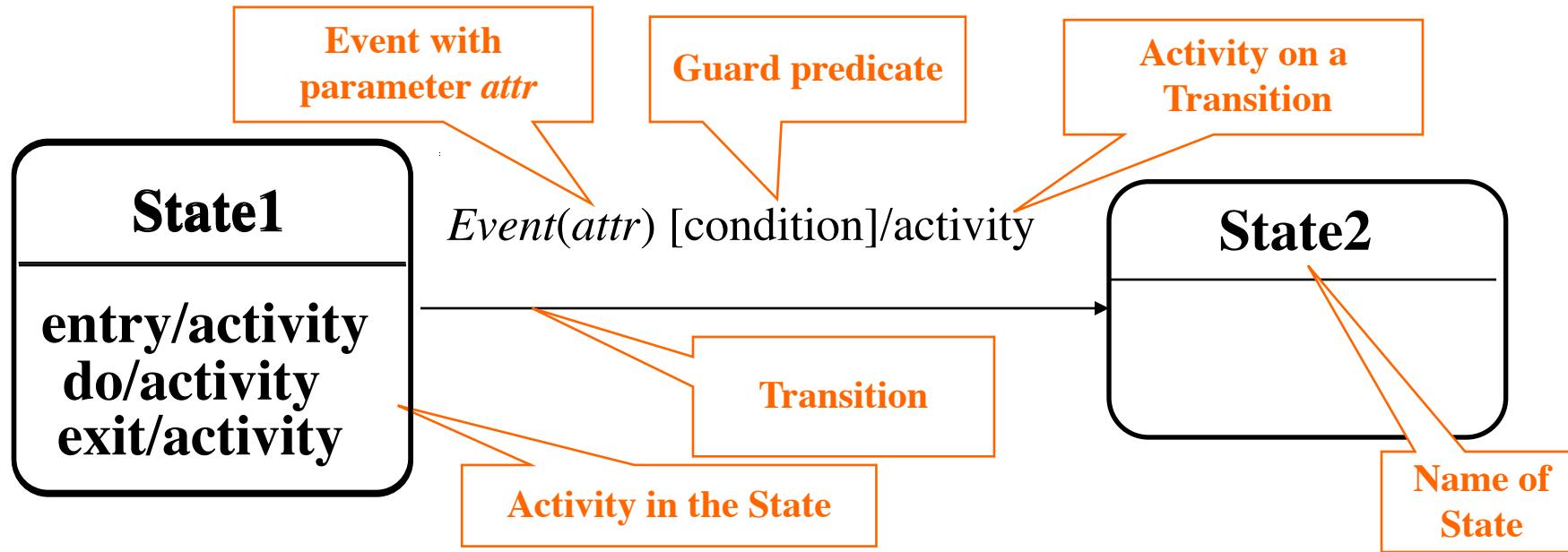


State

- **State**: An abstraction of the attributes of a class
 - State is the aggregation of several attributes of a class
- A state is an equivalence class of all those attribute values that do no need to be distinguished
 - Example: State of a bank
 - State 1: Bank has money
 - State 2: Bank is bankrupt
- State has duration.

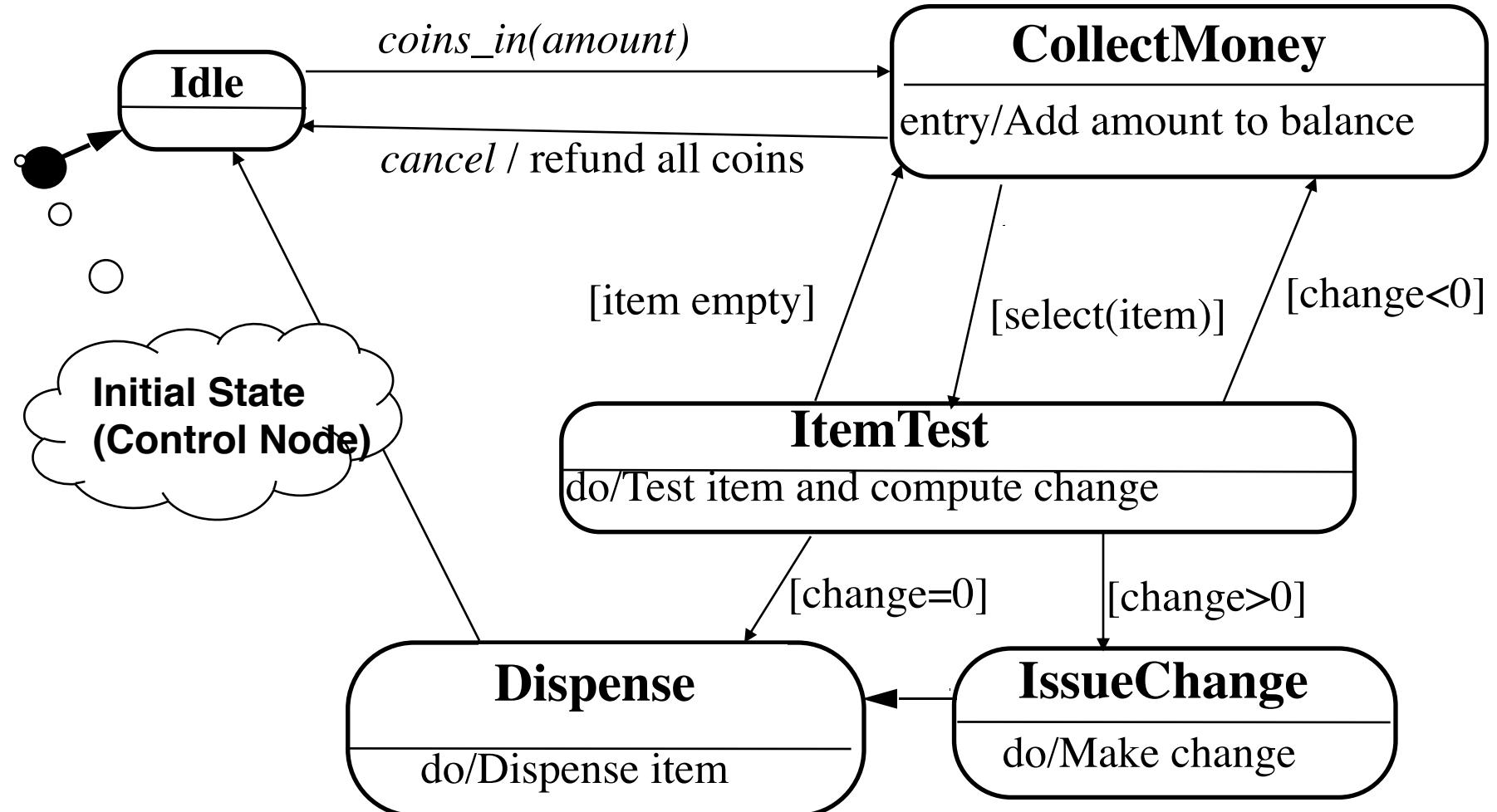


UML Statechart Diagram Notation



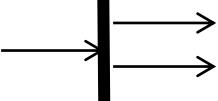
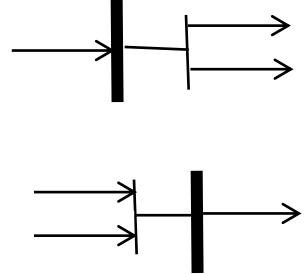
- Modeling Conventions ("EIST Modeling Style Guide"):
 - *Events are written in italics* or with a normal type font
 - Guard predicates are enclosed in brackets []
 - Activities on a transition are always prefixed with a slash /
- UML statecharts are based on work by Harel
 - They are a generalization of finite state automata and Mealy and Moore automatas.

Example: Statechart Diagram of a Vending Machine

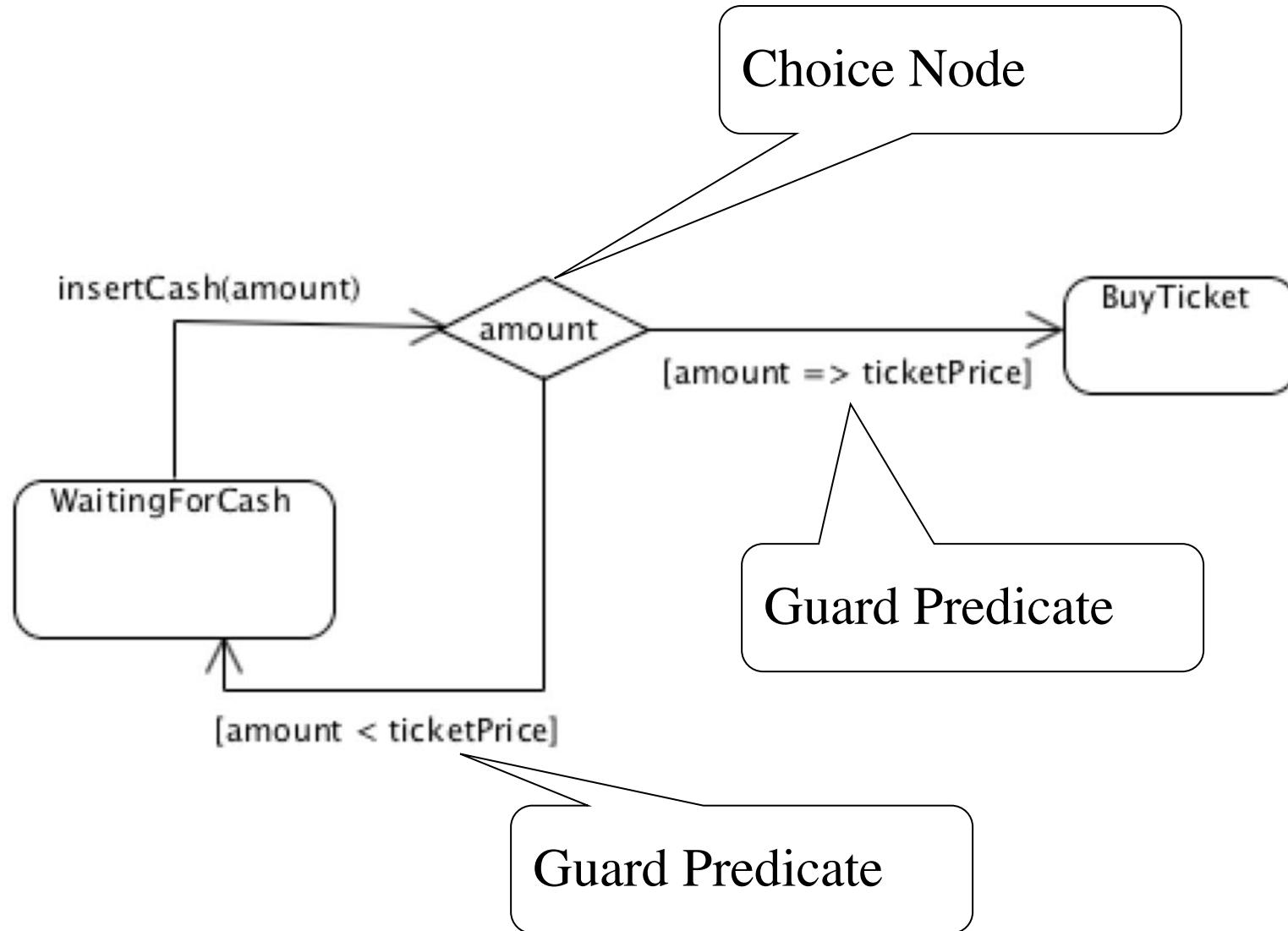


Control Nodes in Statechart Diagrams

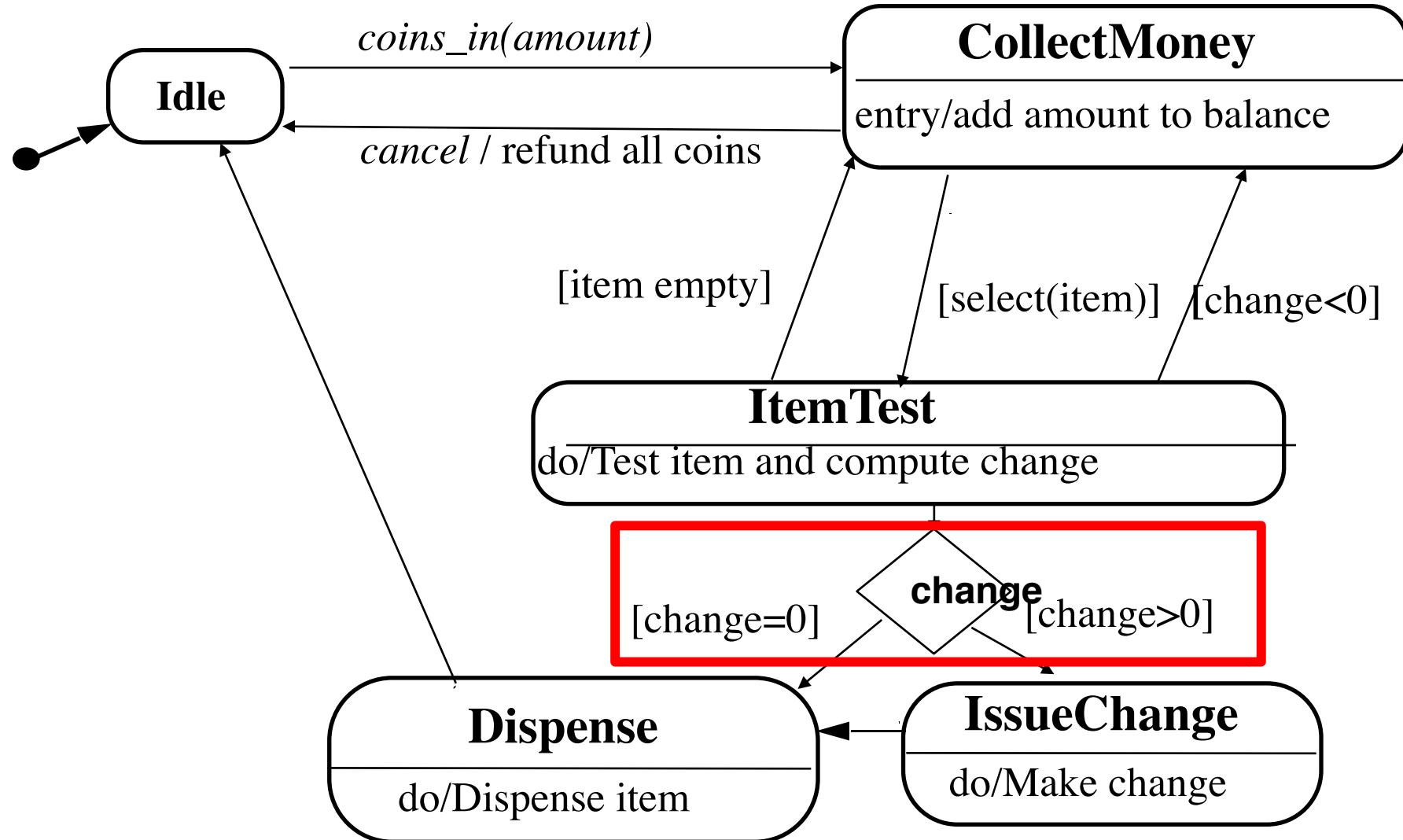
A **control node** is a special node that coordinates the transition between other nodes

Name	Graphical Icon
Initial node	●
Final node	○
Fork node	
Join node	
Choice node	

Example of a Choice Node



Vending Machine with a Choice Node



Statechart Diagram vs Sequence Diagram

- Statechart 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.

Overview of Today's Lecture

UML Notations for dynamic Modeling

1. Sequence Diagrams
2. State Chart Diagrams
-  3. Activity Diagrams
4. Communication Diagrams

System Design (Today and 17 May 2018)

The Scope of System Design

1. Design Goals and Trade-Offs
2. Subsystem Decomposition, Architectural Styles
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 (System startup & termination).

UML Activity Diagrams

Activity diagrams also consist of nodes and edges

- **Nodes** can describe activities and objects
 - Control nodes
 - Executable nodes
 - Most prominent: **Action Node**
 - Object nodes
 - E.g. a document
- An **Edge** is a directed connection between nodes

Action Nodes and Object Nodes

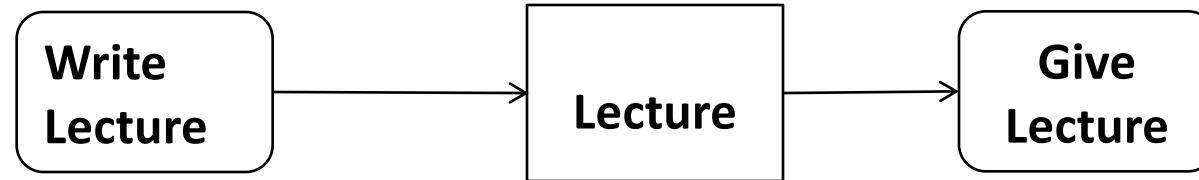
- Object Node



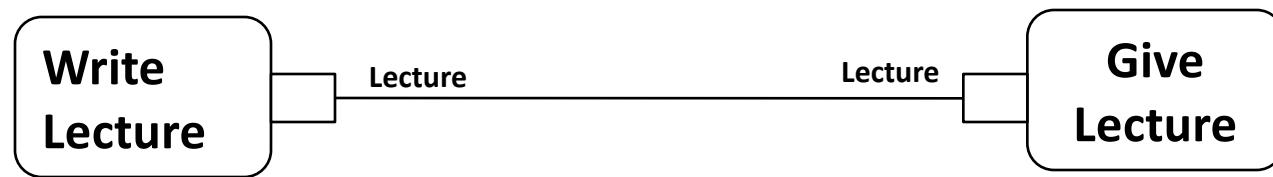
- Action



Alternative Notation for an Object Node



- Different notations with same semantics



- Both notations define object flow in an activity
- The Pin notation is useful for complicated workflows

Control Node Icons in an Activity Diagram

- Initial node



- Final node



- Fork node



- Join node

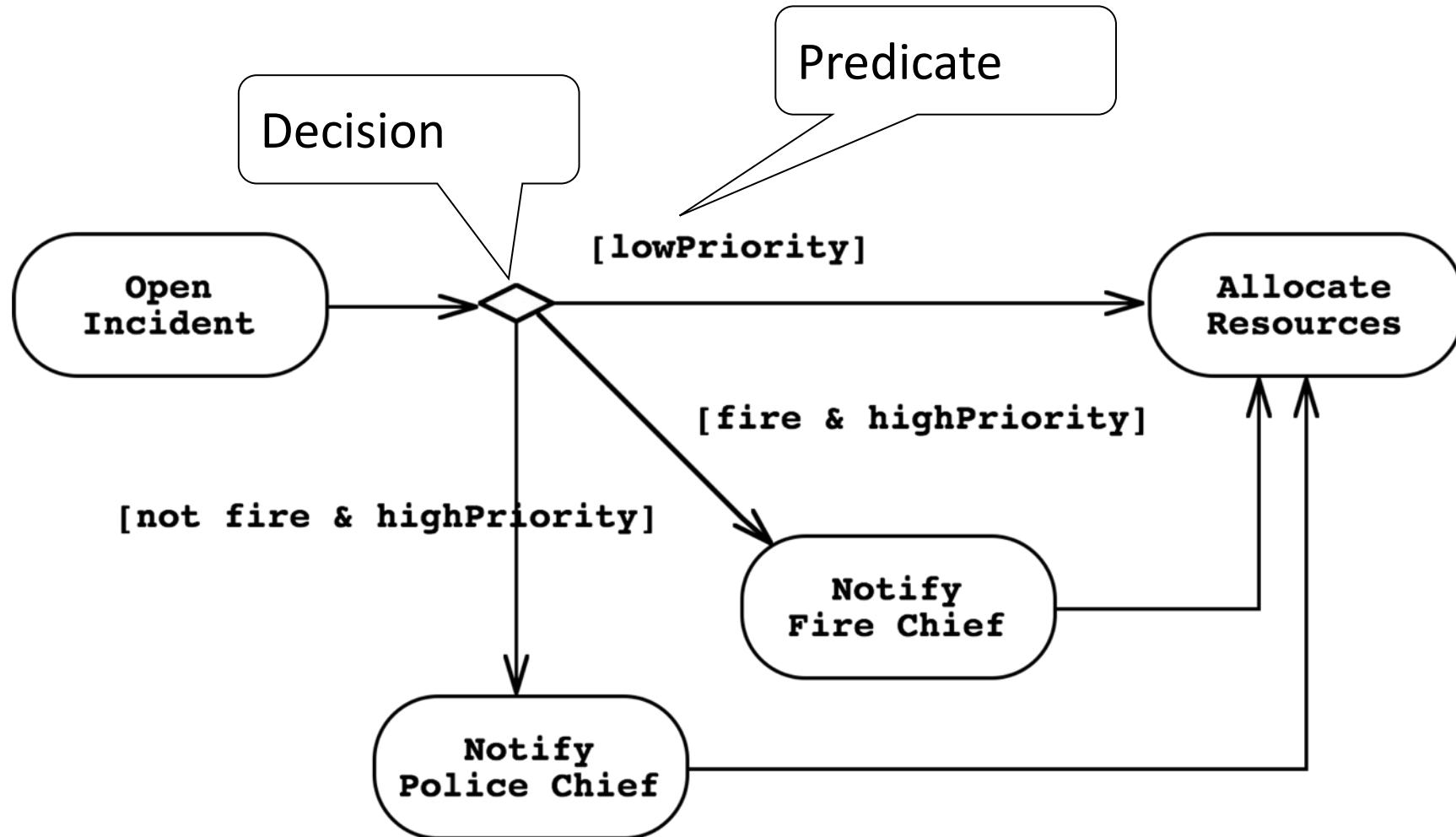


- Decision node



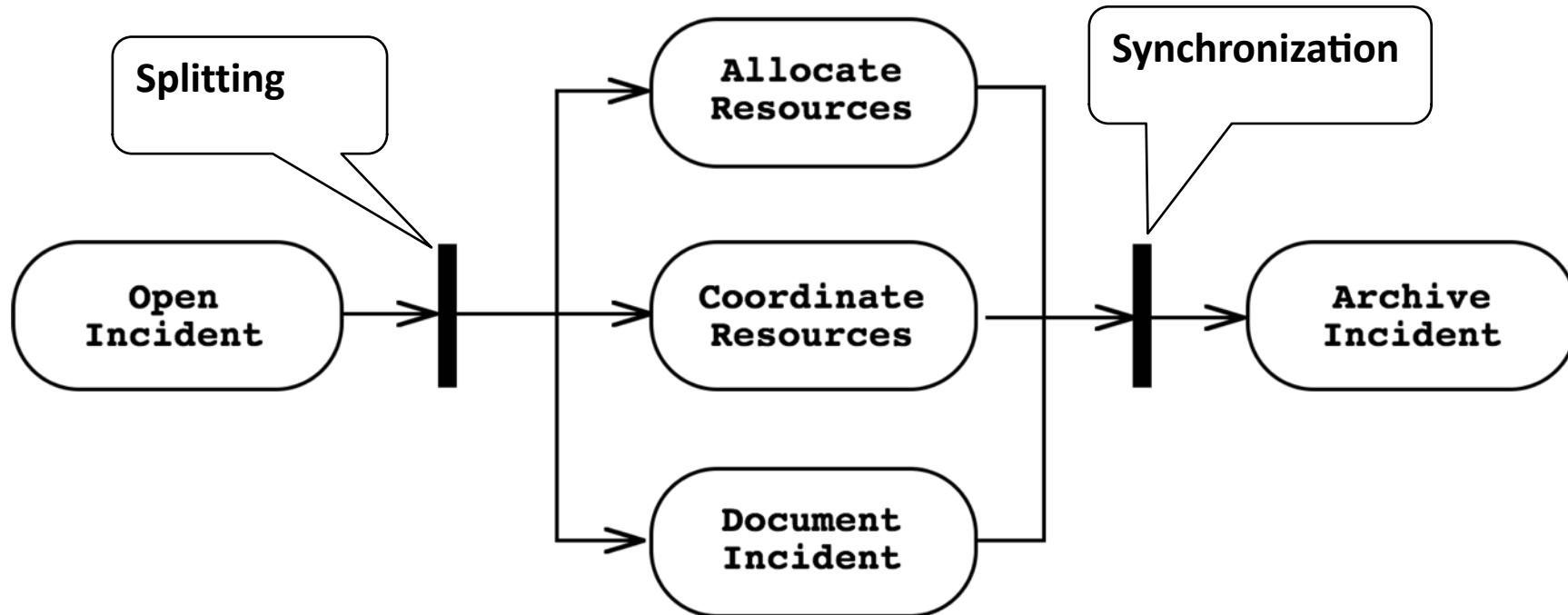
- Merge node

Example of a Decision Node



Example of Fork and Join Nodes

- Synchronization of multiple activities
 - Splitting the flow of control into multiple activities
 - Joining multiple activities into a single activity

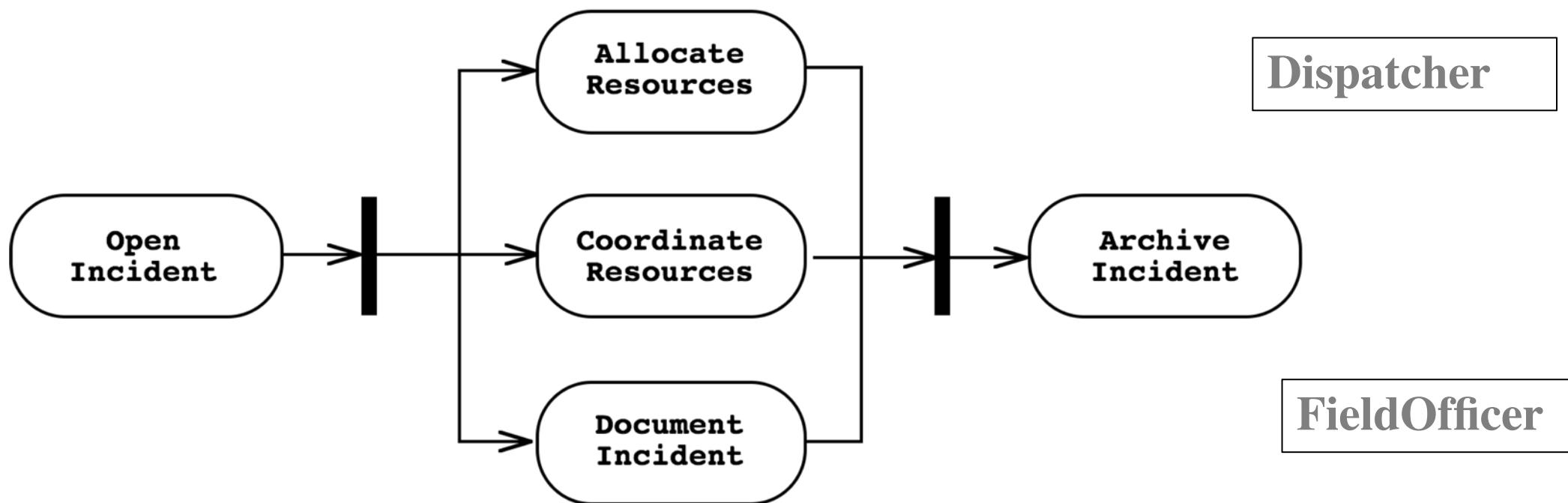


UML **Swimlanes** allow Grouping of Activities

Incident
Open()
Archive()
Document()

- Activities may be grouped into **swimlanes** to denote the object, subsystem or actor that accesses these activities

Resource
Allocate()
Coordinate()

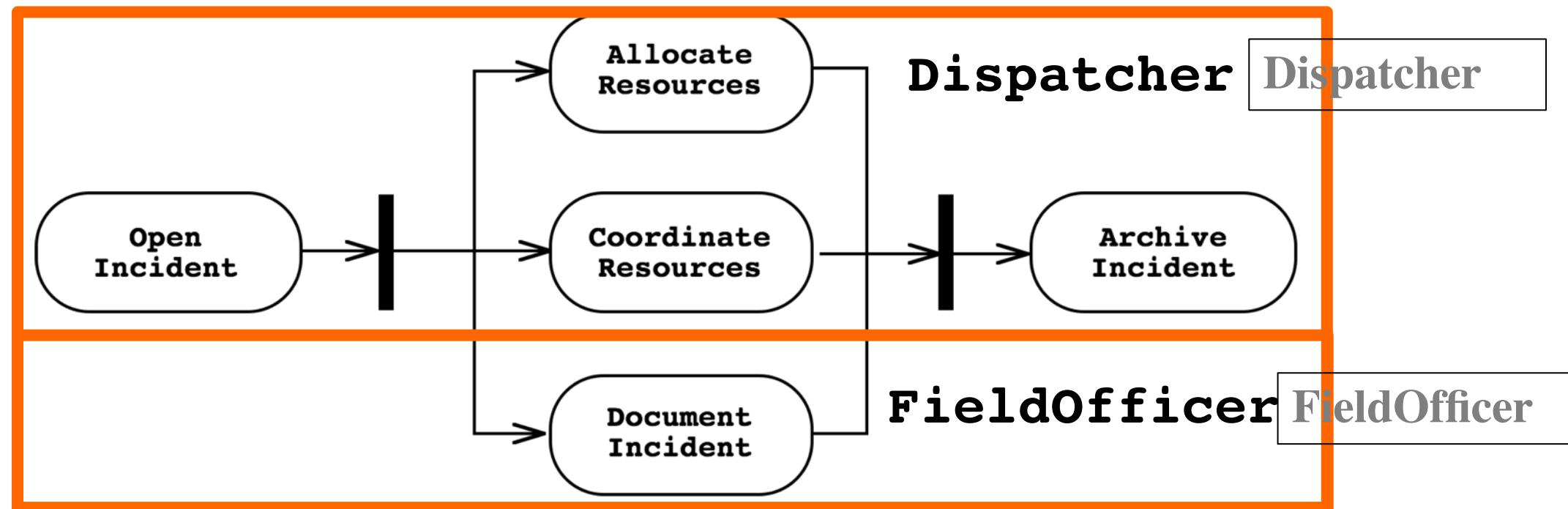


UML **Swimlanes** allow Grouping of Activities

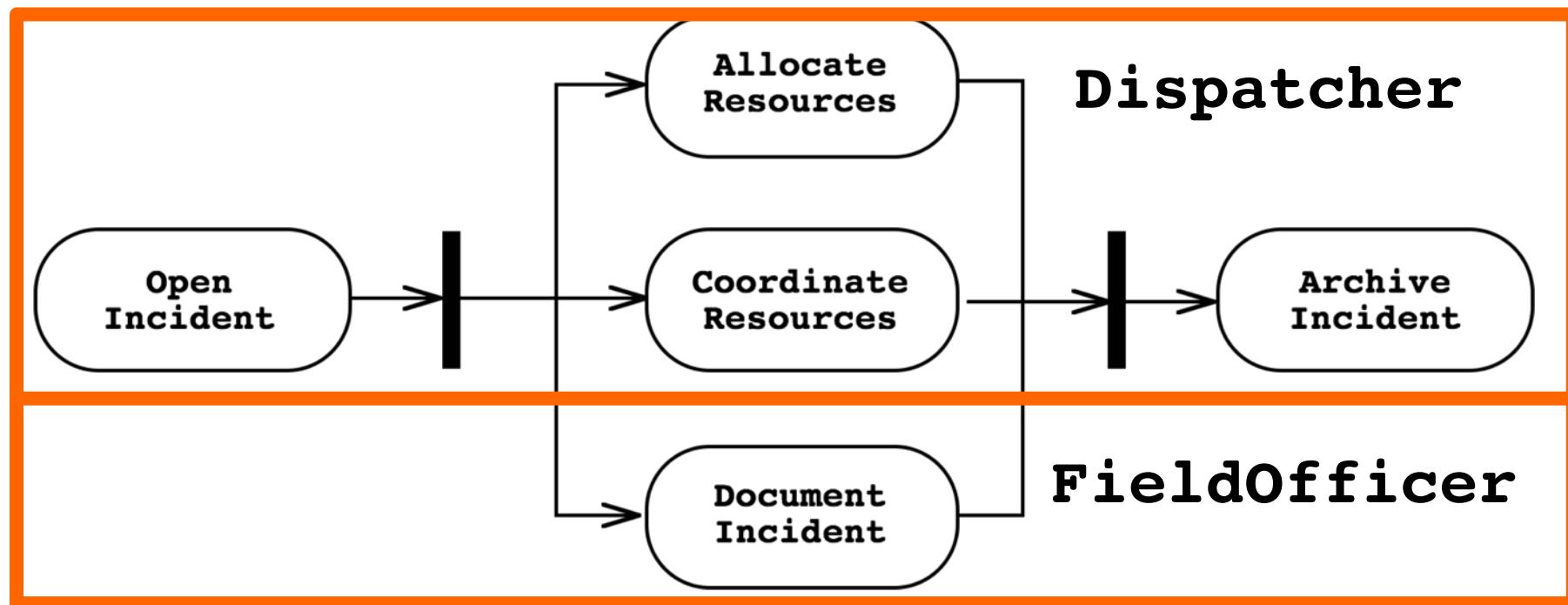
Incident
Open()
Archive()
Document()

Resource
Allocate()
Coordinate()

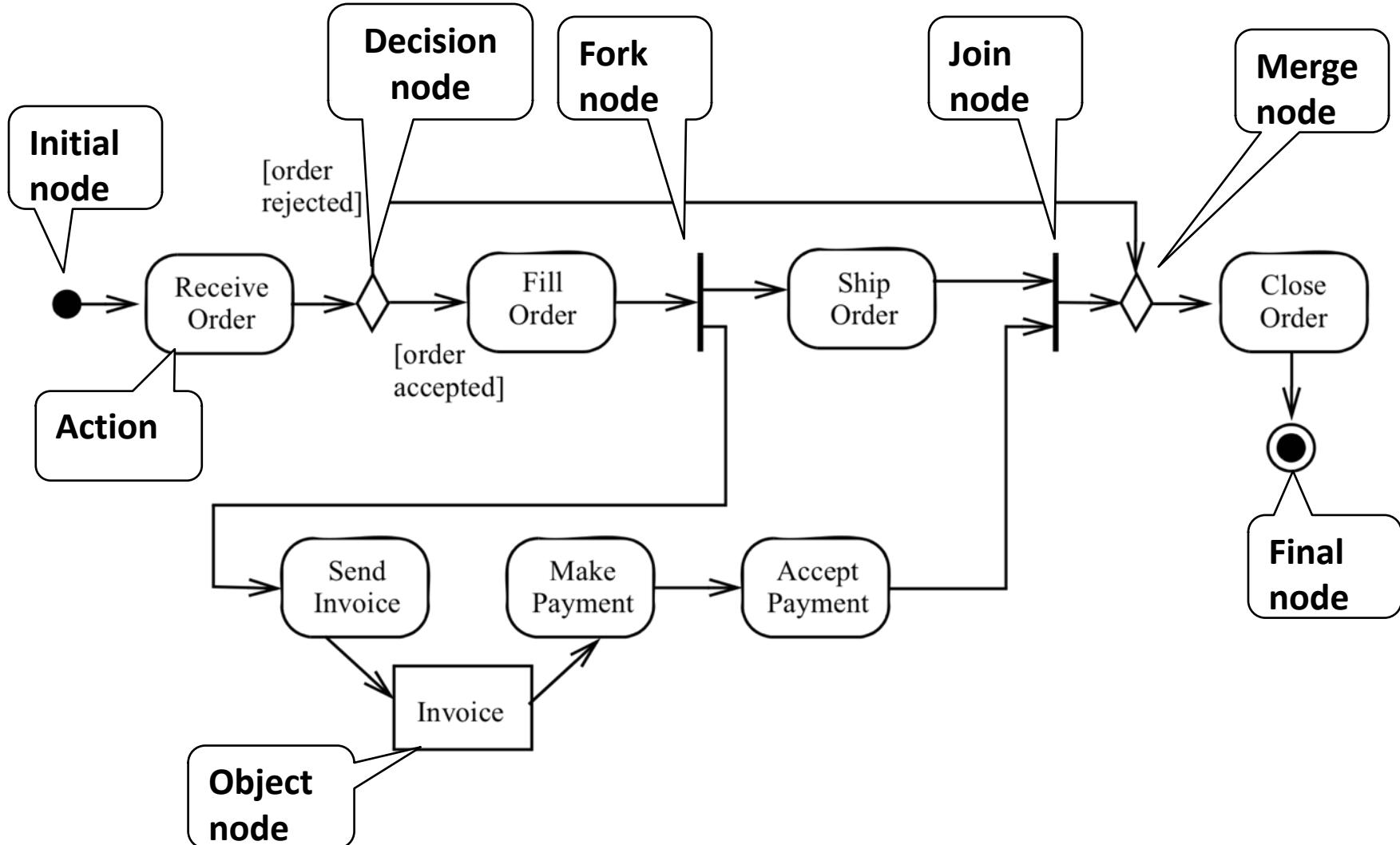
- Activities may be grouped into **swimlanes** to denote the object, subsystem or actor that accesses these activities



Swimlanes: Grouping of Activities

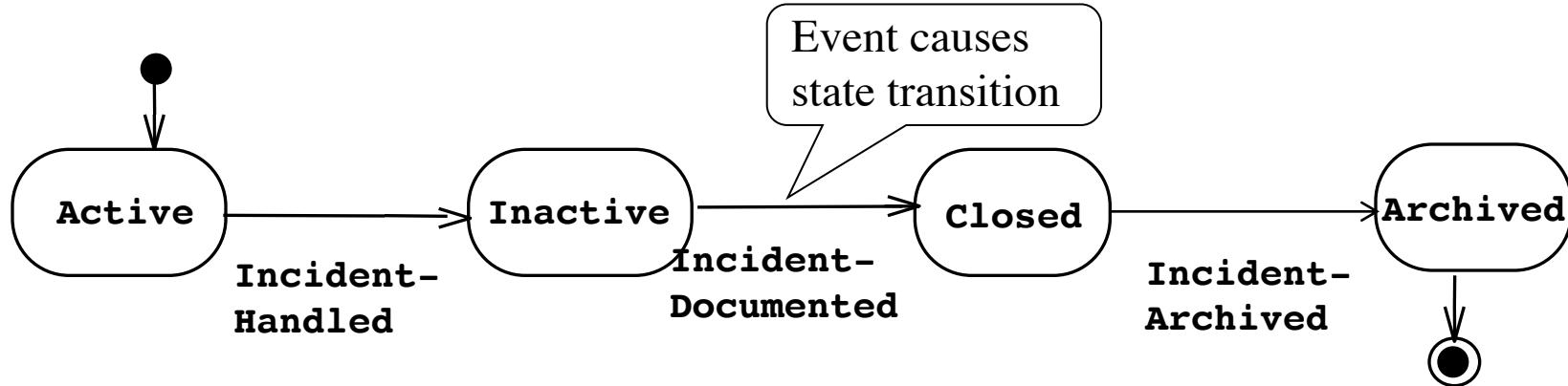


Control Nodes in a Workflow for an Order Management System

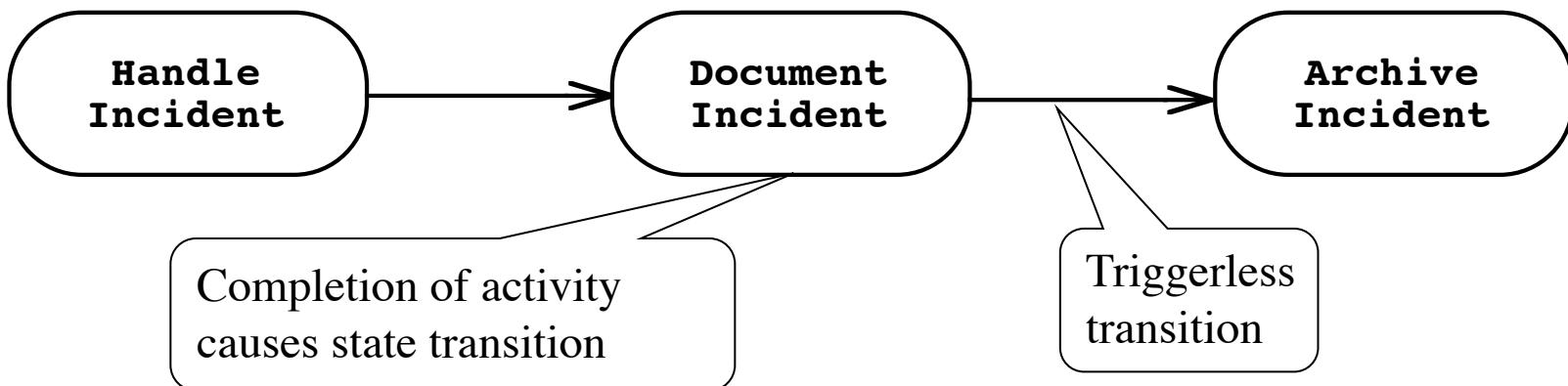


Statechart Diagrams vs Activity Diagrams

Incident Statechart Diagram



Incident Activity Diagram



Overview of Today's Lecture

UML Notations for dynamic Modeling

1. Sequence Diagrams
2. State Chart Diagrams
3. Activity Diagrams
4.  Communication Diagrams

System Design (Today and 17 May 2018)

The Scope of System Design

1. Design Goals and Trade-Offs
2. Subsystem Decomposition, Architectural Styles
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 (System startup & termination).

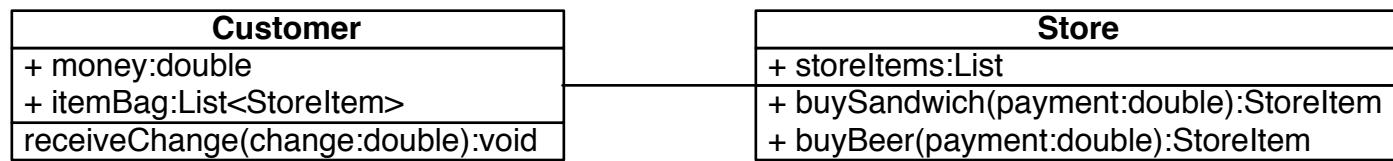
UML Communication Diagram

- A **Communication Diagram** 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
 - Communication diagrams reuse the layout of classes and associations in the class diagram
 - The dynamic behavior is obtained from the dynamic model (UML sequence diagram and UML statechart diagram)
 - Messages between objects are labeled with a number and placed near the link the message is sent over.

UML Communication Diagram

Describe the interaction between different objects by using method invocation

Example class diagram:



- Remember: Class diagrams contain associations:
 - But they do not show the communication (message flow) between the objects/instances

Messages in Communication diagrams

Describe the communication between different objects in terms of messages

Example class diagram:

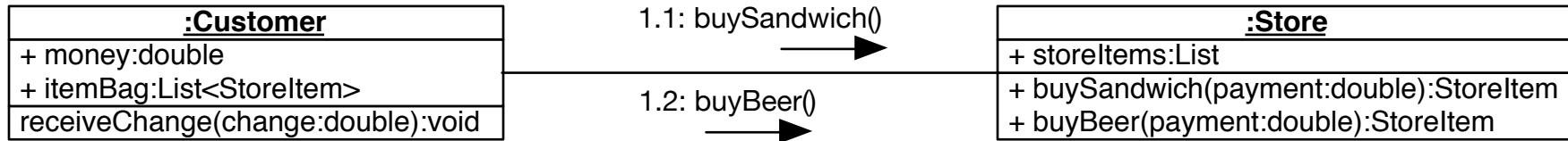


There are three different types of messages:

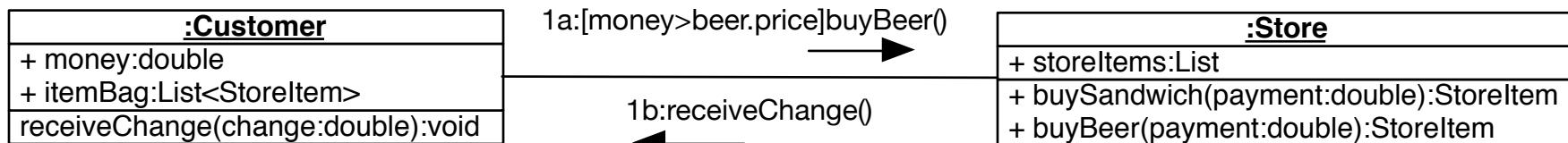
- Sequential Messages
- Conditional Messages
- Concurrent Messages

Message Examples

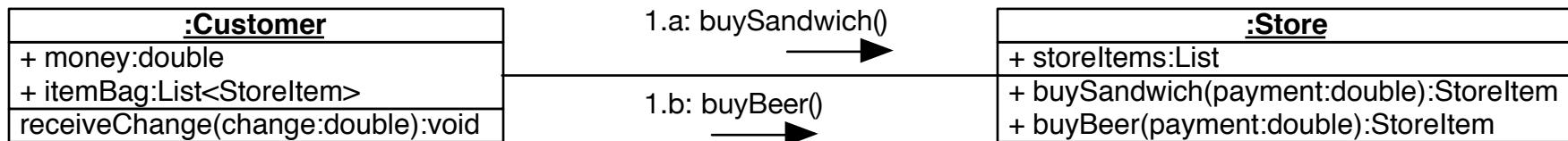
Sequential (blocking) Messages:



Conditional (blocking) Messages:



Concurrent (non-blocking) Messages:



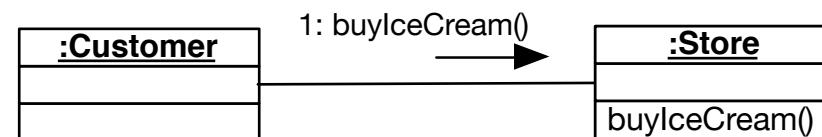
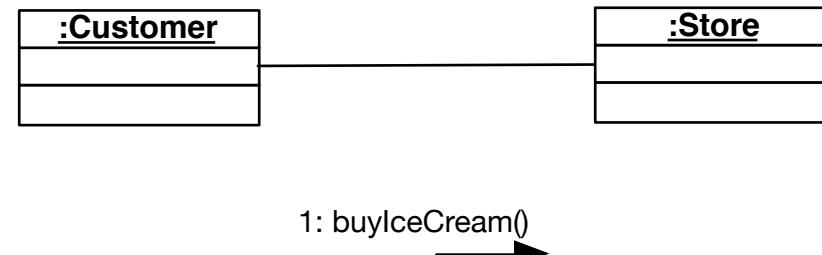
From Class diagrams to Communication diagrams

Actions:

1. Take all the steps from the event flow of a use case
2. Instantiate the participating objects
3. Number the messages from each of the steps of the event flow
4. Is there a corresponding method in the receiver of the message?
No: Refine your Class diagram by adding a public method to the receiver.
5. Draw the message from sender to receiver

Example:

"Customer buys ice cream from the Store."



Communication Diagrams vs Class Diagrams vs Sequence Diagrams

- Difference between communication diagrams and class diagrams:
 - Association labels, roles and multiplicities are not shown in communication diagrams. Associations between objects denote messages depicted as a labeled arrows that indicate the direction of the message, using a notation similar to that used on sequence diagrams
- Difference between communication diagrams and sequence diagrams:
 - Both focus on the message flow between objects
 - Sequence diagrams are good at illustrating the event flow over time. They can show temporal relationships such as causality and temporal concurrencies
 - Communication diagrams focus on the structural view of the communication between objects, not the timing.

Homework

- Use a UML communication diagram to model the dynamic behavior of your collision detection algorithm
- Details in Homework Sheet #4.

UML Notation Summary

1

UML provides a wide variety of notations for representing many aspects of software systems

2

UML can be used to deal with complexity, to communicate with others, and to generate code

3

UML is a programming language. It can become confusing when one uses too many features

4

Use case diagrams describe the **functional model**.
Class diagrams describe the **object model**
State charts, sequence diagrams, activity diagrams and communication diagrams describe the **dynamic model**.

Overview of Today's Lecture

UML Notations for dynamic Modeling

1. Sequence Diagrams
2. State Chart Diagrams
3. Activity Diagrams
4. Communication Diagrams

System Design (Today and 17 May 2018)

 The Scope of System Design

1. Design Goals and Trade-Offs
2. Subsystem Decomposition, Architectural Styles
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 (System startup & termination).

30 Minute Break



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:

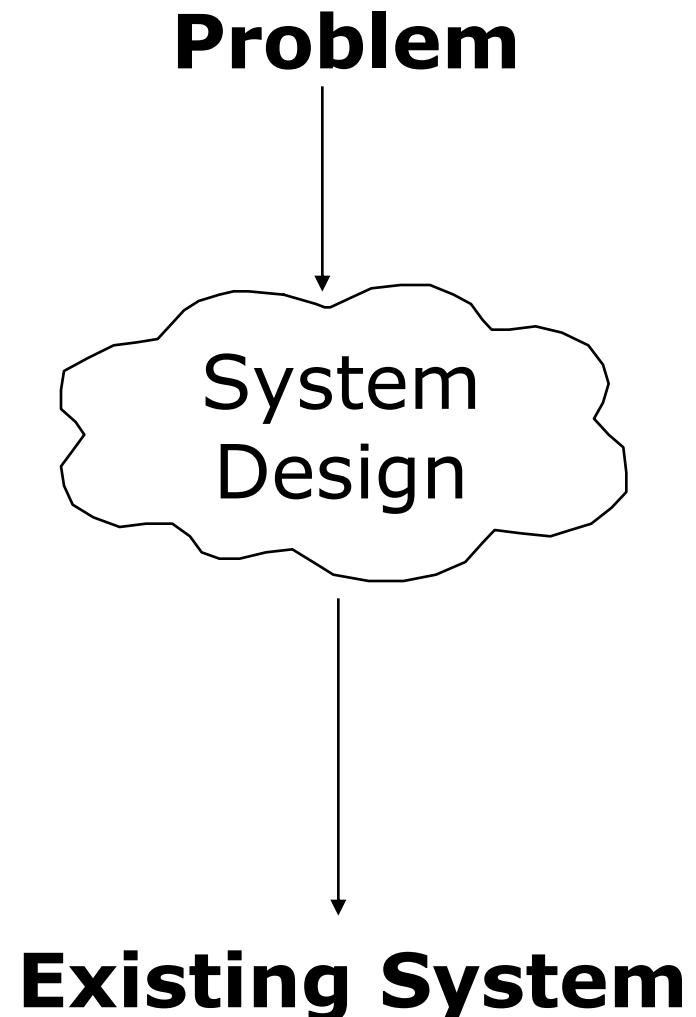
- Fiction which uses the device of fiction
- Best known for: „Sophie’s World“.

Why is Design so Difficult?

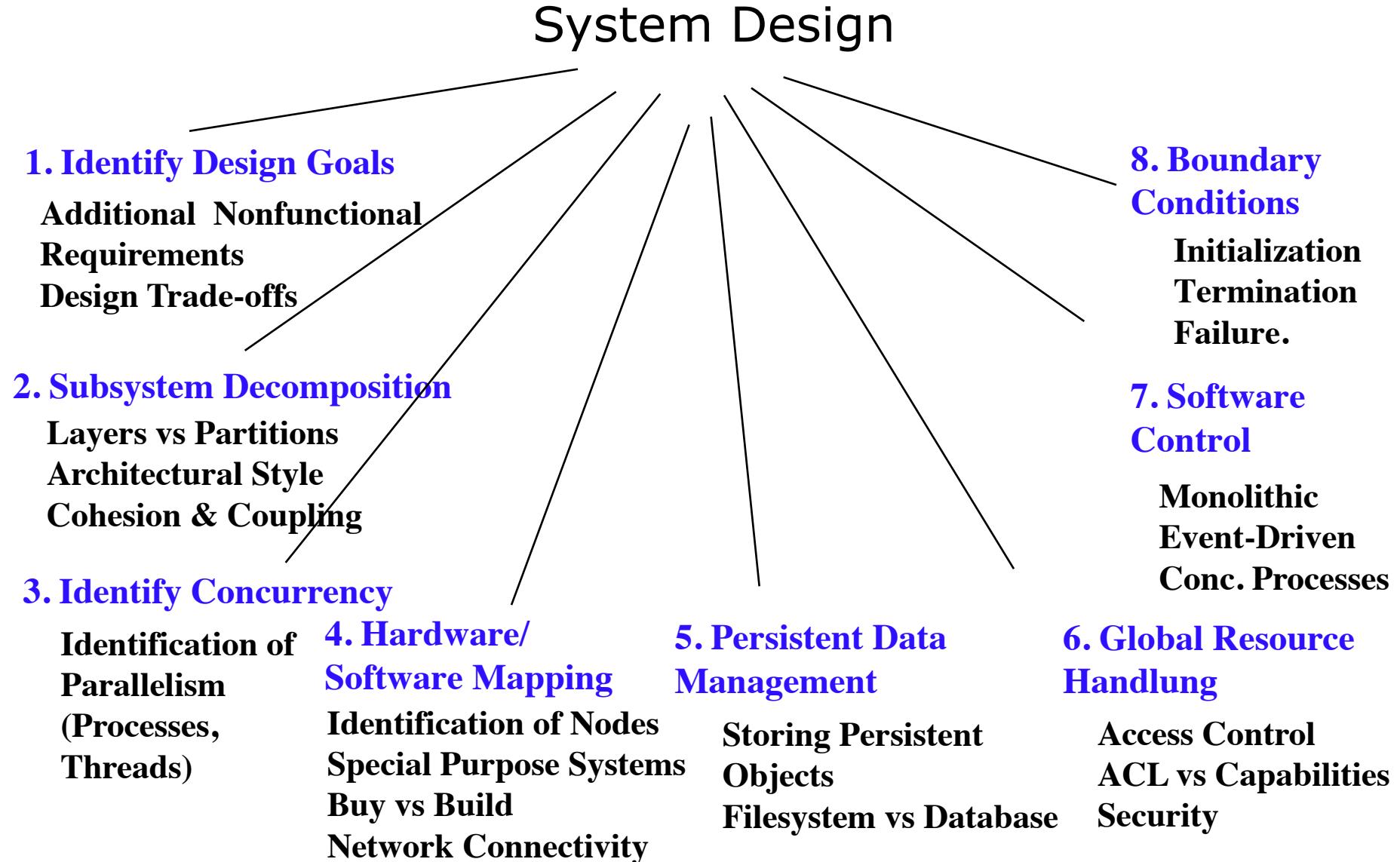
- **Analysis:** Focuses on the application domain
- **Design:** Focuses on the solution domain
 - The solution domain is changing very rapidly
 - Halftime knowledge in software engineering: About 3-5 years
 - Cost of hardware rapidly sinking
 - Design knowledge is a moving target
- **Design window:**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

The Scope of System Design

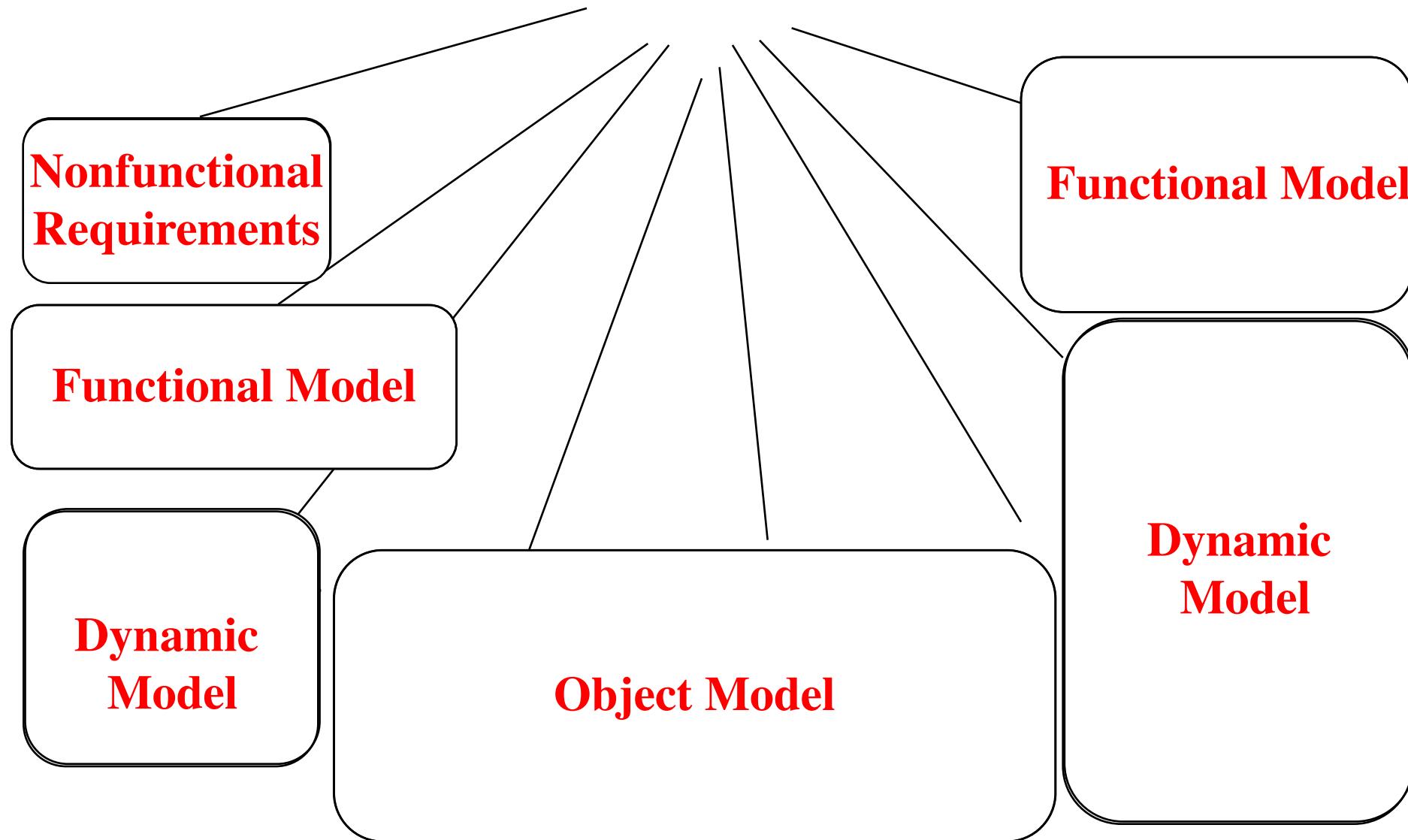
- Bridge the gap
 - between a problem and an existing system in a manageable way
- How?
- Use Divide & Conquer
 - 1) Identify design goals
 - 2) Model the new system design as a set of subsystems
 - 3-8) Address the major design goals.



System Design: Eight Issues



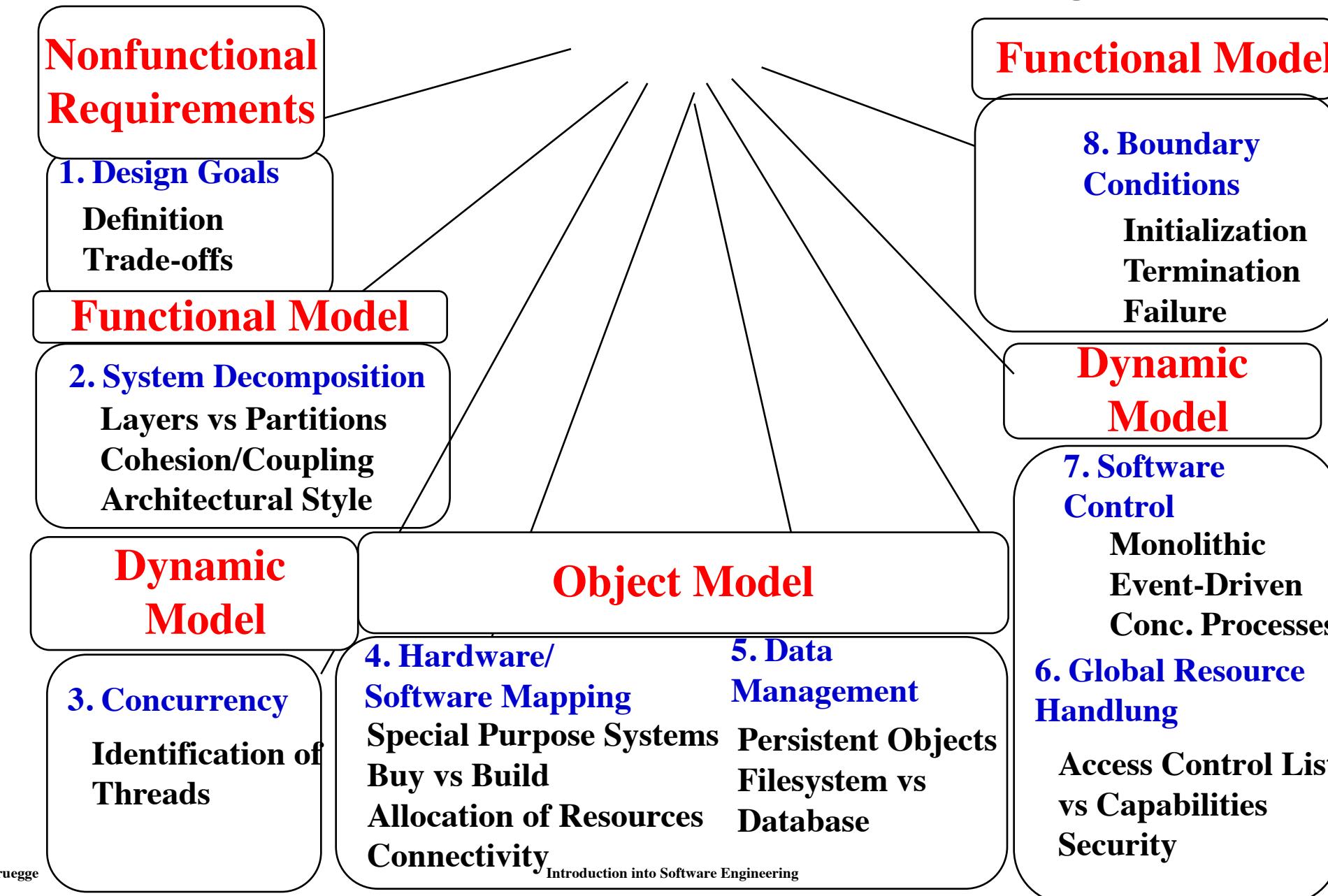
From Analysis to System Design



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



Overview of Today's Lecture

UML Notations for dynamic Modeling

1. Sequence Diagrams
2. State Chart Diagrams
3. Activity Diagrams
4. Communication Diagrams

System Design (Today and 17 May 2018)

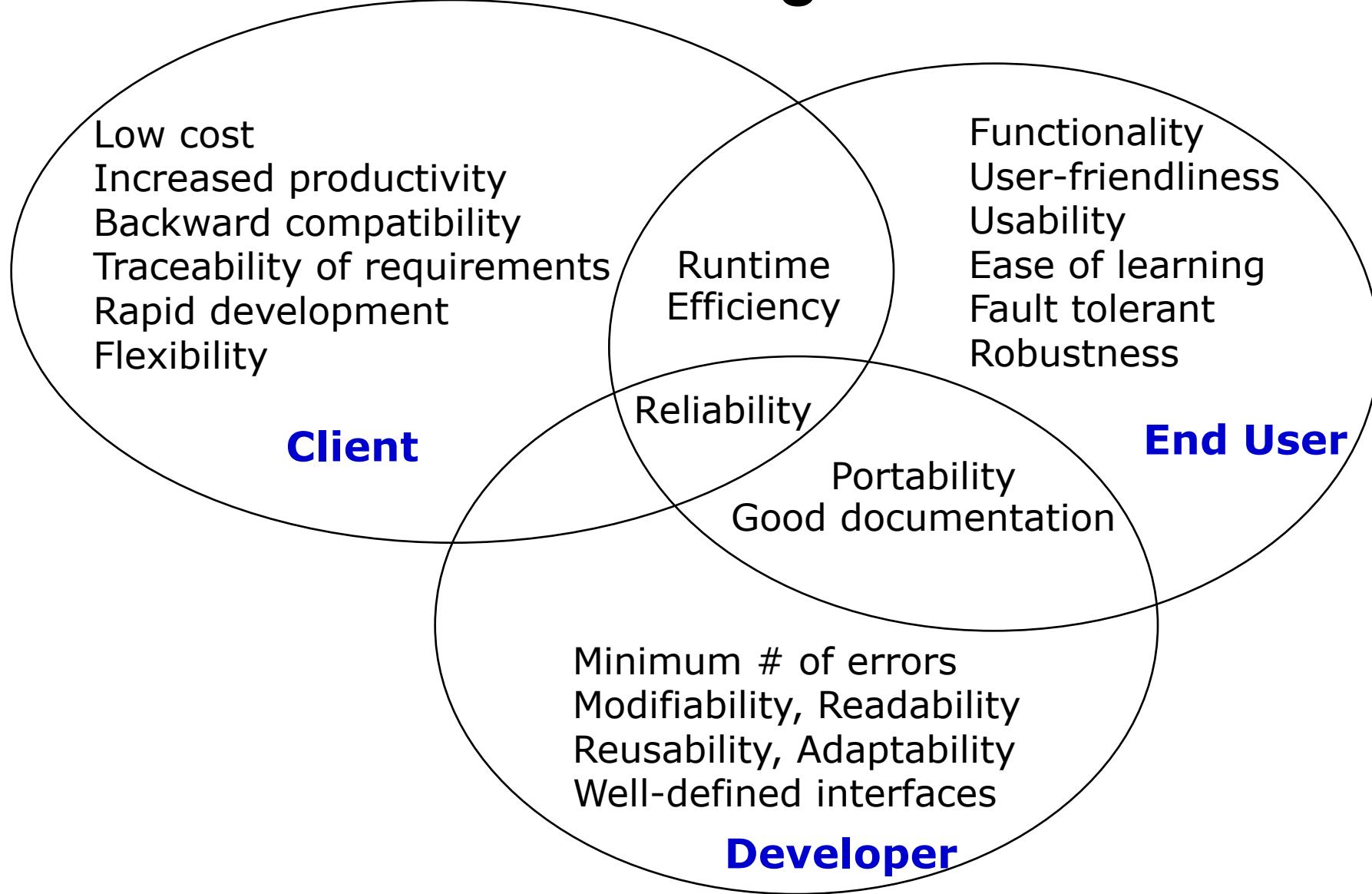
The Scope of System Design

- 
1. Design Goals and Trade-Offs
 2. Subsystem Decomposition, Architectural Styles
 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 (System startup & termination).

Definition: Design Goal

- Any Nonfunctional Requirement is a design goal (Model-based design)
- Design goals govern the system design activities
- Additional stakeholder goals are formulated with respect to
 - Design methodology
 - Design metrics
 - Implementation goals
- Design goals often conflict with each other
-> Typical Design Goal Trade-Offs.

Stakeholders have different Design Goals



Typical Design Goal Trade-offs

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



ToonClips.com

#5401

service@toonclips.com

Functionality v. Usability

Is a system with 100 functions usable?



Functionality v. Usability

Is a system with 100 functions usable?

Some systems are not even usable with two functions:



Cost v. Robustness

A low cost design does not check for errors when the user is entering wrong data

Ebay, early version from 2005

Artikelbezeichnung: BUND 166 ** TOPMARKEN ANSEHEN LOHNT

Aktuelles Gebot: EUR 5,51

Ihr Maximalgebot: EUR (Geben Sie mindestens EUR 6,01 ein. Eingabe bitte ohne 1000er-Trennzeichen, z.B. 1000,00.)

Weiter > Die Bestätigung erfolgt im nächsten Schritt.

eBay bietet automatisch für Sie mit bis zur Höhe Ihres **Maximalgebots**.
[Mehr zum Thema Bieten.](#)

Cost v. Robustness

A low cost design does not check for errors when the user is entering wrong data



Ebay, early version from 2005

Bieten

Artikelbezeichnung: BUND 166 ** TOPMARKEN ANSEHEN LOHNT

Aktuelles Gebot: EUR 5,51

Ihr Maximalgebot: EUR (Geben Sie mindestens EUR 6,01 ein. Eingabe bitte ohne 1000er-Trennzeichen, z.B. 1000,00.)

Weiter > Die Bestätigung erfolgt im nächsten Schritt.

eBay bietet automatisch für Sie mit bis zur Höhe Ihres **Maximalgebots**.
[Mehr zum Thema Bieten.](#)

Artikelbezeichnung: BUND 166 ** TOPMARKEN ANSEHEN LOHNT

Ihr Maximalgebot: EUR 851,00

Verpackung und Versand: Genaue Angaben finden sie auf der Artikelseite. Oder wenden Sie sich an den Verkäufer
Zahlungsmethoden: Überweisung, Andere - Siehe Zahlungshinweise.

Ihr Gebot ist bindend. Bestätigen Sie Ihr Gebot daher bitte nur, wenn Sie den Artikel auch wirklich kaufen möchten.

Bitte warten...

Efficiency v. Portability

What are the advantages and disadvantages of a static vs mobile radar detector?



Rapid development v. Functionality

- Let's say your development time is 5 weeks, you have 5 programmers, your design window is 2 weeks, after design 3 programmers are leaving your company, and your delivery deadline cannot be moved
- You are going to reduce the functionality. Not all the use cases in your model can be implemented nor delivered. (e.g. an emergency brake..)



Cost v. Reusability

- Assume you model the association between 2 classes with an 1-1 multiplicity
 - Easy to code, low cost tests, not very reusable
- Moving from a 1-1 association to a many-many association
 - Additional coding and testing costs



- With design patterns, this trade-off is no longer that painful. You can get reusability with low cost if you use design patterns!

More in the Lecture
about Design Patterns

Backward Compatibility v. Readability

Sometimes it is not easy to achieve backward compatibility e.g. due to different hardware.

One Solution:

```
#Ifdef OldSystem then WriteToPaperTapeWriter  
#elseif Newsystem then WritetoCD-R  
#elseif Newersystem then SendMessageToBLE
```

-> Using design patterns, it is now possible to maintain backwards capability while keeping the code readable!

Overview of Today's Lecture

UML Notations for dynamic Modeling

1. Sequence Diagrams
2. State Chart Diagrams
3. Activity Diagrams
4. Communication Diagrams

System Design (Today and 17 May 2018)

The Scope of System Design

1. Design Goals and Trade-Offs
2. Subsystem Decomposition, Architectural Styles 
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 (System startup & termination).

Definitions: Subsystem and Service

- **Subsystem**
 - Collection of classes, associations, operations, events that are closely interrelated with each other
 - The classes in the object model are the “seeds” for subsystems
- **Service**
 - A group of externally visible operations provided by a subsystem (also called **Subsystem interface**)
 - The use cases in the functional model provide the “seeds” for services.

Subsystem Interface

- **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.*

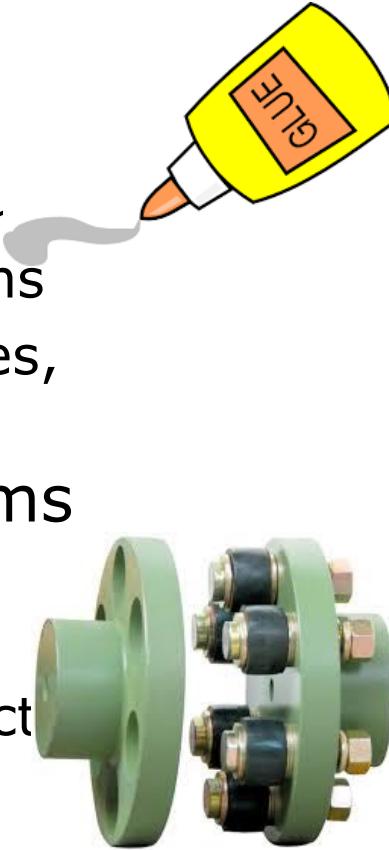
Subsystem Interface Object

Subsystem Interface Object

- The set of public operations provided by a subsystem
- The subsystem interface object describes *all* the services of the subsystem interface
- A subsystem interface object can be realized with the Façade pattern
=> Design Patterns are covered in the Lectures on Object Design.

Coupling and Cohesion of Subsystems

- Goal: Reduce system complexity while allowing change
- **Cohesion** measures dependency among classes
 - **High cohesion:** The classes in the subsystem perform similar tasks and are related to each other via many associations
 - **Low cohesion:** 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 Cohesion of Subsystems

Good System Design

- Goal: Reduce system complexity while allowing change
- **Cohesion** measures dependency among classes
 - **High cohesion:** The classes in the subsystem perform similar tasks and are related to each other via many associations
 - **Low cohesion:** 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

How to achieve high Cohesion

- High cohesion can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Question to ask:
 - Does one class from one subsystem always call another class from another subsystem for a specific service?
 - Yes: Consider moving the classes into one subsystem.

How to achieve Low Coupling

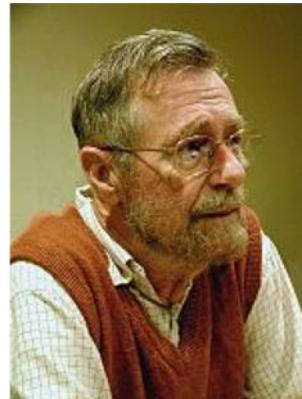
- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (Principle of information hiding, Parnas)
- Questions to ask:
 - Can we make it possible that the calling class calls only operations of the lower level classes?
 - Does the calling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

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



Dijkstra's Idea to reduce High Coupling

- Dijkstra's revolutionary idea in 1968
 - A system should be designed and built as a hierarchy of layers: Each layer uses only the services offered by the lower layers
- The T.H.E. system ☺
 - T.H.E. = Technische Hochschule Eindhoven
- An operating system for single user operation
 - Supporting batch-mode
 - Multitasking with a fixed set of processes sharing the CPU.



Edsger W. Dijkstra, 1930-2002
Formal verification: Proofs for programs
Dijkstra Algorithm, Banker's Algorithm,
Gotos considered harmful, T.H.E.,
1972 Turing Award

Overview of Today's Lecture

UML Notations for dynamic Modeling

1. Sequence Diagrams
2. State Chart Diagrams
3. Activity Diagrams
4. Communication Diagrams

System Design (Today and 17 May 2018)

The Scope of System Design

1. Design Goals and Trade-Offs
2. Subsystem Decomposition → Architectural Styles
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 (System startup & termination).

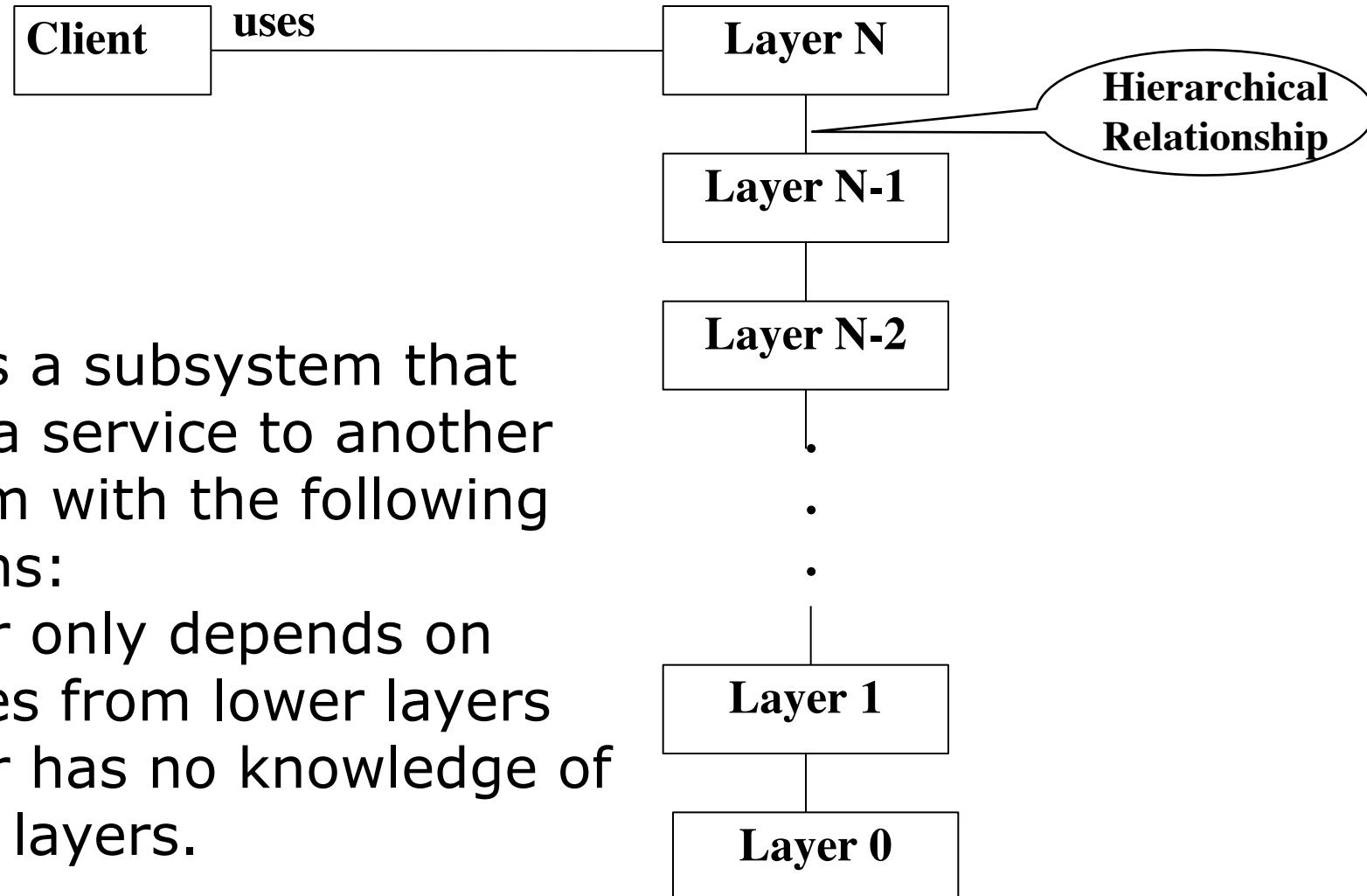
Architectural Style vs Architecture

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

Example: The Layered Architectural Style

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.

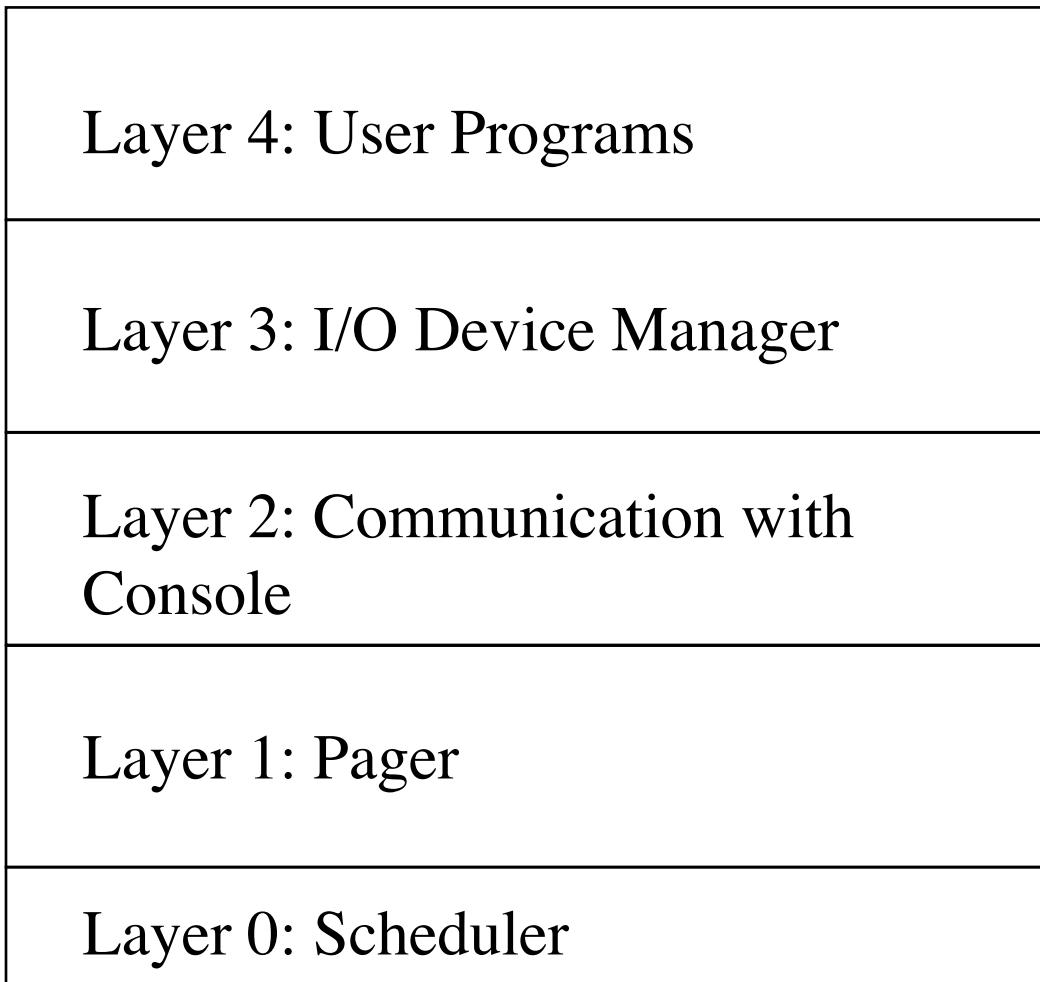


Hierarchical Relationships between Layers

- There are two major types of hierarchical relationships between layers
 - Layer A “depends on” layer B for its full implementation (compile time dependency)
 - Also called **usage dependency** in UML
 - Example: Build dependencies (make, ant, maven)
 - Layer A “calls” layer B (runtime dependency)
 - Example: A web browser calls a web server

The Layers of the T.H.E. System

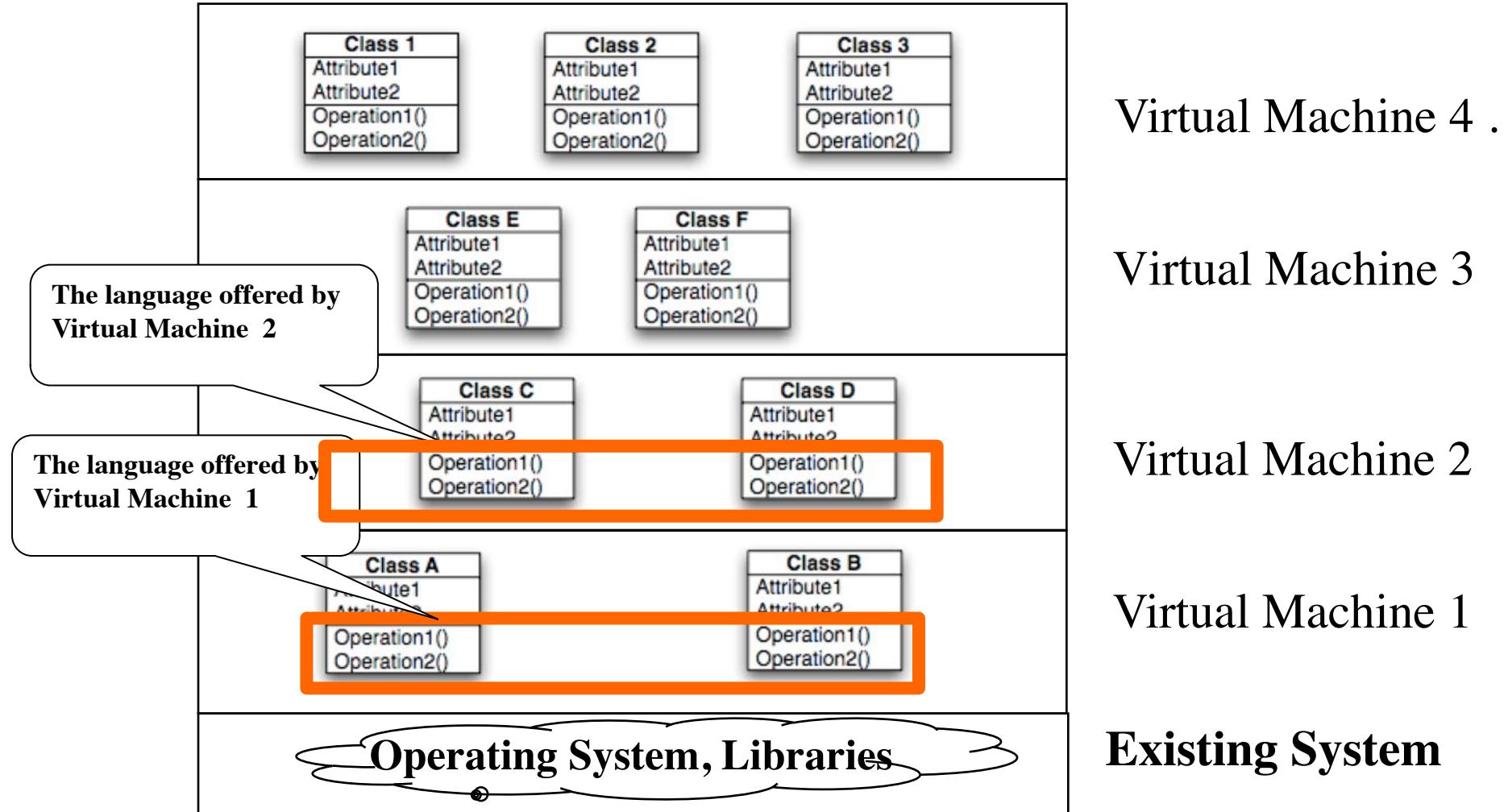
Dijkstra: “An operating system is a hierarchy of layers, each layer using services offered by the lower layers”



Retrospectively,
T.H.E was the first
system that used an
architectural style.

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



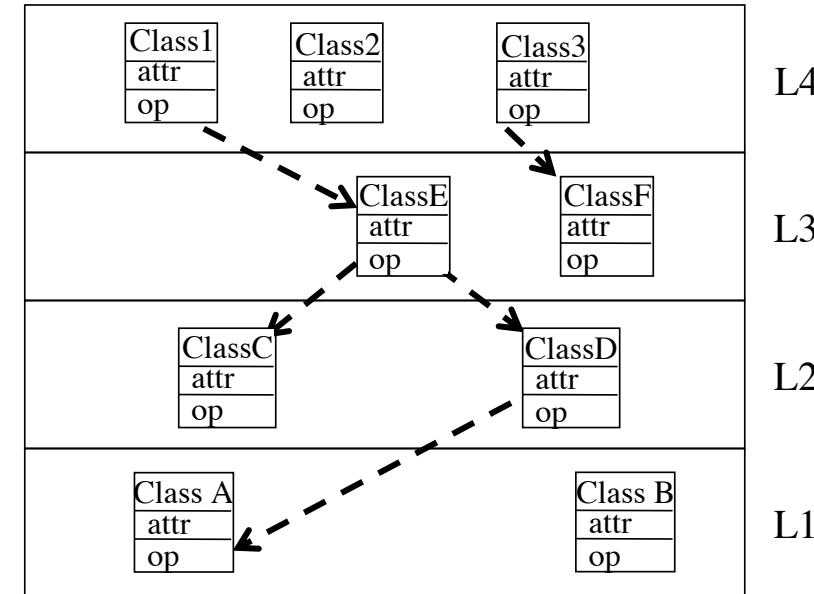
Virtual Machine

- 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 consists of a set of attributes and operations
- The terms **layer** and **virtual machine** can be used interchangeably
 - Also called "level of abstraction".

Closed Architecture (Opaque Layering)

- A **layered architecture is closed**, if each layer can only call operations from the layer below (called “direct addressing” by Buschmann)

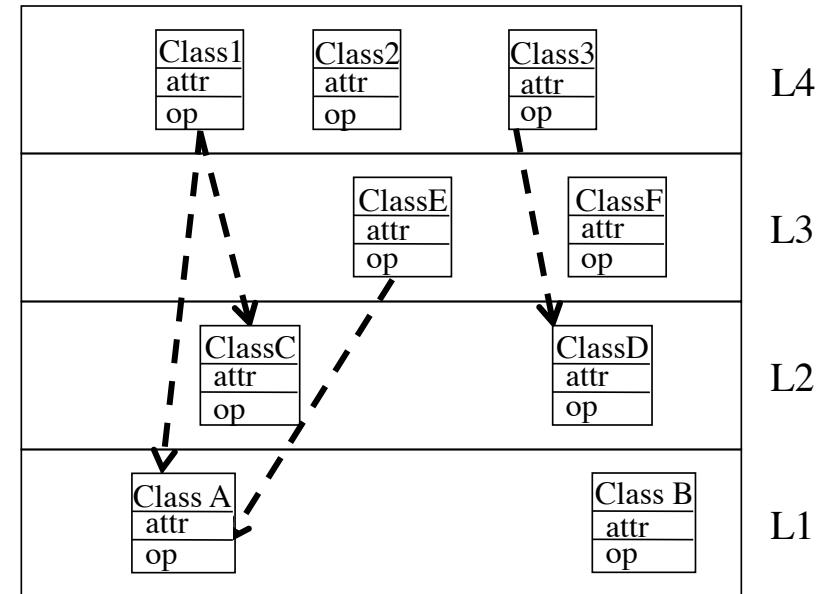
Design goals: Maintainability, flexibility.



Open Architecture (Transparent Layering)

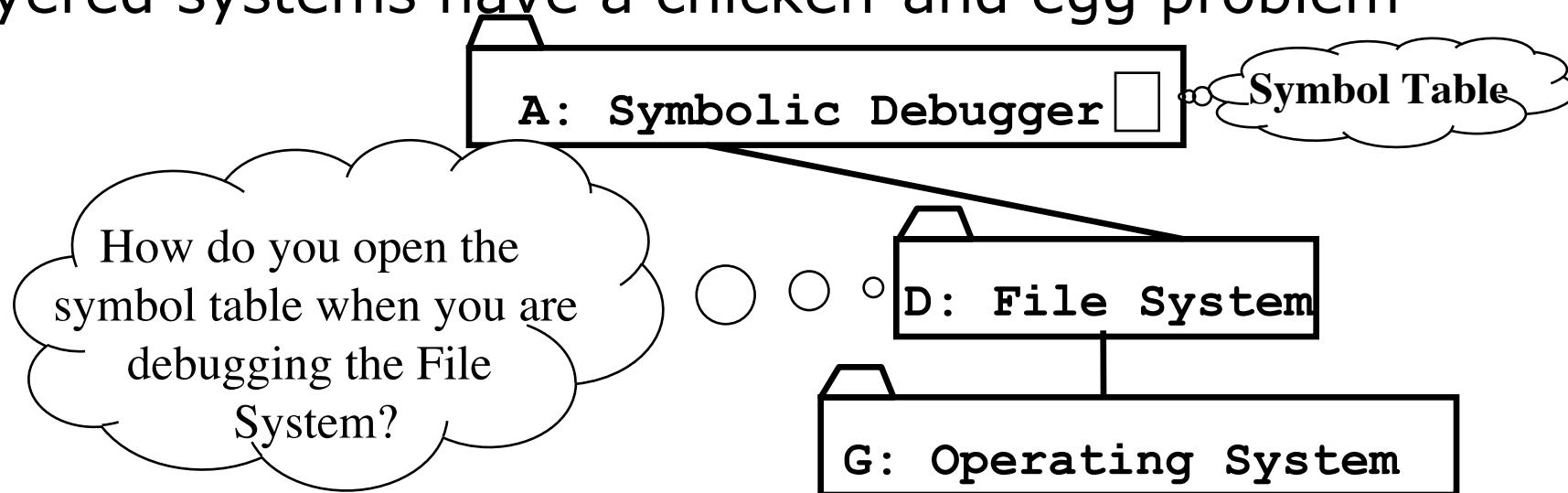
- A **layered architecture** is open if a layer can call operations from any layer below (called “**indirect addressing**” by Buschmann”)

Design goal:
High Performance.



Properties of Layered Systems

- Layered systems are hierarchical. This is desirable:
 - Hierarchy reduces complexity
- Closed architectures are more portable
 - Low coupling
- Open architectures are more efficient
 - High coupling
- Layered systems have a chicken-and egg problem



SOA is a Closed Layered Architecture

SOA = Service Oriented Architecture

- Basic idea: A service provider ("business") offers **Business Services** ("business processes") to a service consumer (application, "customer")
 - The business services are dynamically discoverable, usually offered in **Web-based Applications**
- The business services are created by composing (choreographing) them from lower-level services (**Basic Services**)
- The basic services are usually based on **Legacy Systems**
- **Adapters** are used to provide the "glue" between basic services and the legacy systems.



Web-based Applications

Business Services (Composite Services)

Basic Services

Adapters to Legacy Systems

Legacy Systems

Adapters will be covered in
Lecture 8 (June 14)

Layers vs Tiers

Definition: 3-layered architectural style

- An architectural style, where an application consists of 3 hierarchically ordered layers

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.

3-Layered Architectural Style

- 3-layered architectural styles are often used for the development of Web applications
- Example of a 3-tier architecture:
 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.

4-Layered Architectural Style

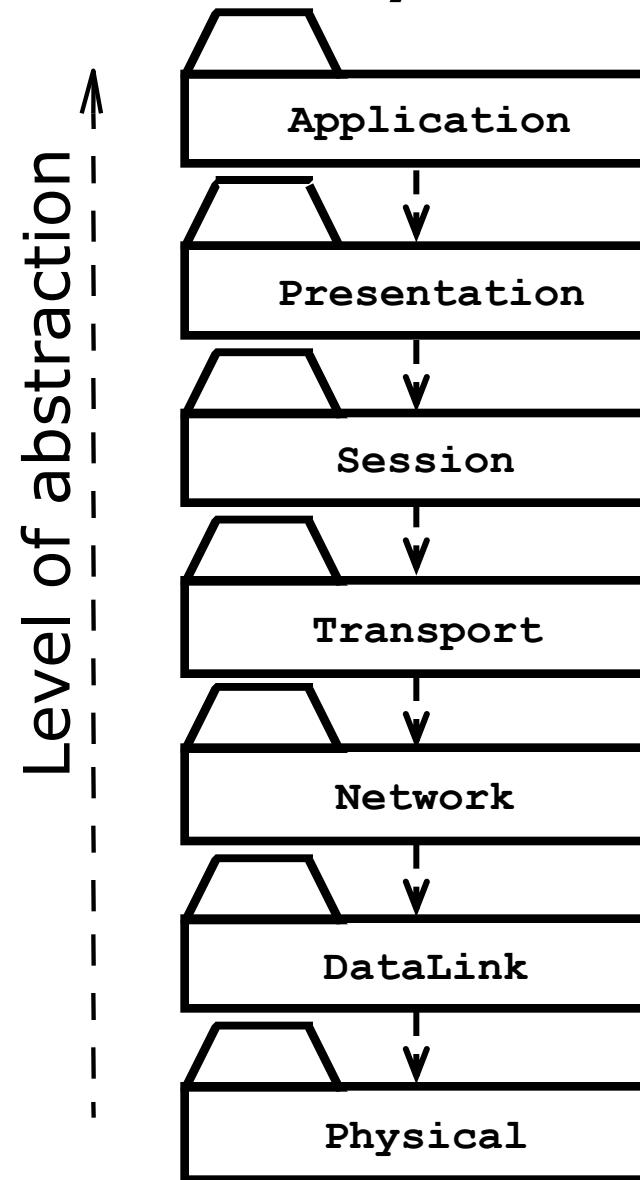
4-layer-architectural style: An architectural style, where an application consists of 4 hierarchically ordered layers

Example of 4-tier architecture:

1. A **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 commercially available 4-tier architectures, this is usually a relational database management system (RDBMS).

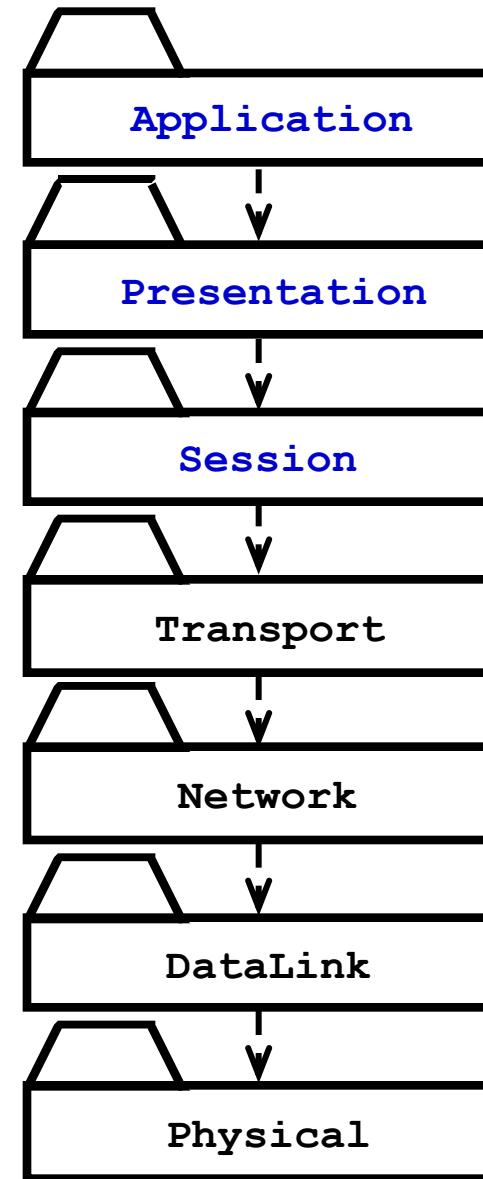
Example of a 7-Layered Architectural Style

- 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



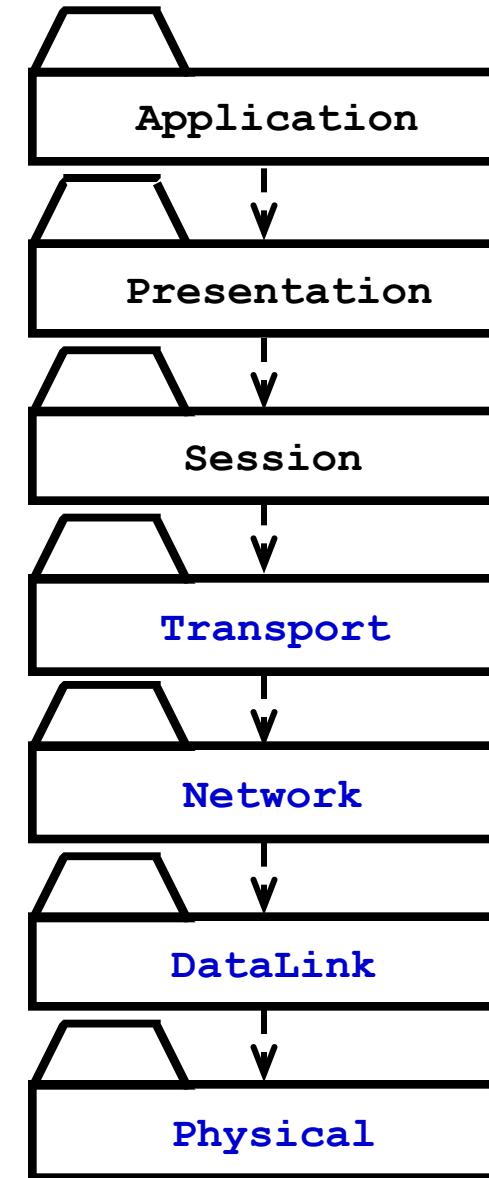
OSI Model Layers and their Services

- 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



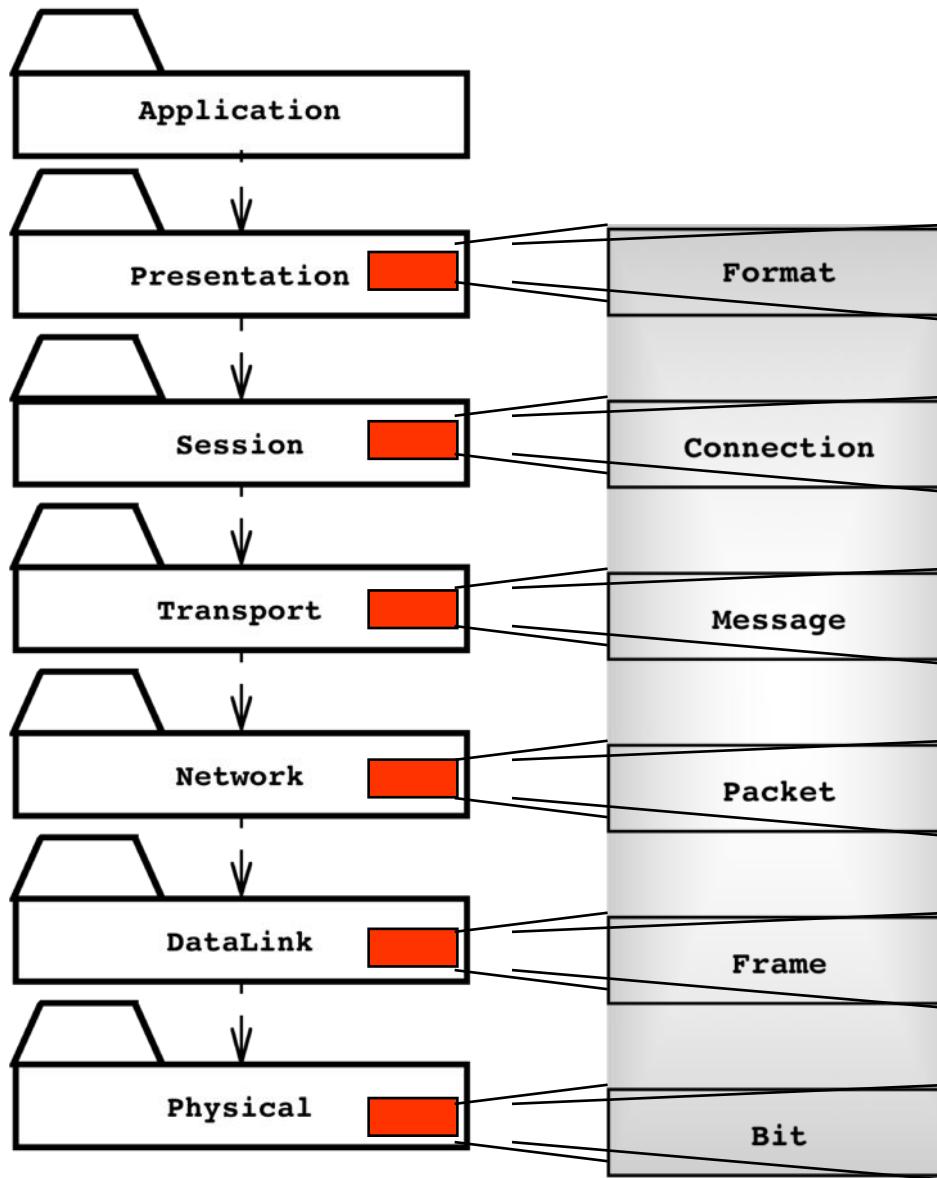
OSI Model Layers and their Services

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

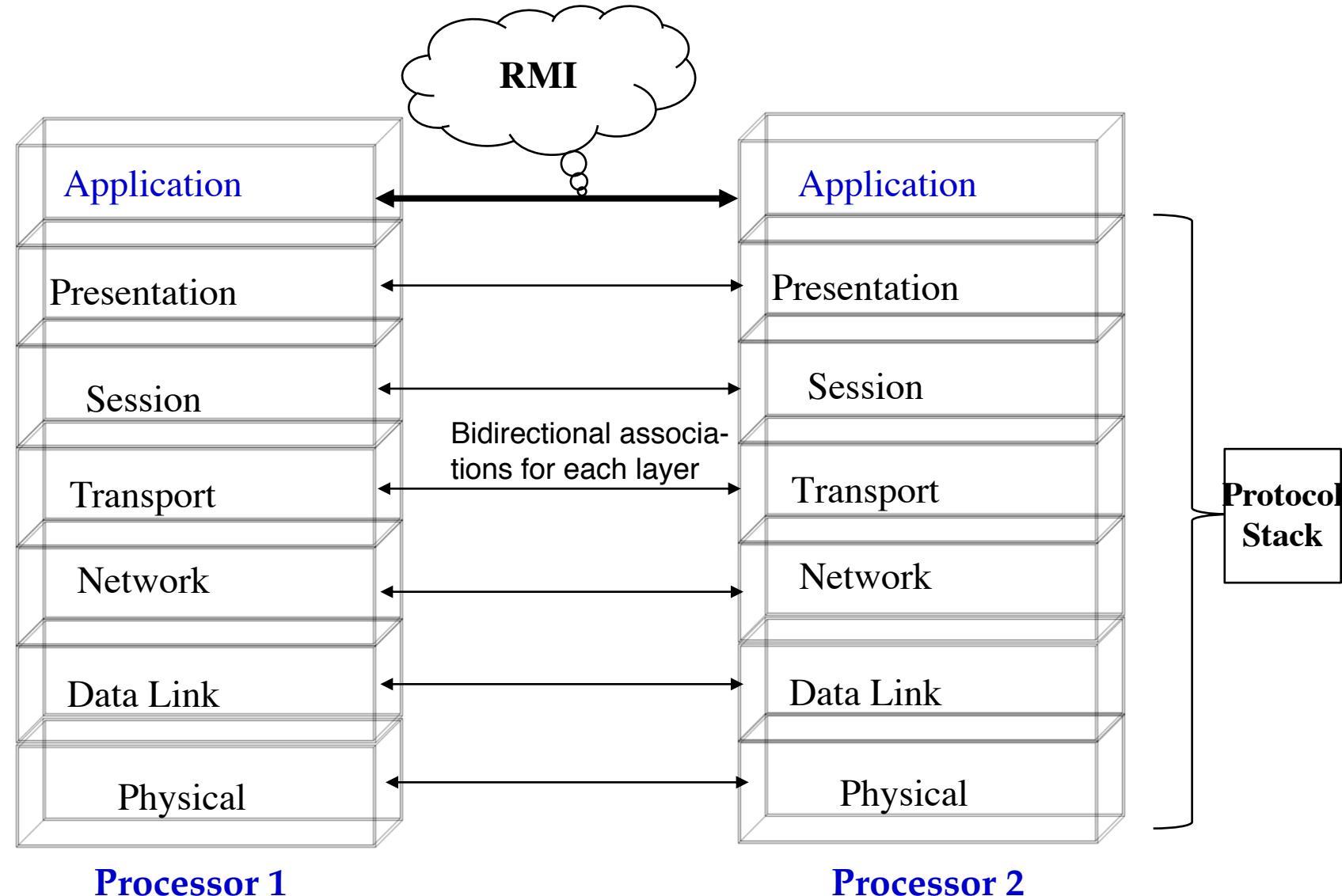


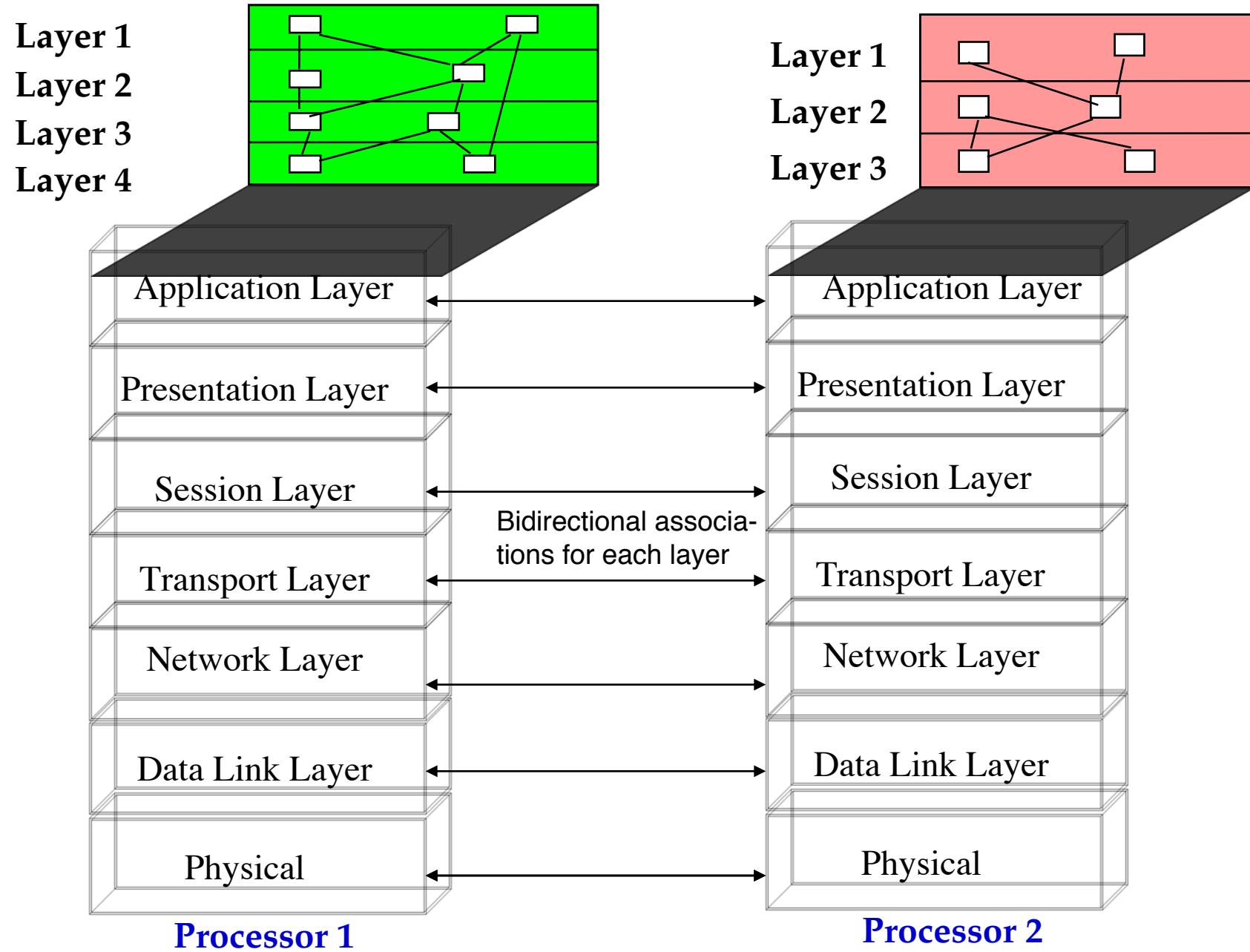
An Object-Oriented View of the OSI Model

- 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 offering public operations for the layer above



The Application Layer provides the Abstractions of the “New System”. It is usually layered itself





5 System Design steps to create a layered architecture

1. Define the abstraction criterion

- Also called "the conceptual distance to the existing system ("platform"). Examples of abstraction criteria:
 - The degree of customization for a specific domain
 - The degree of conceptual complexity

2. Determine the number of abstraction levels

- Each abstraction layer corresponds to one layer of the pattern

3. Name the layers and assign tasks to each of them

- The task of the highest layer is the overall system task, as perceived by the client. The tasks of all the other layers are helper layers. (The lower layers provide the infrastructure needed by the higher layers)

4. Specify the services

- Lower layers should be "slim", while higher layers can cover a broader spectrum of applicability. Also called the "inverted pyramid of reuse"

5. Refine the layering

- Iterate over steps 1 to 4.

Summary

- We introduced UML notations to support dynamic modeling
 - UML sequence diagrams to model the dynamic behavior of an **event flow**
 - UML communication diagrams to model the dynamic behavior of **class diagram**
 - UML state chart diagrams to model the dynamic behavior of a **single class**
- We started with System Design
 - System design reduces the gap between a problem and an existing machine
 - **Design goals** are identified at design time
- Design Goals describe important system qualities
 - **Design trade-offs** can be used to evaluate alternative designs
- Subsystem Decomposition
 - Partitioning a system into manageable parts using of **coupling** and **cohesion**
- First introduction into Architectural Styles and Architectures.

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
- 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
- Frank Buschmann et al:
 - Pattern-Oriented Software Architecture, Vol 1: A System of Patterns, Wiley, 1996
- Judith Hurwitz et. al,
 - Service Oriented Architecture for Dummies, IBM, 2nd edition, online:
ftp://ftp.software.ibm.com/software/cn/soa/lauch/SOA_for_dummies.pdf

Morning Quiz 05

- Start Time: **8:00**
- End Time: **8:10**
- To participate in the quiz, use ArTEMiS
- Click on Open Quiz or Start Quiz
- The Lecture starts at 8:10

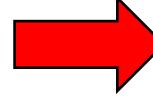
Remaining Time: **46 s**

Saved: never

● Connected

Submit

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	Start exercise
Good Morning Quiz 05		Open Quiz 

Only click on Submit when you have entered all answers!



System Design II: Addressing Design Goals

Bernd Bruegge

Chair for Applied Software Engineering
Technische Universität München

17 May 2018

Roadmap for Today's Lecture

- **Context and Assumptions**

- We completed Chapter 1 to 5 in the OOSE text book by Bruegge and Dutoit

- **Content of this lecture**

- We finish Lecture 05: System Design I (Slide 86-96)
- We introduce how to address design goals
- We complete Chapter 6

- **Objective:** At the end of this lecture you are able to

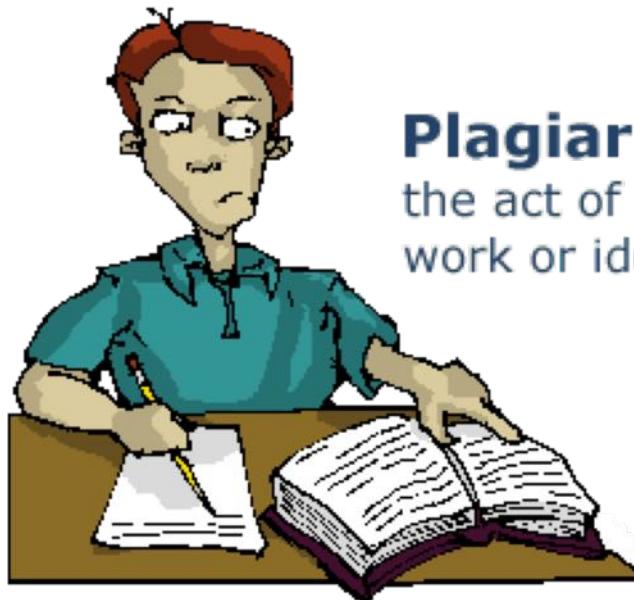
- Understand how nonfunctional requirements impact the design goals
- Choose between different architectural styles based on the system model and the nonfunctional requirements
- Deal with 2 more UML Notations: Component Diagram and Deployment Diagram.

Outline of this lecture

- Miscellaneous
- Decomposition
- Architectural Styles
 - Layered Architecture (Finish Lecture 04 from May 3, 2018), Review: Cohesion and Coupling
 - Client-Server
 - Model-View Controller
 - Repository
 - Pipes-and Filers
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

Plagiarism

- We identified students who copied their homework solutions word by word
- If you plagiarize and we identify this, you will not get a grade bonus any more



Plagiarism:

the act of presenting another's work or ideas as your own.

Review: Mapping the Bonus Points to Grade Improvements

- Students can improve their **final** exam grade with a bonus:
 - 20% regular participation in the morning quiz
 - 20% in-class exercises during the lecture
 - 60% homework submissions
- We will sum up all points and map them to the following **grade improvements**

Total Points	Bonus
0% <= Points < 30%	0.0
30% <= Points < 60%	0.3
60% <= Points < 90%	0.7
90% <= Points <= 100%	1.0

- **Please note:**
 - You have to present your homework at least 2 times in your tutor group (otherwise you will not get a bonus)
 - You **cannot** use the bonus for the repeat exam
 - You must pass the final exam to apply the bonus
 - Your final grade cannot be better than 1.0

Best Collision Strategies in Bumpers Sprint 03

Tutor Group	Student	Collision	Description
Di 03	Lukas Krimphove	Tank Collision	Tanks have to be hit from the right while other cars have to be hit from the left in order to win the collision
Do 01	Alexander Kranzer	Default + Tank Collision	If any of the cars collides with a tank, tank collision is applied → tank always wins If the player collides with cars other than tank, default collision is applied
Mi 05	Fabian Braun	Speed Collision	When the player wins a collision, the player car's speed is incremented and decremented in case he loses the collision. The car with higher speed wins a collision.
Mo 08	Raphael Schmid	Harder Collision	- Player has 8 lives - AI cars have one each - Collision selects the winner randomly - You need luck to win the game
Mo 12	Atanas Yordanov	Collision Right	- Based on right collision - Different cars can be hit multiple times - When hit often enough/too often, the car is crunched

Outline of this lecture

- Miscellaneous
- Decomposition
- Architectural Styles
 - Layered Architecture (Finish Lecture 04 from May 3, 2018)
 - Client-Server
 - Model-View Controller
 - Repository
 - Pipes-and Filers
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

There are 3 Ways to deal with Complexity

- Three ways to deal with complexity
 - ✓ Abstraction
 - ✓ Hierarchy (Taxonomies, Aggregation, Composition)
- Decomposition

Decomposition

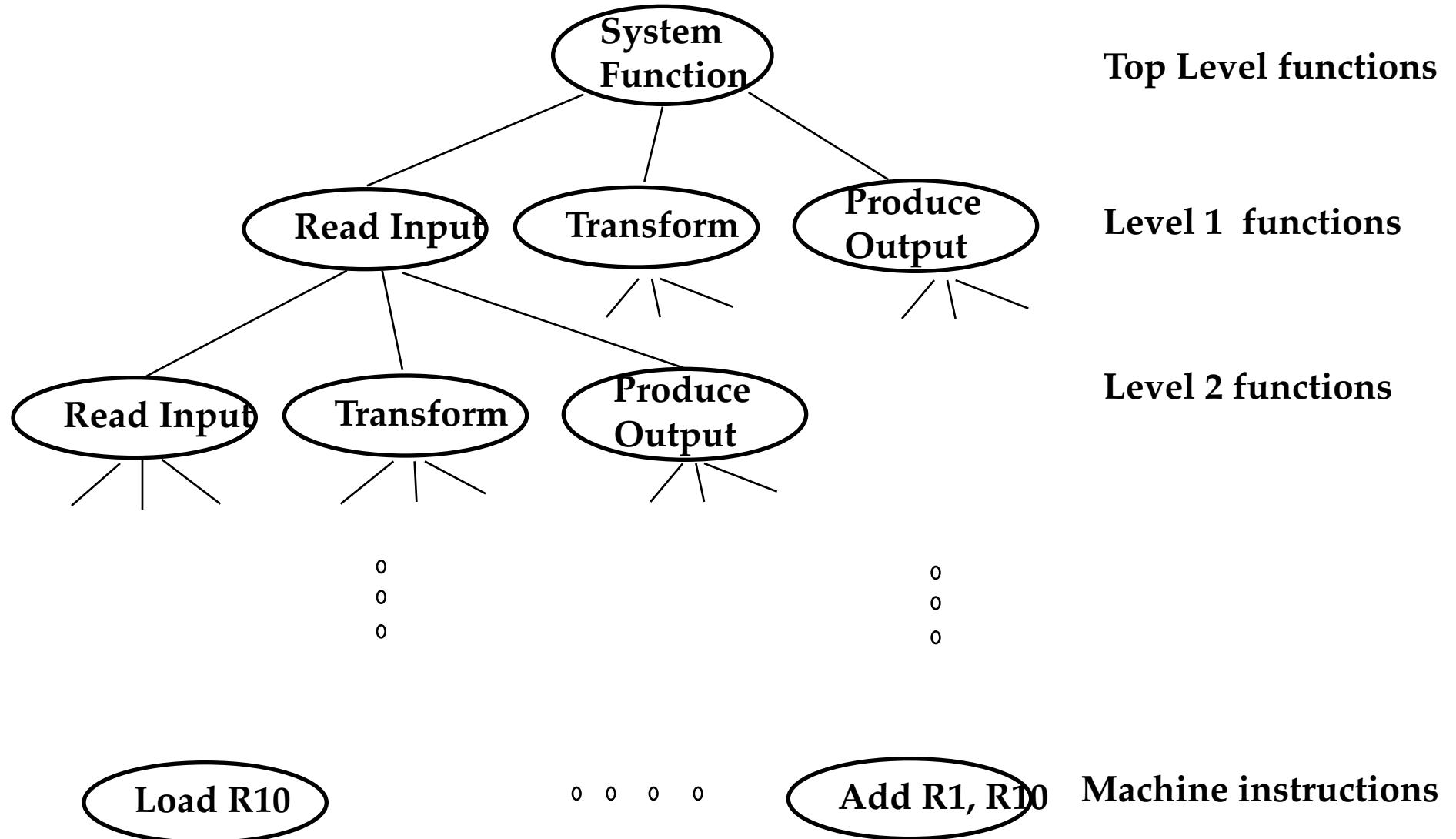
- A technique used to master complexity ("divide and conquer")
- Two major types of decomposition
 - Functional decomposition
 - Object-oriented decomposition.

Functional vs Object-oriented decomposition

- Functional decomposition
 - The system is decomposed into functions
 - Functions can be decomposed into smaller smaller functions
- Object-oriented decomposition
 - The system is decomposed into classes (“objects”)
 - Classes can be decomposed into smaller classes.

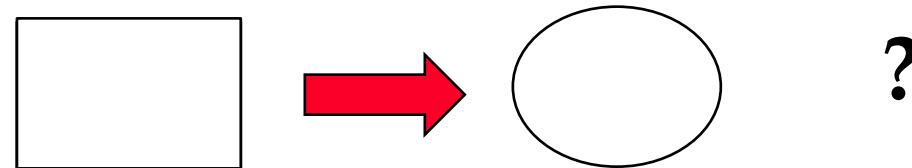
Which decomposition is the right one?

Functional Decomposition

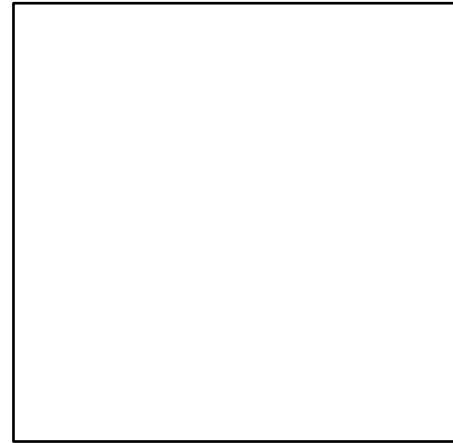


Functional Decomposition

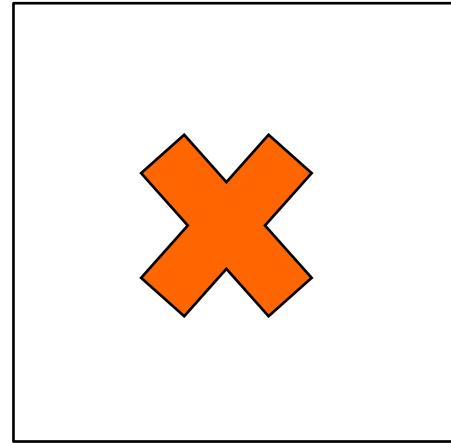
- Example: Graphics Program
 - How do I change a square into a circle?



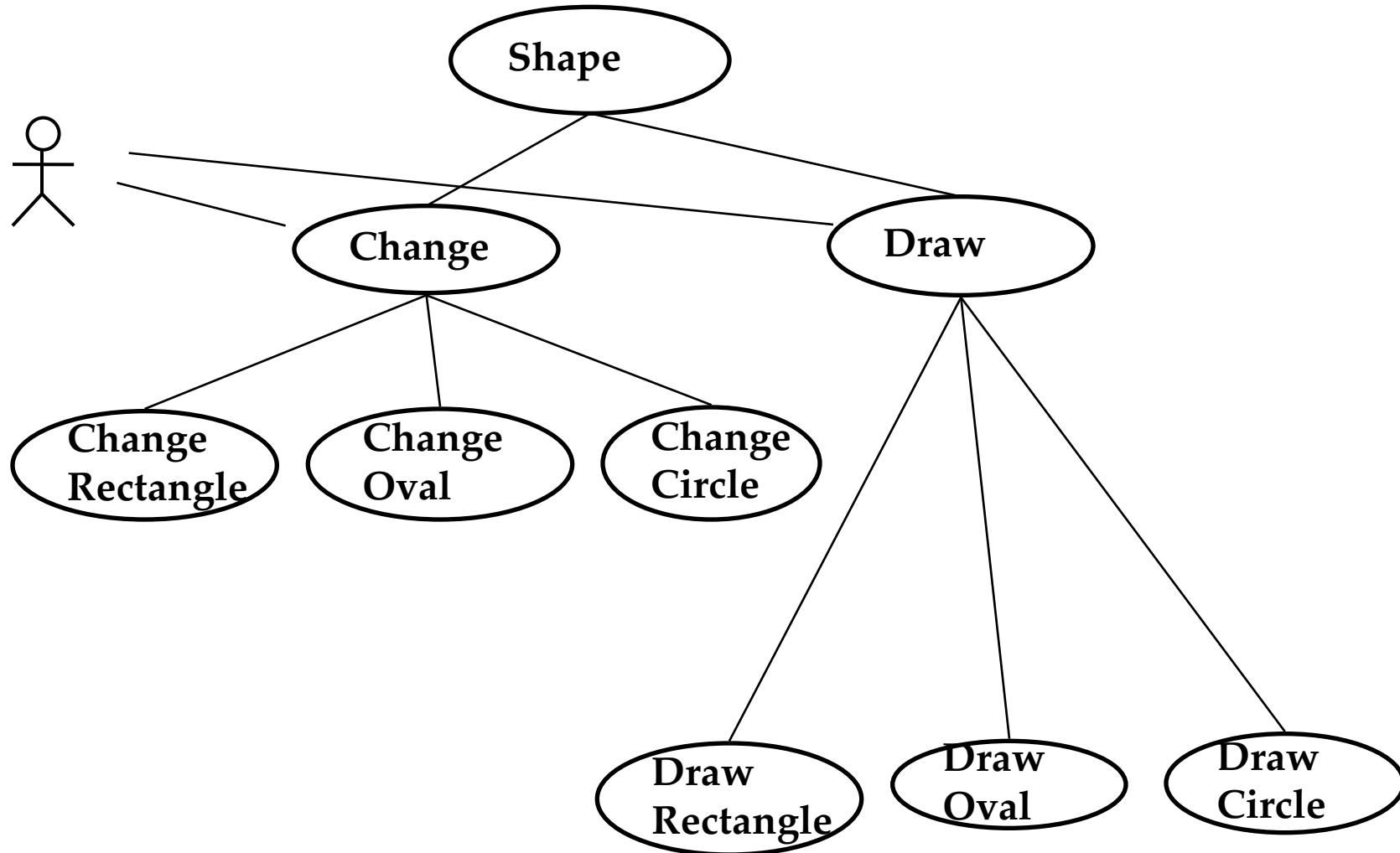
First Attempt: Right Click on Rectangle



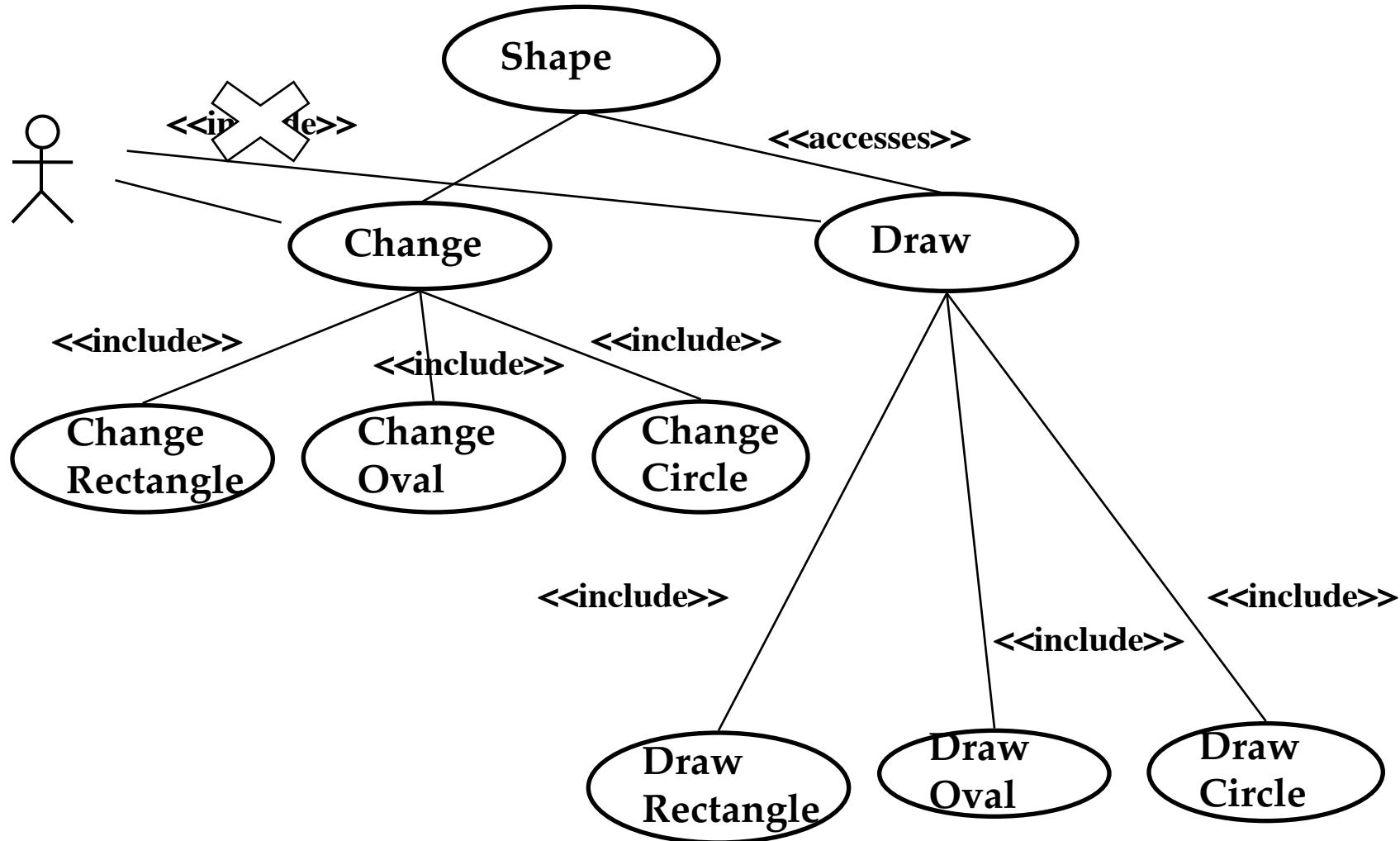
Second Attempt😊



Functional Decomposition: Shape System



Clarification about the <<include>> Stereotype



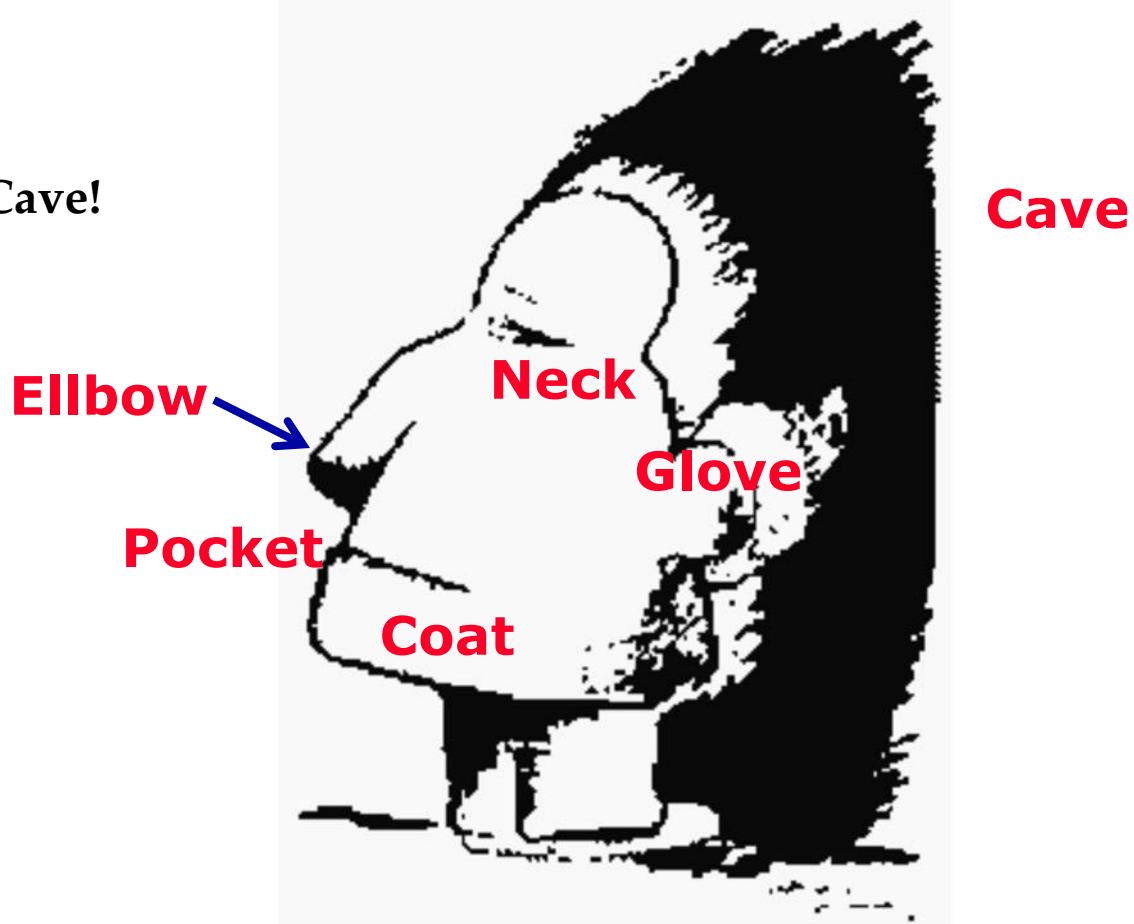
Functional Decomposition

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

What is This?

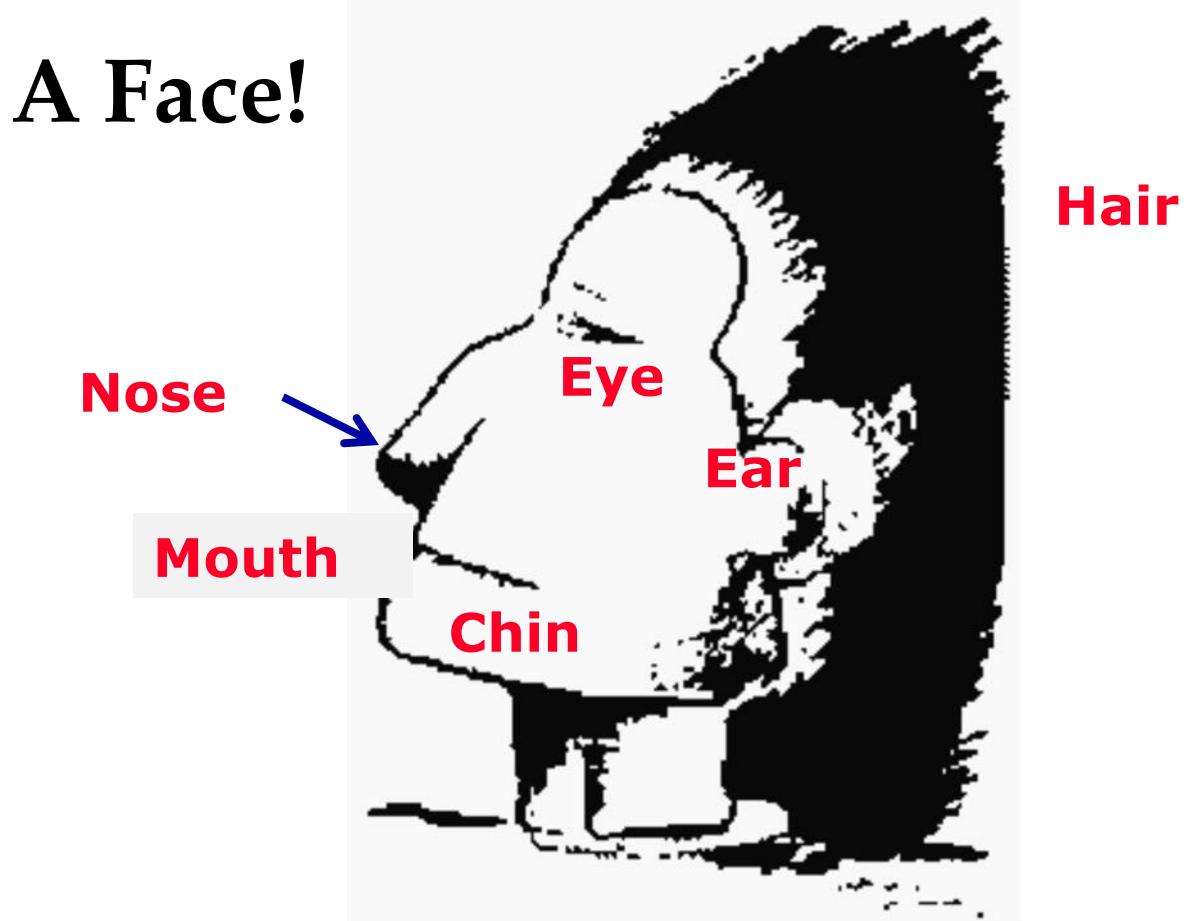
Let's try object-oriented decomposition

An Eskimo
Entering a Cave!

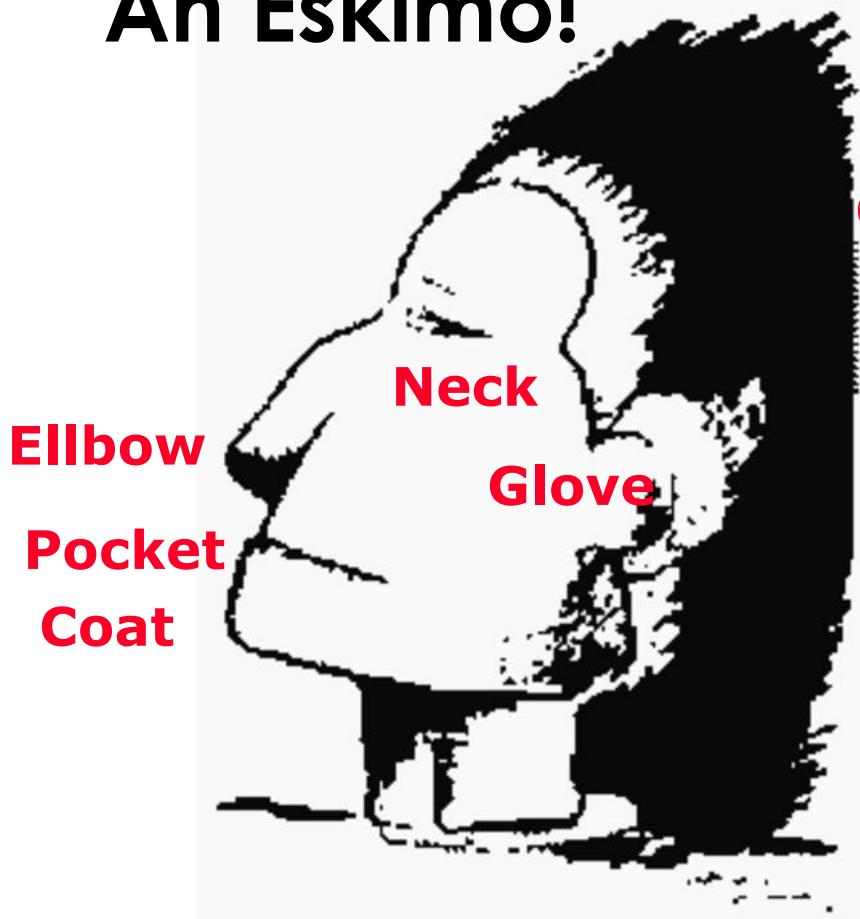


What is This?

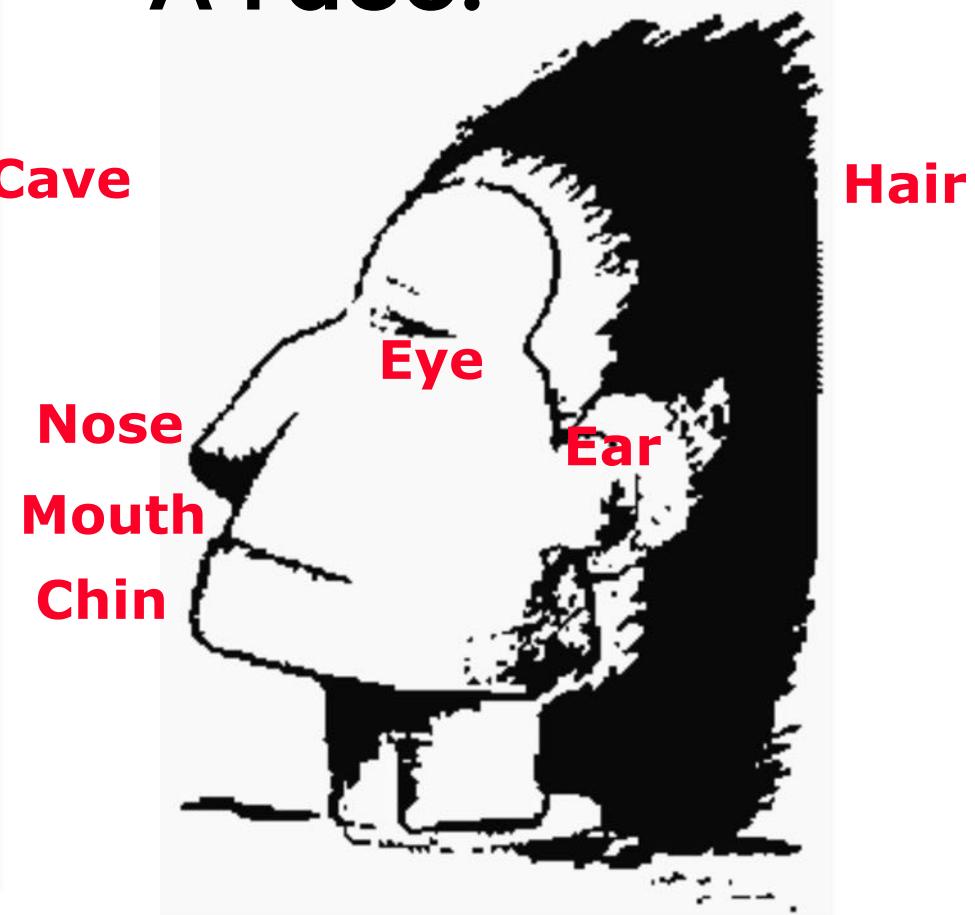
Let's try again



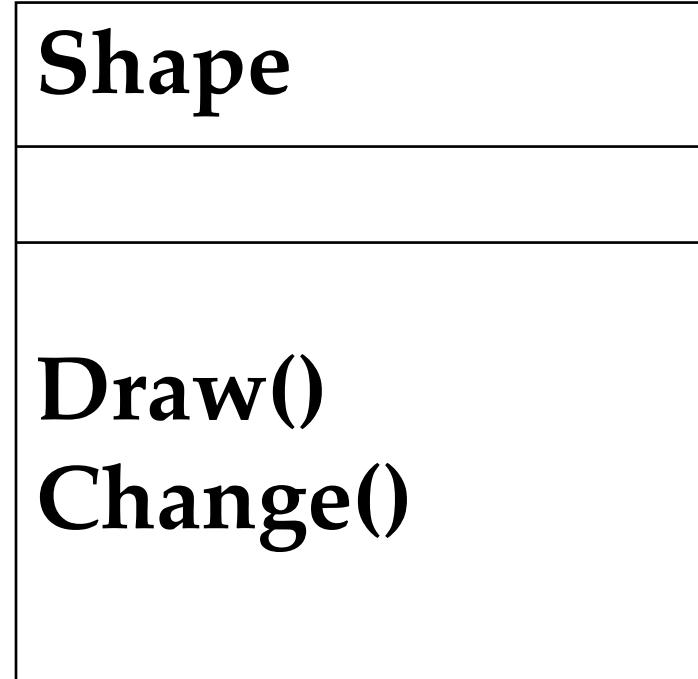
An Eskimo!



A Face!



Object-Oriented View

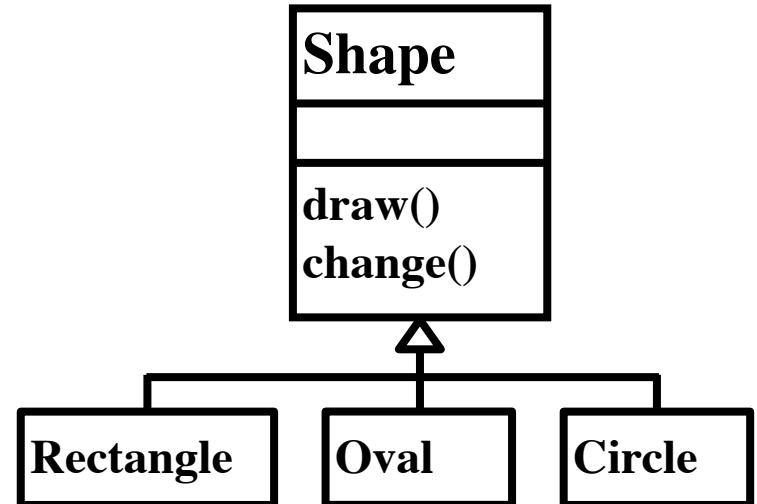
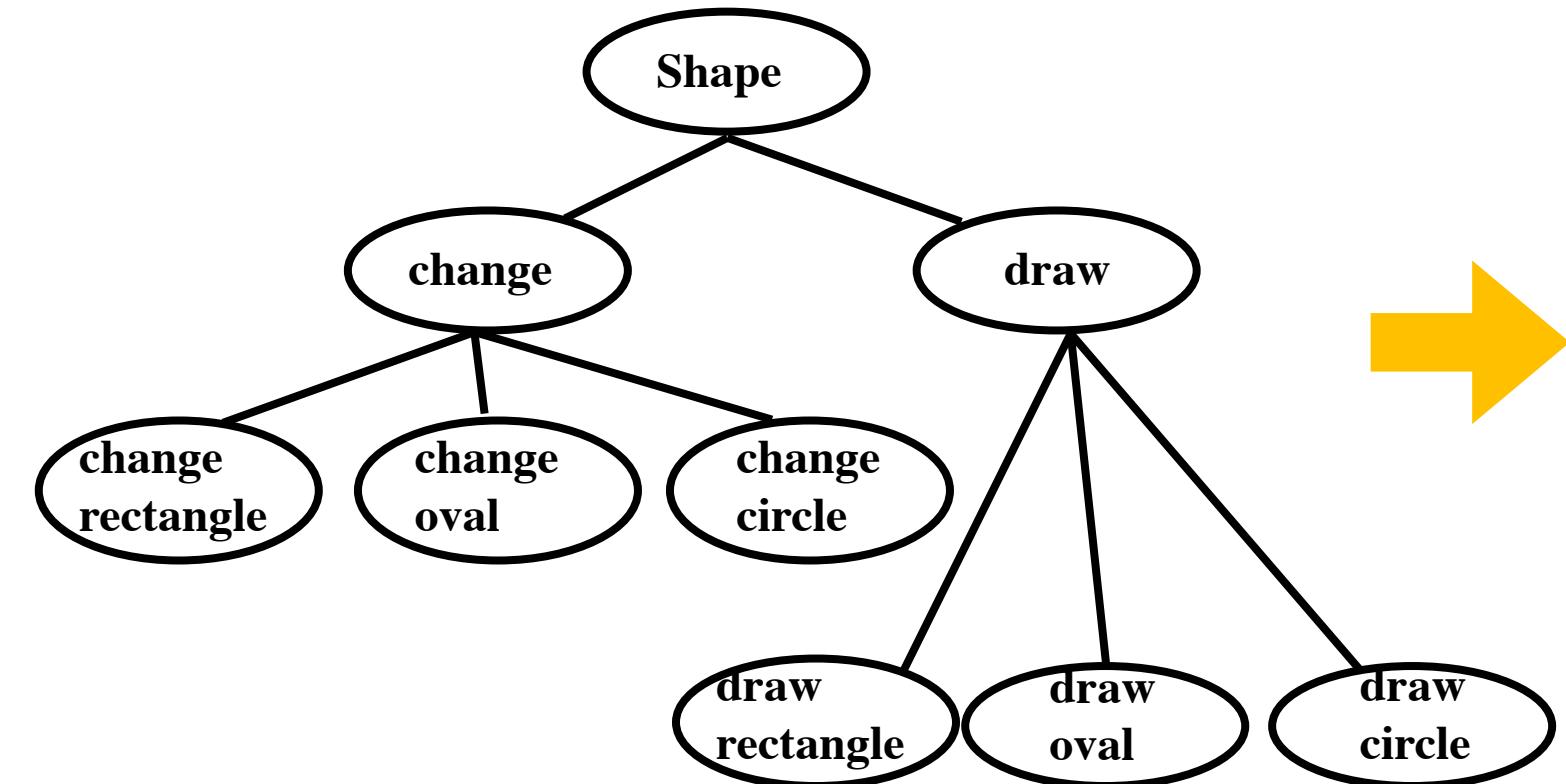


Is this really better?

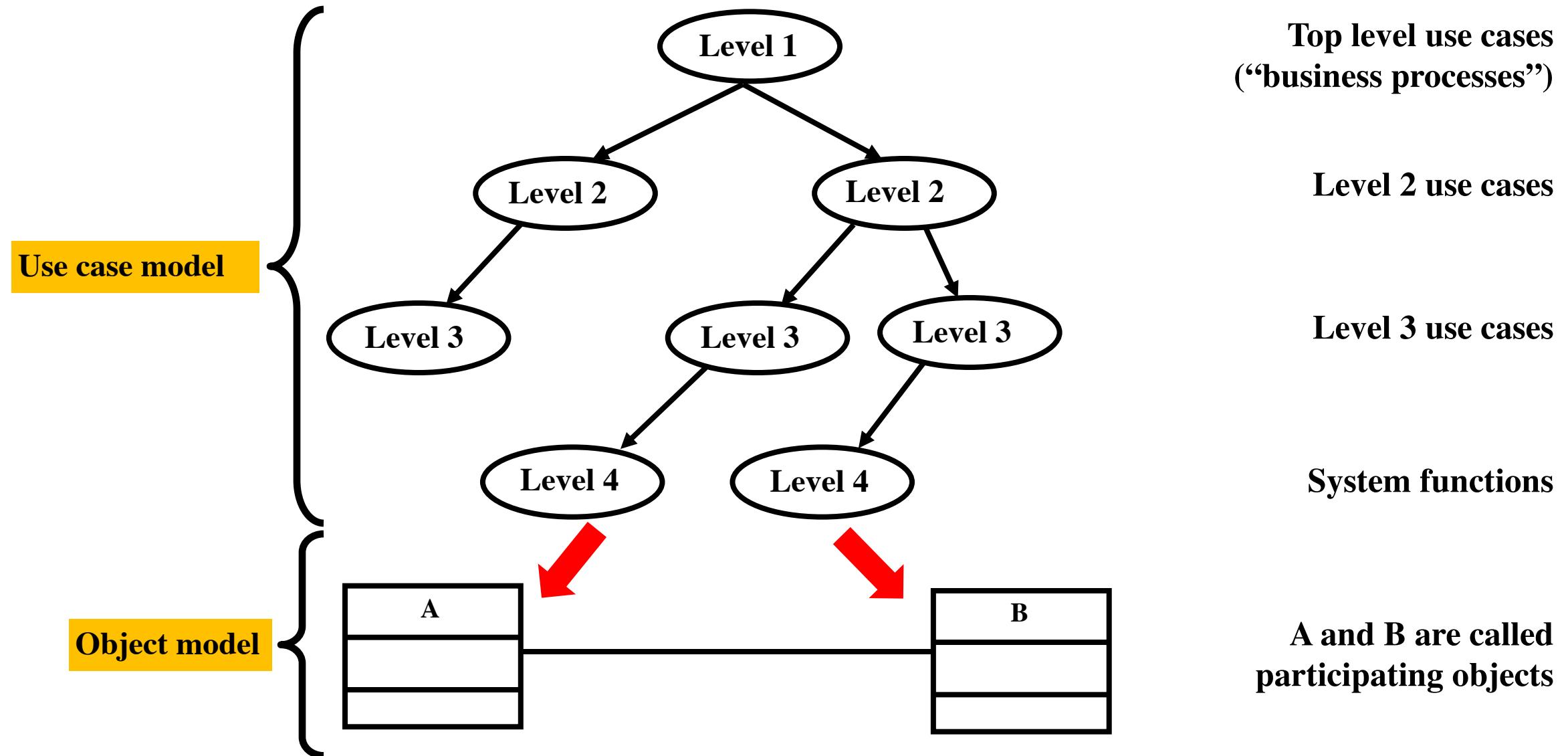
EIST Approach

- First focus on the functional requirements
- Find the corresponding use cases
- Identify the participating objects
- Use these participating objects to start the object model

EIST Approach: From Use Cases to Objects



Best practice: first identify use cases, then identify objects

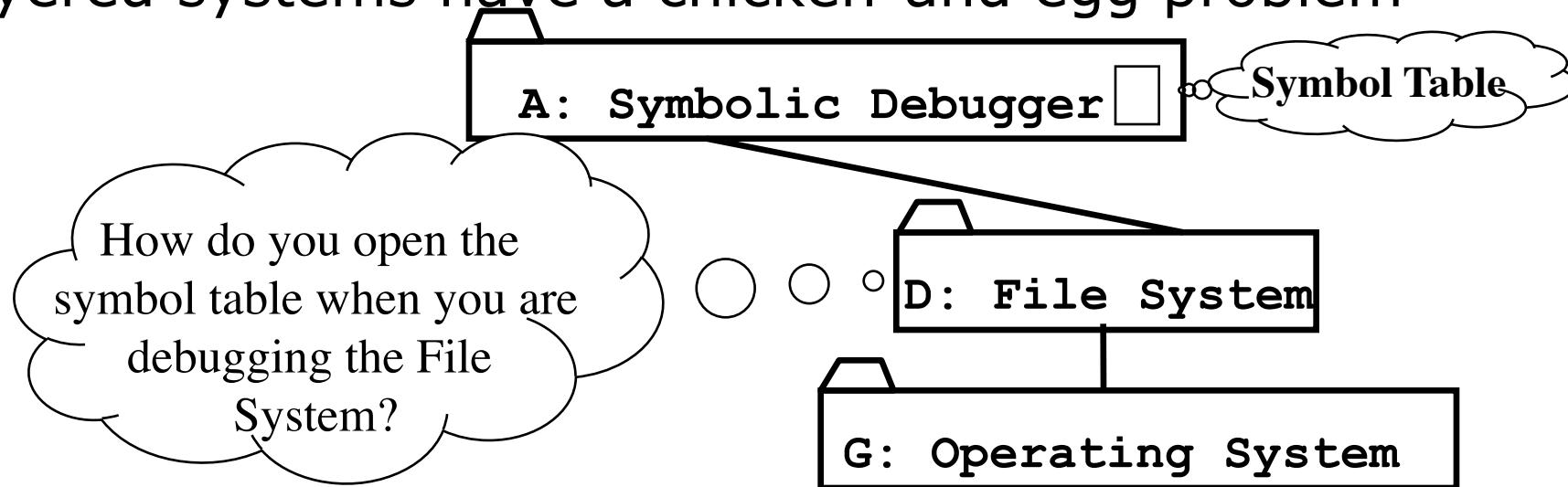


Outline of this lecture

- Miscellaneous
- Decomposition
- Architectural Styles
 - • Layered Architecture (Finish Lecture 04 from May 3, 2018)
 - Client-Server
 - Model-View Controller
 - Repository
 - Pipes-and Filers
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

Properties of Layered Systems

- Layered systems are hierarchical. This is desirable:
 - Hierarchy reduces complexity
- Closed architectures are more portable
 - Low coupling
- Open architectures are more efficient
 - High coupling
- Layered systems have a chicken-and egg problem



Roadmap for Today's Lecture

- **Context and Assumptions**

- We completed Chapter 1 to 5 in the OOSE text book by Bruegge and Dutoit

- **Content of this lecture**

 We finish Lecture 05: System Design I (Slide 86-96)

- We introduce how to address design goals
- We complete Chapter 6

- **Objective:** At the end of this lecture you are able to

- Understand how nonfunctional requirements impact the design goals
- Choose between different architectural styles based on the system model and the nonfunctional requirements
- Deal with 2 more UML Notations: Component Diagram and Deployment Diagram

SOA is a Closed Layered Architecture

SOA = Service Oriented Architecture

- Basic idea: A service provider ("business") offers **Business Services** ("business processes") to a service consumer (application, "customer")
 - The business services are dynamically discoverable, usually offered in **Web-based Applications**
- The business services are created by composing (choreographing) them from lower-level services (**Basic Services**)
- The basic services are usually based on **Legacy Systems**
- **Adapters** are used to provide the "glue" between basic services and the legacy systems.



Adapters will be covered in
Lecture 8 (June 14)

Web-based Applications

Business Services (Composite Services)

Basic Services

Adapters to Legacy Systems

Legacy Systems
Introduction to Software Engineering

Layers vs Tiers

Definition: 3-layered architectural style

- An architectural style, where an application consists of 3 hierarchically ordered layers

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.

3-Layered Architectural Style

- 3-layered architectural styles are often used for the development of Web applications
- Example of a 3-tier architecture:
 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.

4-Layered Architectural Style

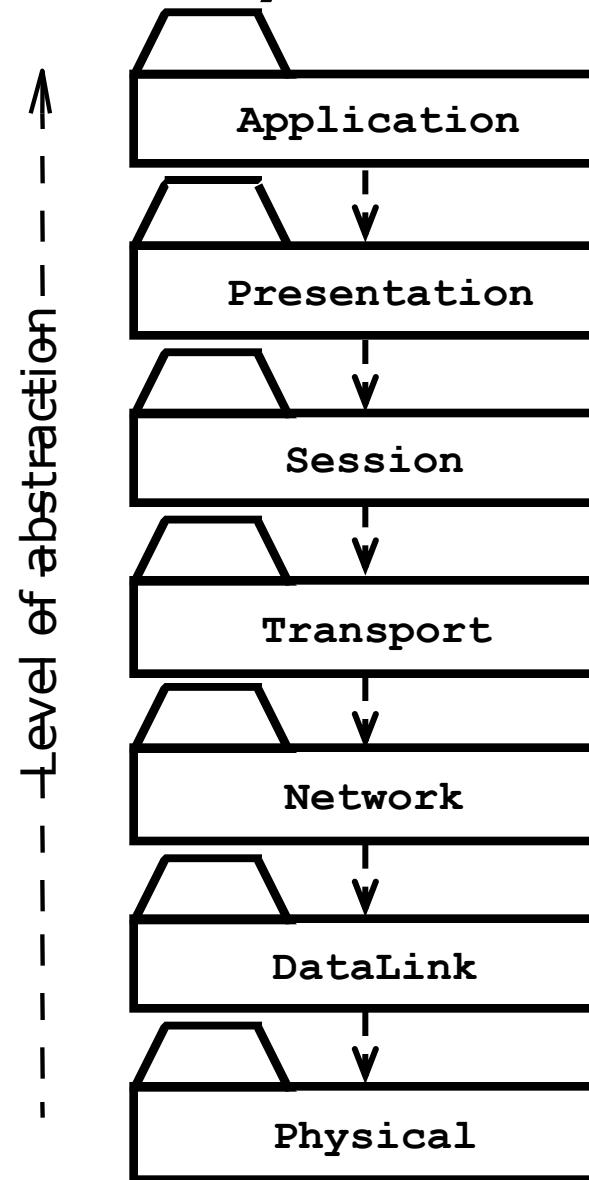
4-layer-architectural style: An architectural style, where an application consists of 4 hierarchically ordered layers

Example of 4-tier architecture:

1. A **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 commercially available 4-tier architectures, this is usually a relational database management system (RDBMS).

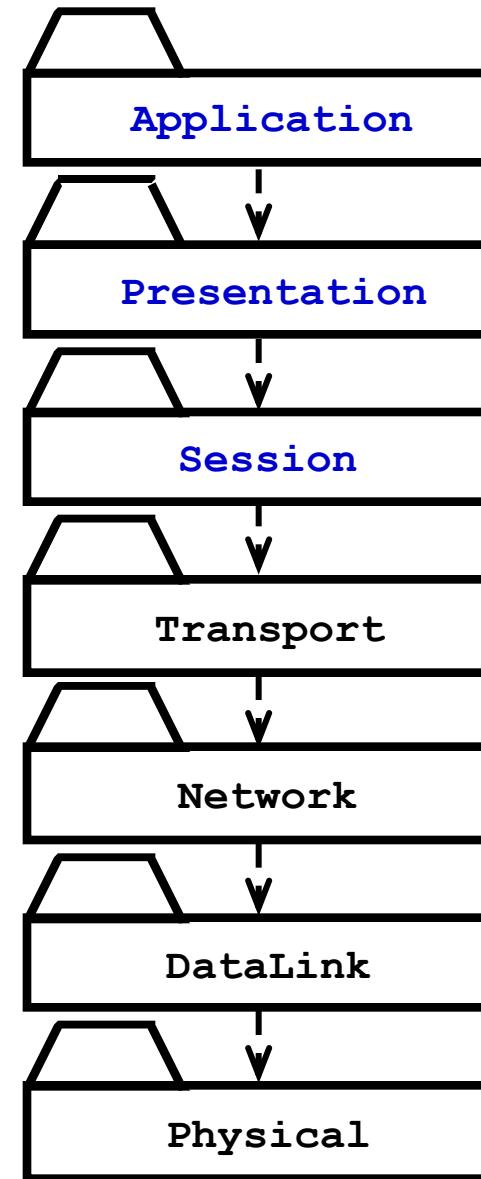
Example of a Layered Architectural Style with 7 Layers

- 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



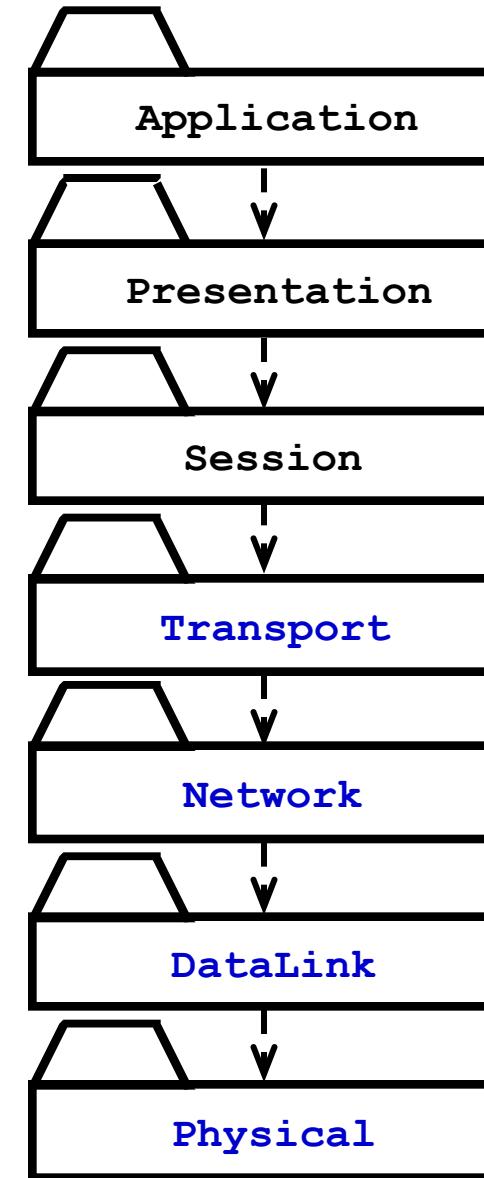
OSI Model Layers and their Services

- 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



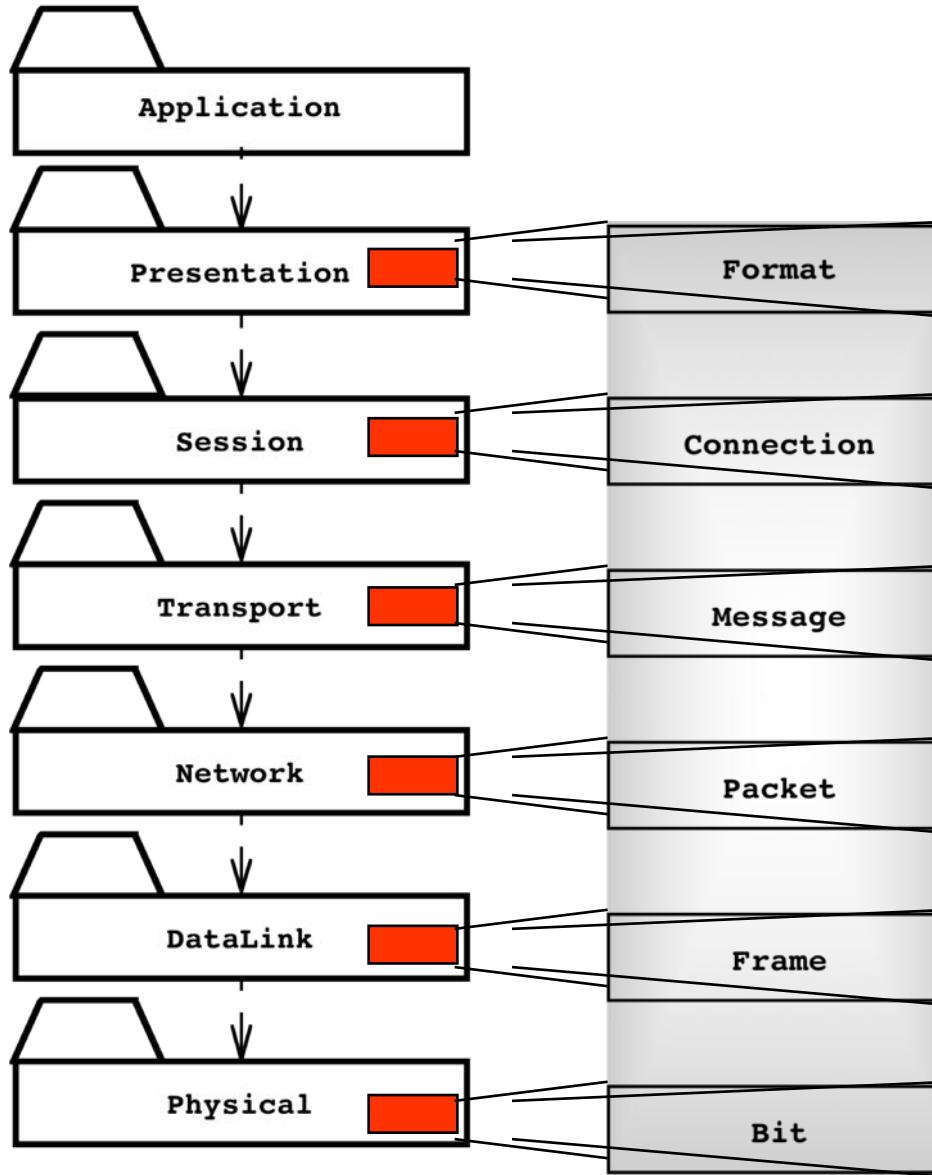
OSI Model Layers and their Services

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

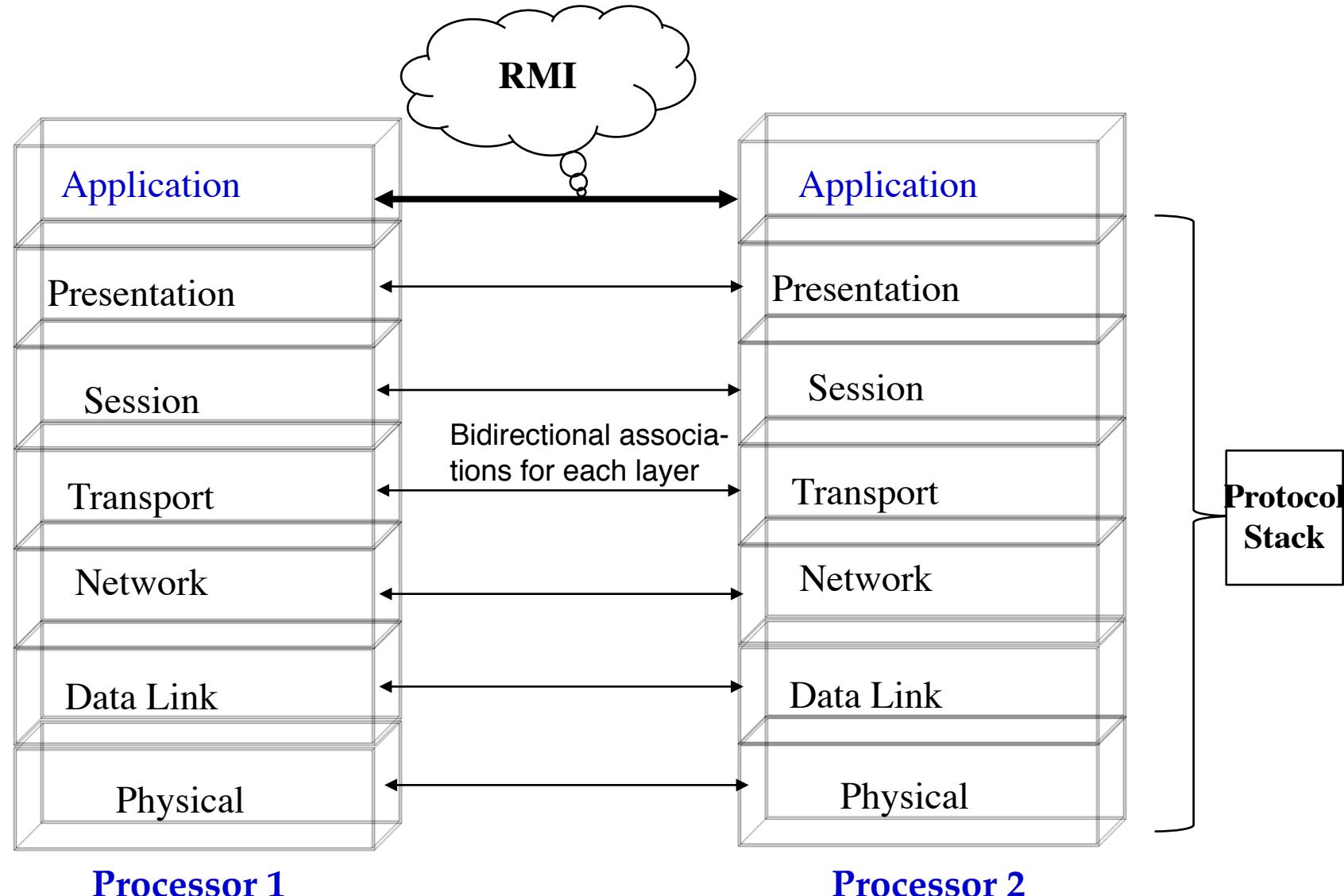


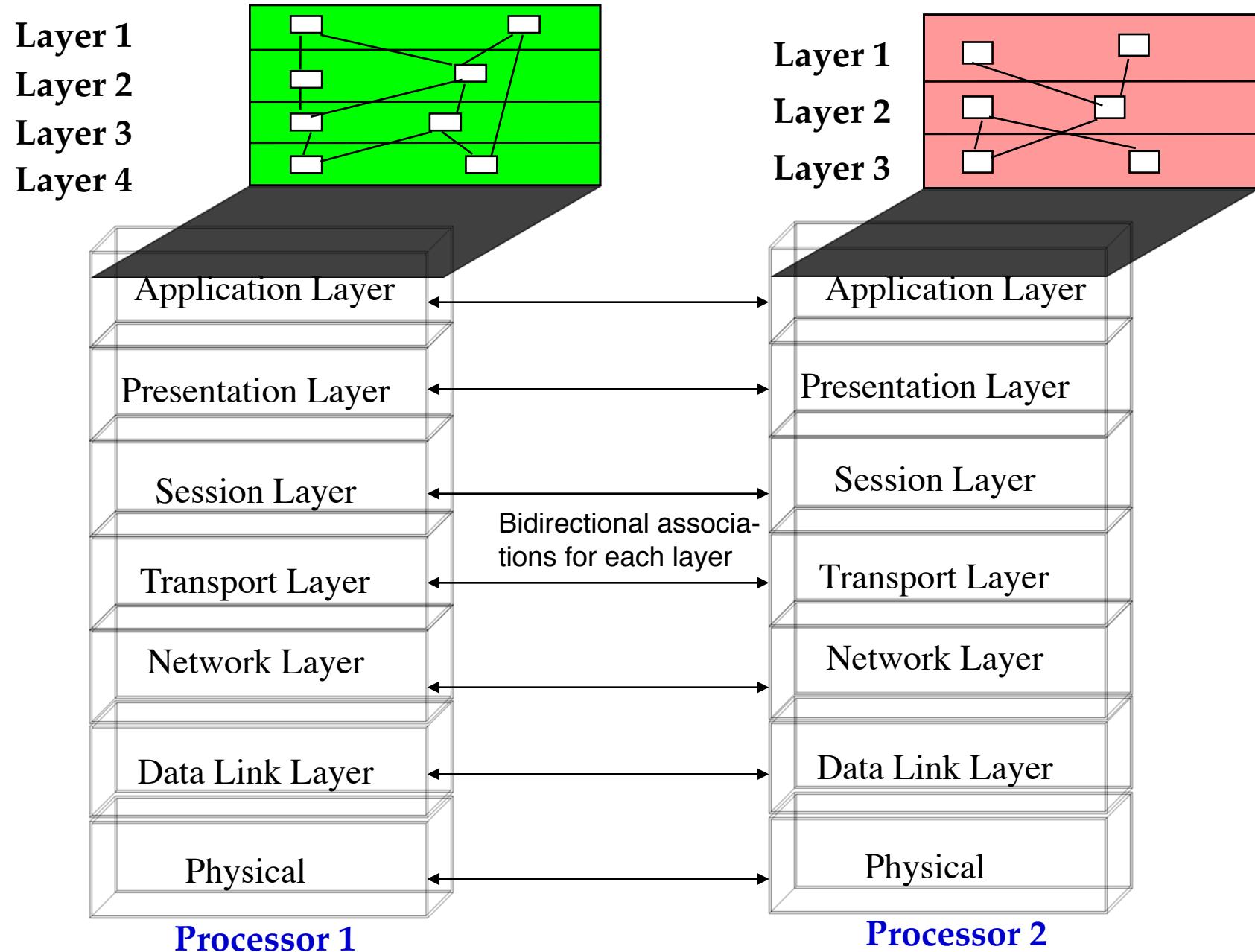
An Object-Oriented View of the OSI Model

- 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 offering public operations for the layer above



The Application Layer provides the Abstractions of the “New System”. It is usually layered itself



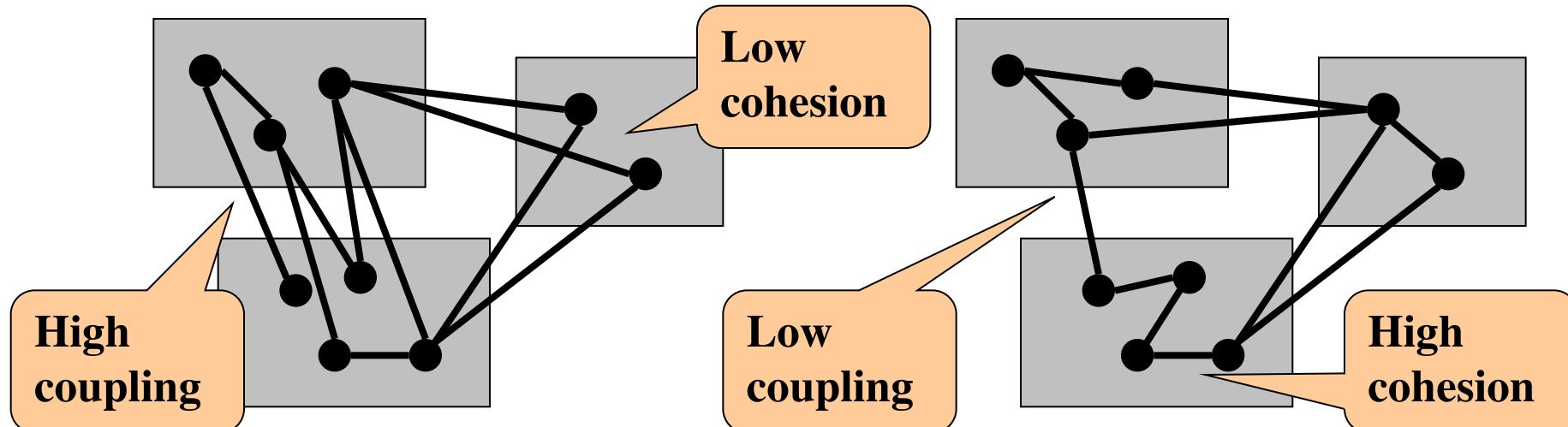


Overview of Today's Lecture

- Decomposition
- Architectural Styles
 - Layered Architecture, Review: Cohesion and Coupling
 - Client-Server
 - Model-View Controller
 - Repository
 - Pipes-and Filters
 - Architectural Styles: Graphs with Components and Connectors
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

Cohesion and Coupling Measure Interdependence

- Cohesion measures the **interdependence** of the objects in **one subsystem**
- Coupling measures the **interdependence of objects between** different **subsystems**



Achieving High Cohesion and Low Coupling

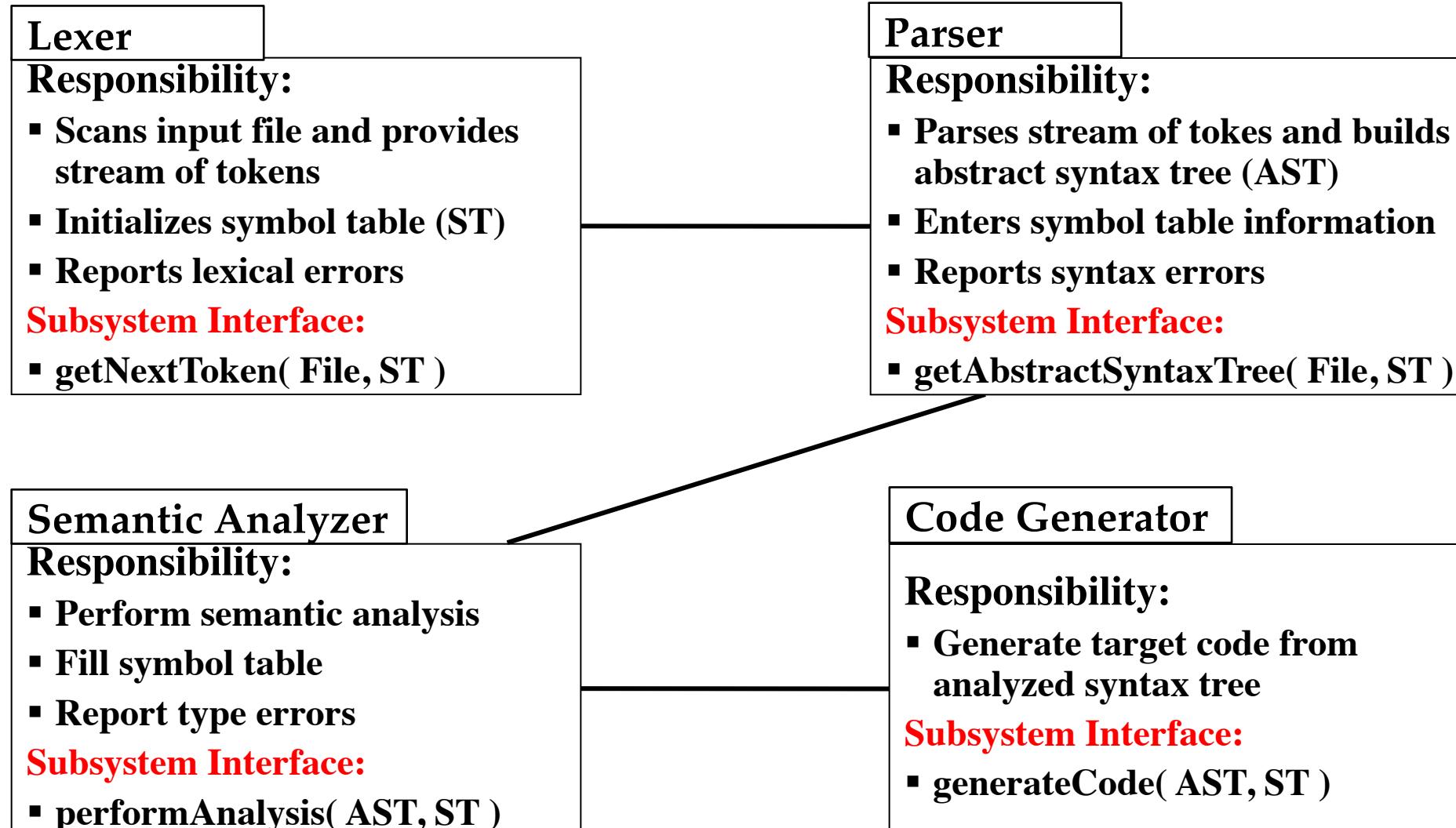
High cohesion

- Operations work on same attributes
- Operations implement a common abstraction or service

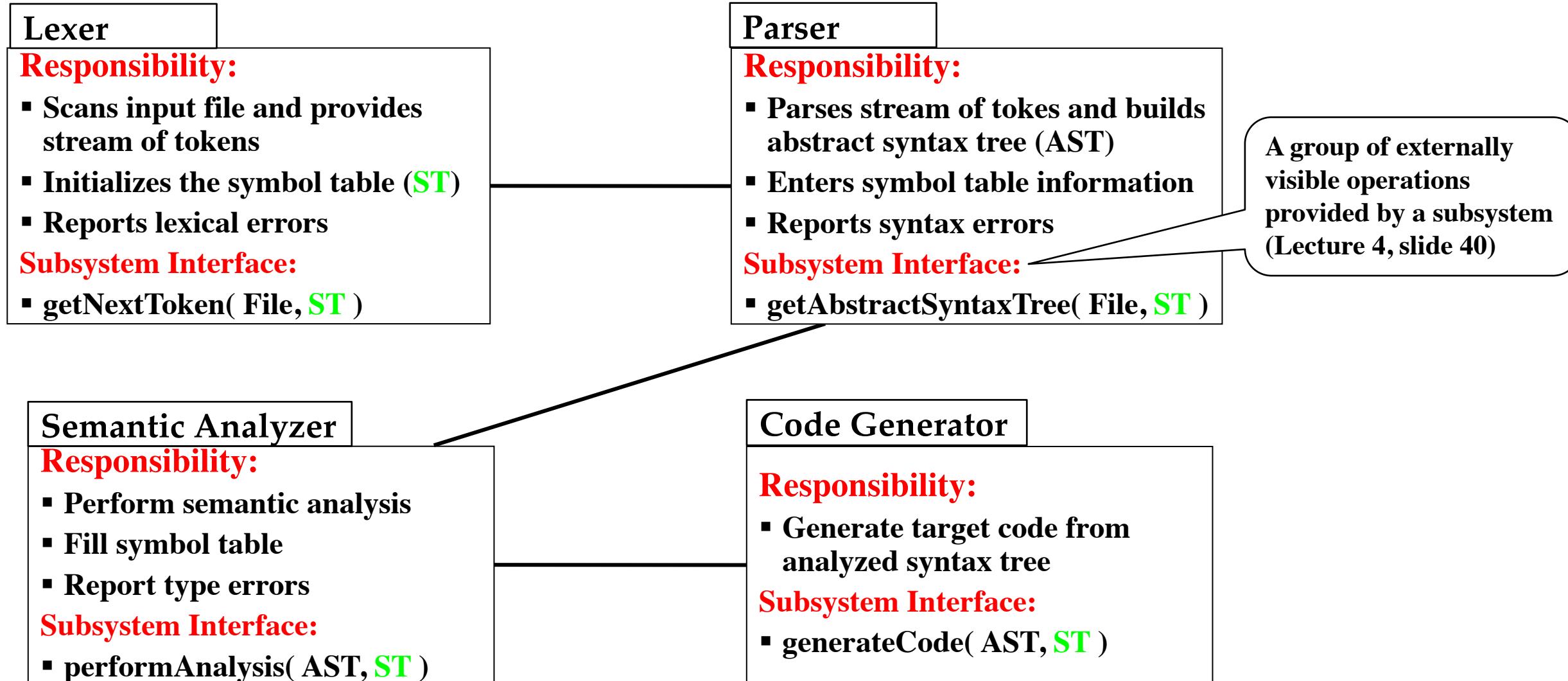
Low coupling

- Small interfaces
- Information hiding
- No global data
- Interactions are mostly within the subsystem rather than across subsystem boundaries.

Example: Subsystem Decomposition of Compiler

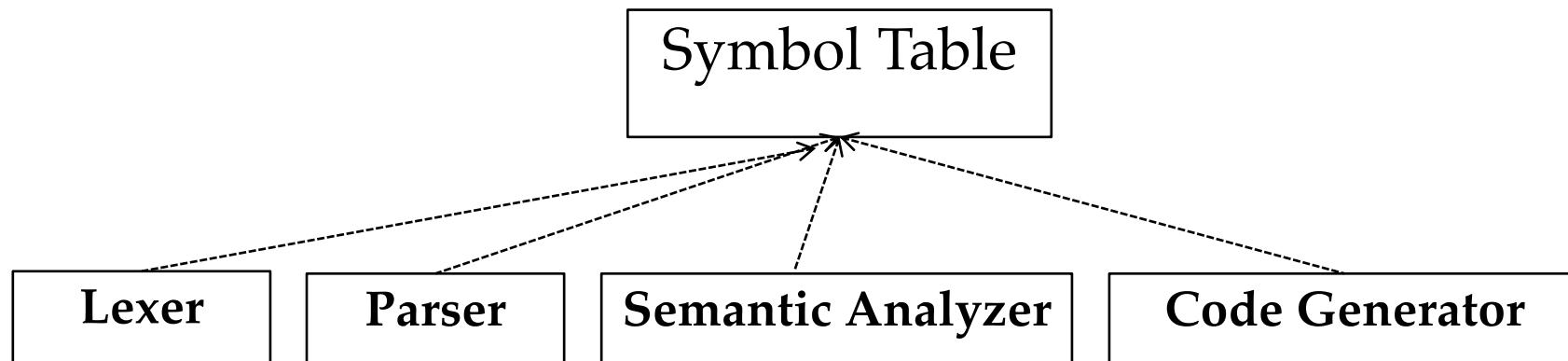


Example: Subsystem Decomposition of Compiler

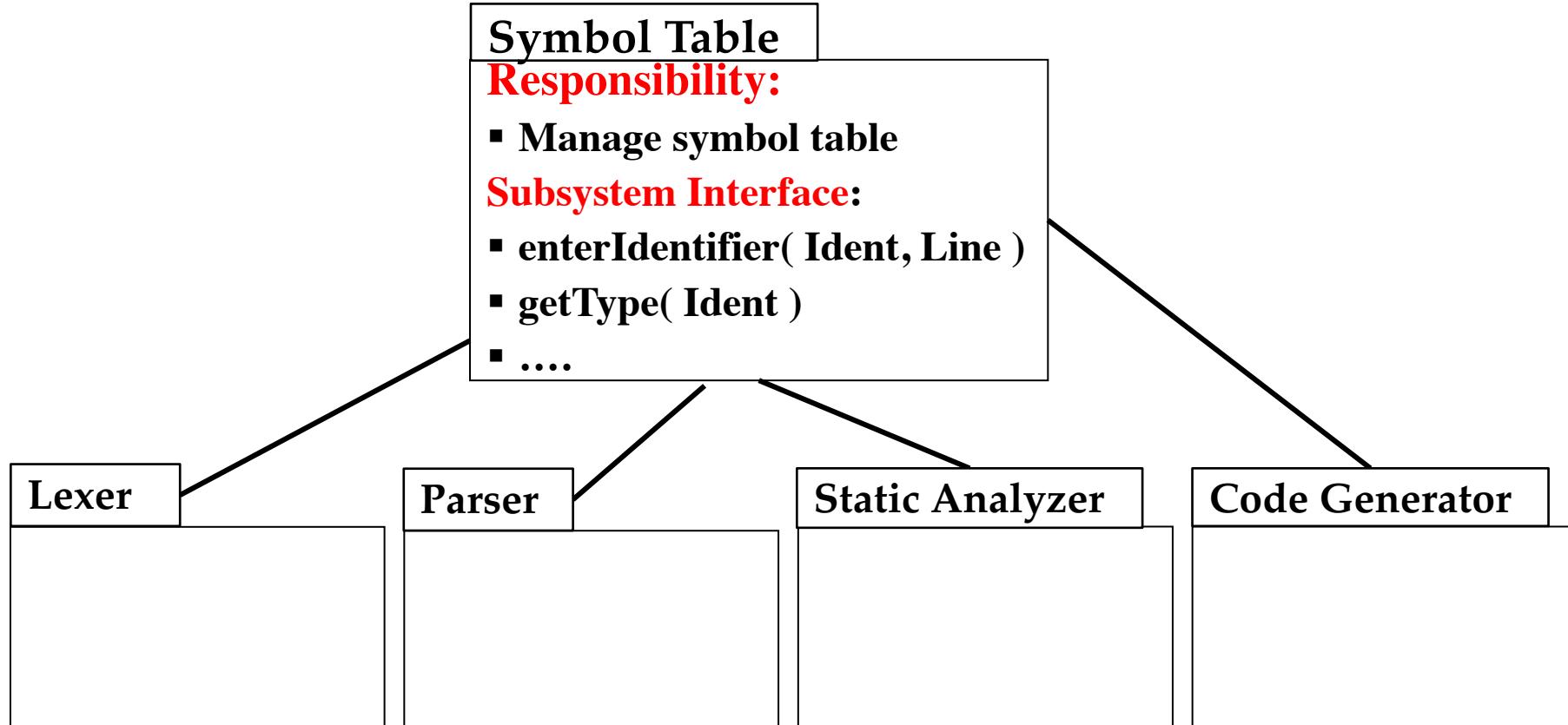


Cohesion & Coupling in Compiler Example

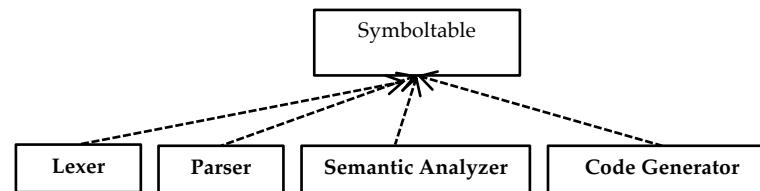
- Coupling
 - Small subsystem interfaces
 - **Coupling is ok**
- Cohesion
 - All subsystems read and update the symbol table
 - Any change in the symbol table representation effects all subsystems
 - **But cohesion is bad**
- We improve the compiler design by introducing a separate subsystem **Symbol Table**



Compiler Design with better Cohesion



We increased the Cohesion by a Model Transformation



More about Model Transformations
in Lecture 7 (June 7)

New design with
high cohesion

Model
transformation
(Model refactoring)

Old design with
low cohesion

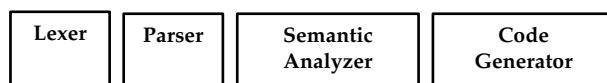
Forward
engineering

Reverse
engineering

Refactoring
(Code Refactoring)

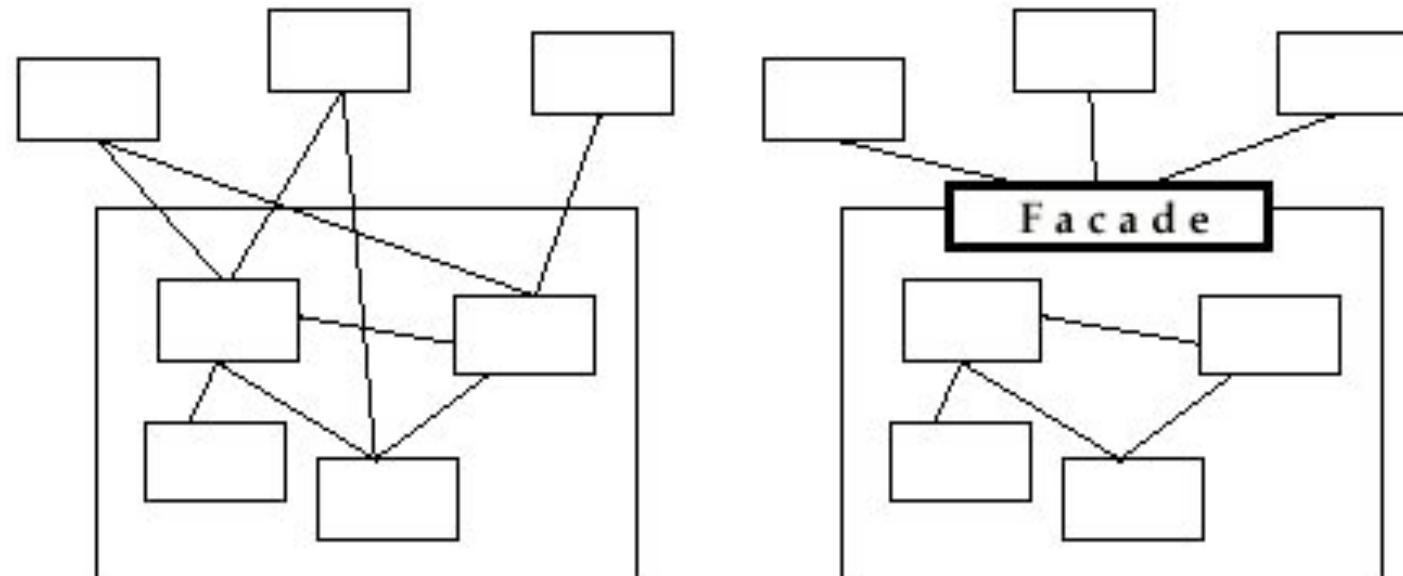
UML Model space

Source code space



The Façade: A Pattern to reduce Coupling

- A façade provides a unified interface for a subsystem
 - A façade consists of a set of public operations
 - Each public operation is delegated to one or more operations in the classes behind the facade
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- A façade allows to hide design spaghetti from the caller

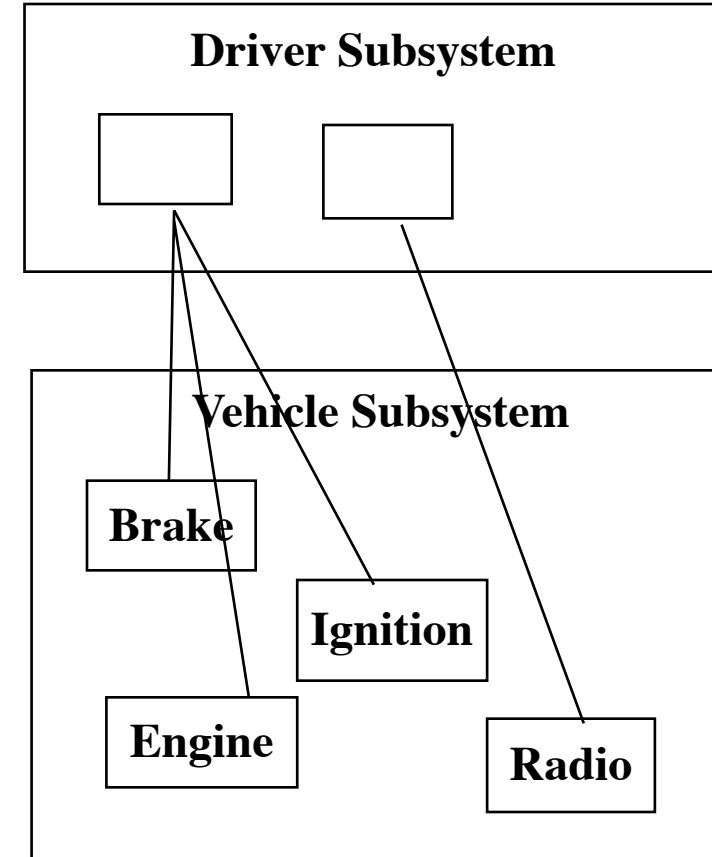


Design Example

The **Driver Subsystem** can call any class operation in the **Vehicle Subsystem**

->

Spaghetti Design!



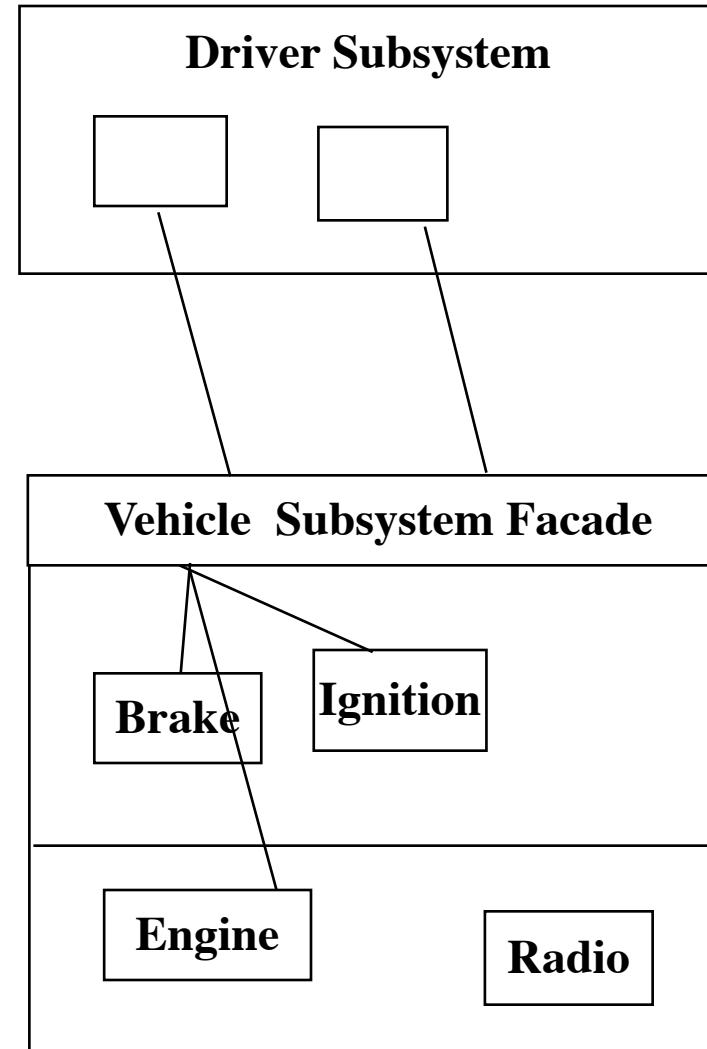
Opaque Architecture with a Façade

- The Vehicle Subsystem decides exactly how it is accessed with the Vehicle Subsystem Façade
- Advantages:
 - Reduced complexity
 - Less recompilations

Additional advantage:

- A façade can be used during integration testing when the internal classes are not implemented
- We can write mock objects for each of the public methods in the façade

➤ Lecture on Testing.



Review Terminology: Architectural Style vs Architecture

- **Subsystem decomposition:** Identification of subsystems, services, and their relationship to each other
- **Architectural Style:** A pattern for a subsystem decomposition
- **Software Architecture:** Instance of an architectural style. Often also used synonymously with architectural style.

Overview of Today's Lecture

- Decomposition
- Architectural Styles
 - Layered Architecture, Review: Cohesion and Coupling
- Client-Server
 - Model-View Controller
 - Repository
 - Pipes-and Filters
 - Architectural Styles: Graphs with Components and Connectors
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

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 the client:**
 - Input from the user (Customized user interface)
 - Front-end processing of input data
- **Functions performed by the server:**
 - Centralized data management
 - Provision of data integrity and database consistency
 - Provision of database security.

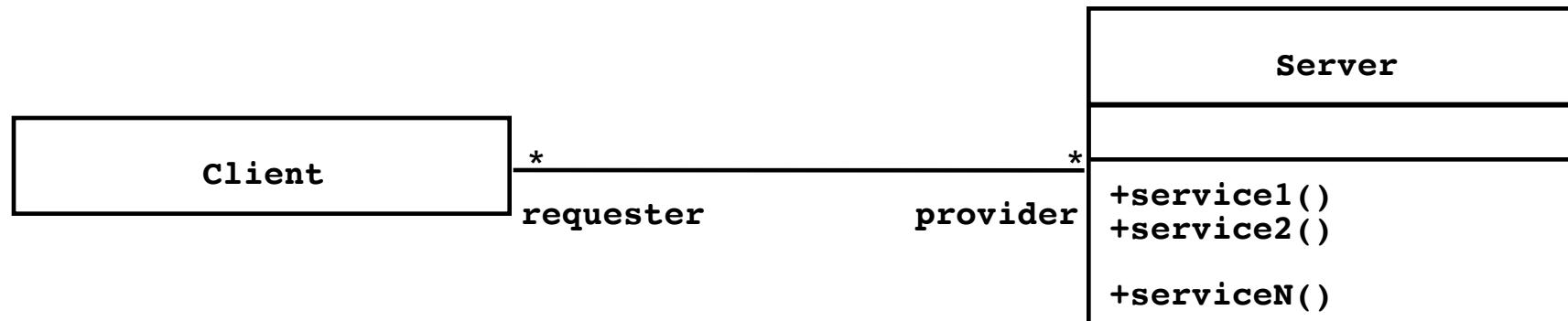
Client/Server Architectural Style

- The **Client/Server architectural style** is a special case of the layered architectural style:
 - One or more **Servers** provide services to instances of subsystems, called **Clients**
- Each Client calls a service offered by the Server; the Server performs the service and returns the result to the Client

The Client knows the *interface* of the Server

The Server does not know the interface of the Client

- The response in general is immediate
- End users interact only with the Client.



Design Goals for Client/Server Architectures

Portability

Server runs on many operating system
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

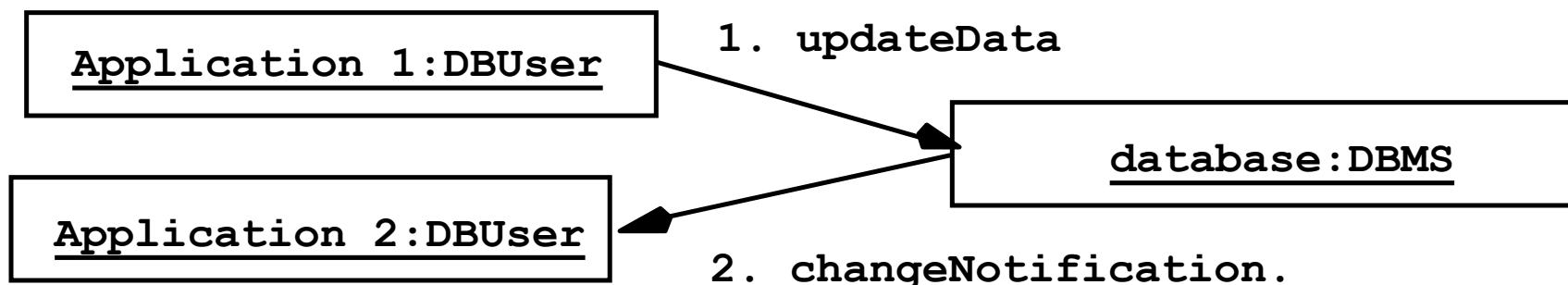
The user interface of the Client supports
a variety of end devices (PDA, Handy,
laptop, wearable computer)

Reliability

Server should be able to survive client
and communication problems.

Problems with Client/Server Architectures

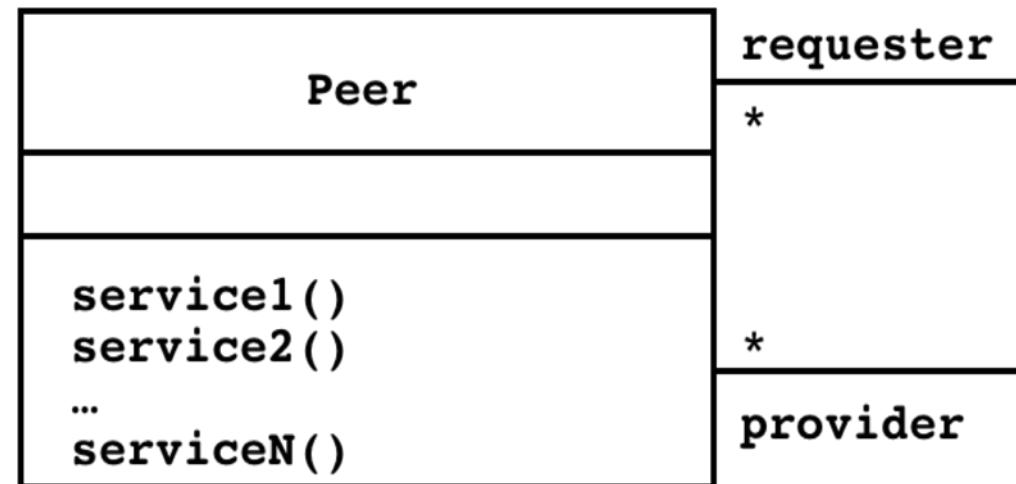
- Client/Server systems use a Request-Response Protocol
- Peer-to-peer communication is often needed
- Example:
 - A database must process queries from application 1 and should be able to send notifications to application 2 when data in the database have changed



Peer-to-Peer Architectural Style

The **Peer-to-Peer architectural style** is generalization of the Client/Server Architectural Style: Clients can be servers and servers can be clients.

Introduction of a new abstraction: Peer



Overview of Today's Lecture

- Decomposition
- Architectural Styles
 - Layered Architecture, Review: Cohesion and Coupling
 - Client-Server
- Model-View Controller
 - Repository
 - Pipes-and Filters
 - Architectural Styles: Graphs with Components and Connectors
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

Model-View-Controller Architectural Style

- **Problem:** In systems with high coupling any change to the boundary objects (user interface) often force changes to the entity objects (data)
 - The user interface cannot be re-implemented 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 (MVC) style decouples data access (entity objects) and data presentation (boundary objects)
 - **View:** A subsystem containing boundary objects
 - **Model:** A subsystem containing entity objects
 - **Controller:** A subsystem mediating between the views (data presentation) and the models (data access).

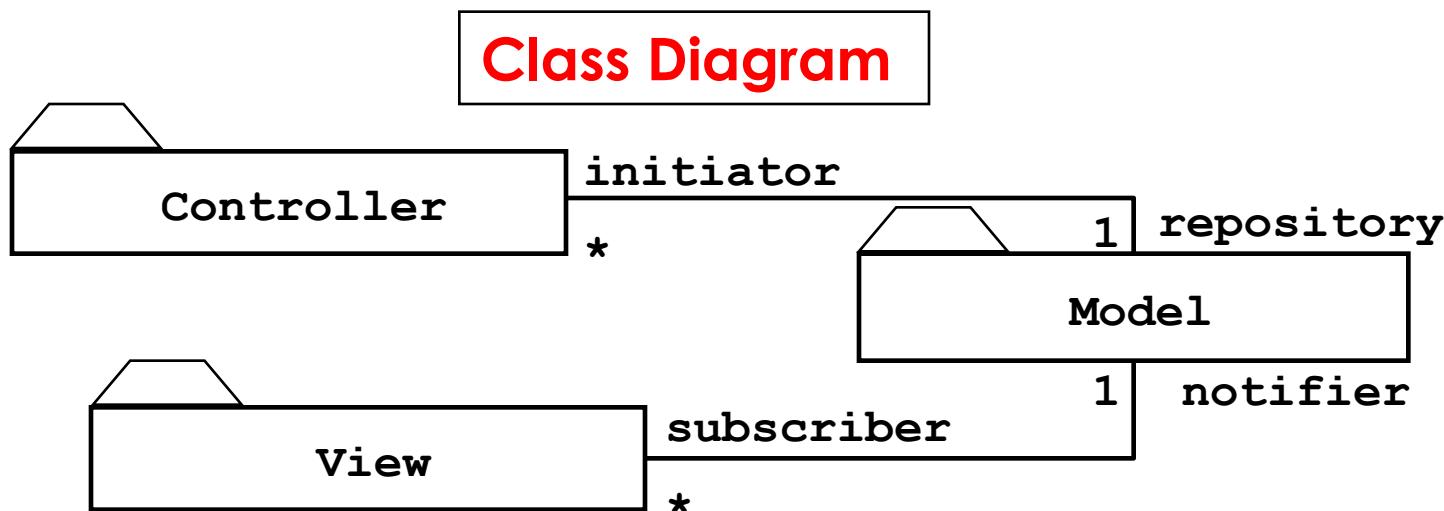
Model-View-Controller Architectural Style

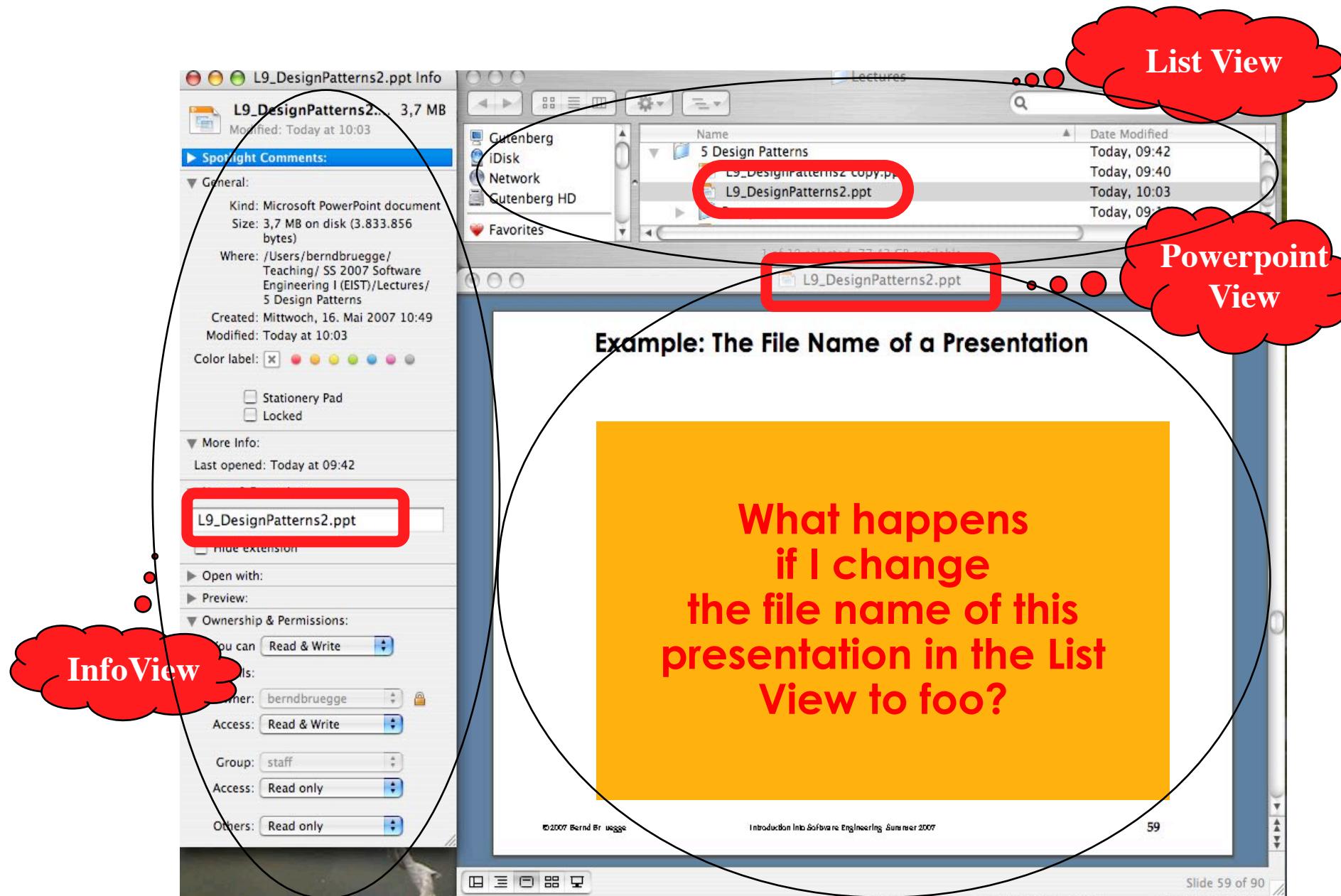
- Subsystems are classified into 3 different types

Model subsystem: Responsible for application domain knowledge

View subsystem: Responsible for displaying information to the user

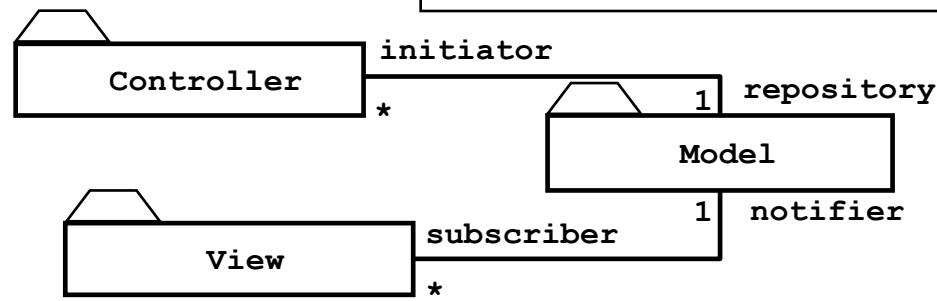
Controller subsystem: Responsible for interacting with the user and notifying views of changes in the model



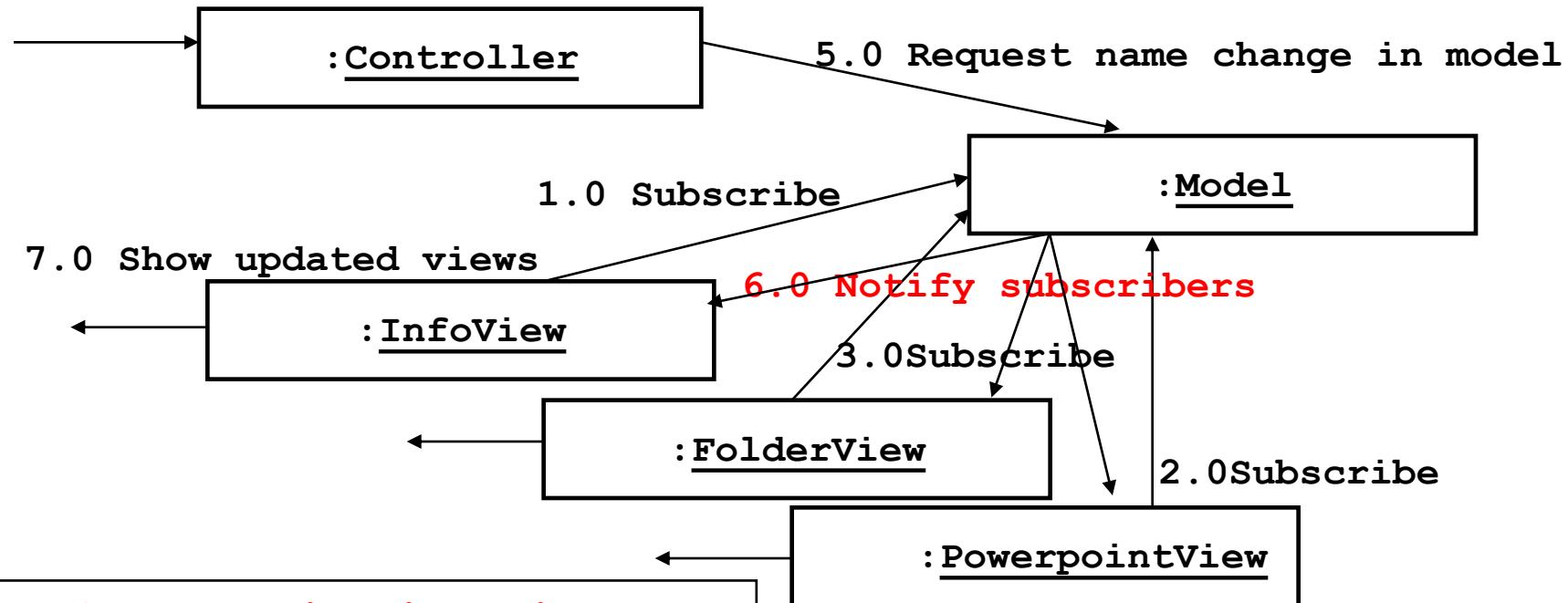


Modeling the Sequence of Events

UML Class Diagram



4.0 User types new filename



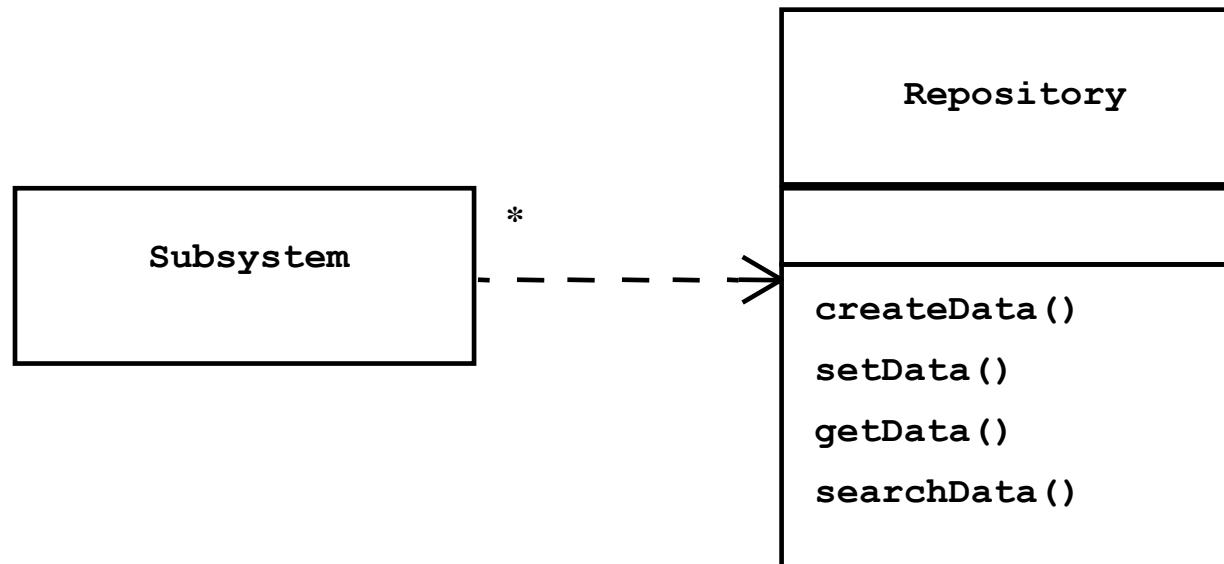
UML Communication Diagram

MVC vs. 3-Tier Architectural Style

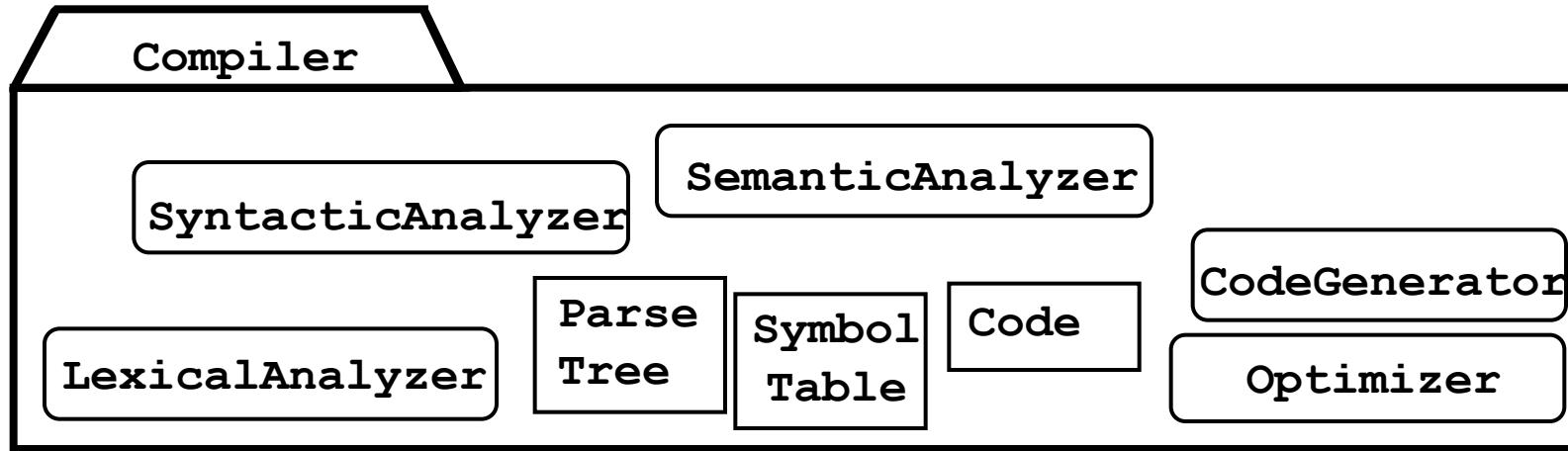
- The **MVC** architectural style is **nonhierarchical** (triangular):
 - View subsystem sends updates to the Controller subsystem
 - Controller subsystem updates the Model subsystem
 - View subsystem is updated directly from the Model
- The **3-tier** architectural style is **hierarchical** (linear):
 - 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 run on physically separate platforms.

Repository Architectural Style

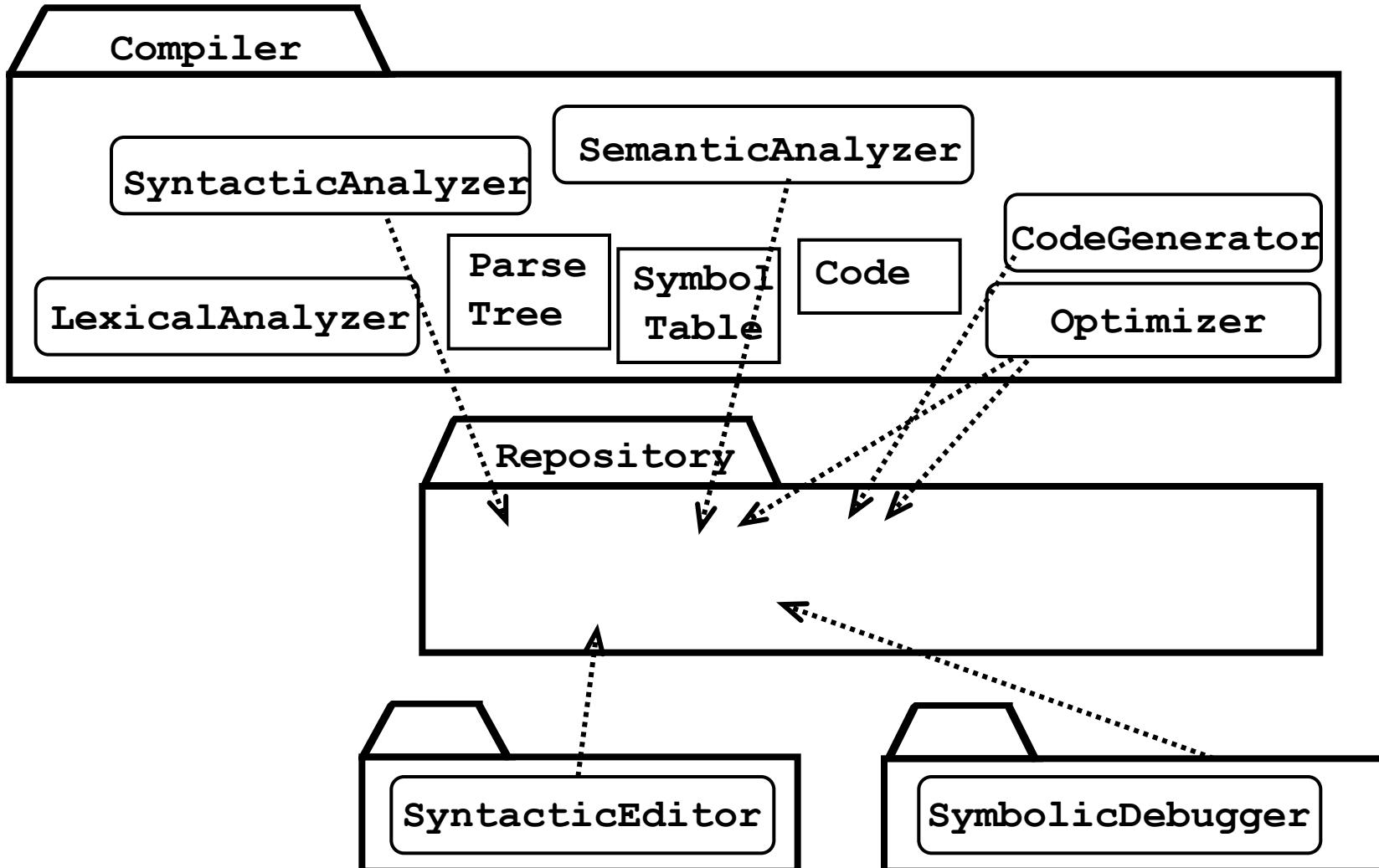
- The basic idea behind this architectural style is to support a collection of independent programs that *work cooperatively* on a common data structure called the **Repository**
- In the architectural style these programs are called subsystems. **Subsystems** access and modify data from the **Repository**. The subsystems are *loosely coupled* (they interact only through the repository).



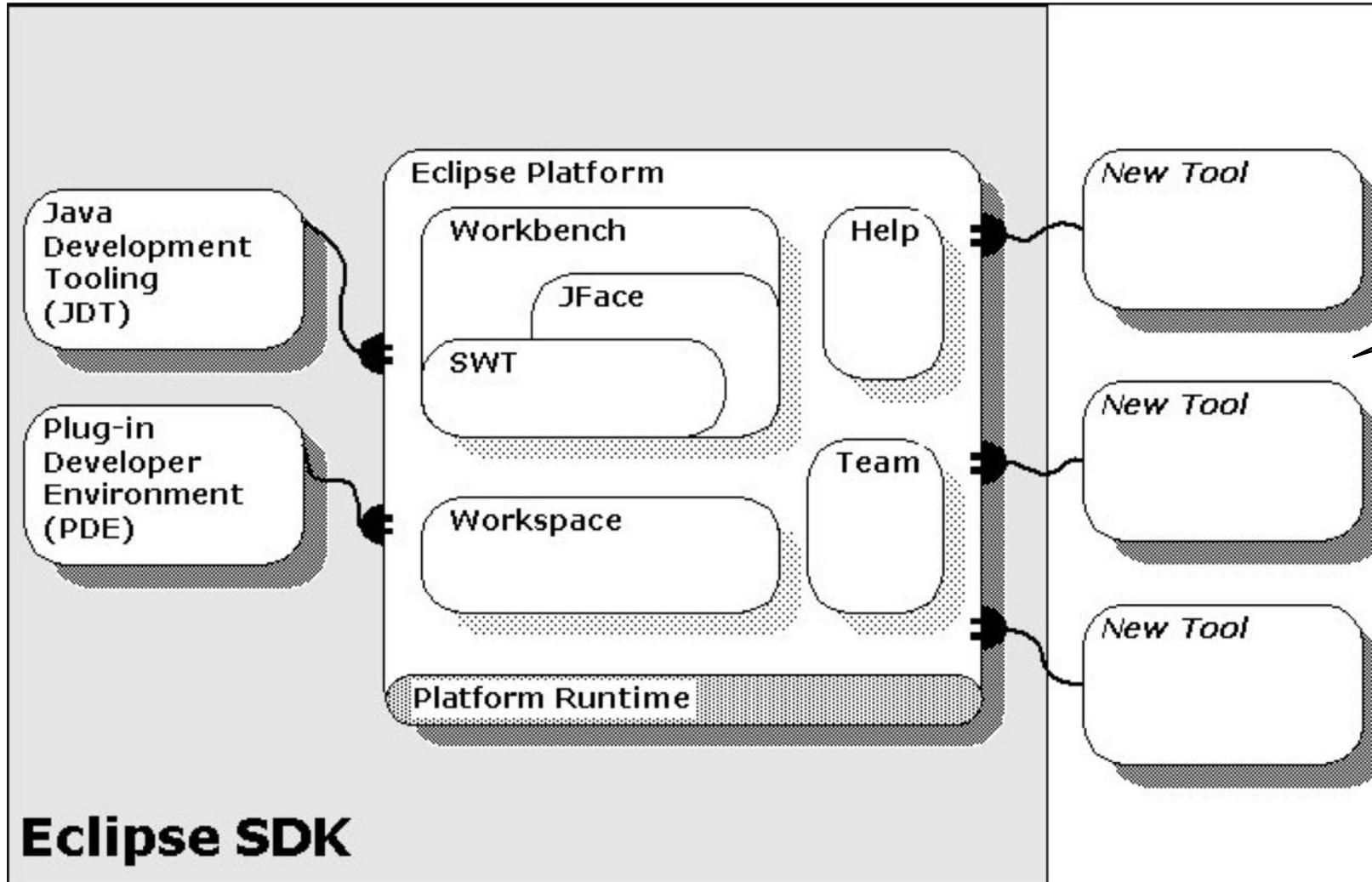
Repository Architecture Example: Integrated Development Environment (IDE)



Repository Architecture Example: Integrated Development Environment (IDE)



Eclipse uses a repository architectural style



Challenge: Turn this drawing into a UML diagram!
(Practice this in your tutor group)

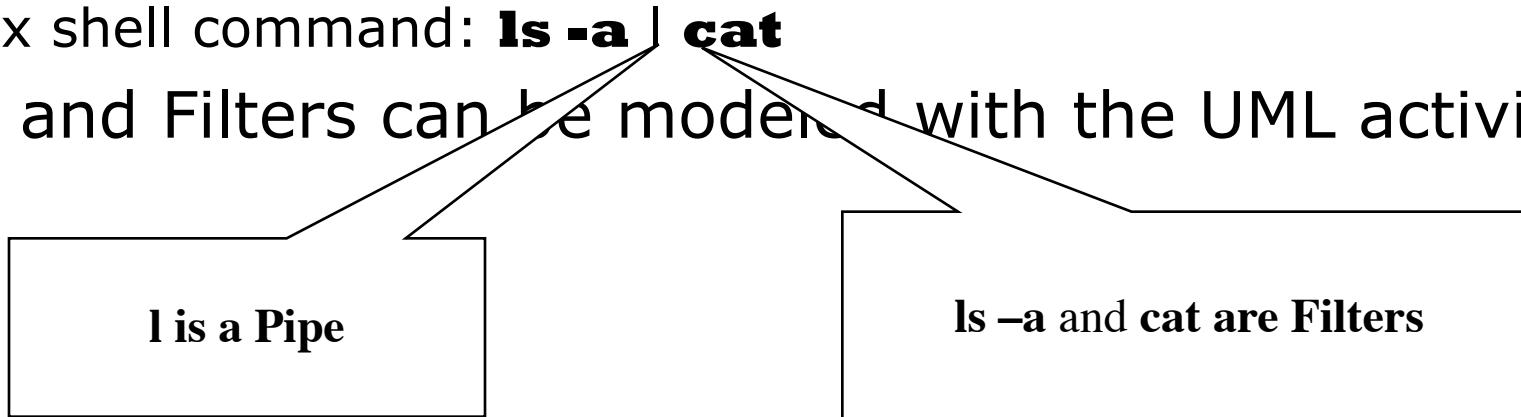
<https://help.eclipse.org/neon/topic/org.eclipse.platform.doc.isv/guide/arch.htm>

Overview of Today's Lecture

- Decomposition
- Architectural Styles
 - Layered Architecture, Review: Cohesion and Coupling
 - Client-Server
 - Model-View Controller
 - Repository
- Pipes-and Filters
 - Architectural Styles: Graphs with Components and Connectors
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

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**
- Pipes and Filters can be modeled with the UML activity diagram



Overview of Today's Lecture

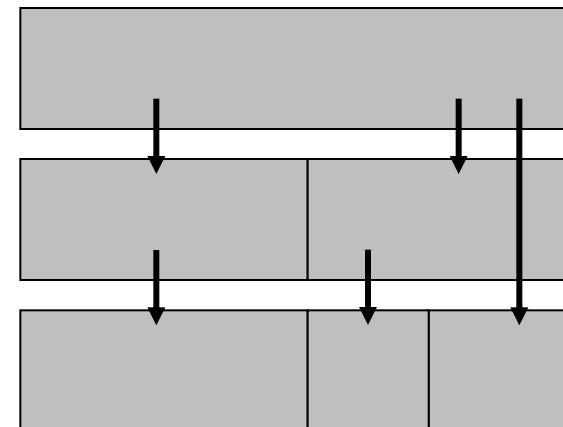
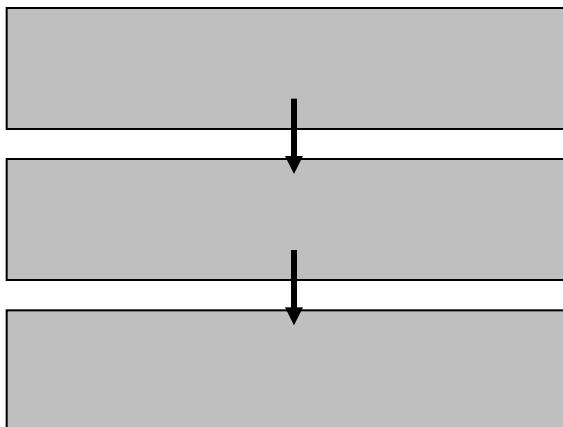
- Cohesion and Coupling
- Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - Repository
 - Pipes-and Filers
- ➡ Architectural Styles: Graphs with Components and Connectors
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

Elements of an Architectural Style

- Components (Subsystems)
 - Computational units with a specified interface
 - Examples: filters, databases, layers, objects
- Connectors (Communication)
 - Interactions between the components (subsystems)
 - Examples: method calls, pipes, event broadcasts, shared data
- From: Mary Shaw and David Garlan: *Software Architecture*, Prentice Hall, 1996.

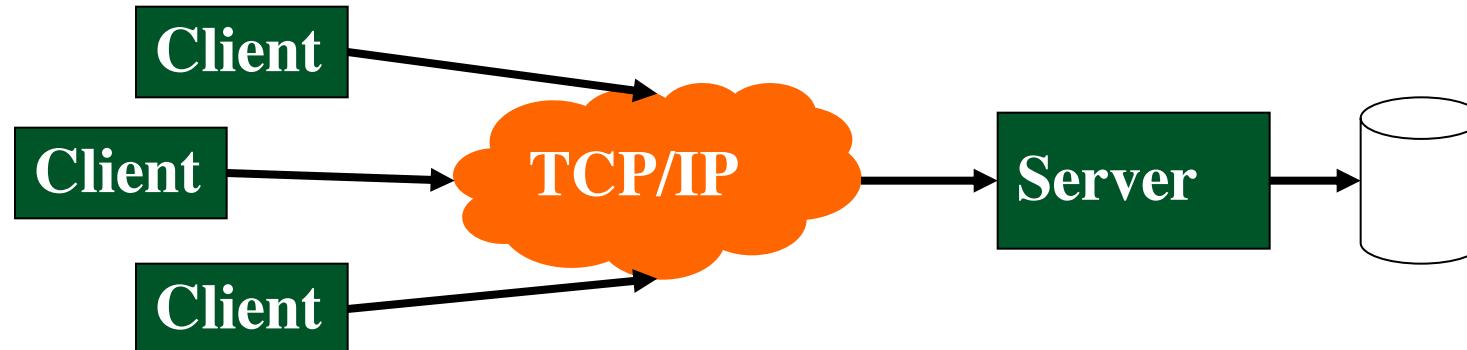
Layered Architectural Style in terms of Components and Connectors

- **Subsystems (Components)**
 - Group of subtasks which implement an abstraction at some layer in the hierarchy
- **Connectors**
 - Protocols that define how the layers interact



Client/Server Architectural Style in terms of Components and Connectors

- **Components**
 - Subsystems are independent processes
 - Servers provide specific services such as printing, etc.
 - Clients use these services
- **Connectors**
 - Data streams, typically over a communication network



Model-View-Controller Architectural Style in terms of Components and Connectors

- **Components (Subsystems)**
 - The **Model** contains the core functionality and data
 - One or more **views** display information to the user
 - One or more **controllers** handle user input
- **Connectors**
 - **Events:** Change-propagation mechanism via **events** ensures consistency between user interface and model
 - **Method calls:**
 - Read Data(), Initiate Operation(), Update()
 - If the user changes the model through the controller of one view (UpdateView()), the other views will be updated automatically.

Overview of Today's Lecture

- Cohesion and Coupling
- Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors

→ Concurrency

- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

30 Minute Break



System Design

✓ 1. Design Goals

Definition
Trade-offs

✓ 2. Subsystem Decomposition

Architectural Styles
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

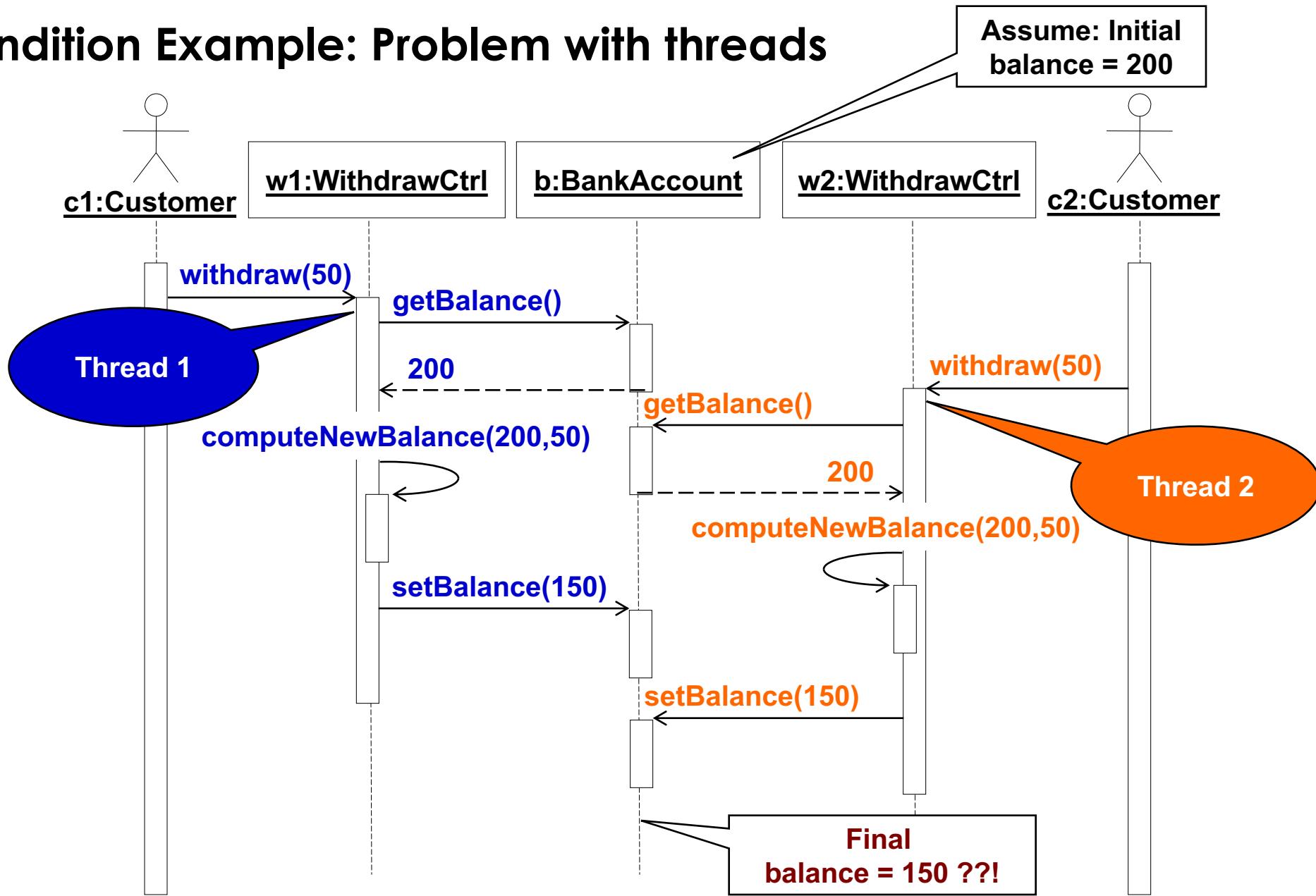
3. Concurrency

- Used to address nonfunctional requirements such as: Performance, Response time, latency, availability
- Two objects are **inherently concurrent** if they can receive events at the same time without interacting
 - Good source for identification: Objects in a sequence diagram that are independent from each other
- Inherently concurrent objects can be assigned to separate threads of control
 - Objects with **mutual exclusive activity** can be folded into a single thread of control.

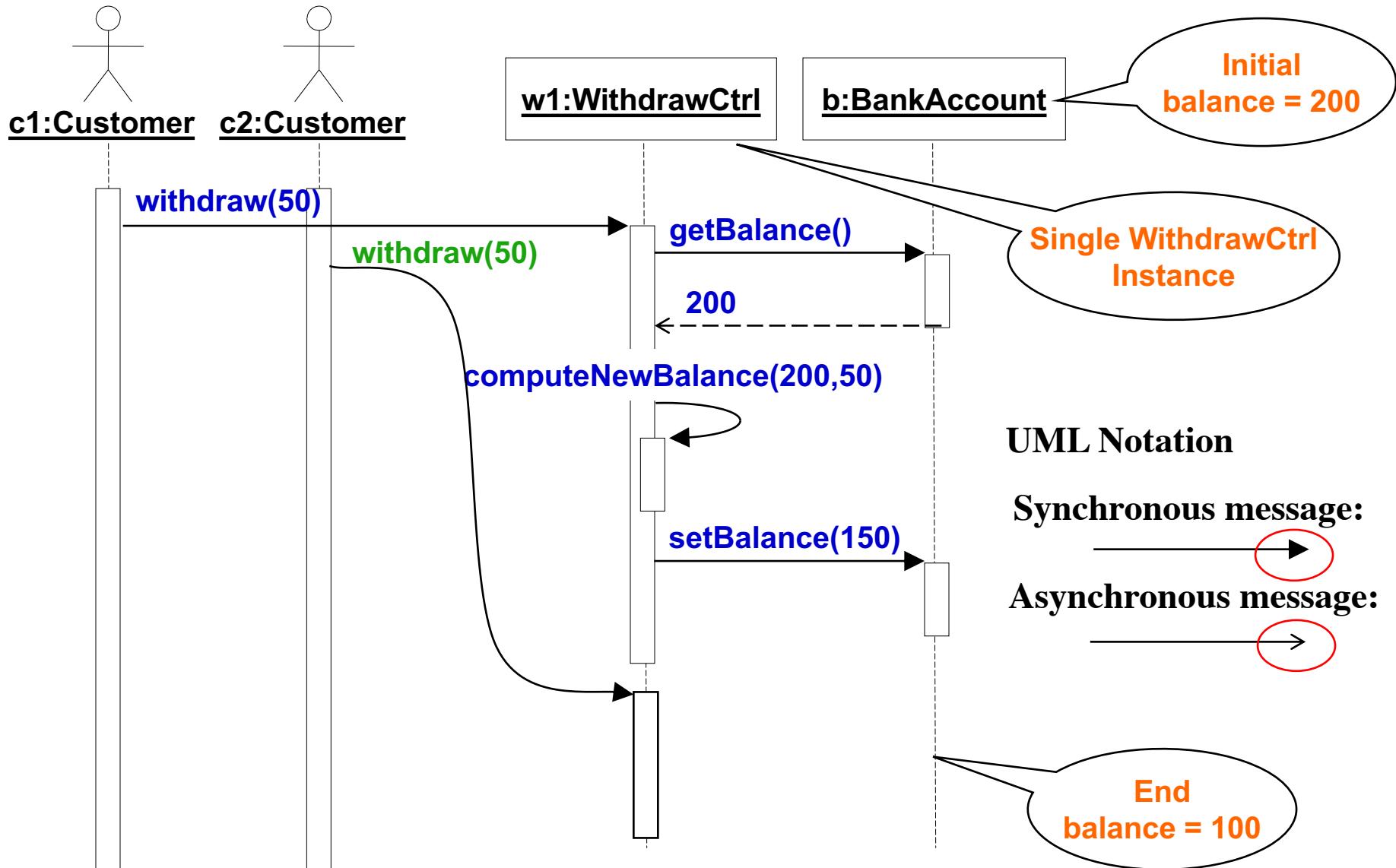
Thread of Control

- A **thread of control** is a path through a set of state diagrams where only a single object is active at any time
 - A thread remains within a state diagram until the object sends an event to a second object, which then becomes active (The first object then waits for another event)
 - **Thread splitting:** Object does a non-blocking send of an event to another object, so it does not wait
- Concurrent threads can lead to race conditions
- A **race condition** 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.

Race Condition Example: Problem with threads



Solution: Synchronization of Threads



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 concurrency. Two types:
- **Physical concurrency:**
 - Threads are provided by hardware (Multiprocessors, multi-cores and computer networks)
- **Logical concurrency:**
 - Threads are provided by software (usually provided by threads packages)
- **Study threads well, before you use them.**
 - Course IN2147, Parallel Programming (Prof. Michael Gerndt).

Implementing Concurrency (2)

- In both cases, - physical concurrency as well as logical concurrency - we have to solve the scheduling of the threads:
 - Which thread runs when?
- Today's operating systems provide a variety of scheduling mechanisms:
 - Round robin, time slicing, collaborating processes, interrupt handling
- Concurrency introduces problems such as starvation, deadlocks, fairness
- Sometimes you have to solve the scheduling problem ourselves
 - Software control is system design topic 7
 - Course: IN0009, Betriebssysteme(Operating Systems), Prof. Uwe Baumgarten.

Overview of Today's Lecture

- Cohesion and Coupling
 - Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors
 - Concurrency
- Hardware/Software Mapping
- Persistent Data Management
 - Global Resource Handling
 - Software Control
 - Boundary Conditions

4. Hardware Software Mapping

This system design activity addresses two questions:

1. How shall we realize the subsystems: With hardware or with software?
2. How do we map the object model onto the chosen hardware and/or software?
 - Mapping the Objects:
 - Processor, Memory, Input/Output devices
 - Mapping the Associations:
 - Network connections.

Mapping the Objects

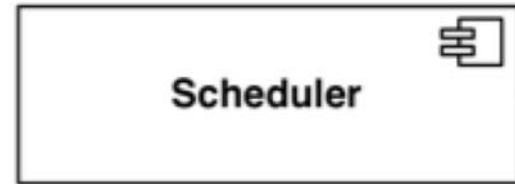
- **Control Objects -> Processor**
 - Is the computation rate too demanding for a single processor?
 - Can we get a speedup by distributing objects across several processors?
 - How many processors are required to maintain a steady state load?
- **Entity Objects -> Memory**
 - Is there enough memory 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?

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 linktime 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

Component Diagram

- Illustrates dependencies between components at design time, compilation time and runtime

• Deployment Diagram

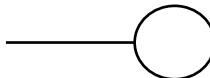
- Illustrates the distribution of components on concurrent processes at run-time
- Deployment diagrams use nodes and connections to depict the physical resources in the system

UML Component Diagram

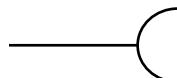
- Used to model the top-level view of the system design in terms of components and dependencies among the **components**. Components can be
 - source code, linkable libraries, executables
- The dependencies (edges in the graph) are the **connectors** are shown as dashed lines with arrows from the client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

UML Interfaces: Lollipops and Sockets

- A UML interface describes a group of operations provided or required by a UML component
 - There are two types of interfaces: provided and required interfaces.
 - A **provided interface** is modeled using the lollipop notation

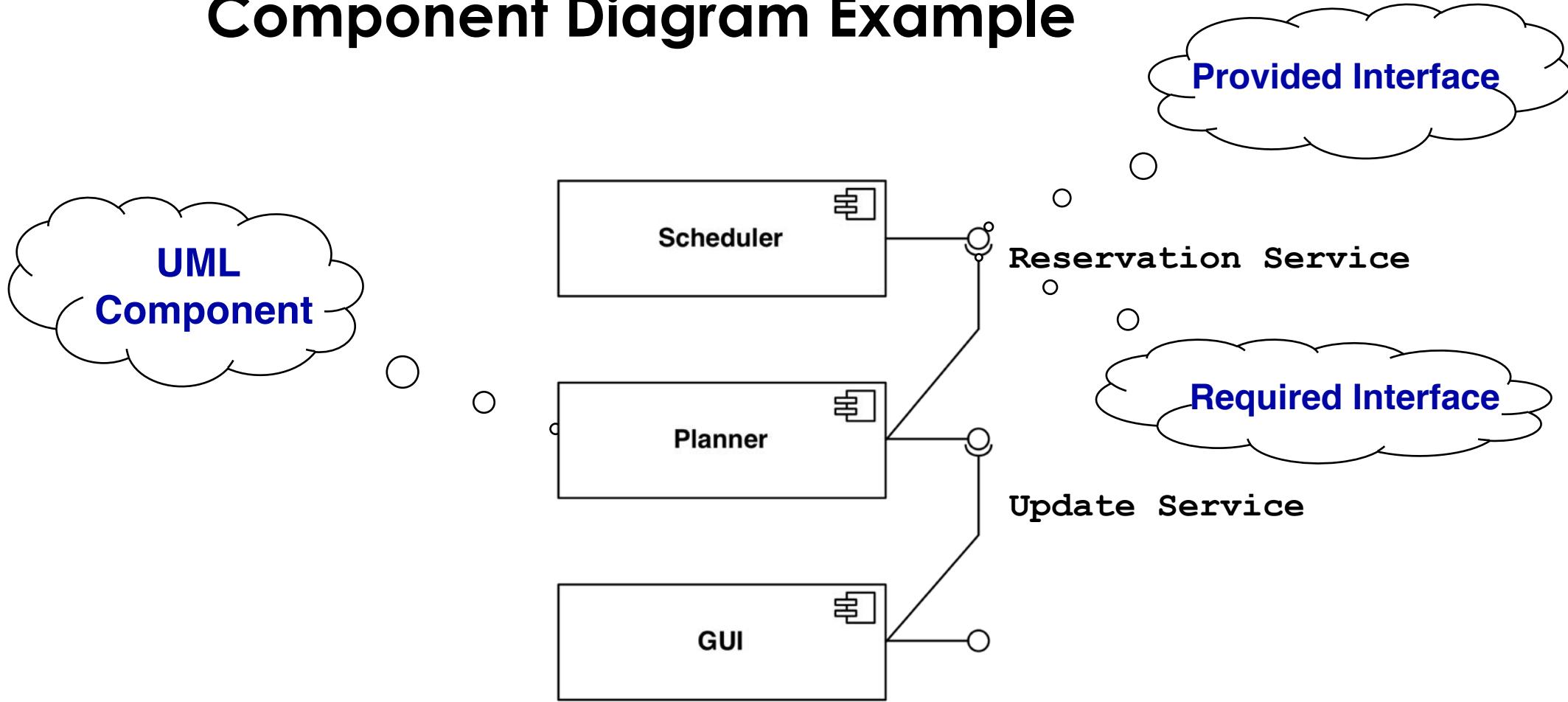


- A **required interface** is modeled using the socket notation



- 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



Two New UML Diagram Types

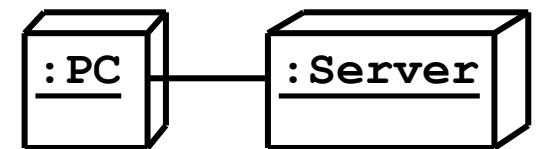
- Component Diagram
 - Illustrates dependencies between components at design time, compilation time and runtime

Deployment Diagram

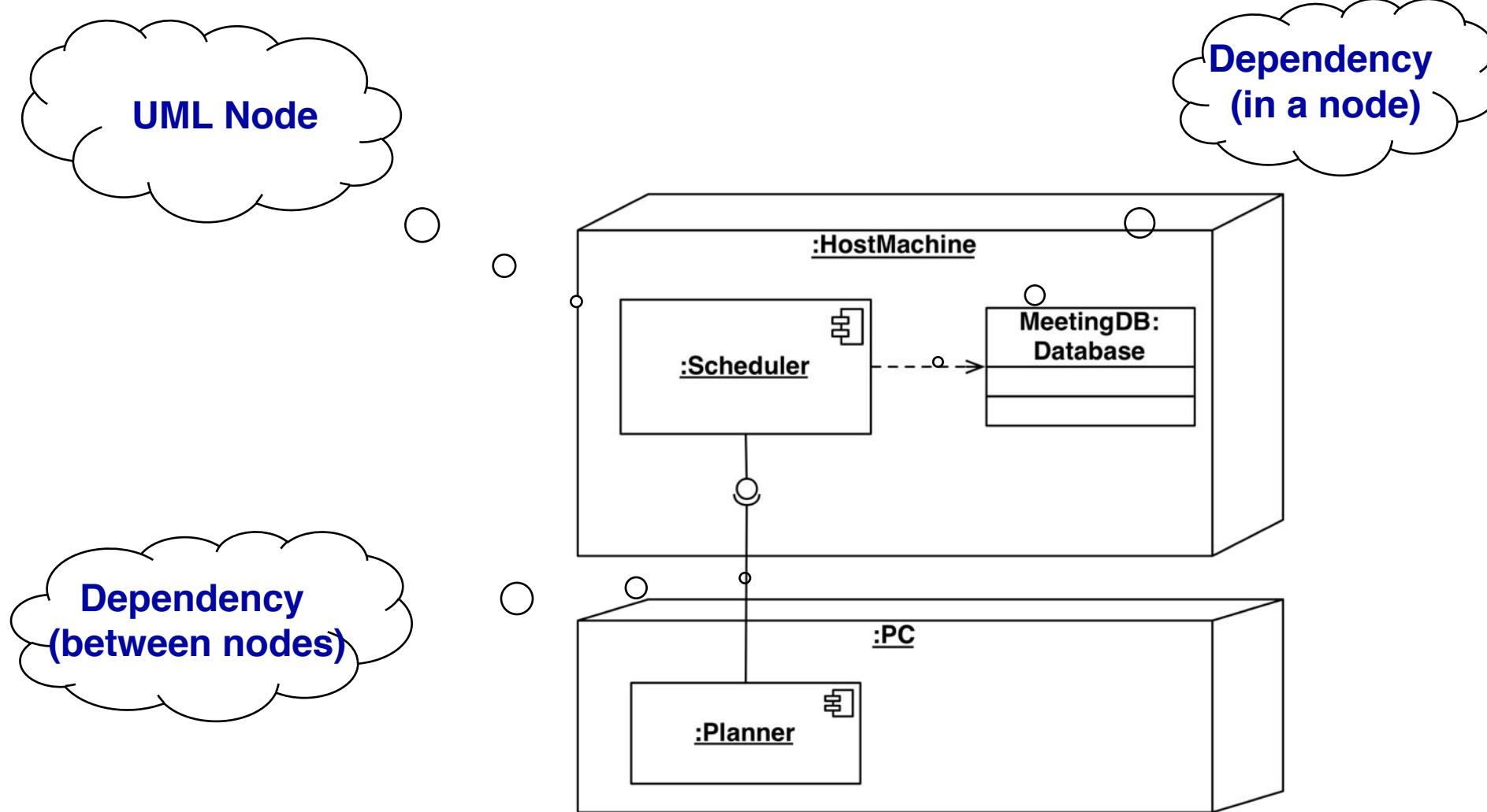
- Illustrates the distribution of components at run-time
- Deployment diagrams use nodes and connections to depict the physical resources in the system

Deployment Diagram

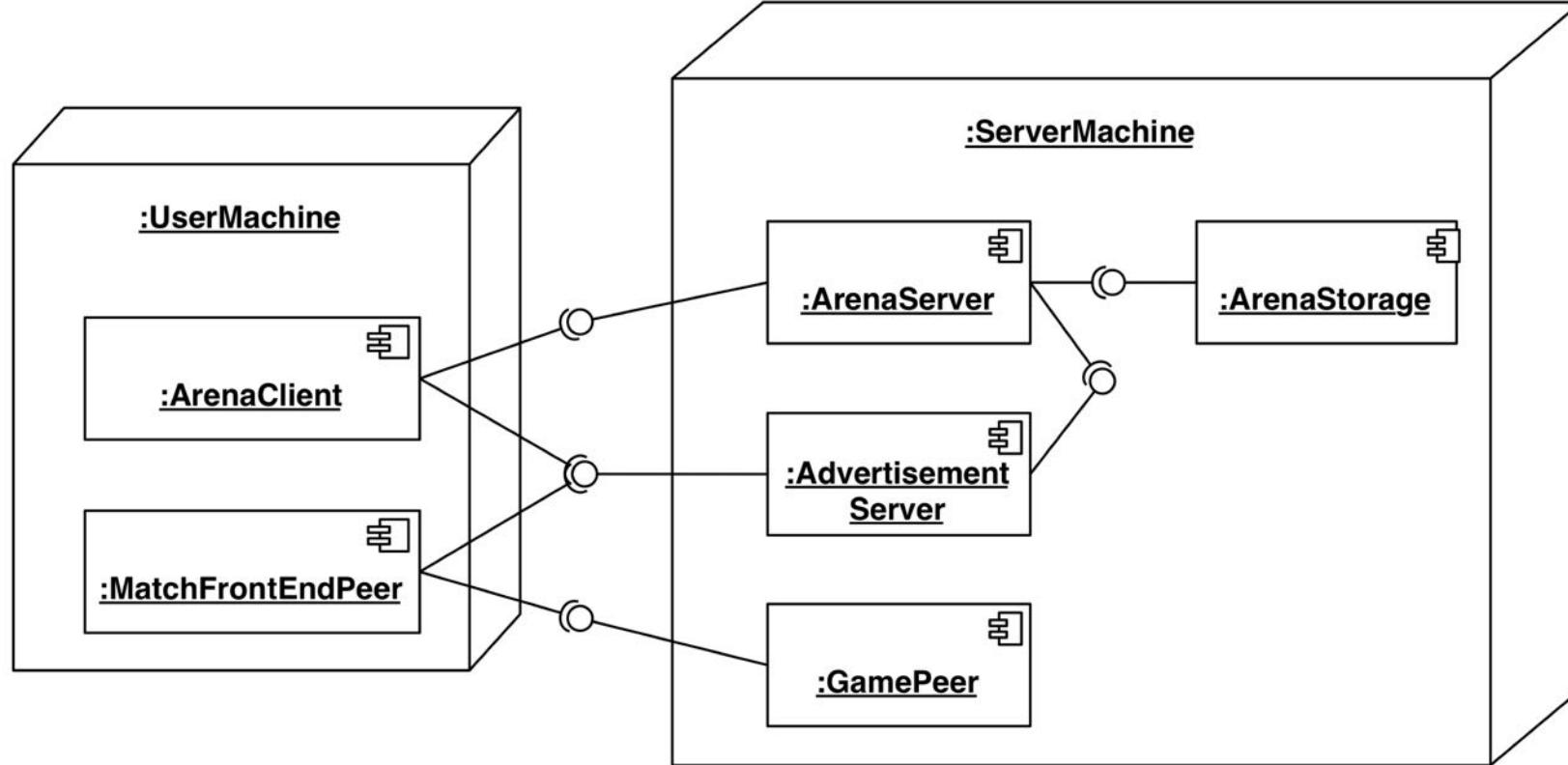
- Deployment diagrams are useful for showing a design after these system design decisions have been made:
 - Subsystem decomposition
 - Concurrency
 - Hardware/Software Mapping
- A **deployment diagram** is a graph of nodes and connections (“communication associations”)
 - Nodes are shown as 3-D boxes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



Combination of Deployment and Component Diagram



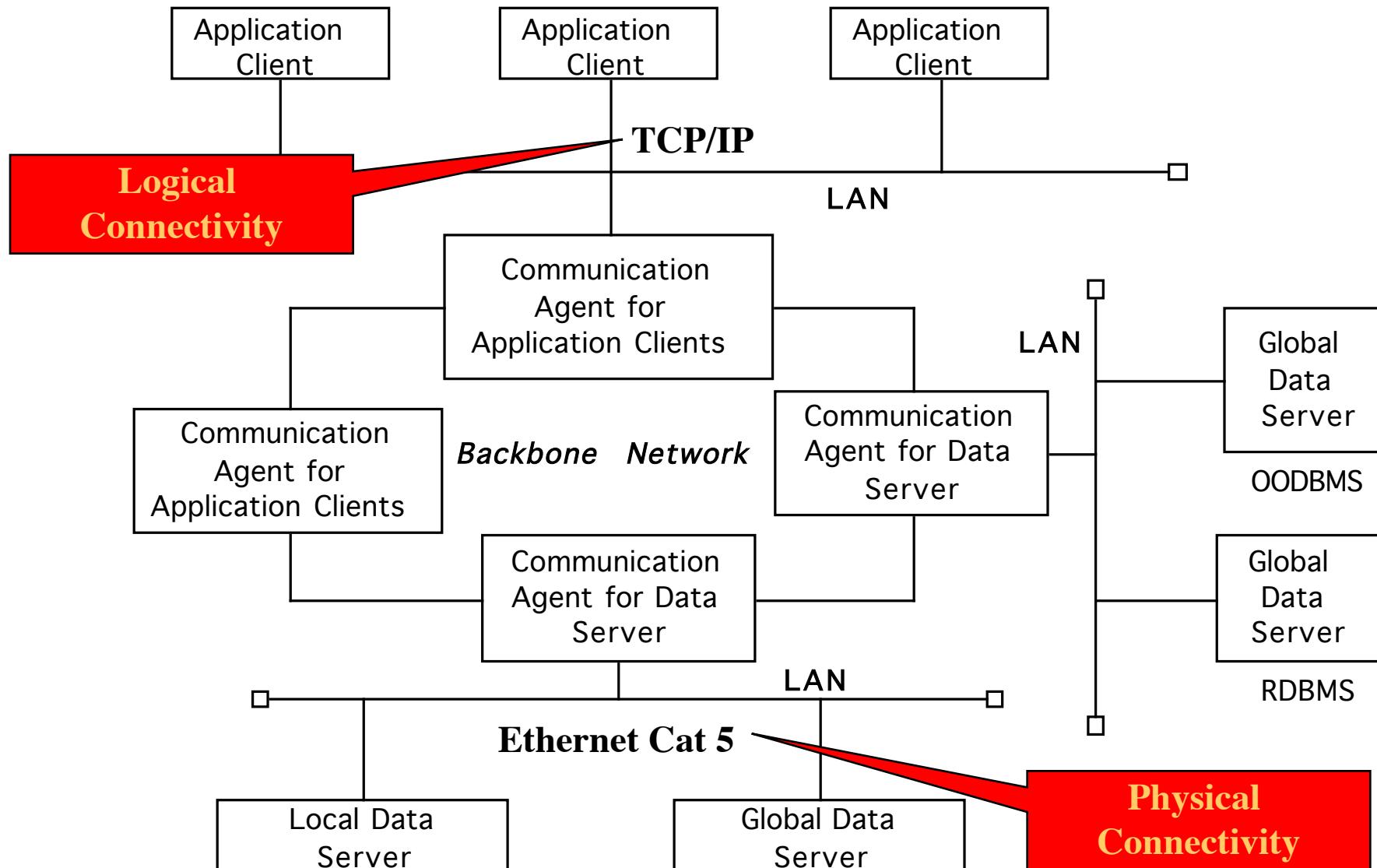
Another Example of a Combination of Deployment and Component Diagram



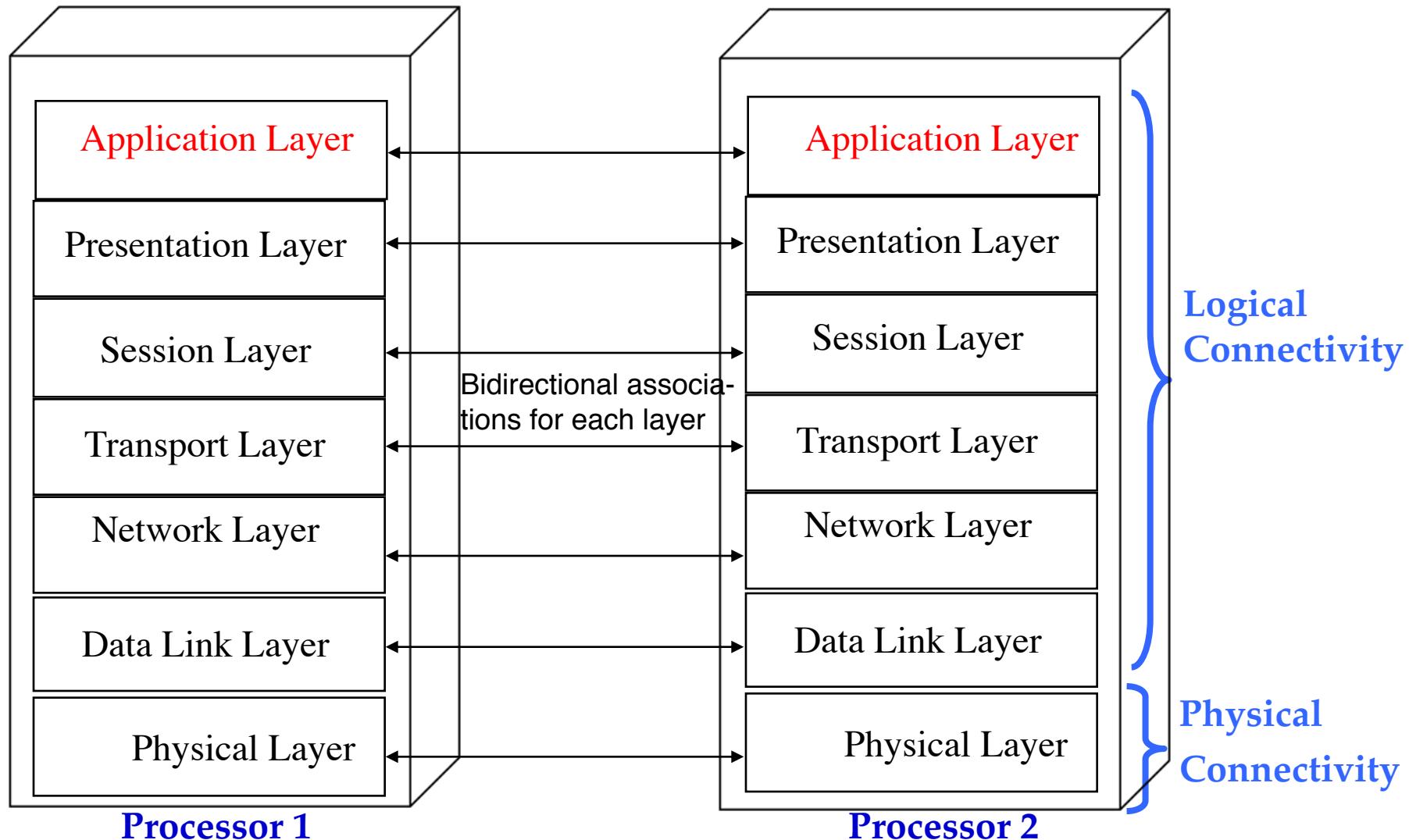
Mapping the Associations: Connectivity

- Describe the physical connectivity
 - Physical layer in the OSI reference model:
 - IN0010: Foundations: Networking and Distributed Systems (Prof. Carle)
 - Describes which associations in the object model are mapped to physical connections
- Describe the logical connectivity (subsystem associations)
 - Associations that do not map to 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.

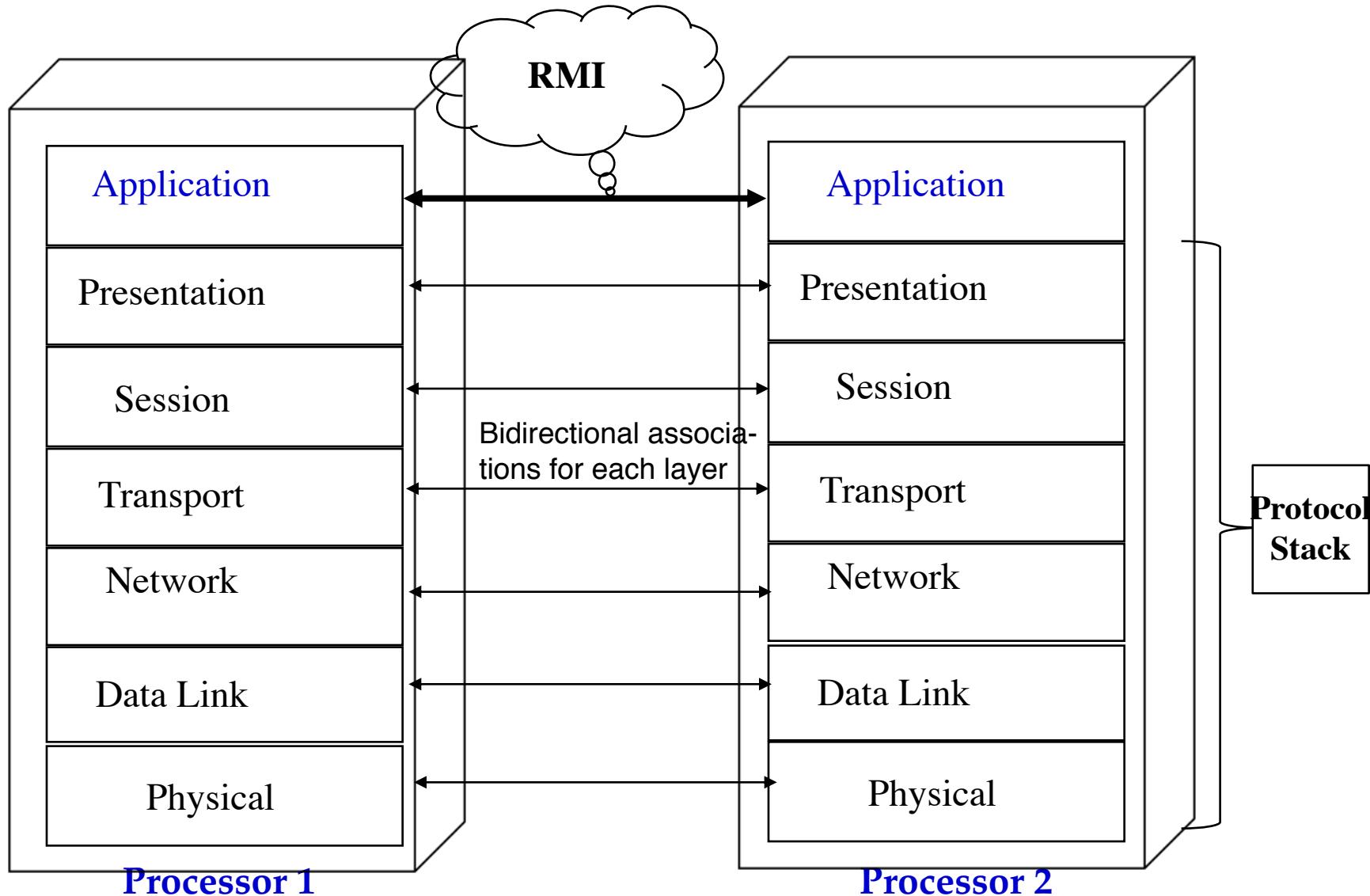
Example: Informal Connectivity Drawing



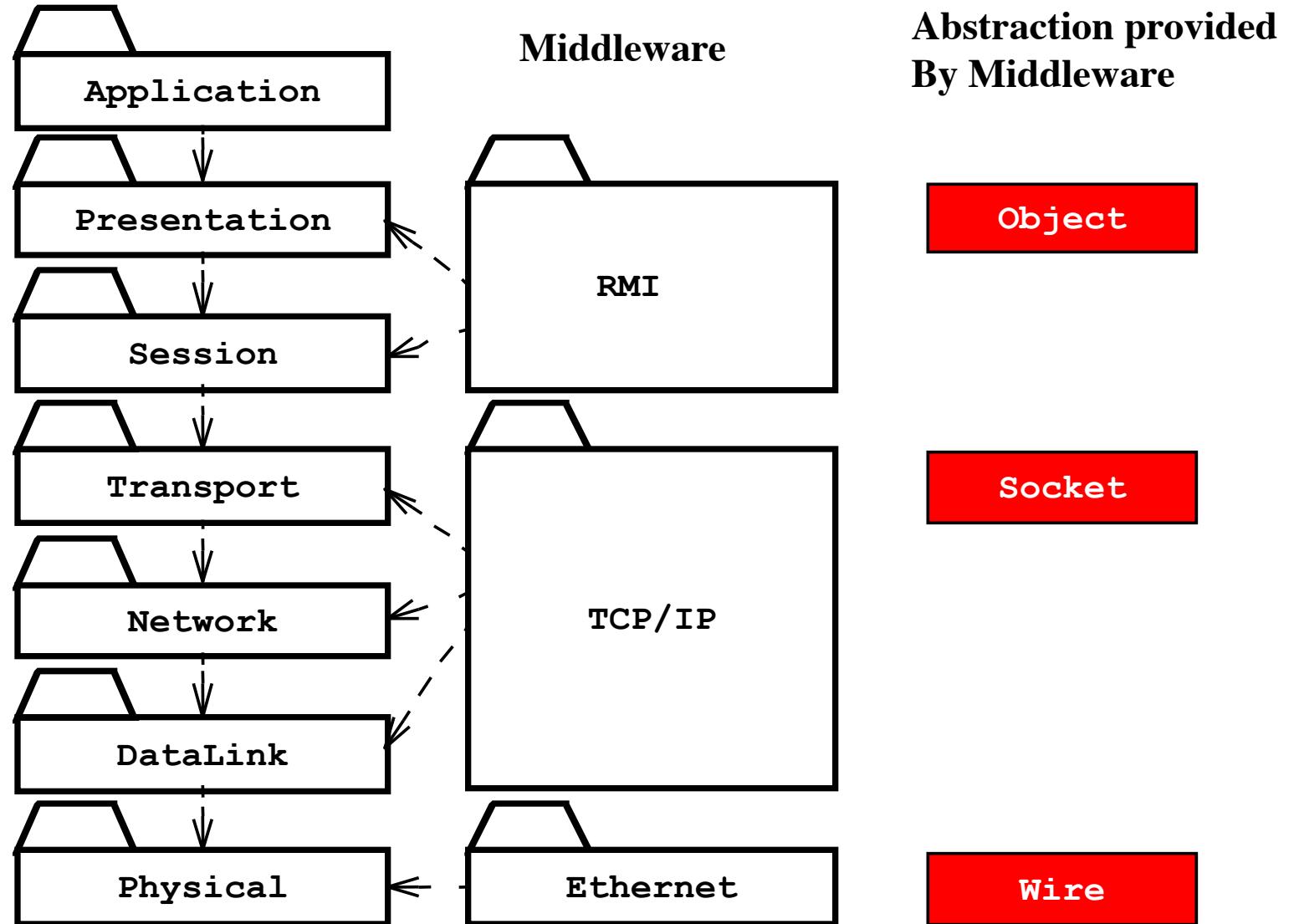
Logical vs Physical Connectivity and the relationship to Subsystem Layering



The Application Layer provides the Abstractions of the “New System”. It is usually layered itself



Middleware Allows Focus On Higher Layers



Overview of Today's Lecture

- Cohesion and Coupling
 - Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors
 - Concurrency
 - Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
 - Software Control
 - Boundary Conditions

5. Persistent Data Management

- All classes of type entity in the system model need to be persistent:
 - **Persistency**: A class is persistent, if the values of their attributes have a lifetime beyond a single execution
- A persistent class can be realized with one of the following mechanisms:
 - **File system**:
 - If the data are used by multiple readers but a single writer
 - **Database system**:
 - If the data are used by concurrent writers and readers.

Mapping an Object Model to a Database

- UML object models can be mapped to relational databases:
 - Some degradation occurs because all UML constructs must be mapped to a single relational database construct - the **table**
- Mapping of classes, attributes and associations
 - Each *class* is mapped to a table
 - Each class *attribute* is mapped onto a column in the table
 - An *instance* of a class represents a row in the table
 - A *many-to-many association* is mapped into its own table
- Methods are not mapped.
- Object Relational Mappings (ORM):
 - More Details in the Lecture on Model Transformations June 7 2018

Overview of Today's Lecture (05 17 2018)

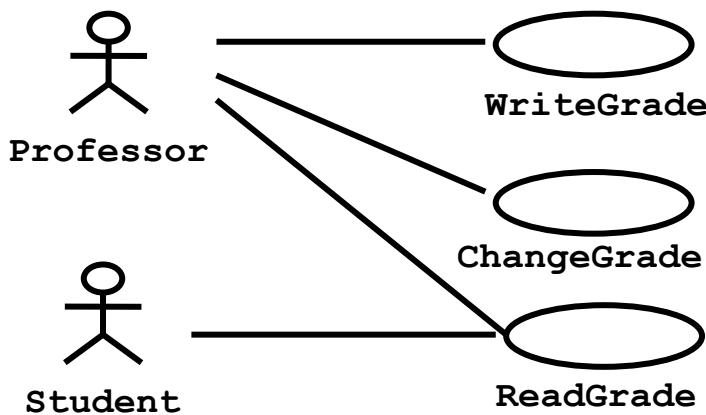
- Cohesion and Coupling
- Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
 - Software Control
 - Boundary Conditions

6. Global Resource Handling

- Addresses access control
- Describes access rights for different classes of actors
- Describes how objects can be guarded against unauthorized access.

Defining Access Control

- In multi-user systems different actors usually have different access rights to functions and data
- How do we model these access rights?
 - During *analysis* we model access rights by associating use cases with the actors



- During *system design* we model access rights by determining which objects are shared among actors
 - For this purpose we introduce the **access matrix**.

Access Matrix

- An **access matrix** models the access of actors on classes:
 - 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 by the actor on instances of the class.

Access Matrix Example

The diagram illustrates an Access Matrix Example. It features three conceptual bubbles: 'Actors' (containing Operator, LeagueOwner, Player, and Spectator), 'Classes' (containing Arena, League, Tournament, and Match), and 'Access Rights' (containing various methods like <<create>>, archive(), end(), etc.). Arrows point from the 'Actors' bubble to the matrix rows and from the 'Classes' and 'Access Rights' bubbles to the matrix columns.

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner	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()

Access Matrix

- In general, the access matrix is a sparse matrix
- There are 3 different implementations of the access matrix:
 - Global access table
 - Access control list
 - Capabilities.

Access Matrix Implementations

- **Global access table:** Represents every non-empty 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()

		Arena
		Operator
LeagueOwner	<<create>>	createUser()
	view ()	view ()

Access Matrix Implementations (2)

→ Access control list:

- Associates a *list of* (actor,operation) pairs with the *class* being 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,operations) pair with an actor*
 - A capability allows an actor to gain control access to an object of the class by calling one of the class operations described in the capability.

Access Control List 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()

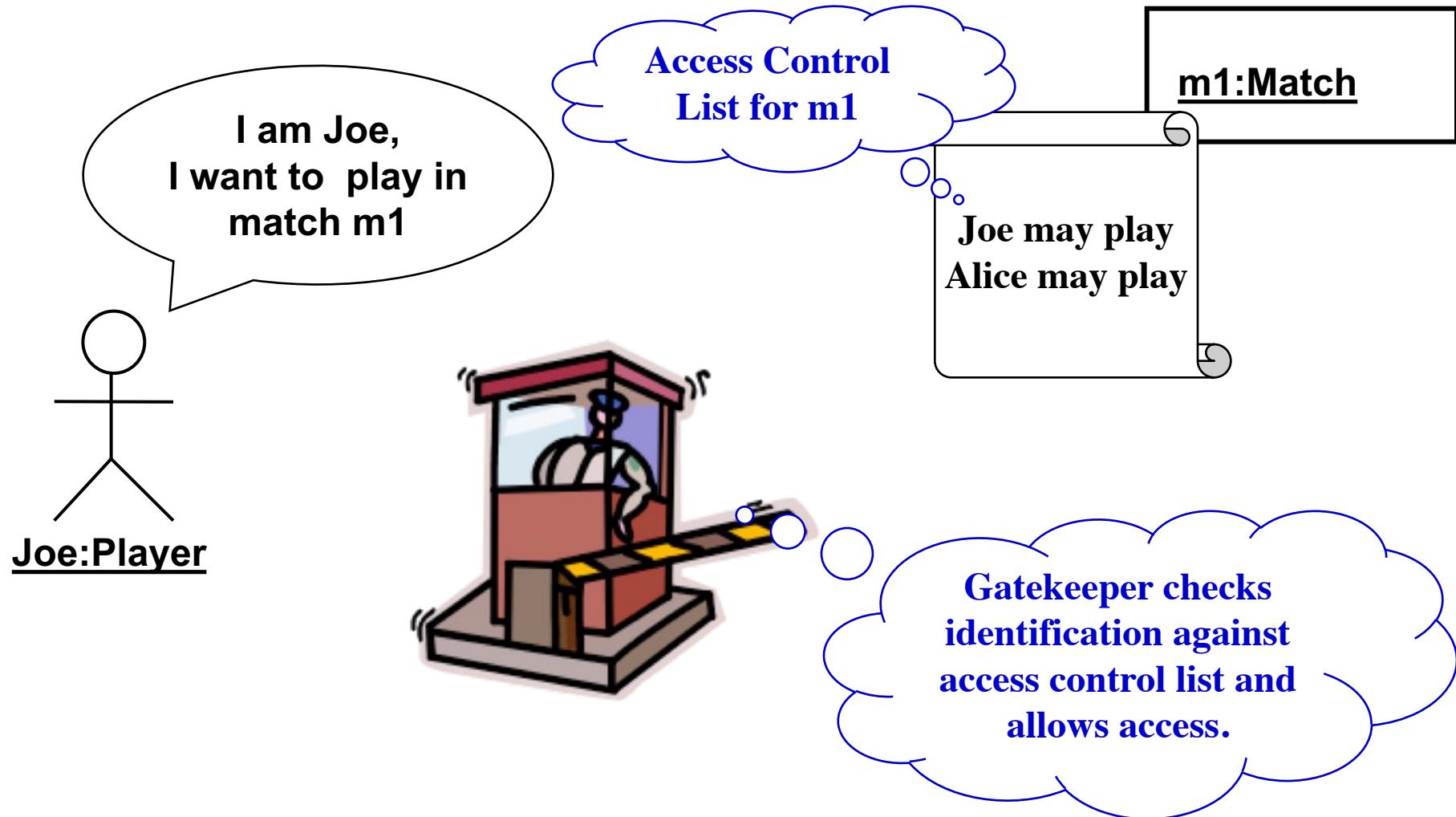
Access Control List Example

	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()

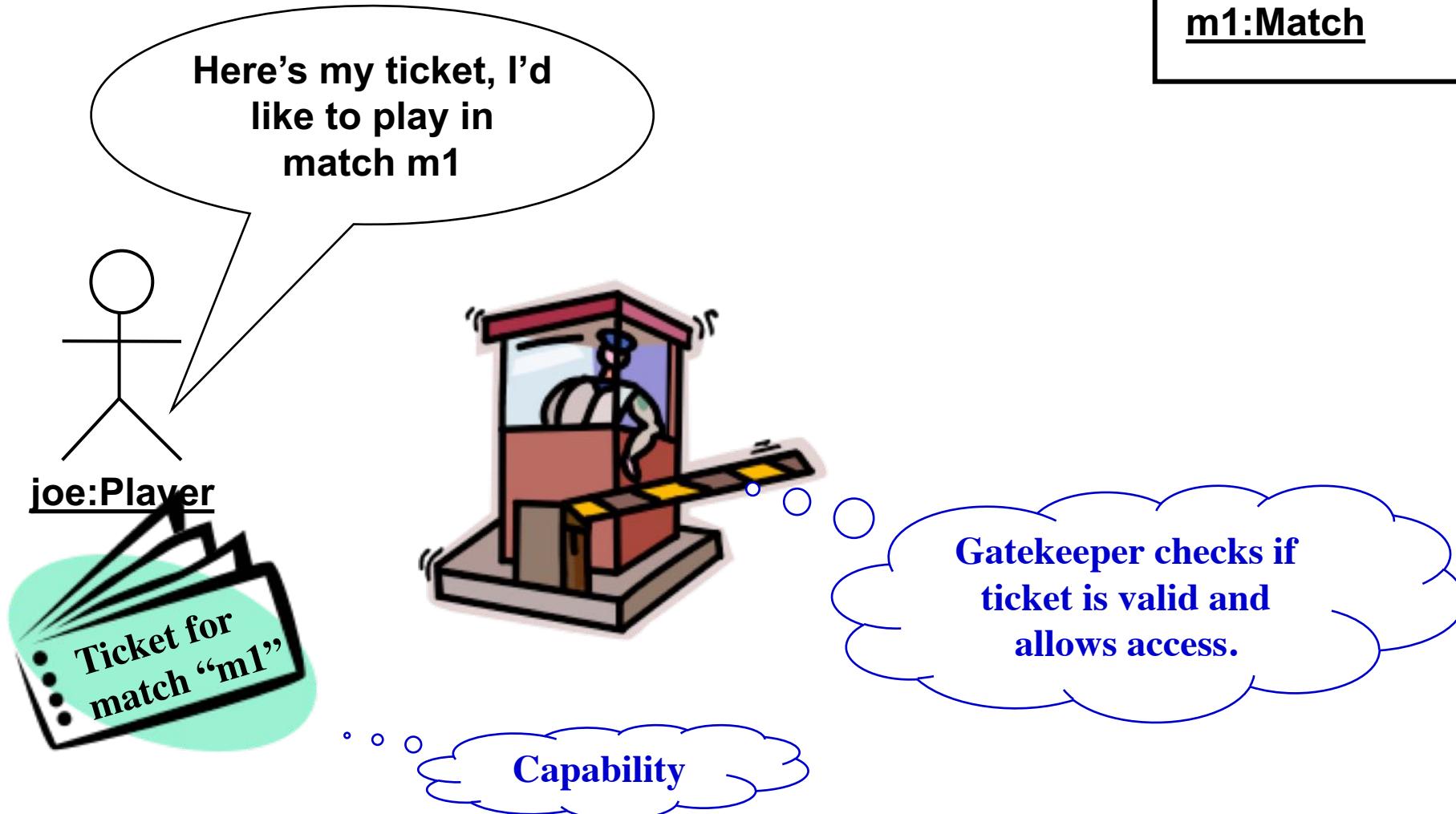
Access Control List Example



Access Control List Realization (ctd)

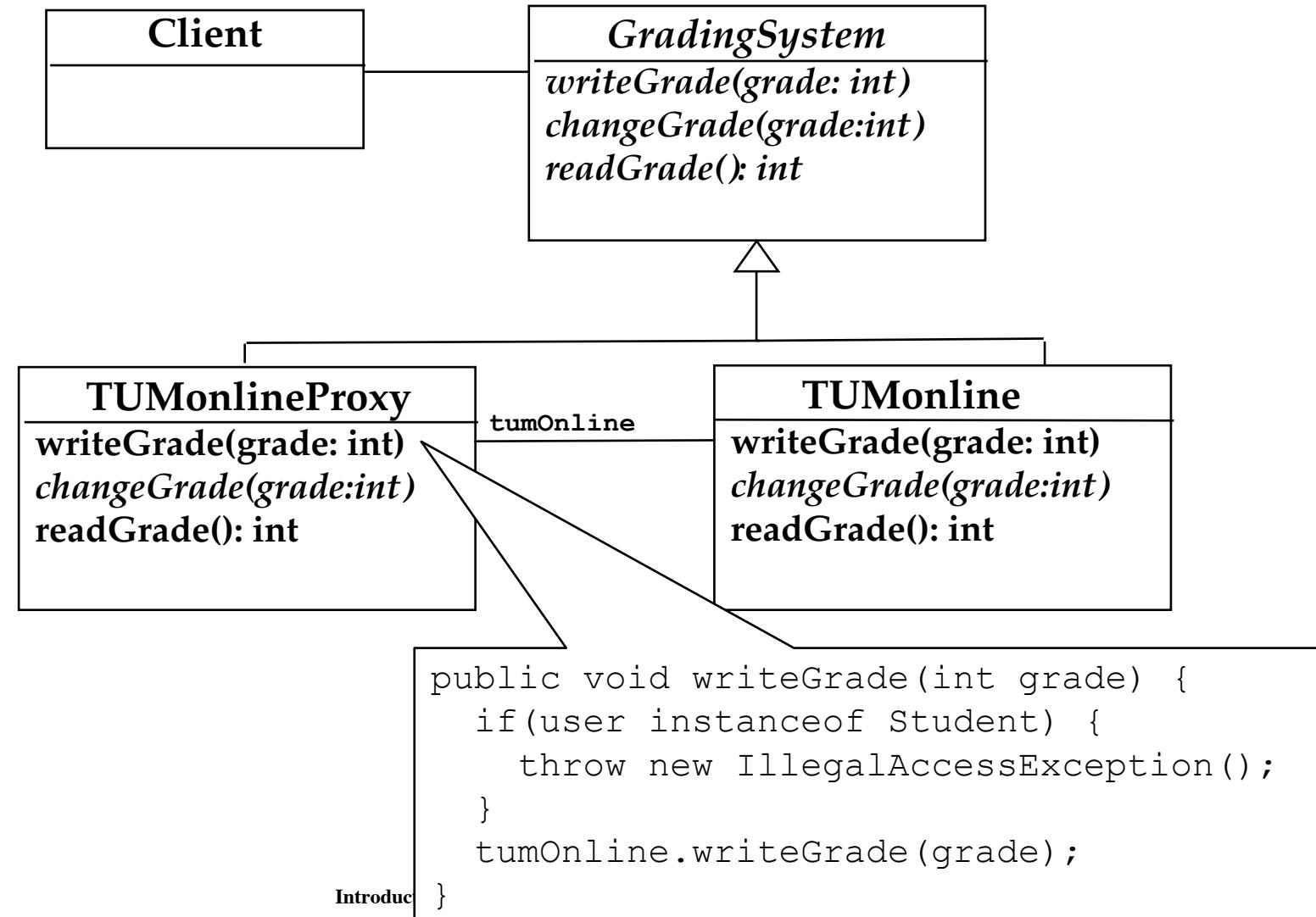


Capability Realization



Example: Grade Information Access

- UML Class Diagram



Overview of Today's Lecture

- Cohesion and Coupling
- Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

7. Software Control

You have 2 system design choices:

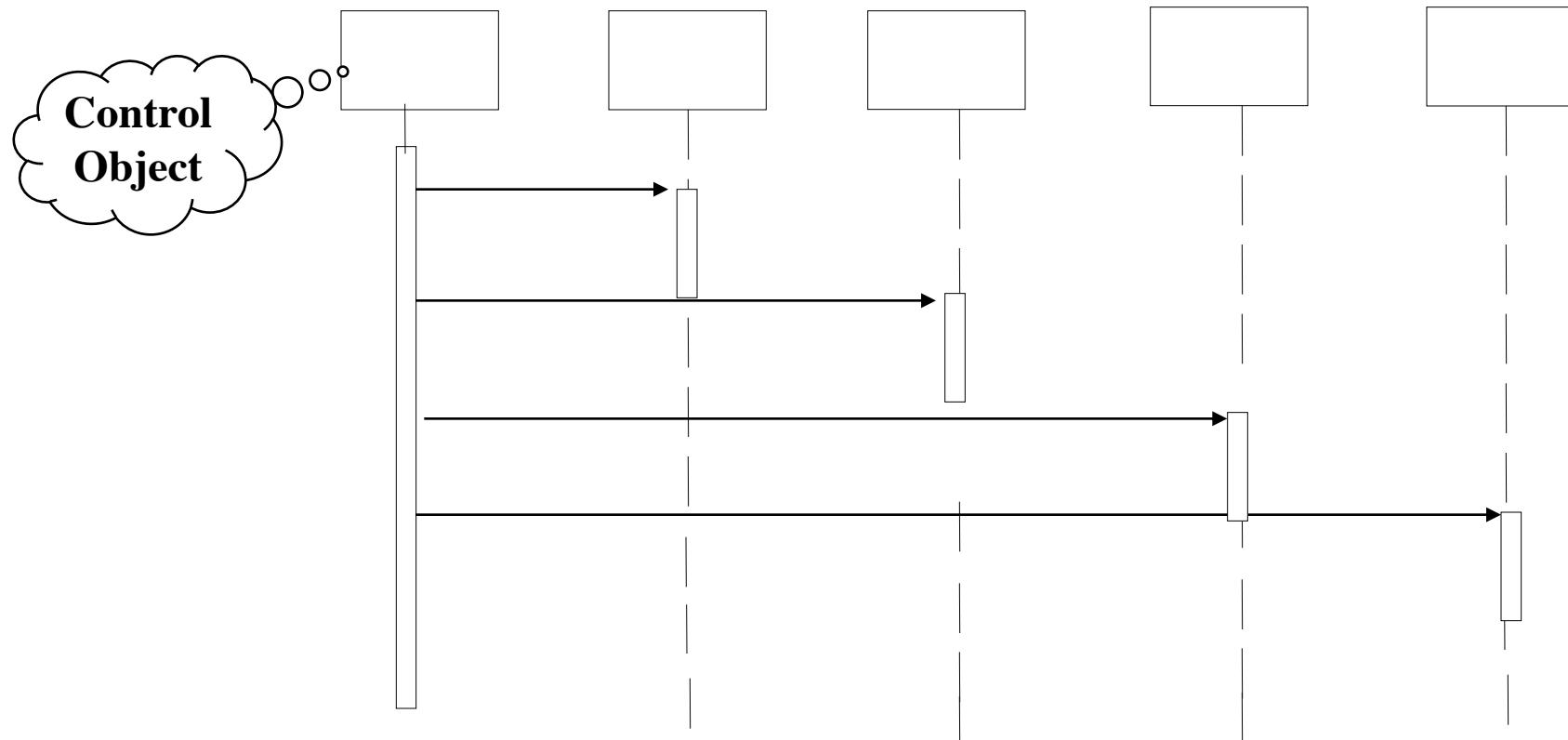
- Implicit software control
 - rule-based systems, logic programming
- Explicit software control
 - Centralized control
 - Decentralized control.

Sequence Diagrams can be used to determine the Decentralization of a System

- The structure of the sequence diagram helps us to determine how decentralized the system is
- We distinguish 2 sequence diagram structures (Ivar Jacobson):
 - Fork Diagrams and Stair Diagrams.

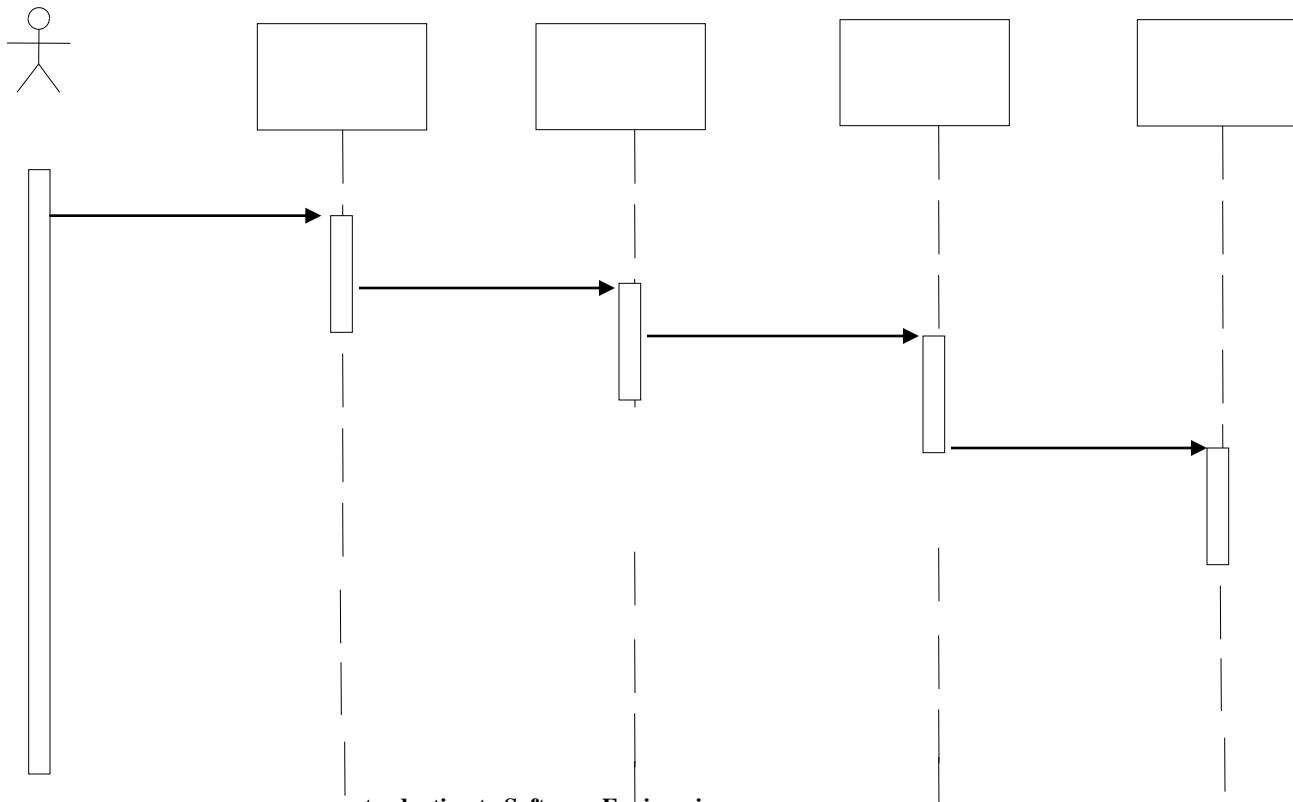
Fork Diagram

- **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

- **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



Explicit software control: Centralized vs. decentralized

- **Centralized control:**
 - **Procedure-driven:** Control resides within the program code
 - Example: Programming languages that have a main program
 - **Event-driven:** Control resides within a dispatcher calling other functions via socalled callbacks
 - Example: Event loop in an User Interface listening to keyboard and mouse events
- **Decentralized control:**
 - Control resides in several independent objects
 - Examples: Message based systems, RMI
 - Decentralized control promises a possible speedup by mapping the objects on different processors, but require 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: Additional communication overhead
 - 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.

Overview of Today's Lecture

- Cohesion and Coupling
- Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors

- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control

 Boundary Conditions

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.

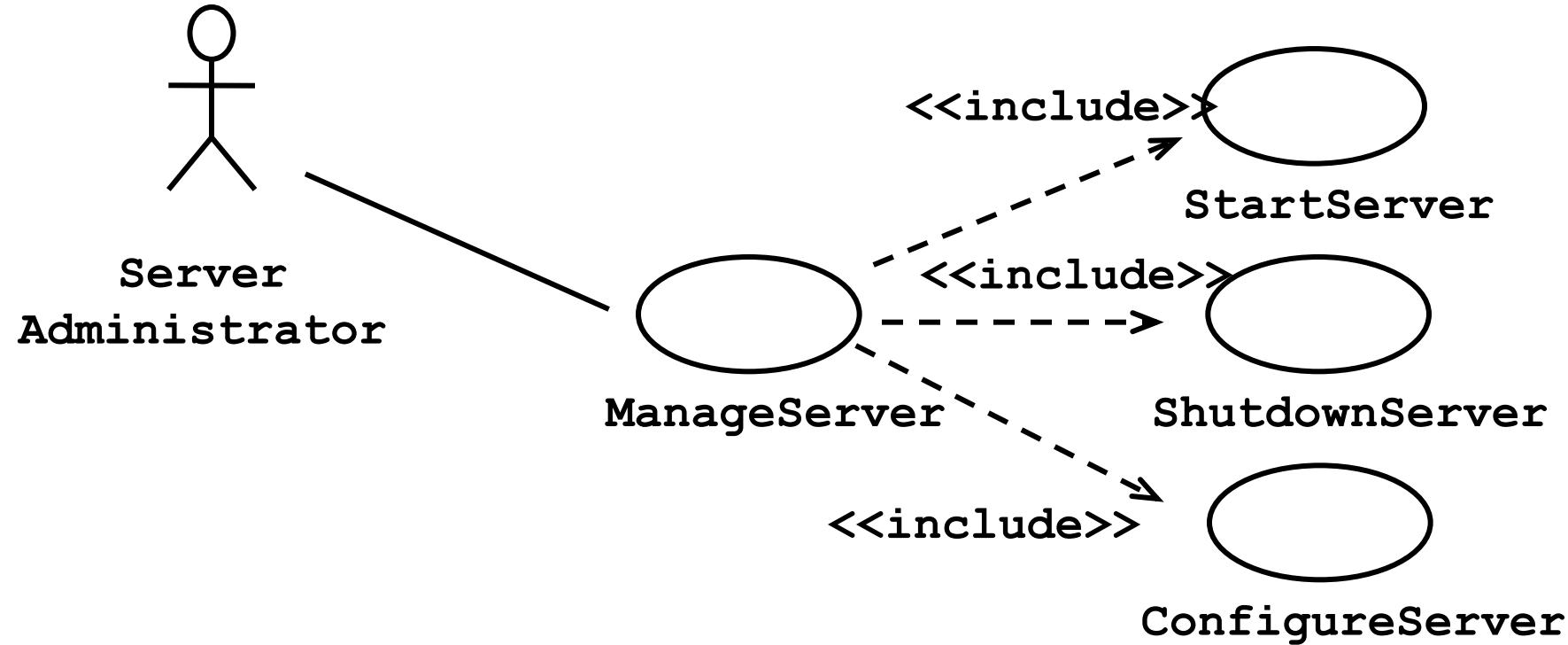
Modeling Boundary Conditions

- Boundary conditions are best modeled as use cases
- 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 a client server system

- Let us assume, we identified the subsystem Server 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 Server.

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.

Morning Quiz 06a

- Start Time: **8:00**
- End Time: **8:10**
- To participate in the quiz, use ArTEMiS
- Click on Open Quiz or Start Quiz
- The Lecture starts at 8:10

Remaining Time: **46 s**

Saved: never

● Connected

Submit

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	Start exercise
Good Morning Quiz 06a		Open Quiz 

Only click on Submit when you have entered all answers!

Object-Oriented Software Engineering

Using UML, Patterns, and Java

Object Design

Bernd Bruegge

Chair for Applied Software Engineering
Technische Universität München

24 May 2018

EIST



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)
 - System Design (Chapter 6 and 7)
- **Today: Object Design (Chapter 8)**
- No Lecture on 31 May 2018
 - Fronleichnam
- Next Lectures
 - 07 June 2018:
 - Model Transformations and Refactorings
 - 14 June 2018: Design Patterns
 - 21 June 2018: Lifecycle Modeling
 - 28 June 2018:
 - Configuration-, Build- and Release Management
 - 05 July 2018: Testing
 - 12 July 2018: Project Management
 - 19 July 2018: Repetitorium

Roadmap for Today's Lecture

- **Context and Assumptions**

- We completed Chapter 1 to 7 in the OOSE text book by Bruegge and Dutoit

- **Content of this lecture**

- We will finish Lecture 06: System Design II (Slide 106-132)
- We introduce how to use patterns and how to specify interfaces
- We complete Chapter 8 and Chapter 9

- **Objective:** At the end of this lecture you are able to

- Explain different types of reuse
- Distinguish between different types of inheritance
- Differentiate between structural, behavioral and creational design patterns
- Select an appropriate design pattern for a specific problem.

Overview of Today's Lecture

→ Miscellaneous

- System Design
 - Global Resource Handling
 - Software Control
 - Boundary Conditions
- Object Design
 - Purpose
 - Object Design Activities
 - Types of Reuse: Black Box and White Box Reuse
 - The Use of Inheritance in Object Design
 - Discovering Inheritance: Generalization vs Specialization
- Design Patterns

No tutor groups next week (May 28 – June 01)

- No tutor groups in the week between May 28 and June 01
- Homework Sheet 05
 - Will be discussed in the week after June 04 – June 06
- You have 2 weeks to solve Homework Sheet 06

University Election: How to vote

TUM Campus-Management-System TUMonline Achtung! Wartung: 8.Mai, 16:00 - 21:00

Präident/in:
Wolfgang A. Herrmann

Suche ▾ Logout Hilfe ▾ Auswahl Detailansicht Bearbeitung

Technische Universität München Visitenkarte/Arbeitsplatz

Herr Herrmann, Wolfgang A., Prof. Dr. Dr. h.c. mult.

E-Mail wolfgang.a.herrmann@tum.de



Forschung & Lehre Studium Ressourcen Dienste

- Abschlussarbeiten
- Beitragsstatus
- Organisationen & Zuständigkeiten
- Tipp der Woche

- LV-Bookmarks
- Bewerbungen
- TUM-Kennung
- Token-Verwaltung

- LV-Bookmarks Planansicht
- Studierendenkartei
- Benutzungsrichtlinien
- TUMcard Passfoto upload

- ORCID ID
- LV-An/Abmeldung
- Terminkalender
- Software

- Prüfungsan-/abmeldung
- persönliche Einstellungen
- Universitätsbibliothek

- Prüfungsergebnisse
- Anmelde-Log
- Kennwort ändern

- Studienerfolgsnachweis
- Correspondenzadresse
- TUM Mailbox (Exchange)

- Anmerkungen / Zeugnisausdruck
- @ E-Mail-Adressen
- Online-Speicher (NAS)

©2018 Technische Universität München. Alle Rechte vorbehalten. | TUMonline powered by CAMPUSonline® | Anleitungen | Datenschutz | Impressum | Feedback

<http://www.asta.tum.de/en/studentische-vertretung/university-elections/postal-vote>

University Election: How to vote

The screenshot shows the TUMonline portal interface. On the left, there is a sidebar with a tree view of university departments. The main area displays a table of documents under the heading 'Dokumente'. A blue arrow points from the 'Briefwahl beantragen' button in the 'Wahlbenachrichtigung und Briefwahlantrag' row to the right. Below this, there is another table for 'Leistungsbestätigungen' and a section for 'Briefe'.

Dokumente

Bezeichnung	Semester	Sprache	Aktion
Immatrikulationsbescheinigung	Sommersemester 2018	Deutsch	Drucken
Immatrikulationsbescheinigung für MVV	Sommersemester 2018		Drucken
Studienverlaufsbescheinigung			Drucken
Zahlungsbestätigung	Sommersemester 2018		Drucken
Antrag auf Beurlaubung			Drucken
Antrag auf Exmatrikulation			Drucken
Wahlbenachrichtigung und Briefwahlantrag		Deutsch	Drucken → Briefwahl beantragen
Rentenbescheinigung			Drucken

Leistungsbestätigungen

Bezeichnung	Semester	Studium	Aktion
Einzelleistungsnachweis		Bitte wählen...	Drucken

Briefe

Betreff	Erstellungsdatum	Erstdruckdatum durch den Studierenden	Studien-ID	Ersteller	Aktion
Keine Einträge vorhanden					

©2018 Technische Universität München. Alle Rechte vorbehalten. | TUMonline powered by CAMPUSonline® | [Anleitungen](#) | [Datenschutz](#) | [Impressum](#) | [Feedback](#)

<http://www.asta.tum.de/en/studentische-vertretung/university-elections/postal-vote>

University Election: How to vote

Briefwahlbeantragen

Ich beantrage die Aushändigung/Übersendung der Briefwahlunterlagen

Auswahl Die Briefwahlunterlagen sollen an meine übliche Korrespondenzanschrift geschickt werden
 werde ich persönlich im Wahlamt, Stammgelände, Raum 0228, Arcisstr. 21, 80333 München abholen
 Die Briefwahlunterlagen sollen an eine davon abweichende Anschrift geschickt werden:

[Senden und Schließen](#) [Abbrechen](#)

X

<http://www.astा.тum.de/en/studentische-vertretung/university-elections/postal-vote>

Overview of Today's Lecture

- Miscellaneous
- System Design (finishing the lecture from May 17)
 - Global Resource Handling
 - Software Control
 - Boundary Conditions
- Object Design
 - Purpose
 - Object Design Activities
 - Types of Reuse: Black Box and White Box Reuse
 - The Use of Inheritance in Object Design
 - Discovering Inheritance: Generalization vs Specialization
- Design Patterns

Status of the System Design Lecture from May 17 2018

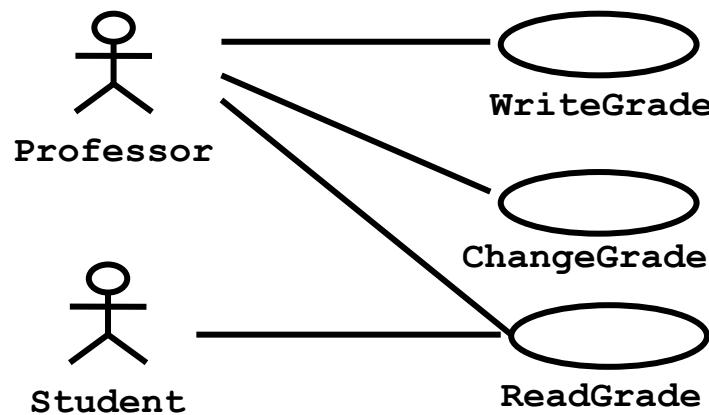
- ✓ Cohesion and Coupling
- ✓ Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors
- ✓ Concurrency
- ✓ Hardware/Software Mapping
- ✓ Persistent Data Management
- ➡ Global Resource Handling
 - Software Control
 - Boundary Conditions

6. Global Resource Handling

- Addresses access control
- Describes access rights for different classes of actors
- Describes how objects can be guarded against unauthorized access.

Defining Access Control

- In multi-user systems different actors usually have different access rights to functions and data
- How do we model these access rights?
 - During *analysis* we model access rights by associating use cases with the actors



- During *system design* we model access rights by determining which objects are shared among actors
 - For this purpose we introduce the **access matrix**.

Access Matrix

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

Access Matrix Example

The diagram illustrates an Access Matrix Example. It features three conceptual bubbles: 'Actors' (containing Operator, LeagueOwner, Player, and Spectator), 'Classes' (containing Arena, League, Tournament, and Match), and 'Access Rights' (containing various methods like <<create>>, archive(), end(), etc.). Arrows point from the 'Actors' bubble to the matrix rows and from the 'Classes' and 'Access Rights' bubbles to the matrix columns.

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner	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()

Access Matrix

- In general, the access matrix is a sparse matrix
- There are 3 different implementations of the access matrix:
 - Global access table
 - Access control list
 - Capabilities.

Global Access Table

- **Global access table:** Represents every non-empty cell in an access 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()

Operator	Arena
<<create>> createUser() view ()	
view ()	LeagueOwner

Access control list

- Access control list:
 - Associates a *list of* (actor,operation) pairs with the *class* being accessed
 - Every time an instance of this class is accessed, the access list is checked for the corresponding actor and operation.

Access Control List Example

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner	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()

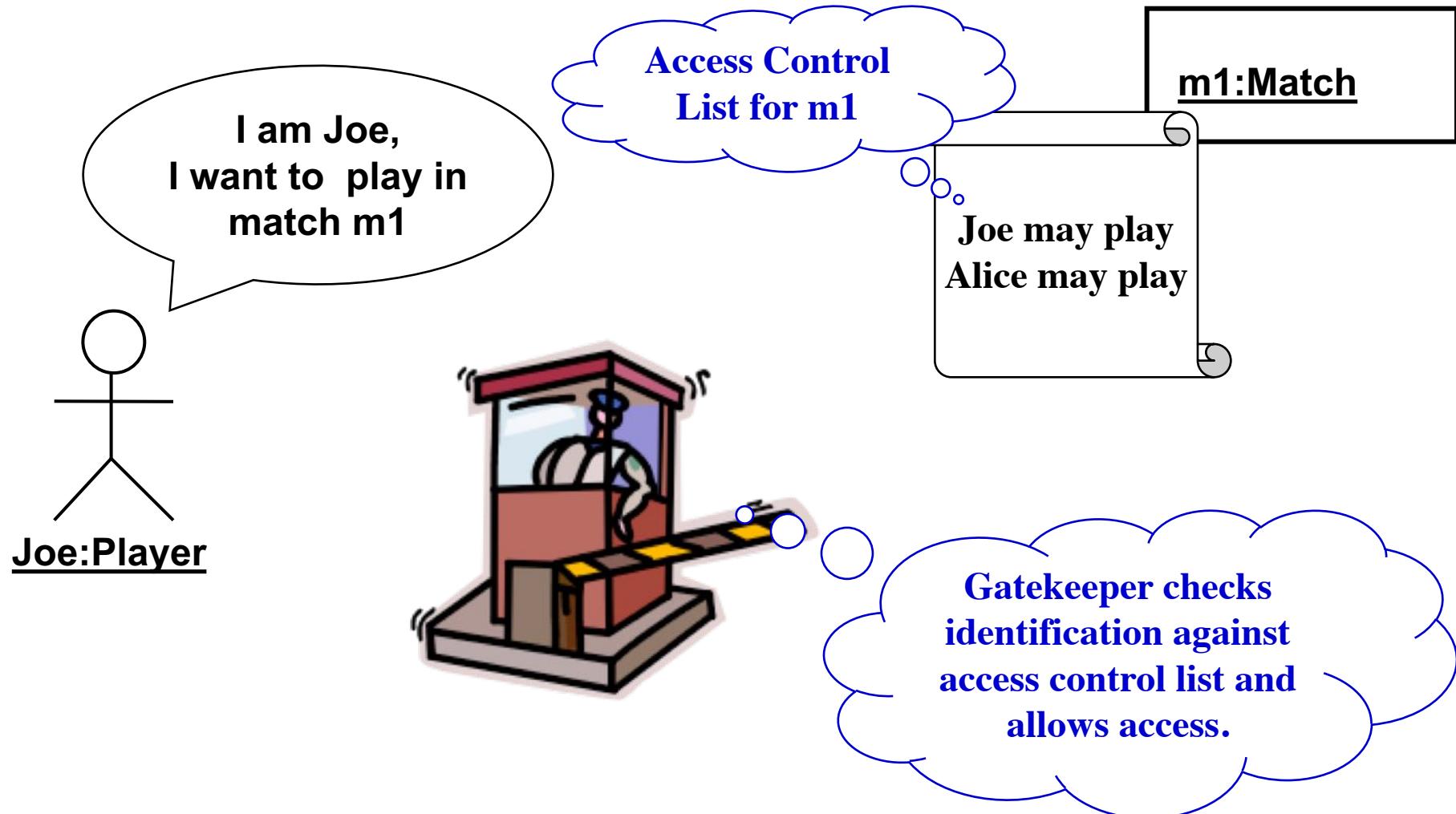
Access Control List Example

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner	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()

Access Control List Example



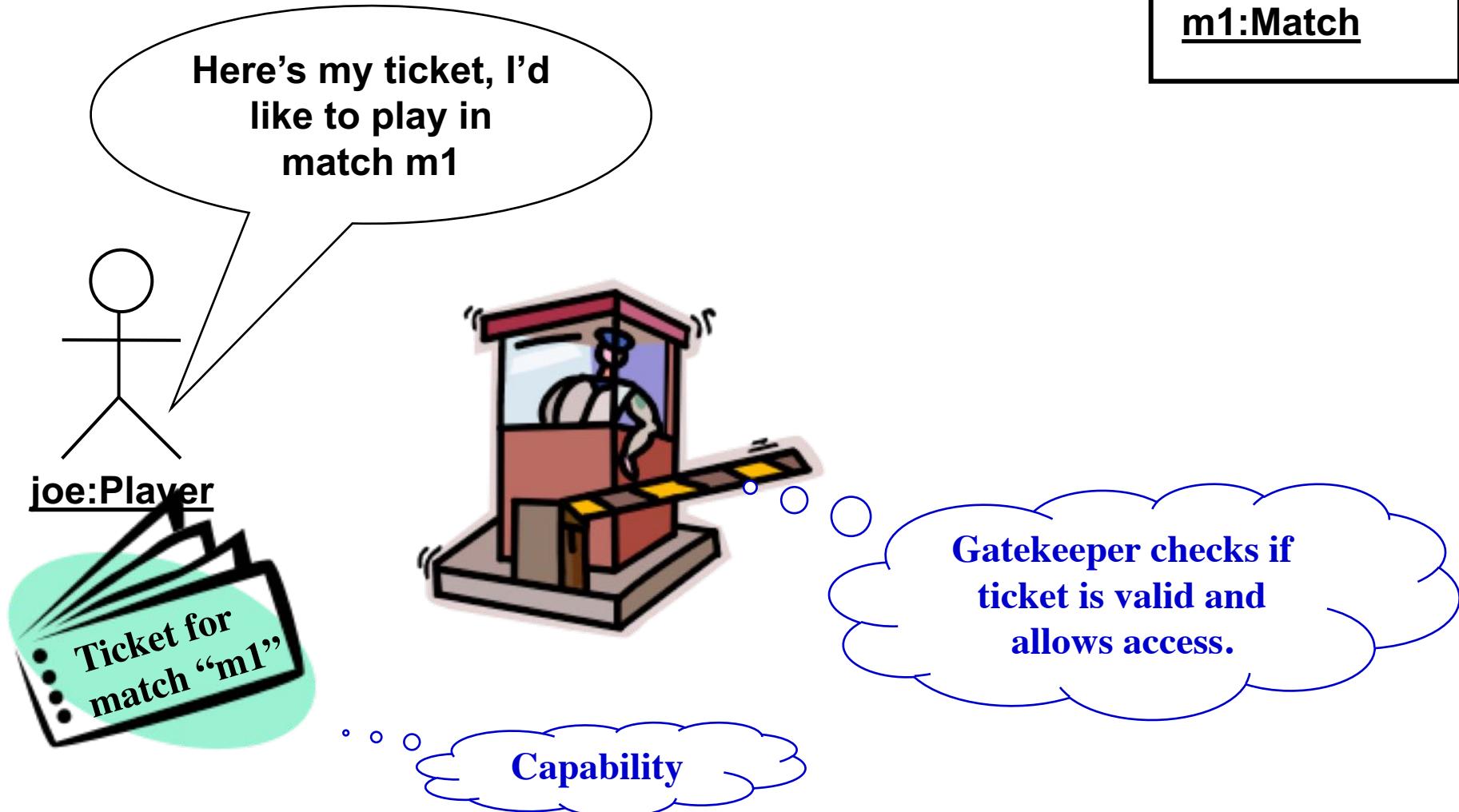
Access Control List Realization (ctd)



Capabilities

- **Capability:**
 - A capability associates a *(class,operations) pair with an actor*
 - A capability allows an actor to gain control access to an object of the class by calling one of the class operations described in the capability.

Capability Realization



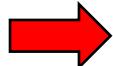
Overview of Today's Lecture

- Cohesion and Coupling
- Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors
- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control
- Boundary Conditions

7. Software Control

To control a software system the software designer has 2 choices:

- Implicit software control
 - rule-based systems, neural networks, logic programming

 **Explicit software control**

- Centralized control
- Decentralized control.

Explicit Software Control

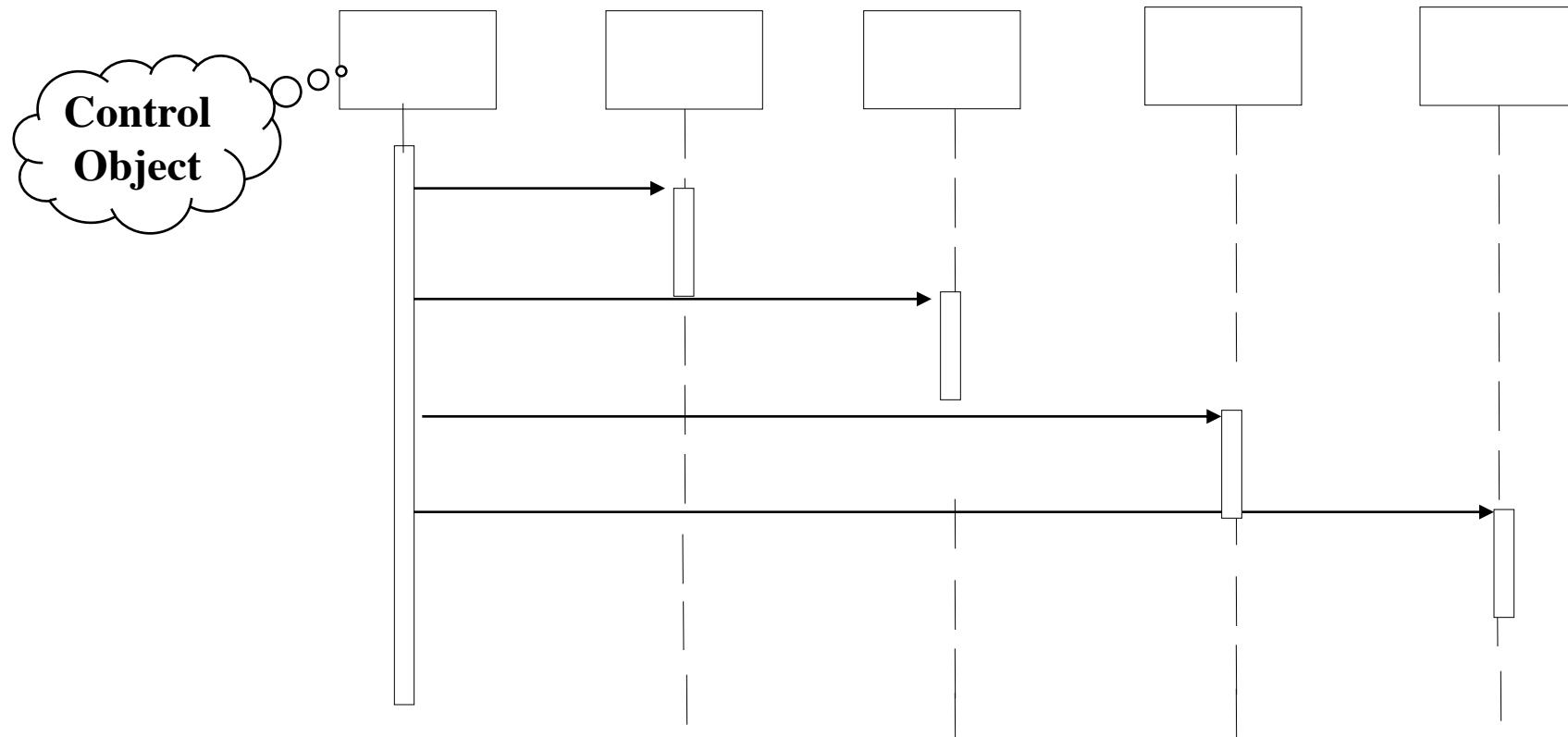
- **Centralized control:**
 - **Procedure-driven:** Control resides within a single component/subsystem
 - Example: Programming languages that have a main program
 - **Event-driven:** Control resides within a dispatcher calling other functions via socalled callbacks
 - Example: User Interface event loop listening to keyboard and mouse events
 - Pro: Change in the control structure is very easy
 - Con: The single control object is a possible performance bottleneck
- **Decentralized control:**
 - Control resides in several independent objects
 - Examples: Message based systems, Remote Method Invocation (RMI)
 - Con: Additional communication overhead
 - Pro: Fits nicely into object-oriented development, promises a possible speedup by mapping the objects on different processors.

Sequence Diagrams can be used to determine the Decentralization of a System

- The structure of the sequence diagram helps us to determine how decentralized the system is
- We distinguish 2 sequence diagram structures (Ivar Jacobson):
 - [Fork Diagrams](#) and [Stair Diagrams](#).
- If the sequence diagram looks like a fork => Centralized design
- If the sequence diagram looks like a stair => Decentralized design.

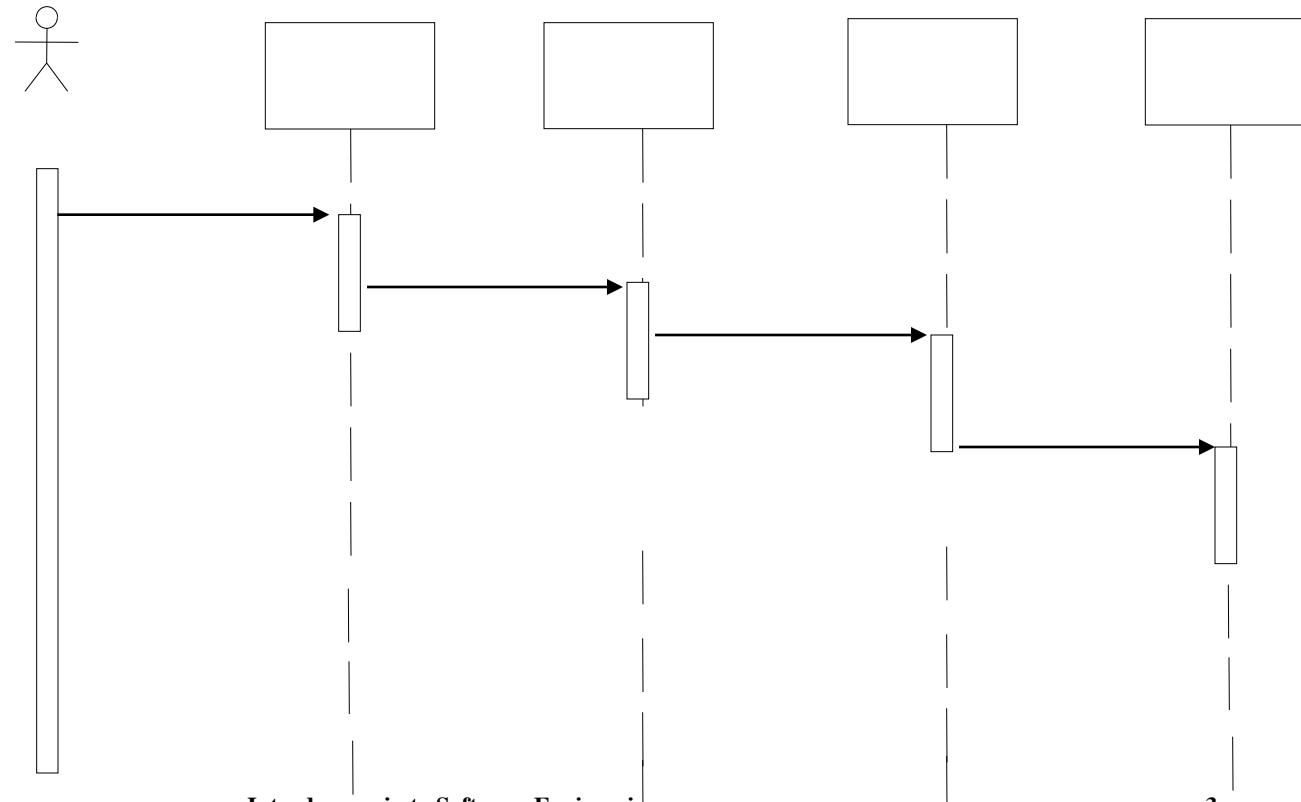
Fork Diagram

- **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

- **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



Overview of Today's Lecture

- Cohesion and Coupling
- Architectural Styles
 - ✓ Layered Architecture (Lecture on May 3, 2018)
 - ✓ Client-Server and Peer-to-Peer
 - ✓ Model-View Controller
 - ✓ Repository
 - ✓ Pipes-and Filers
 - ✓ Architectural Styles: Graphs with Components and Connectors

- Concurrency
- Hardware/Software Mapping
- Persistent Data Management
- Global Resource Handling
- Software Control

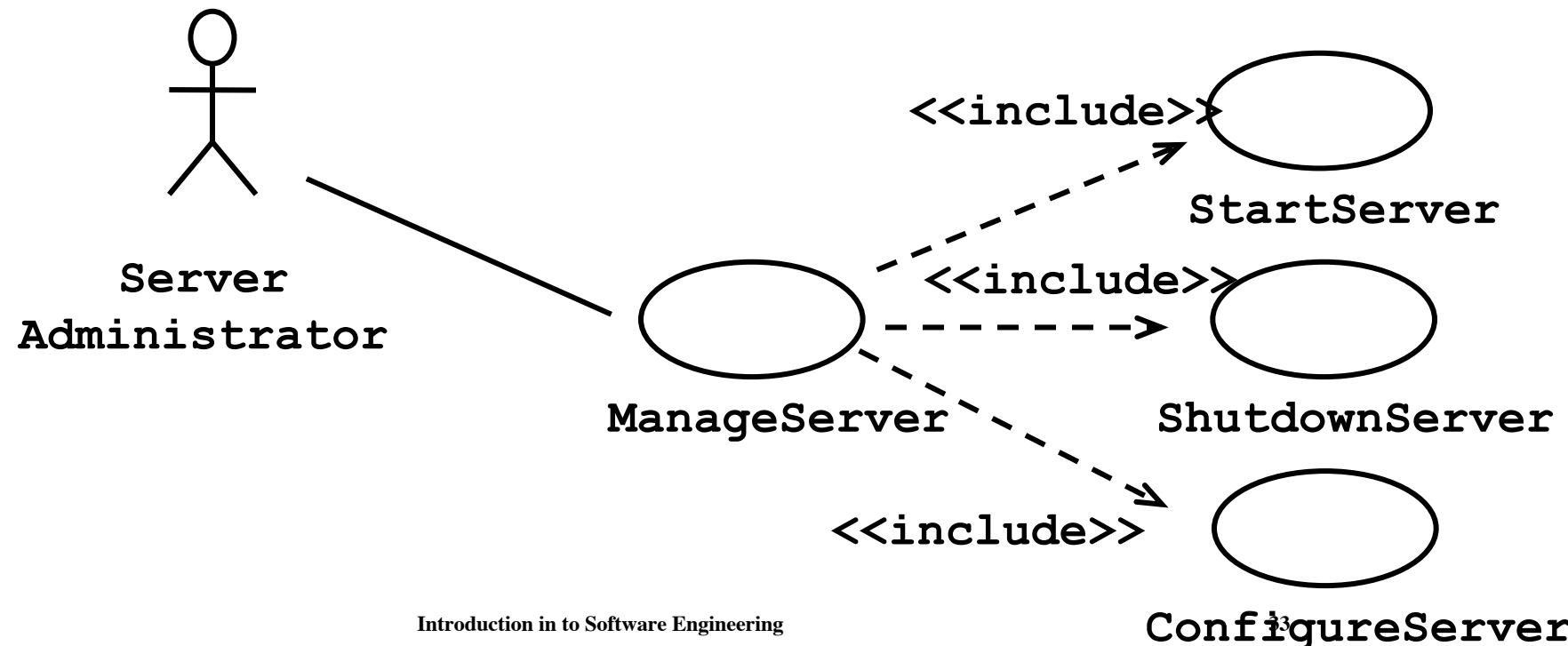
 Boundary Conditions

8. Boundary Conditions

- **Initialization**
 - The system is brought from an initial state to a steady state
- **Termination**
 - Resources are cleaned up and other systems are notified upon termination
- **Failure**
 - Possible failures: Bugs, errors, external problems
- Boundary conditions are best modeled as use cases
 - Actor: usually 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 a client server system

- Assume we identified the subsystem **Server** during system design
- During the hardware software mapping we dedicate a special node for **Server**
- We define a boundary use case `ManageServer`
 - `ManageServer` includes use cases to start up and shutdown **Server**



Summary of System Design II

- 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.

Quiz 06b

- To participate in the quiz, use ArTEMiS
- Click on Open Quiz or Start Quiz

Introduction to Software Engineering		
Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	 Start exercise
Quiz 06b		 Open Quiz

Remaining Time: **46 s**

Saved: never

● Connected

Submit

Only click on Submit when you have entered all answers!

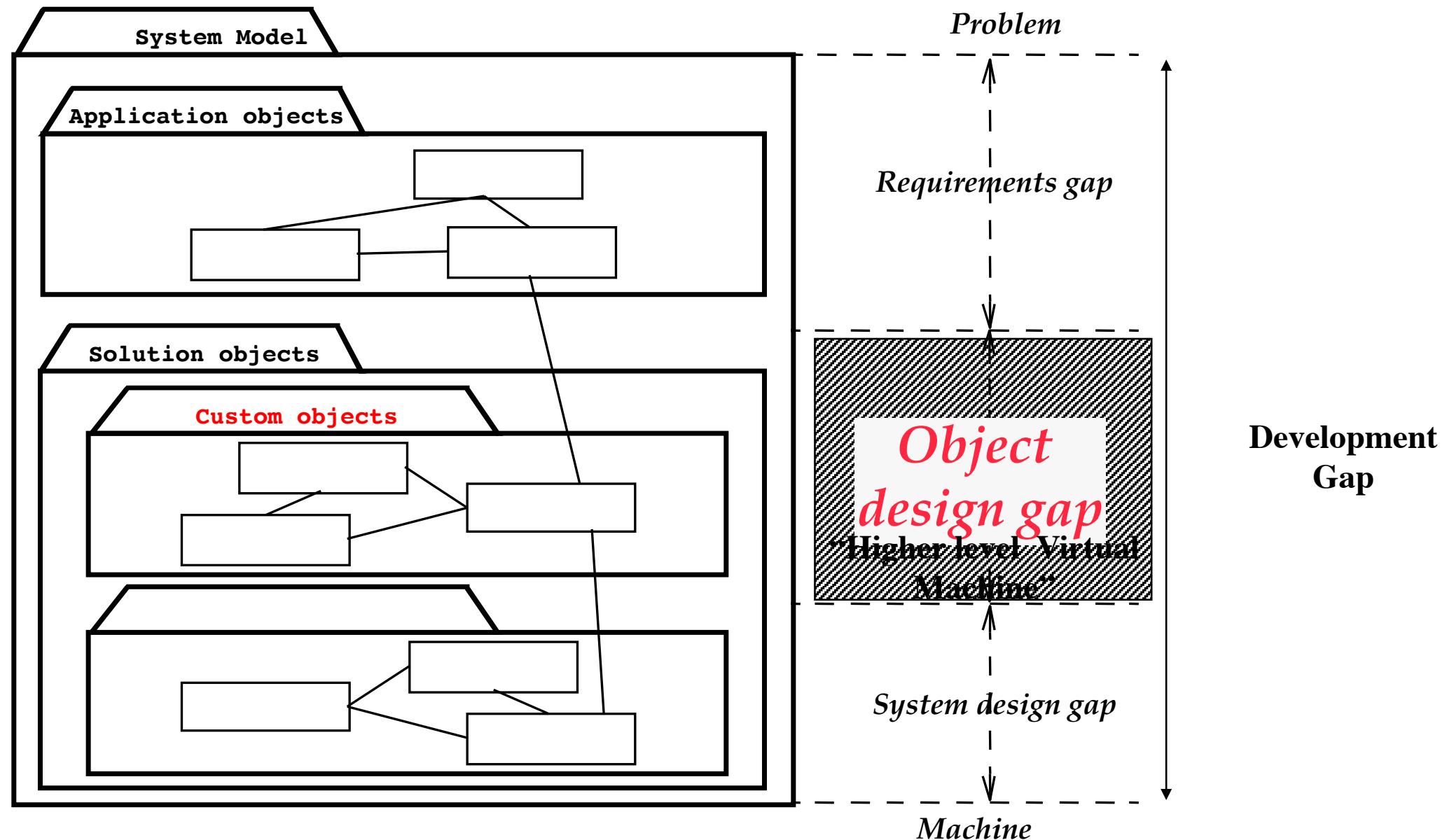
Overview of Today's Lecture

- ✓ Miscellaneous
- ✓ System Design (finishing the lecture from May 17)
 - ✓ Global Resource Handling
 - ✓ Software Control
 - ✓ Boundary Conditions
- Object Design
 - ➡ Purpose
 - Object Design Activities
 - Types of Reuse: Black Box and White Box Reuse
 - The Use of Inheritance in Object Design
 - Discovering Inheritance: Generalization vs Specialization
 - Design Patterns

Purpose of 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.

Object Design closes the Gap between Analysis and System Design

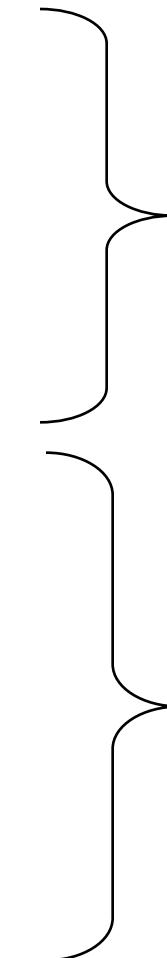


Overview of Today's Lecture

- Purpose of Object Design
- Object Design Activities
 - Types of Reuse: Black Box and White Box Reuse
 - The Use of Inheritance in Object Design
 - Discovering Inheritance: Generalization vs Specialization
 - Design Patterns

Object Design consists of 4 Activities

1. Reuse: Identification of existing solutions
 - Use of inheritance
 - Off-the-shelf components and additional solution objects
 - Use of 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.



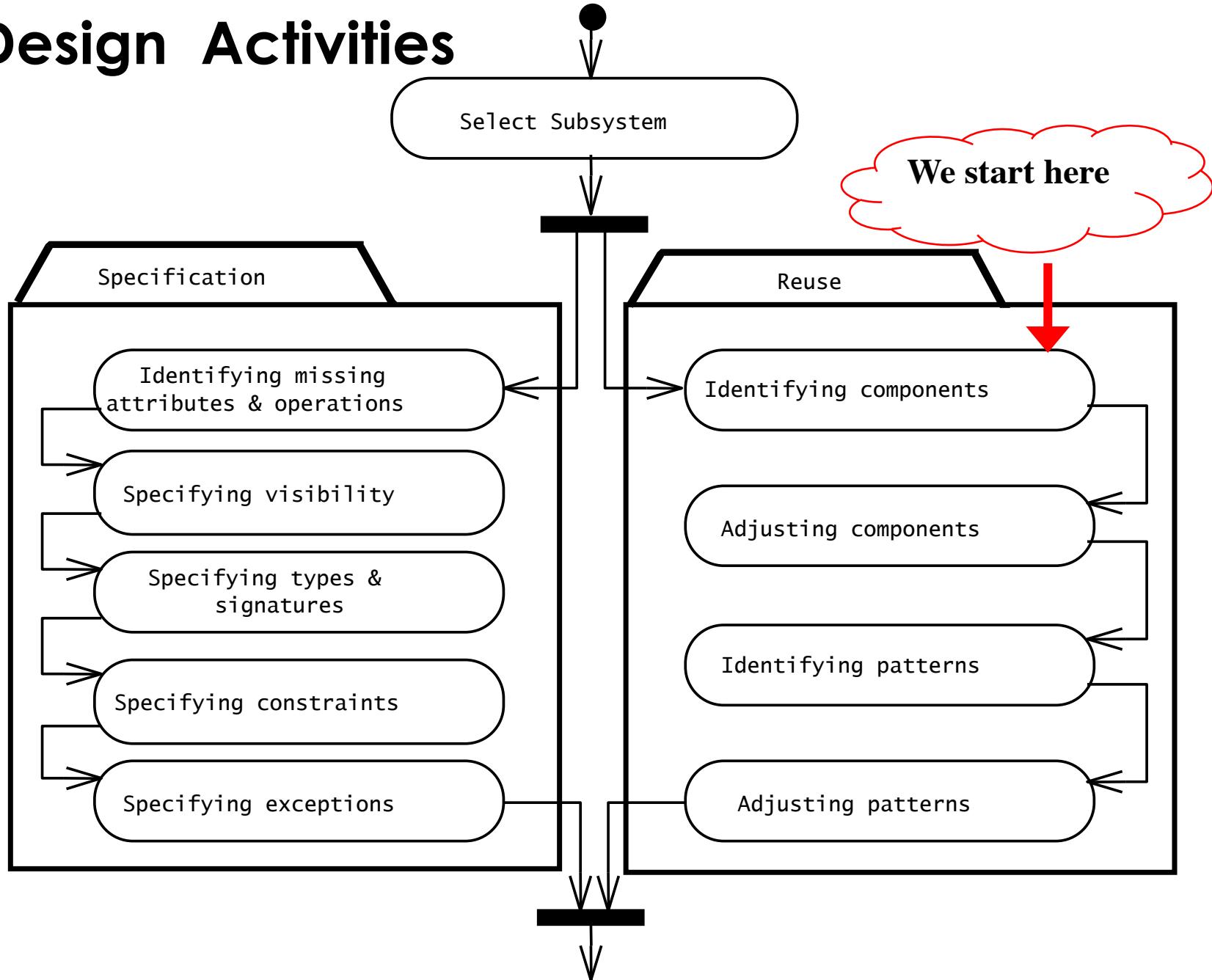
**Focus on
Reuse and
Specification**

Today

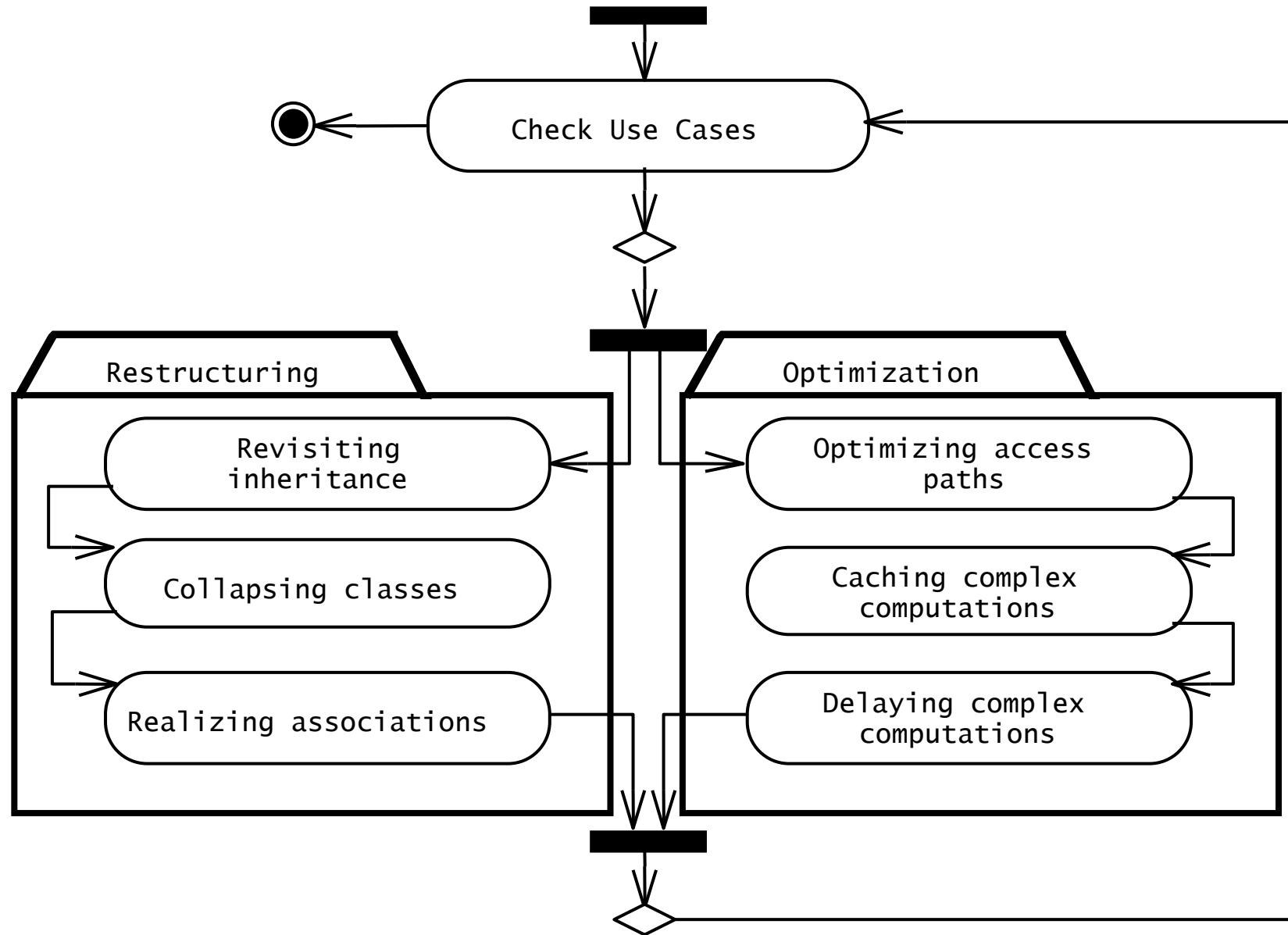
**Towards Mapping
Models to Code**

Lecture 7
**Model Transformations
and Refactoring**

Object Design Activities



Object Design Activities ctd



Identifying Components

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

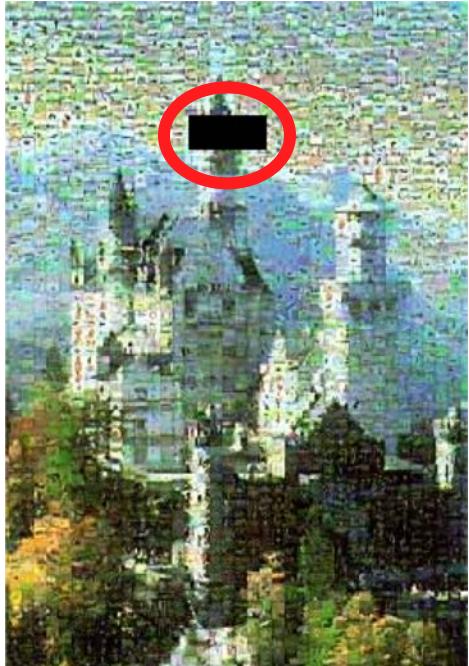
=> **Component-Based Software Engineering (CBSE):**

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

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

Design with Standard Components is similar to solving a Jigsaw Puzzle

Standard Puzzles:
„Corner pieces have two straight edges“



What do we do
if that is not true?.



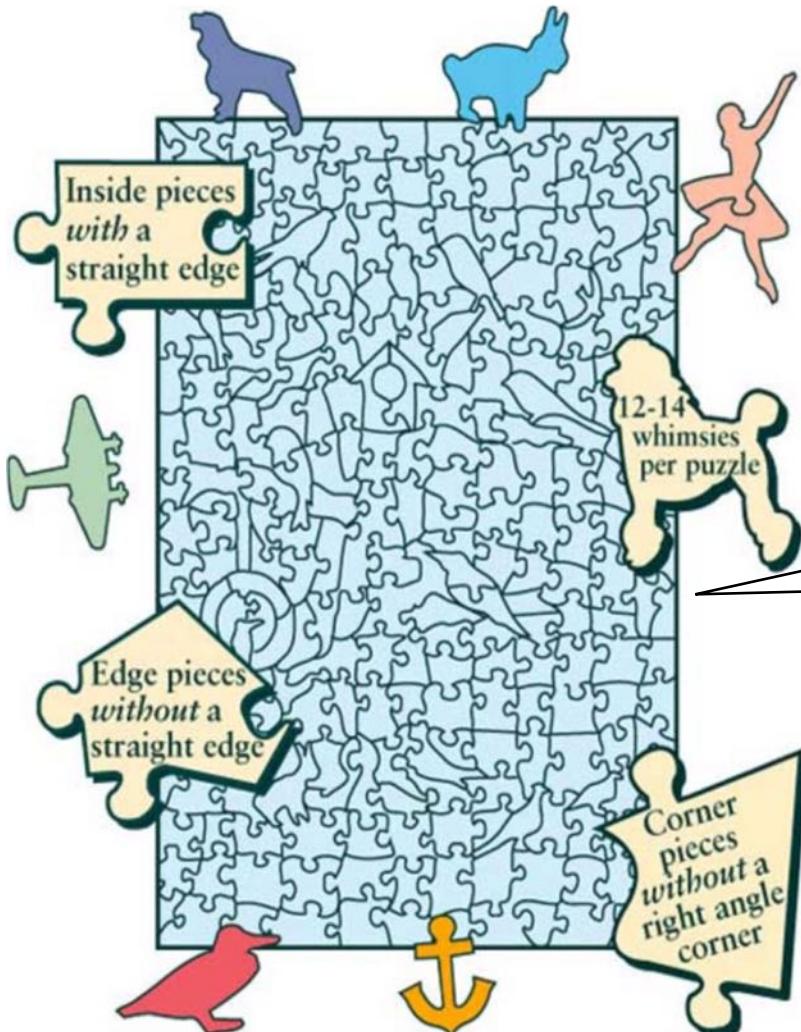
“Find”
the puzzle piece



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 (finalize the design)

Problem solving with non-Standard Components



Advanced
Jigsaw Puzzles

Modeling the Real World

- Modeling the real world leads to a system that reflects today's realities but not necessarily tomorrow's
- There is a need for **reusable and extendable** designs
 - Also called "flexible designs"
- There are several types for reuse in software engineering.

Overview of Today's Lecture

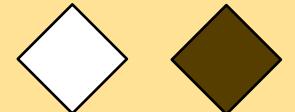
- Purpose of Object Design
 - Object Design Activities
- Types of Reuse: Black Box and White Box Reuse
- The Use of Inheritance in Object Design
 - Discovering Inheritance: Generalization vs Specialization
 - Design Patterns

Reuse in Object Design

→ 3 Types of Reuse:

- Reuse of design knowledge
- Reuse of existing classes
- Reuse of existing interfaces
- 2 techniques to close the object design gap:
 - **Composition** (also called **black box reuse**)
 - A new class is created by the aggregation of the existing classes. The new class offers the aggregated functionality of the existing classes
 - **Inheritance** (also called **white box reuse**)
 - A new class is created by subclassing. The new class reuses the functionality of the superclass and may offer new functionality.

Modeled with



Modeled with



Reuse of Existing Classes

- I have a list, but the customer wants 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 have implemented a list of elements of type `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 a previous project
 - Can I add `AddressBook` as a subsystem to my e-mail program replacing the vendor-supplied address book?
 - Can I reuse this class for the billing software in my customer relationship management system?

Reuse of Interfaces

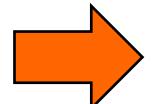
- I am an off-shore programmer in Hawaii. I have a contract to implement an electronic parts catalog
 - How can my contractor make sure that I implement it correctly?
- I would like to develop a window system for Linux that behaves the same way as in Windows
 - How can I make sure that I follow the conventions for Windows and not those of Mac OS X?
- I have to develop a new service for cars, that automatically calls a help center when the car is upside down?
 - Can I reuse the help desk software that I developed for a company in the telecommunication industry?

Reuse in Object Design

3 Types of Reuse:

- Reuse of design knowledge
- ✓ Reuse of existing classes
- ✓ Reuse of existing interfaces

2 techniques to close the object design gap:

 **Composition** (also called **black box reuse**)

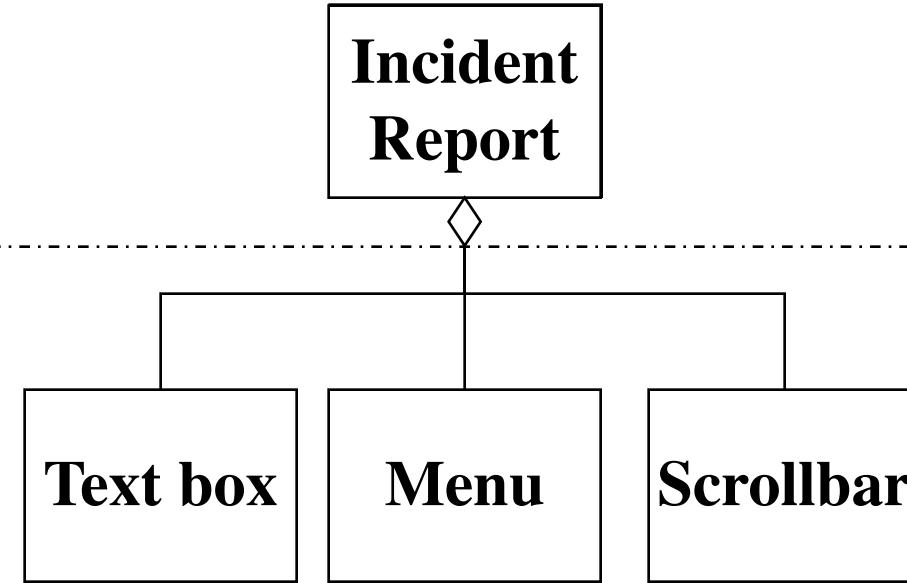
- A new class is created by the aggregation of the existing classes. The new class offers the aggregated functionality of the existing classes

2. Inheritance (also called **white box reuse**)

- A new class is created by subclassing. The new class reuses the functionality of the superclass and may offer new functionality.

Example of Composition (Black-box Reuse)

Requirements Analysis
(Language of Application
Domain)



Object Design
(Language of Solution
Domain)

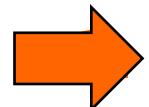
Reuse in Object Design

3 Types of Reuse:

- Reuse of design knowledge
- ✓ Reuse of existing classes
- ✓ Reuse of existing interfaces

2 techniques to close the object design gap:

- **Composition** (also called **black box reuse**)
 - A new class is created by the aggregation of the existing classes. The new class offers the aggregated functionality of the existing classes



Inheritance (also called **white box reuse**)

- A new class is created by subclassing. The new class reuses the functionality of the superclass and may offer new functionality.

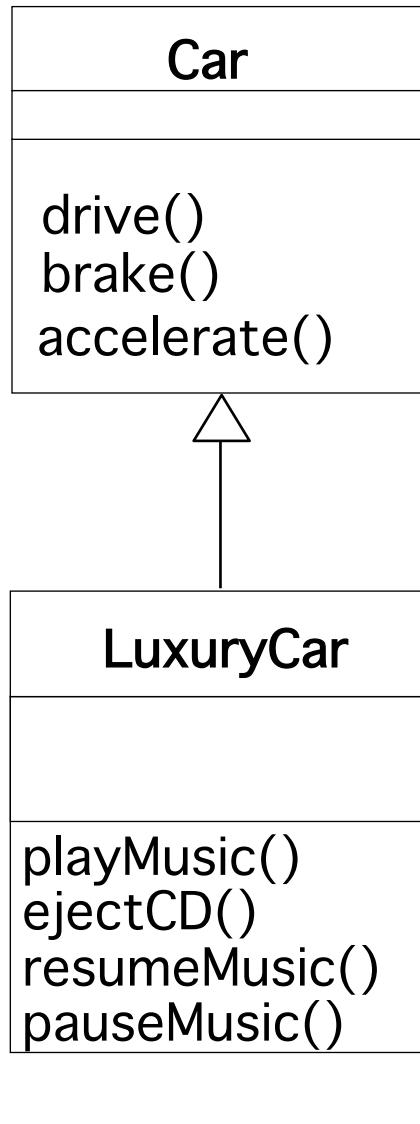
The Use of Inheritance in Software Engineering

- Inheritance is used to achieve two different goals
 - Description of Taxonomies
 - Interface Specification and Reuse

Description of Taxonomies

- Used during *requirements elicitation and analysis*
- Activity: Identify application domain objects that are hierarchically related
- Goal: Make the analysis object model more understandable
- **Interface Specification**
 - Used during *object design*
 - Activity: Identify the signatures of all identified objects
 - Goal: Increase the reusability, enhance the modifiability and the extensibility.

Description of Taxonomies



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() {...}  
}
```

The Use of Inheritance in Software Engineering

- Inheritance is used to achieve two different goals
 - Establish taxonomies
 - Reuse
- **Description of Taxonomies**
 - Used during *requirements analysis*
 - Activity: Identify application domain objects that are hierarchically related
 - Goal: Make the analysis object model more understandable

→ Interface Specification

- Used during *object design*
- Activity: Identify the signatures of all identified objects
- Goal: Increase the reusability, enhance the modifiability and the extensibility.

Interface Specification

→ Implementation Inheritance

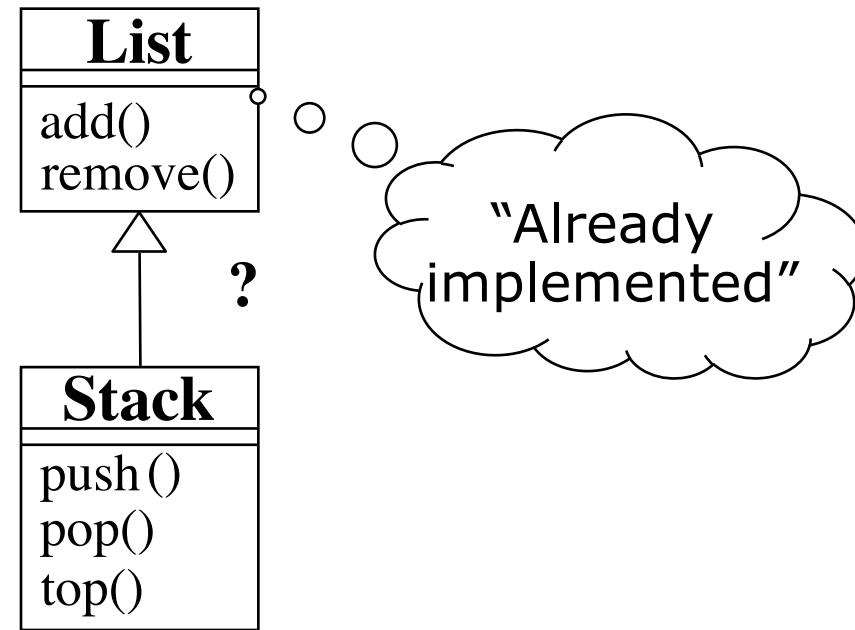
- Subclassing from an implementation
- **Reuse:** Implemented functionality in the super class
- **Delegation**
 - Catching an operation and sending it to another object where it is already implemented
 - **Reuse:** Implemented functionality in an existing object
- **Specification Inheritance**
 - Subclassing from a specification
 - The specification is an abstract class where all the operations are specified but not yet implemented
 - **Reuse:** Specified functionality in the super class.

Implementation Inheritance

A class is already implemented that does almost the same as the desired class

Example:

- I have a List, I need a Stack
- How about subclassing **Stack** from **List** and implementing push(), pop() and top() using the implementations of add() and remove()?



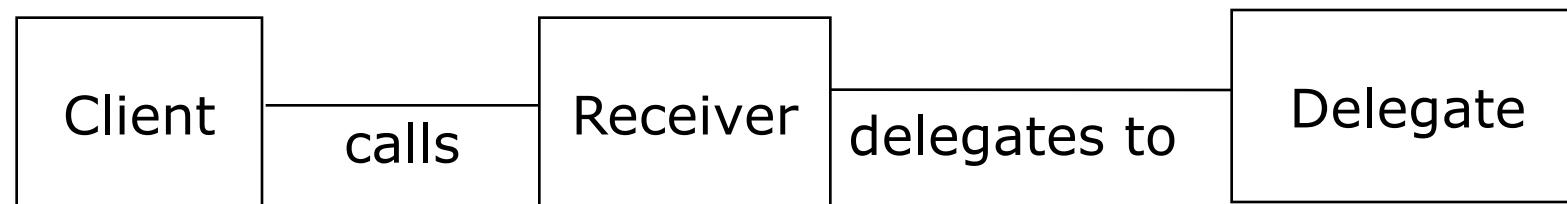
Problem with implementation inheritance:

- The inherited operations might exhibit unwanted behavior
- Example: What happens if I call remove() instead of pop()?

Delegation

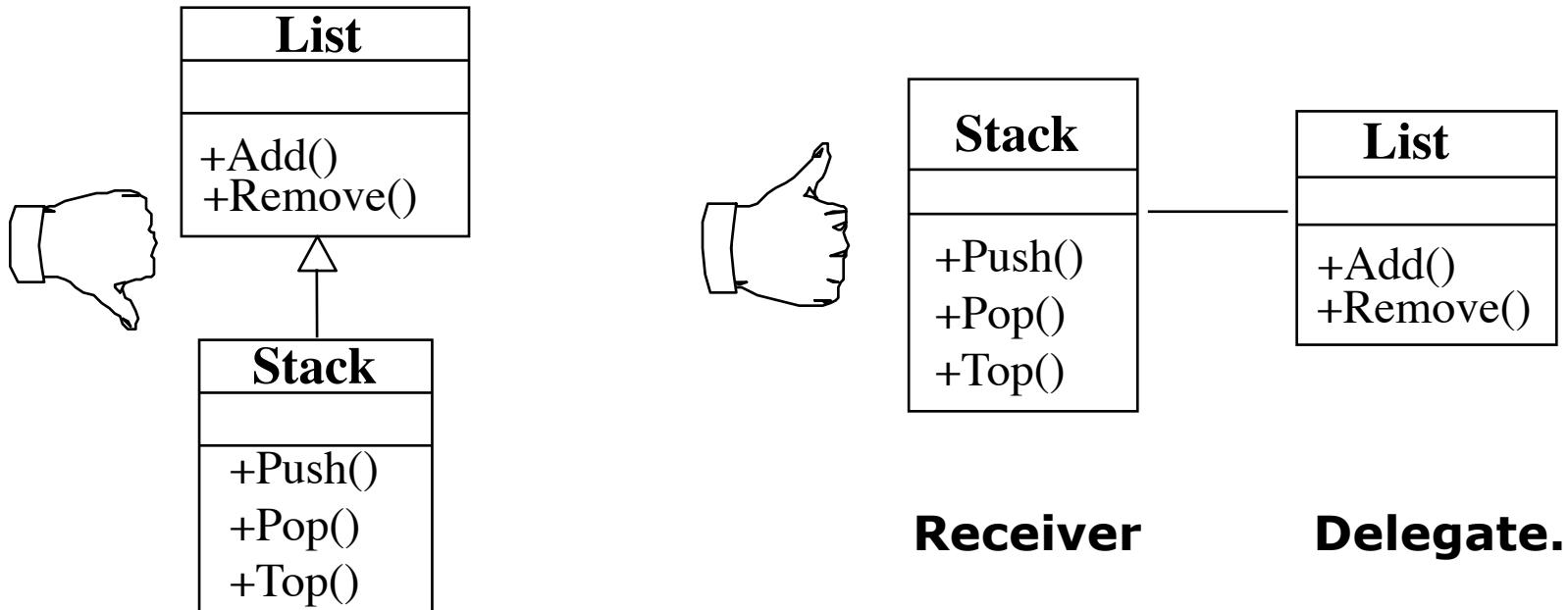
- Delegation is a way of making composition as powerful for reuse as inheritance
- In delegation, three objects are involved:
 - The Client calling the Receiver
 - The Receiver sending the request to the Delegate
 - The Delegate executing the request

The existence of the Receiver makes sure, that the Client cannot misuse the Delegate object.



Implementation Inheritance vs. Delegation

- **Implementation Inheritance:** Extends a super class with a subclass containing a new operation or overriding an existing operation
- **Delegation:** Catches an operation and sends it to another object
- Which of the following models is better?



Implementation Inheritance vs. Delegation

- Delegation
 - + Flexible, because any object can be replaced at run time by another one (as long as it has the same type)
 - Inefficient, because objects are encapsulated
- Inheritance
 - + Straightforward to use
 - + Supported by many programming languages
 - + Easy to implement new functionality in the subclass
 - Inheritance exposes the public methods of the parent class
 - Changes in the parent class force the subclass to change as well, at least it requires recompilation.

Interface Specification

✓ Implementation Inheritance

- ✓ Subclassing from an implementation
- ✓ Reuse: Implemented functionality in the super class

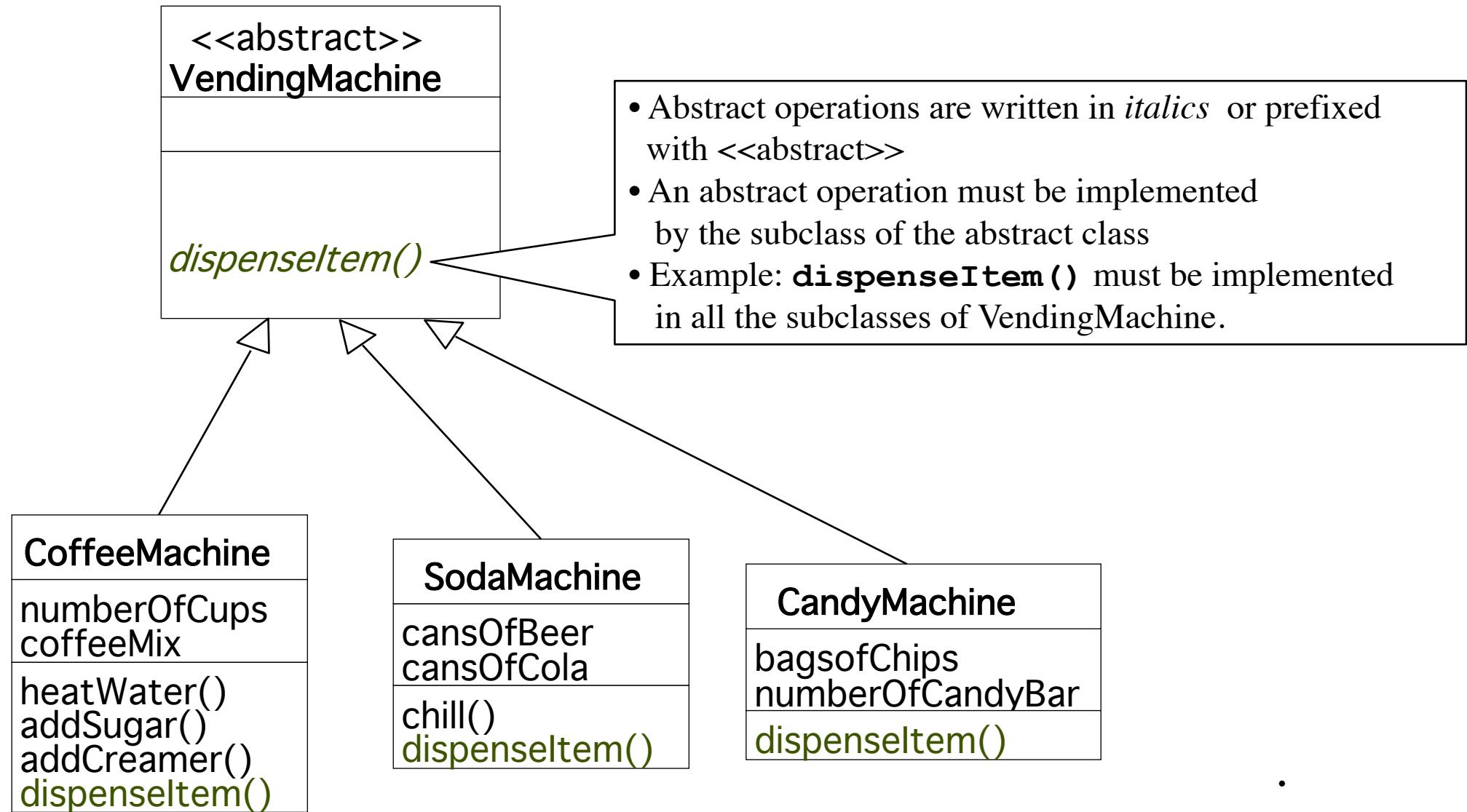
✓ Delegation

- ✓ Catching an operation and sending it to another object where it is already implemented
- ✓ Reuse: Implemented functionality in an existing object

→ Specification Inheritance

- Subclassing from a specification
 - The specification is an **abstract class** where at least one operation is abstract, that is, the operation is specified but not yet implemented
- Reuse: Specified functionality in the super class.

Example: Specification Inheritance with an Abstract Class



20 Minute Break



Overview of Today's Lecture

- Purpose of Object Design
- Object Design Activities
- Types of Reuse: Black Box and White Box Reuse
- The Use of Inheritance in Object Design
- Discovering Inheritance: Generalization vs Specialization
- Design Patterns

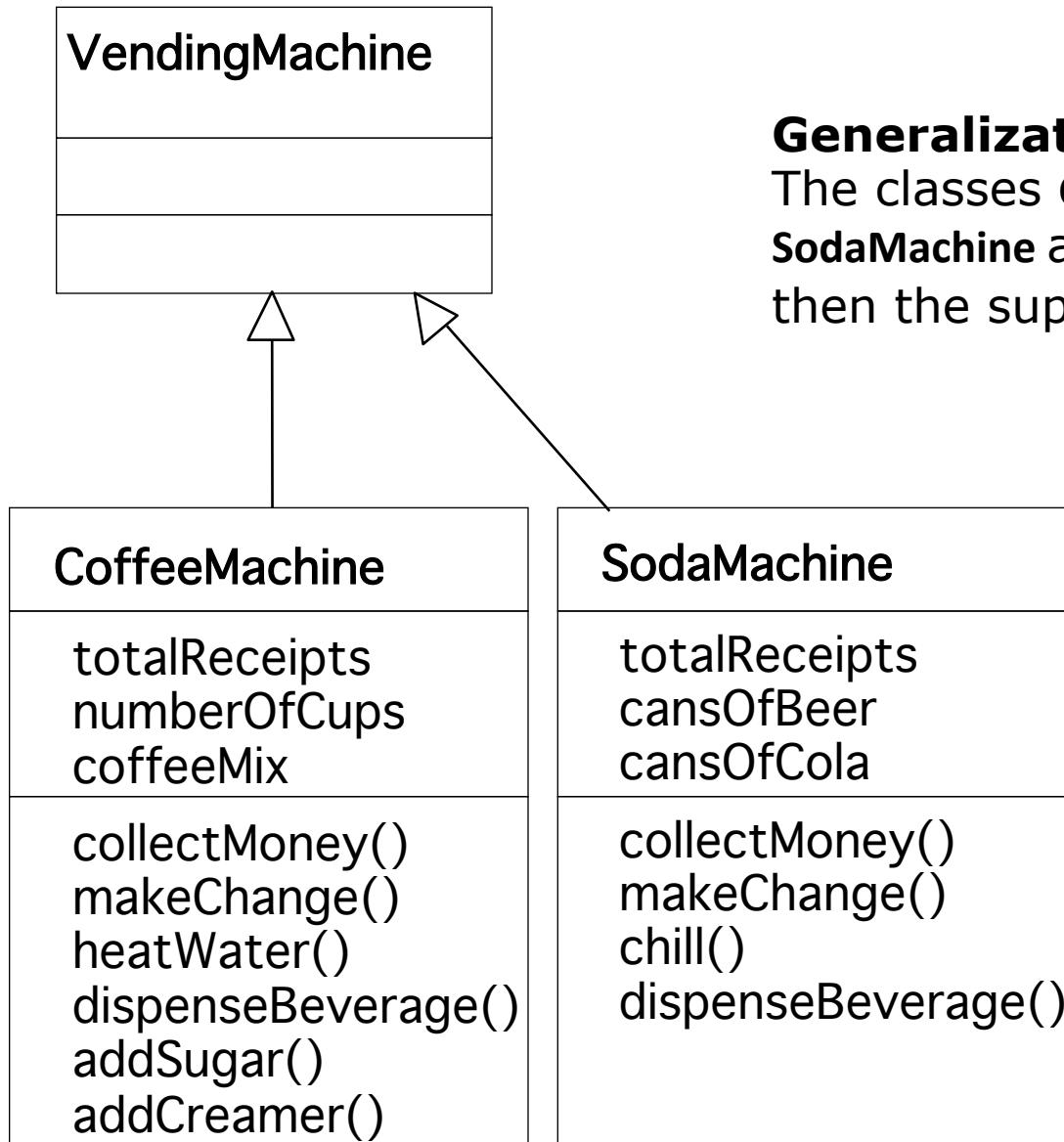
Discovering Inheritance

- To “discover” inheritance associations, we can proceed in two ways, which we call specialization and generalization
- **Generalization**: The discovery of an inheritance association between two classes, where the sub class is discovered first
- **Specialization**: The discovery of an inheritance association 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 (Elephant, Lion, Tiger), then we discover that these animals have common properties
 - We give these common properties a name: **mammal**.
 - Engineering:
 - We do an inventory on the machines produced by two companies that were recently merged
 - We then define a common interface to be shared by all the machines.

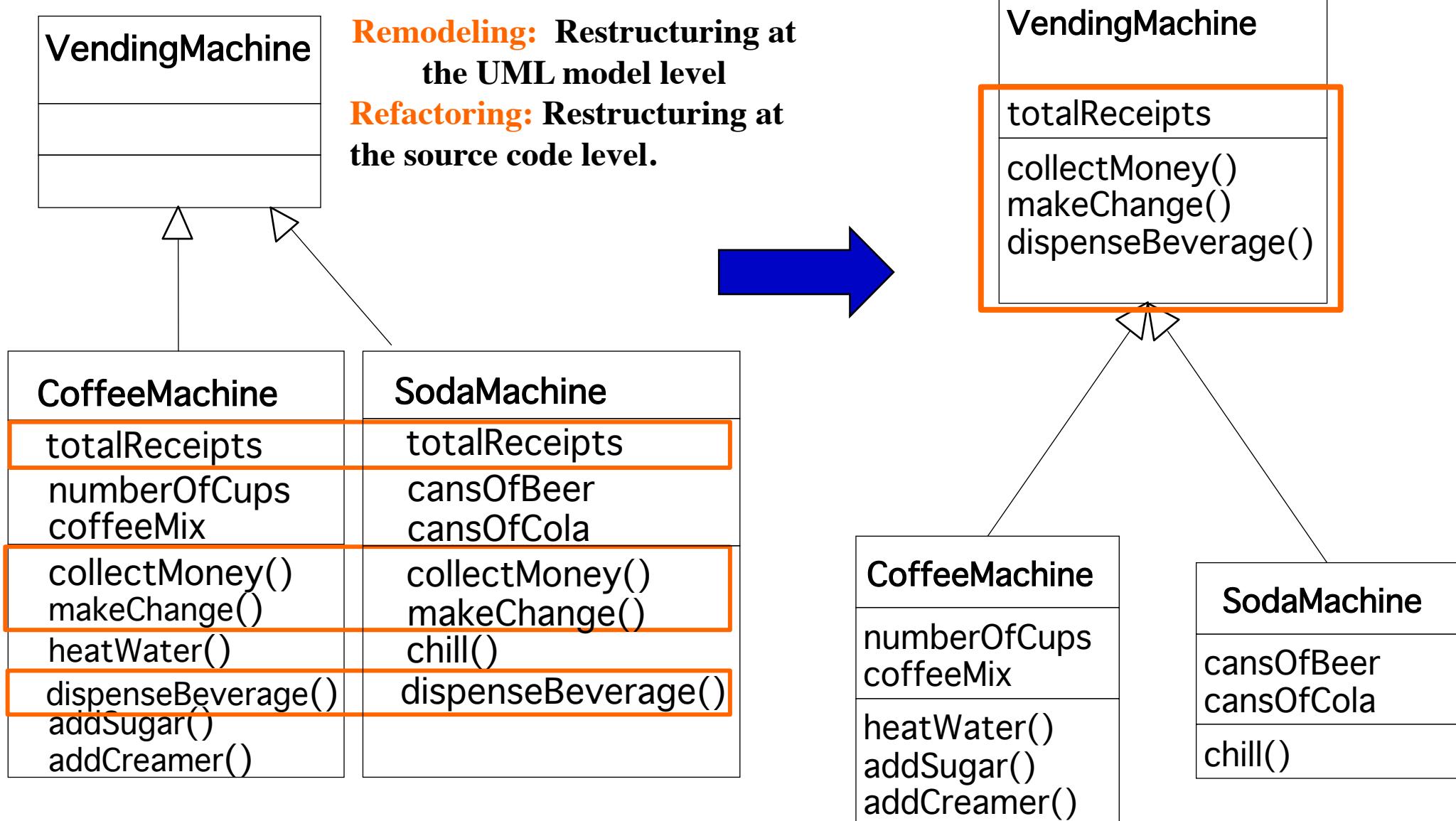
Generalization Example: Modeling Vending Machines



Generalization:

The classes **CoffeeMachine** and **SodaMachine** are discovered first, then the superclass **VendingMachine**.

Generalization usually leads to a Model Transformation



Specialization

- Specialization occurs, when we find a subclass that is very similar to an existing class
 - Example: The Quarks Theory postulated certain particles and events. They were found
 - Not all, some particles are still being looked for
- Specialization can also occur unintentionally:
 - Summersault at 300 km/h (24 Hours of Le Mans, 1999)



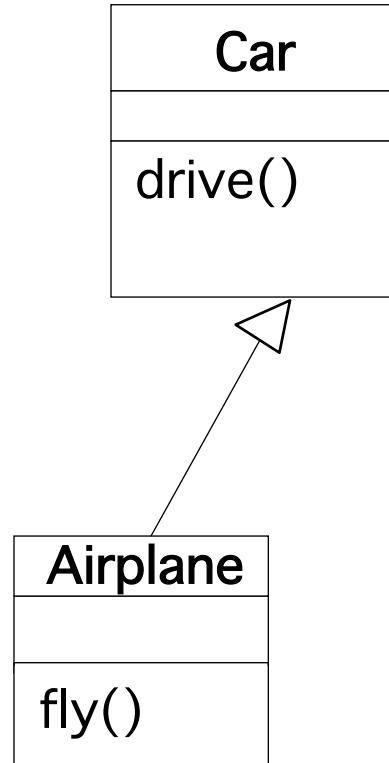
<http://www.youtube.com/watch?v=Ow3rxq7U1mA>

Quiz 06c on Artemis: Specialization

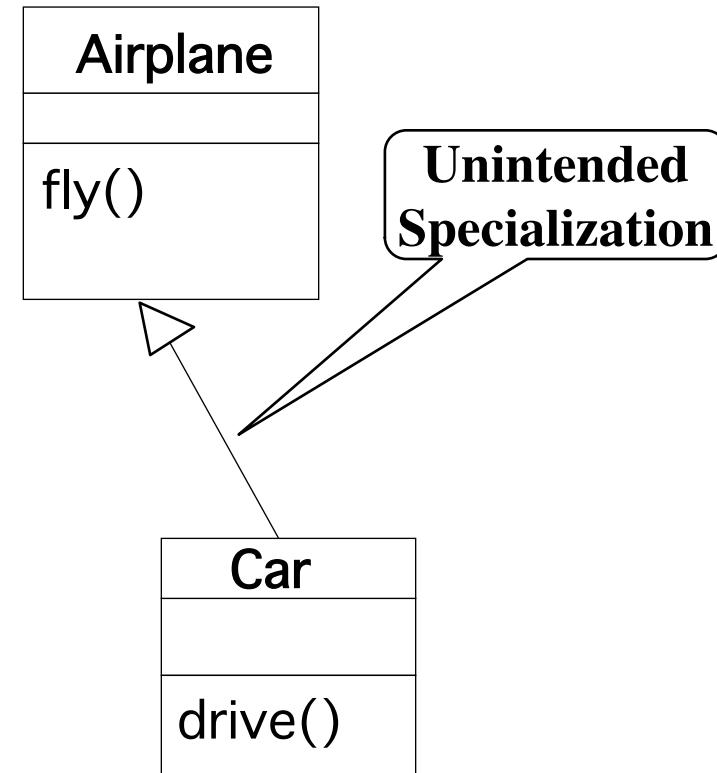
Which of these 2 taxonomies describes the car in the previous slide?

„Once the forces lifting a car exceed those holding it to the ground the car will fly“
-- Unknown

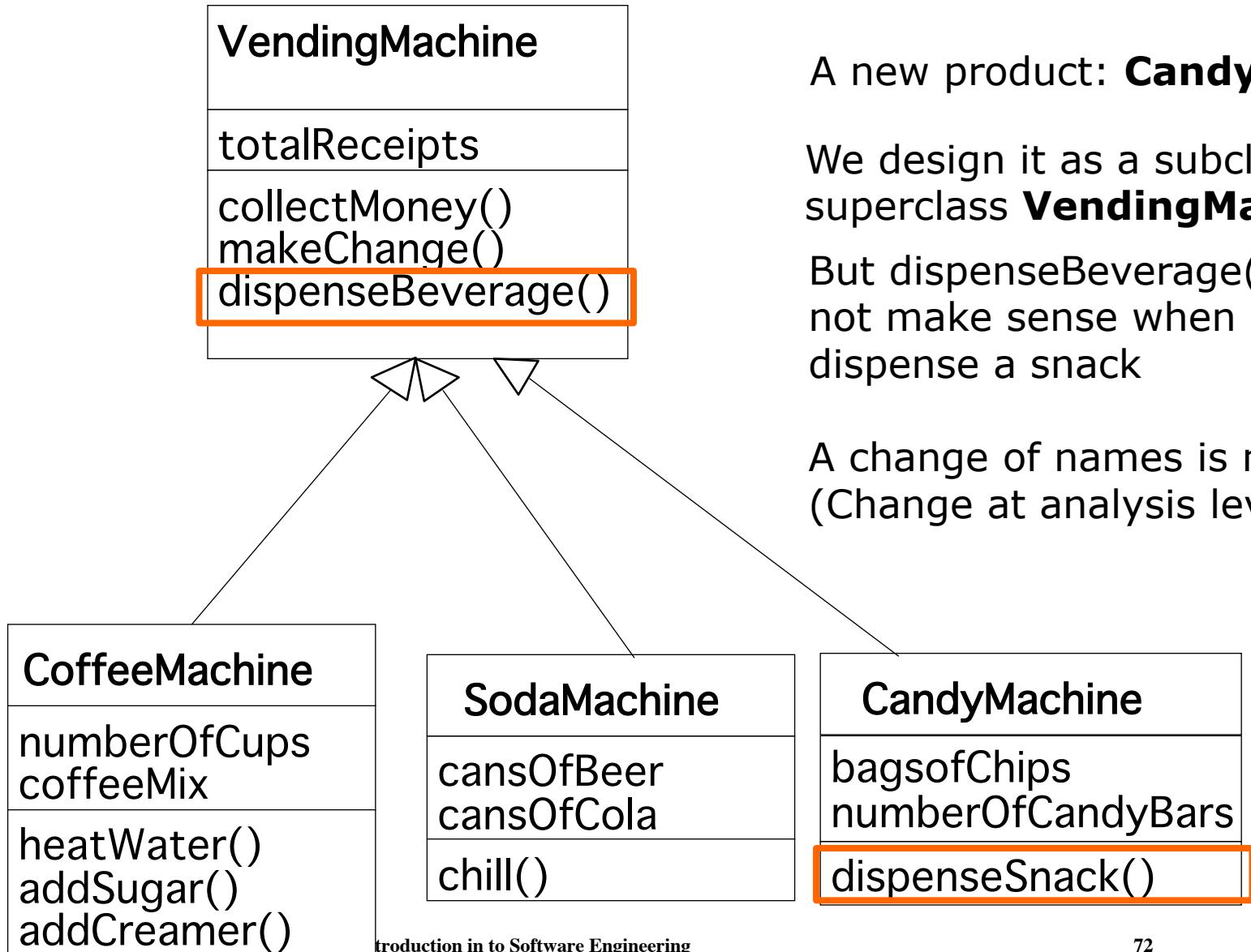
Choice 1



Choice 2



Another Specialization & Generalization Example



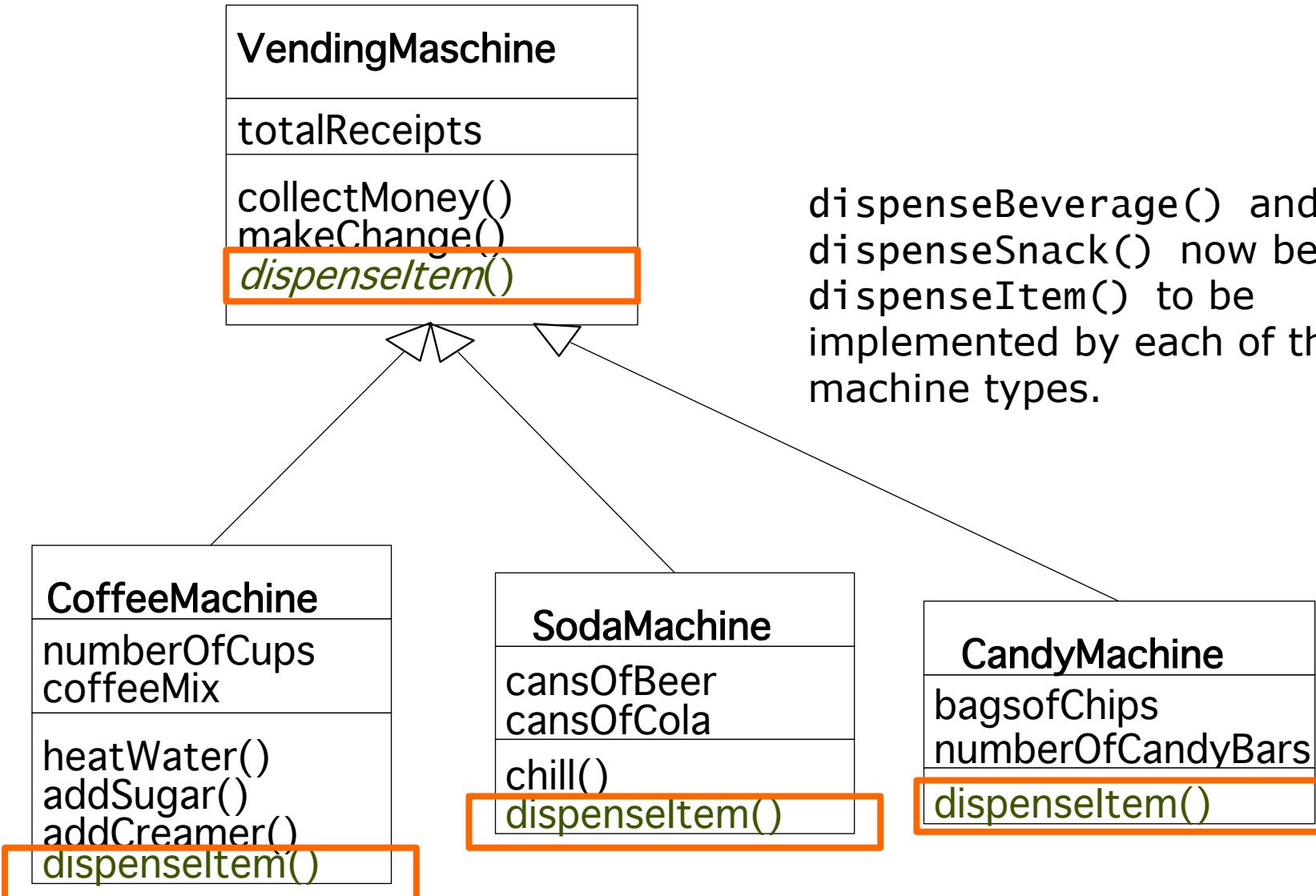
A new product: **CandyMachine**

We design it as a subclass of the superclass **VendingMachine**

But `dispenseBeverage()` does not make sense when we dispense a snack

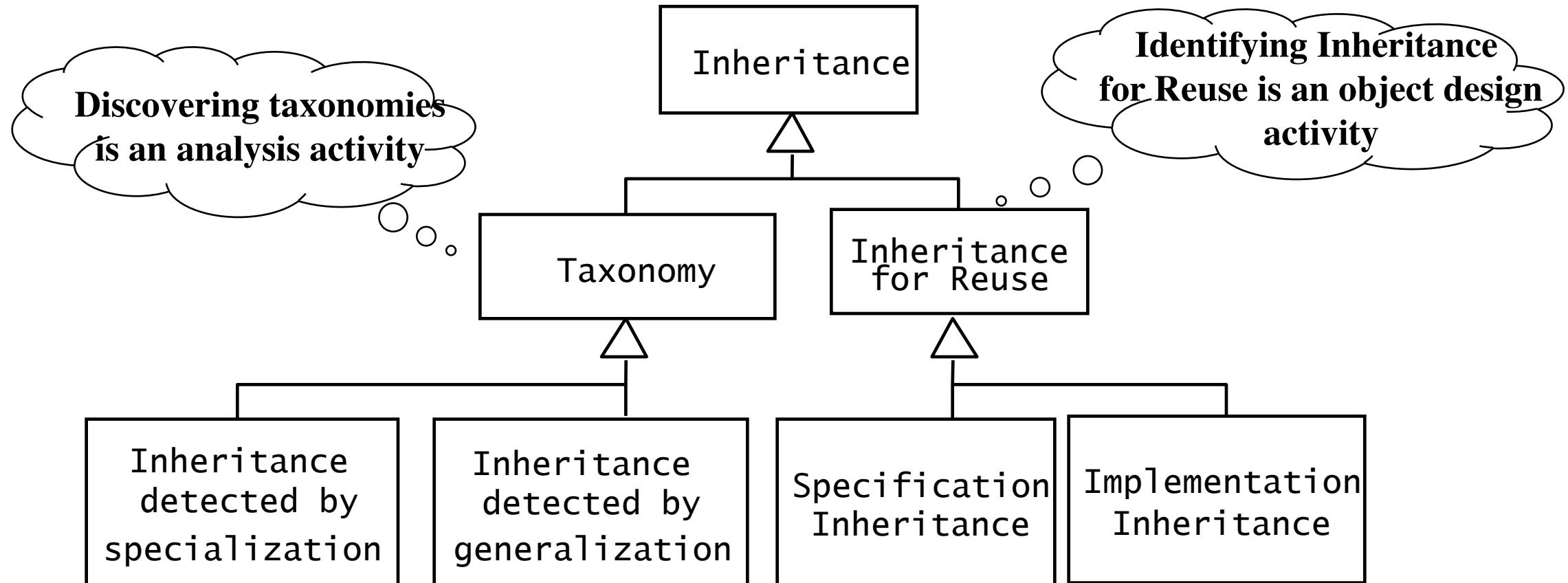
A change of names is now useful
(Change at analysis level!).

Consequence of Adding the New Machine: Renaming



`dispenseBeverage()` and `dispenseSnack()` now become `dispenseItem()` to be implemented by each of the machine types.

Metamodel for Inheritance



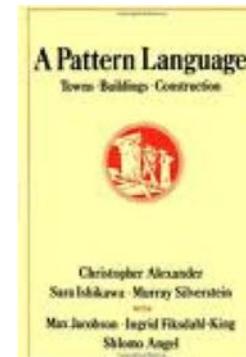
Overview of Today's Lecture

- Purpose of Object Design
 - Object Design Activities
 - Types of Reuse: Black Box and White Box Reuse
 - The Use of Inheritance in Object Design
 - Discovering Inheritance: Generalization vs Specialization
- Design Patterns

Design Patterns originated in Architecture

- **Christopher Alexander:**

- “Buildings have been built for thousands of years by users who were not architects”
- “Users know more about what they need from buildings and towns than an architect”
- “Because good buildings are based on a set of design principles that can be described with a pattern language”



Christopher Alexander

- * 1936 Vienna, Austria
- More than 200 building projects
- Creator of the „Pattern language“
- Professor emeritus at UCB

Although Christopher Alexander's pattern language is about architecture and urban planning, his ideas are applicable to many other disciplines, in particular software engineering.

Design Patterns

- Book by John Vlissedes, Erich Gamma, Ralph Johnson and Richard Helm, also called the Gang of Four
- Based on Christopher Alexander's pattern language



John Vlissedes
* 1961-2005
• Stanford
• IBM Watson
Research Center



Erich Gamma
* 1961
• ETH
• Telligent, IBM
• JUnit, Eclipse,
• Jazz



Ralph Johnson
* 1955
• University of Illinois,
• Smalltalk, Design
Patterns, Frameworks
OOPSLA veteran

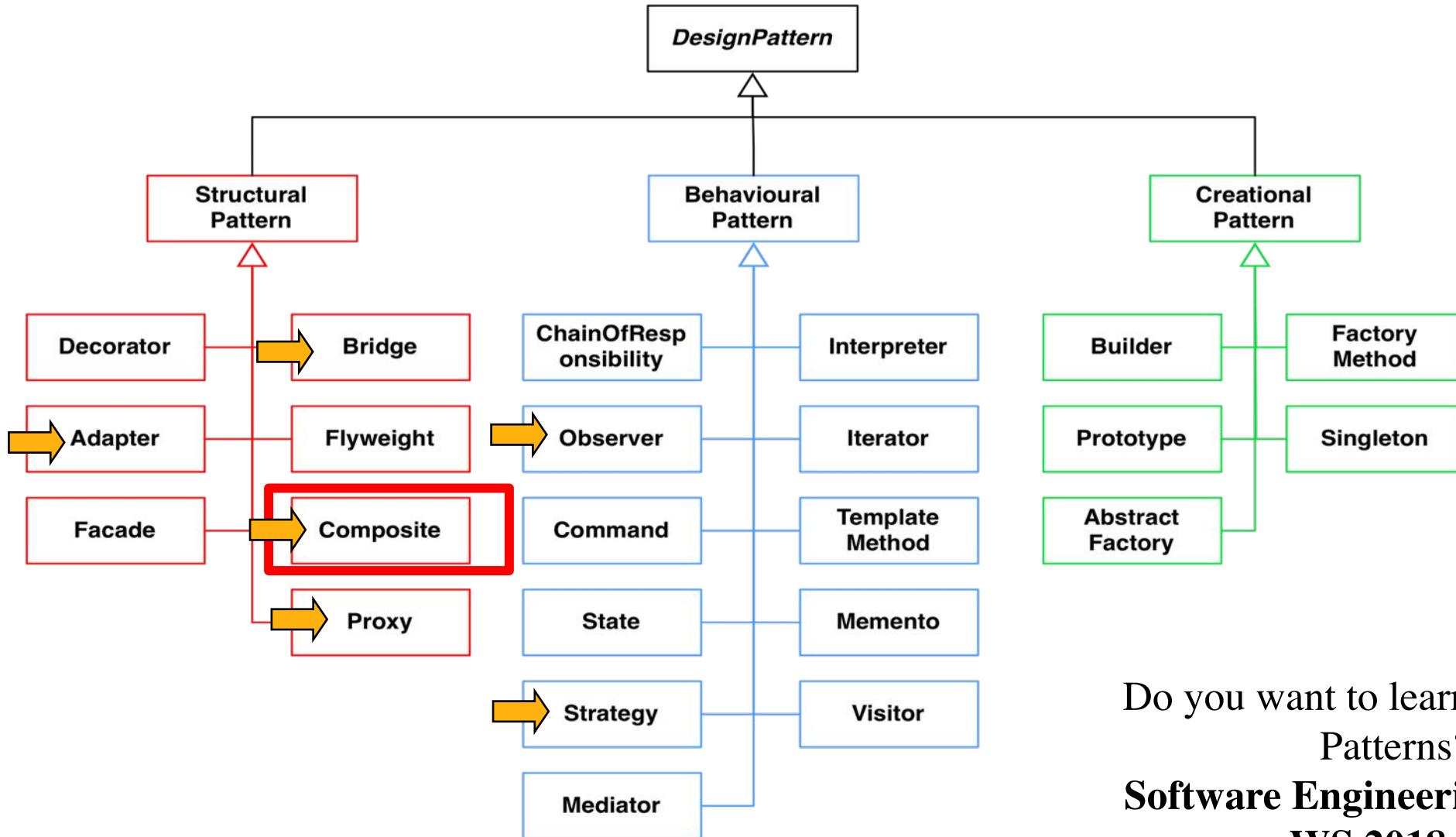


Richard Helm
• University of Melbourne
• IBM Research, Boston
• Consulting Group (Australia)

3 Types of Design Patterns

- **Structural Patterns**
 - Reduce coupling between two or more classes
 - Introduce an abstract class to enable future extensions
 - Encapsulate complex structures
- **Behavioral Patterns**
 - Allow a choice between algorithms
 - Allow the assignment of responsibilities to objects ("Who does what?")
 - Model complex control flows that are difficult to follow at runtime
- **Creational Patterns**
 - Allow to abstract from complex instantiation processes
 - Make the system independent from the way its objects are created, composed and represented.

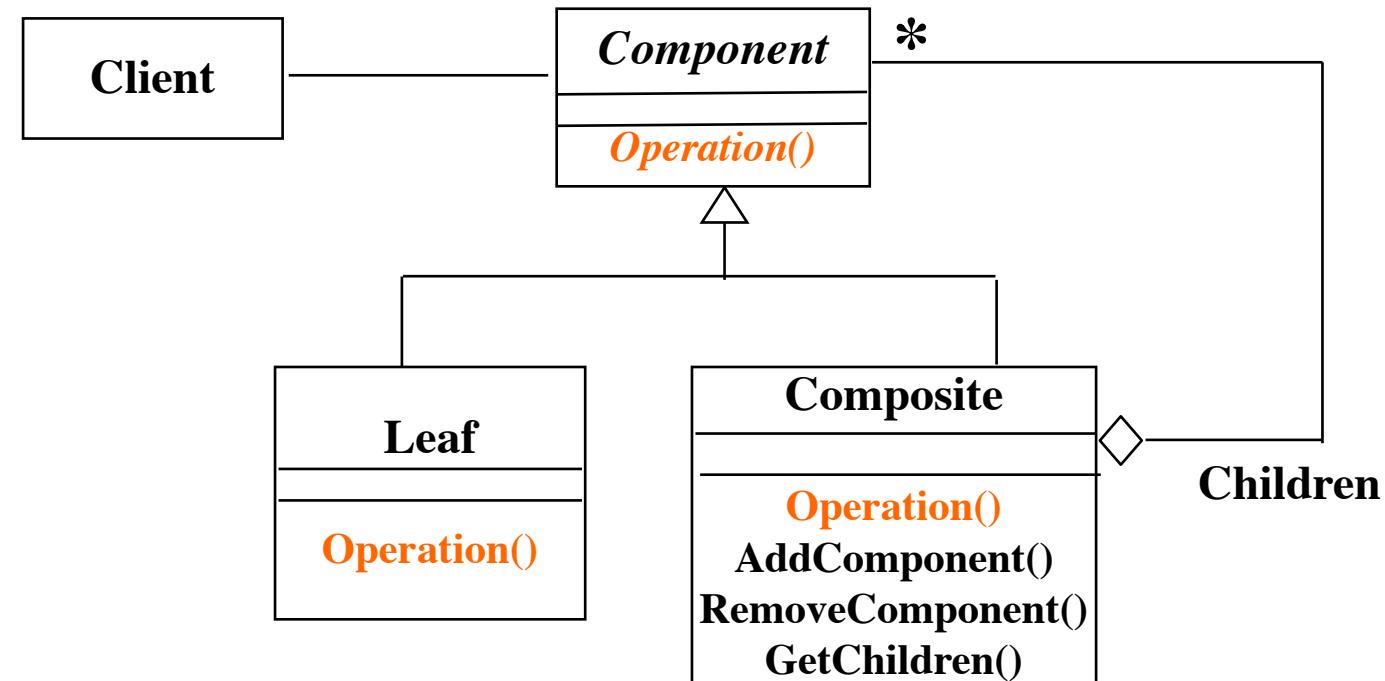
Design Patterns Taxonomy (23 Patterns)



Do you want to learn more about
Patterns?
Software Engineering Patterns
WS 2018-19

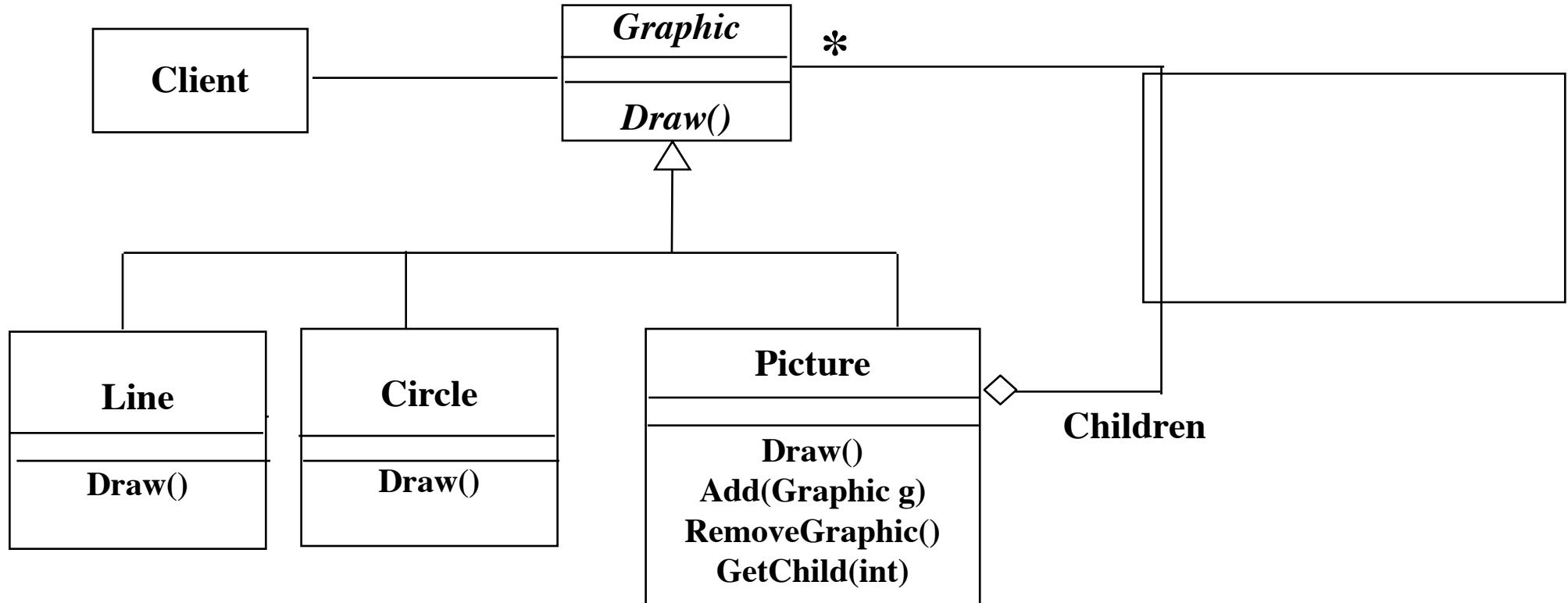
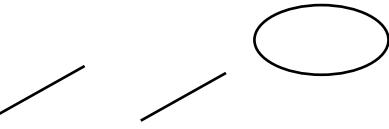
Composite Pattern

- **Observation:**
 - Many problems can be solved by modeling them as hierarchies with arbitrary depth and width
- **Solution:**
 - The Composite Pattern lets a client treat an individual class called Leaf and compositions of Leaf classes **uniformly**

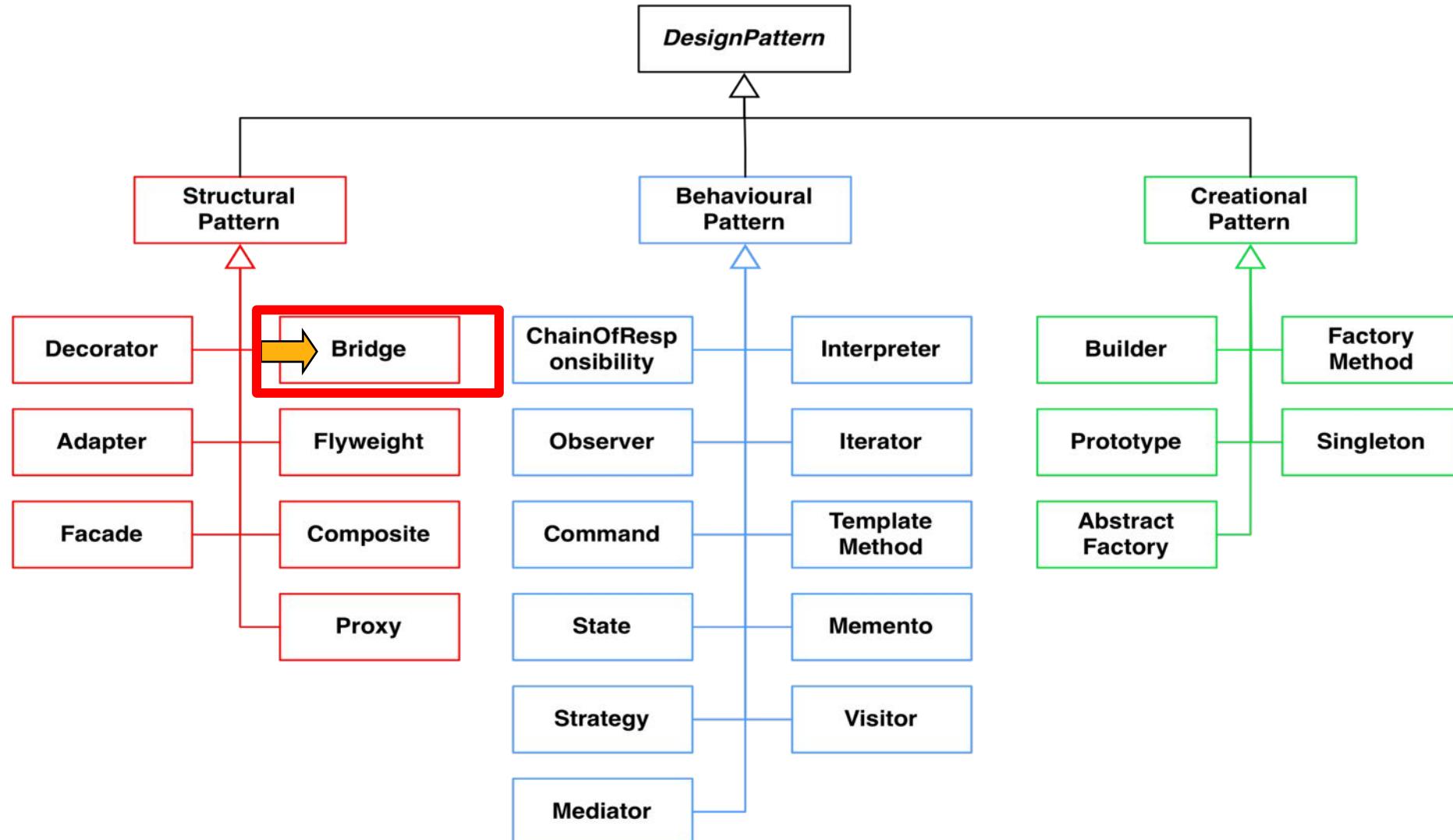


Composite Pattern in Graphic Applications

Graphic represents both, **Line** and **Circle** (Leaf), as well as **Picture** (Composite).



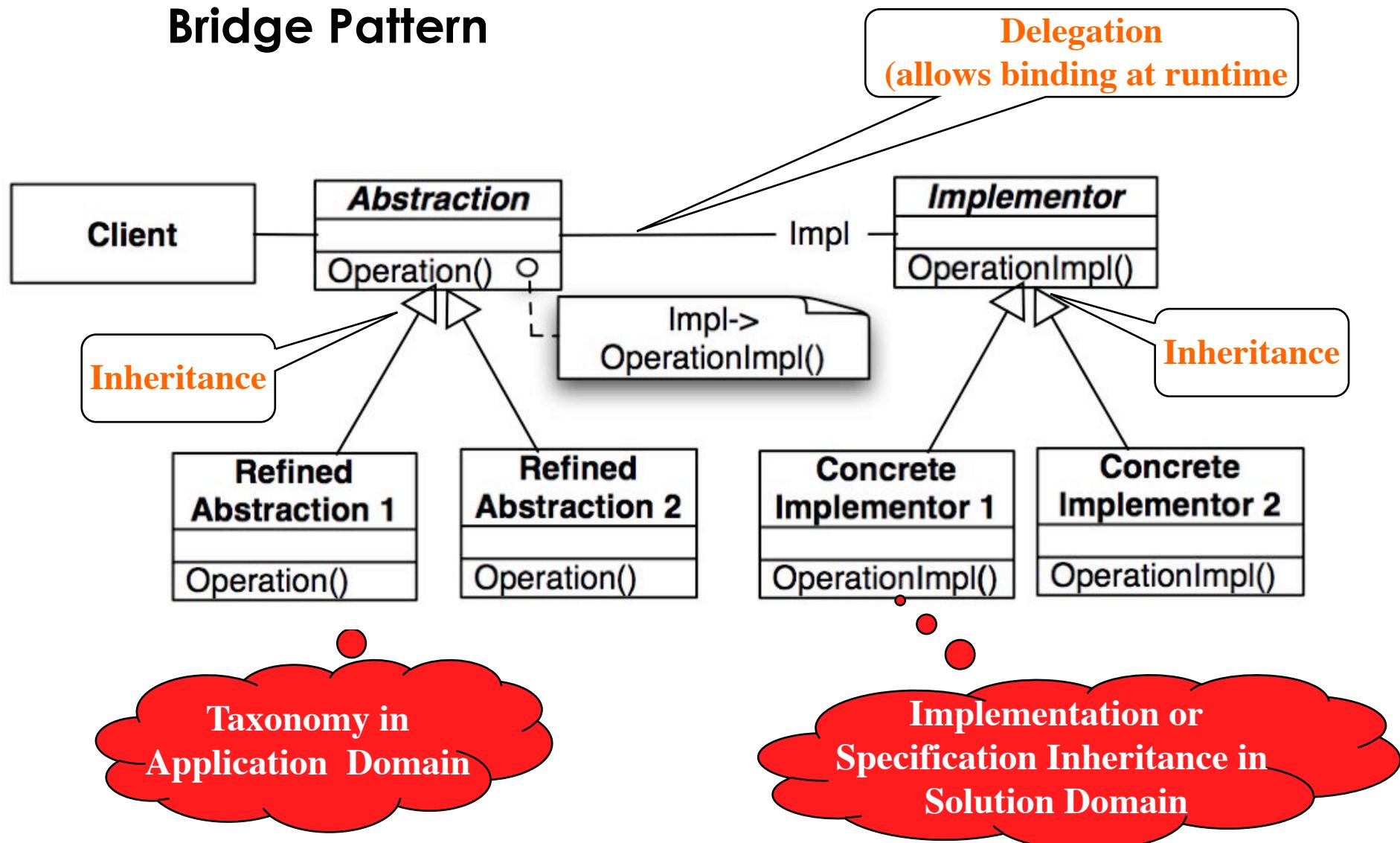
Taxonomy of Design Patterns



Bridge Pattern

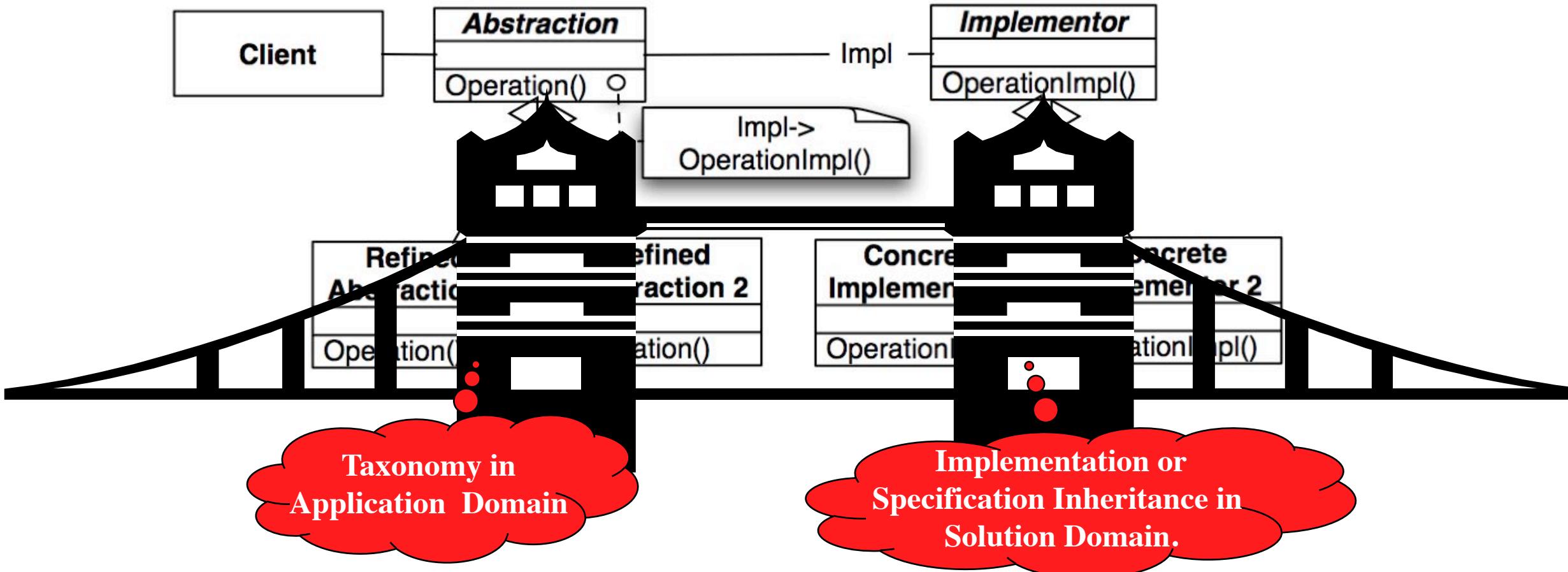
- **Problem:** Many design decisions are made final at design time ("design window") or at compile time
 - Example: Binding a client to one of many implementation classes of an interface
- Sometimes it is desirable to delay design decisions until run time
 - Example: We have two clients. One client uses a very old implementation, the other client uses a more recent implementation of an interface
- The bridge pattern allows to delay the binding between an interface and its subclass to the start-up time of the system (e.g. in the constructor of the interface class).

Bridge Pattern

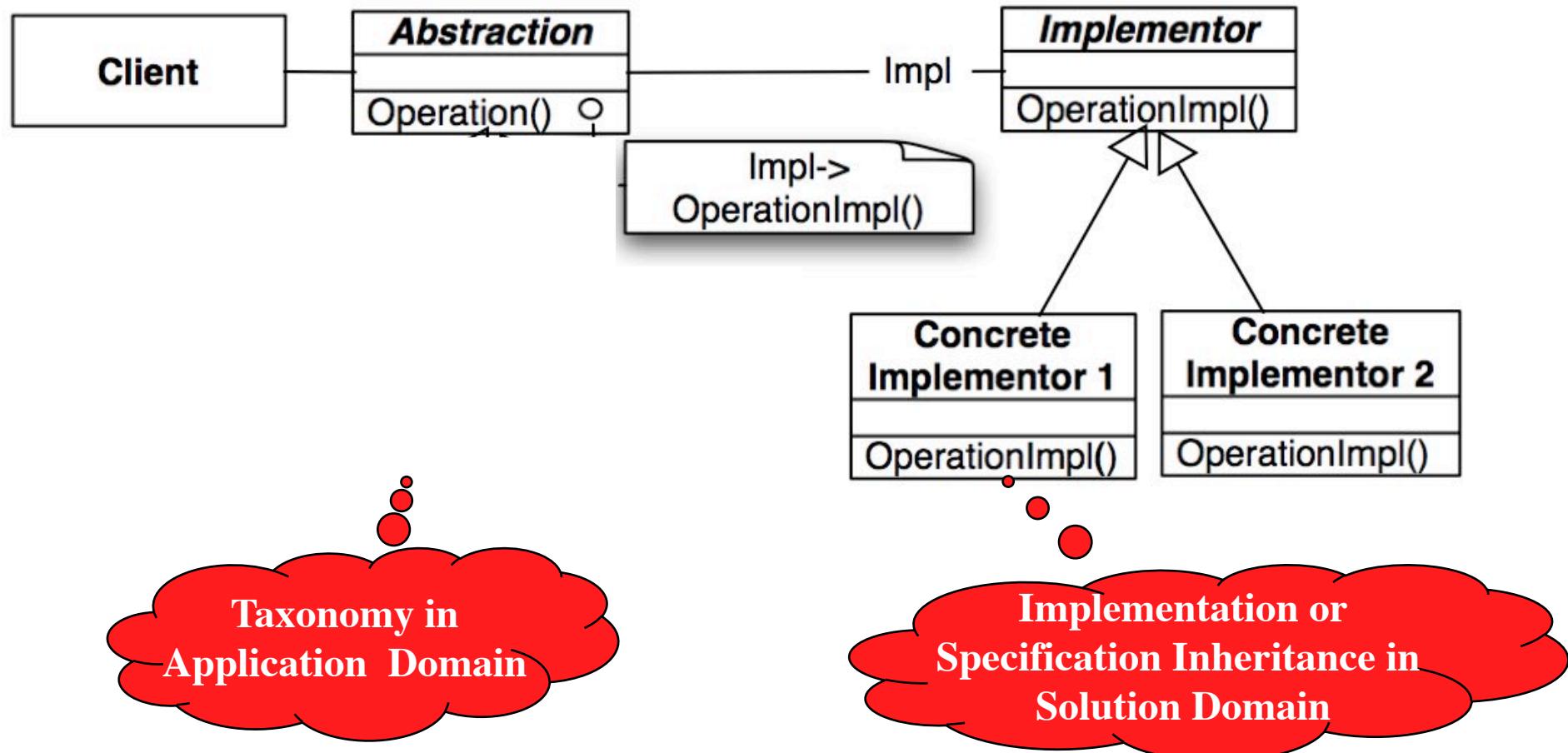


Why the Name Bridge Pattern?

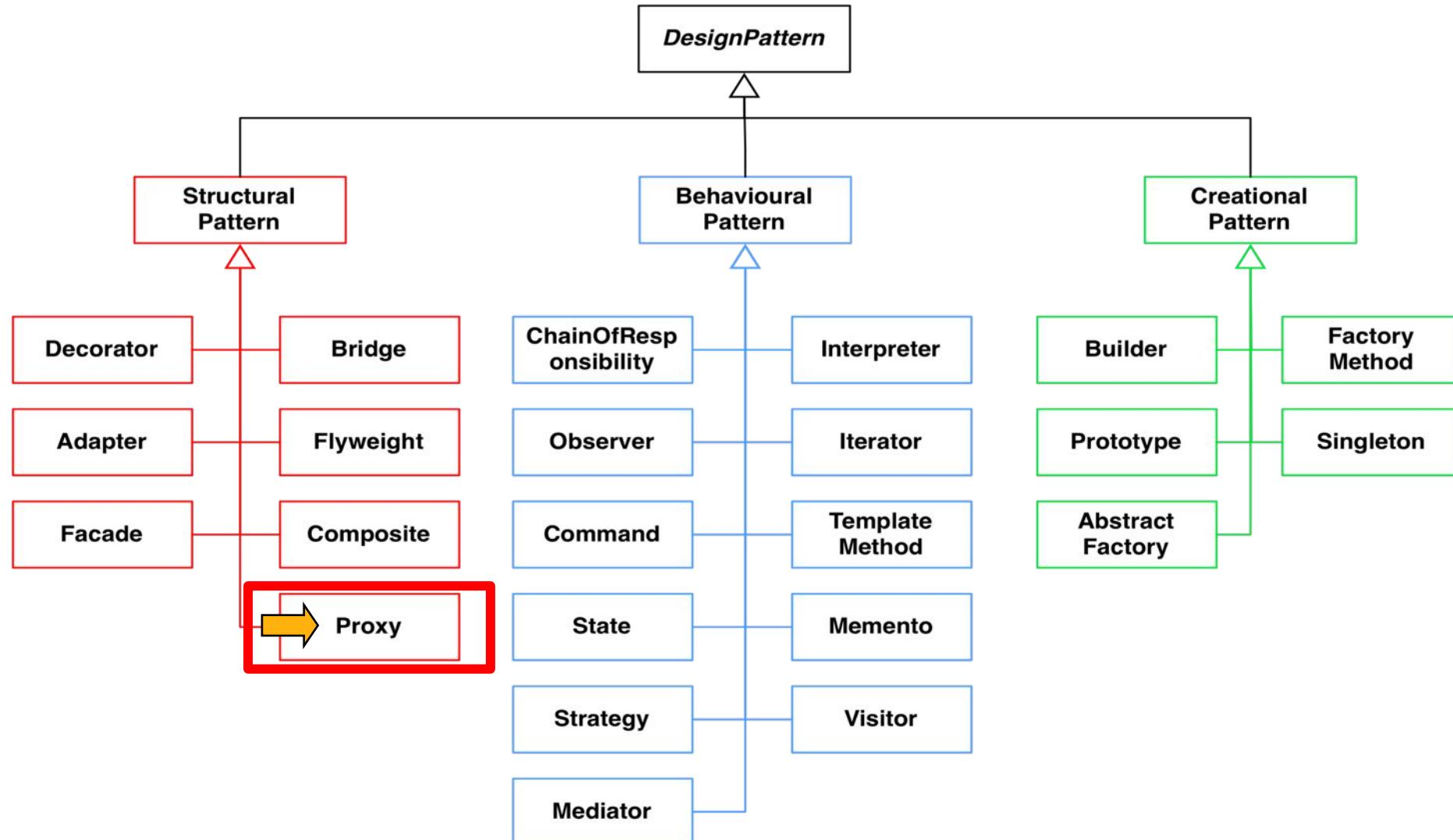
It provides a bridge between the abstraction (in the application domain) and the implementor (in the solution domain)



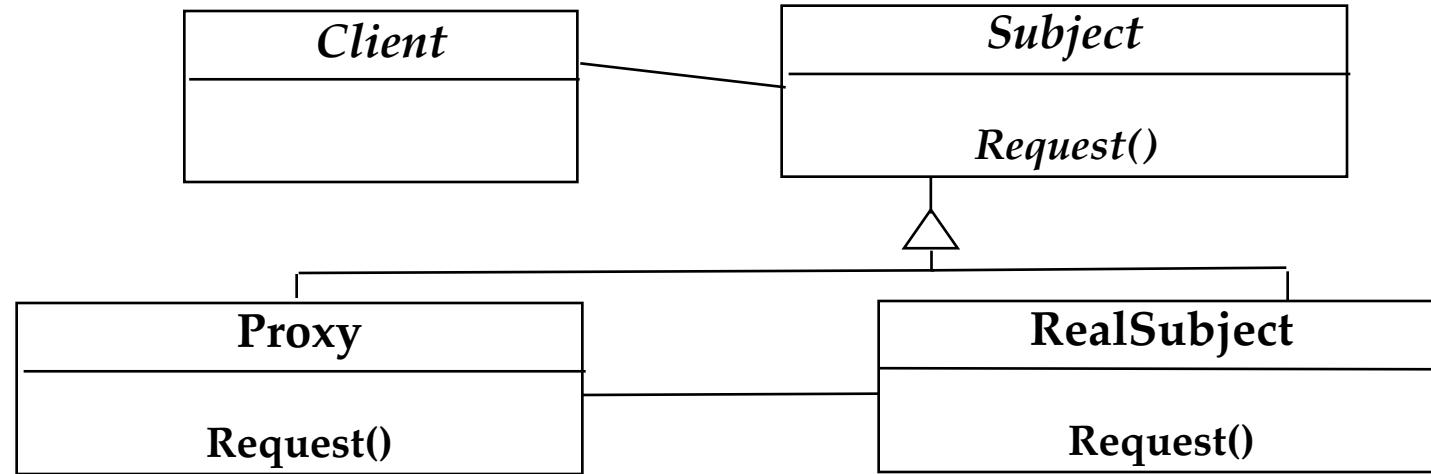
“Degenerated” Bridge Pattern: No Application Domain Taxonomy



Taxonomy of Design Patterns



Proxy Pattern



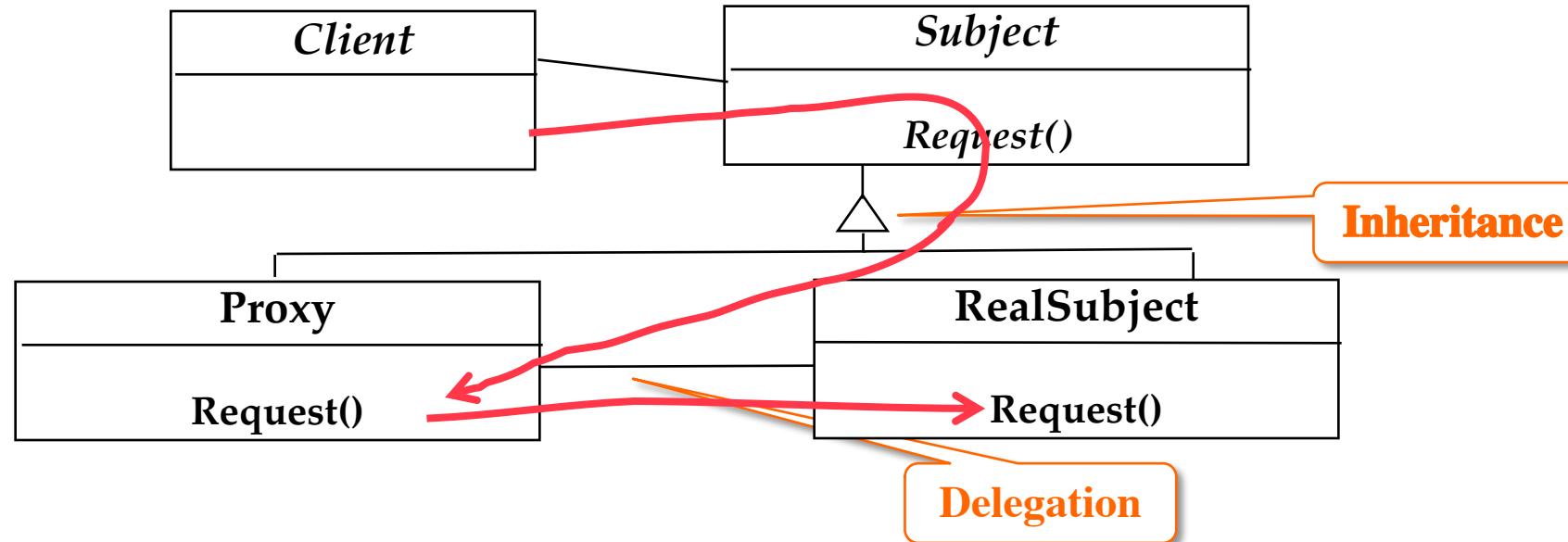
Proxy and RealSubject are subclasses of Subject

The Client always calls Request() in an instance of type Proxy

The Implementation of Request() in Proxy then uses Delegation to access Request() in RealSubject.

Proxy Pattern

*The Proxy pattern uses
Inheritance and Delegation*



Proxy and RealSubject are subclasses of Subject

The Client always calls Request() in an instance of type Proxy

The Implementation of Request() in Proxy then uses Delegation to access Request() in RealSubject.

The Proxy Pattern solves two Problems

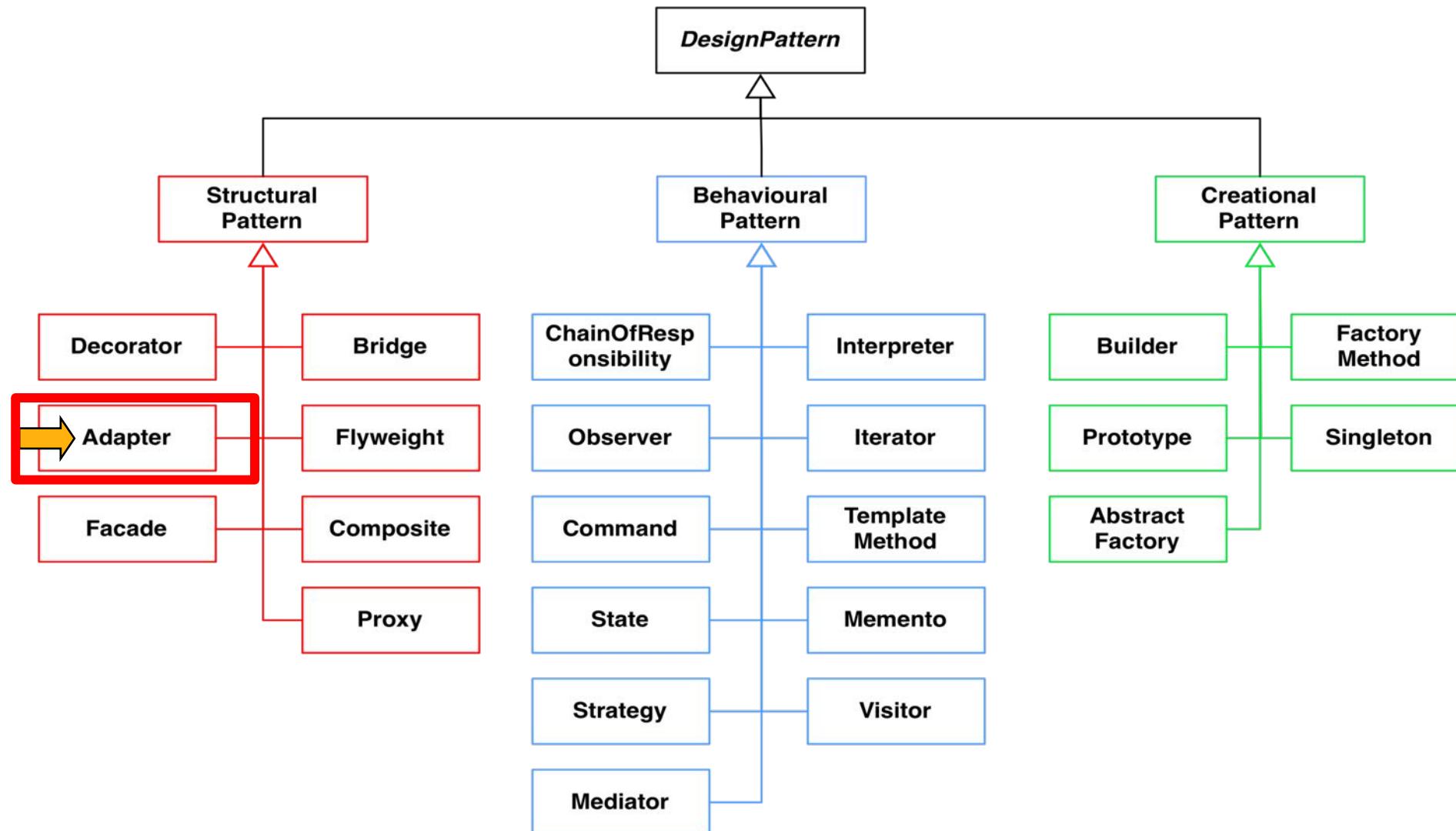
- **Problem #1:** The object is complex, its instantiation is expensive
 - Solution:
 - Delay the instantiation until the object is actually used
 - If the object is never used, then the costs for its instantiation do not occur
- **Problem #2 :** The object is located on another node (i.e on a web server), accessing the object is expensive
 - Solution:
 - Instantiate and initialize a “smaller” local object, which acts as a representative (“proxy”) for the remote object
 - Try to access mostly the local object
 - Access the remote object only if really necessary.

Applicability of the Proxy Pattern

- **Caching (Remote Proxy)**
 - The proxy object is a local representative for an object in a different address space
 - Caching is good if information does not change too often
 - If the information changes, the cache needs to be flushed
- **Substitute (Virtual Proxy)**
 - The object is expensive to create or to download. The proxy object acts as stand-in
 - Good for information that is not immediately accessed
 - Good for objects that are not visible (not in line of sight, far away)
 - Example: Google Maps, Fog of War
- **Access Control (Protection Proxy)**
 - The proxy object provides access control to the real object
 - Good when different objects should have different access and viewing rights
 - Example: Grade information shared by administrators, teachers and students.

Access Control
List (see slide 18)

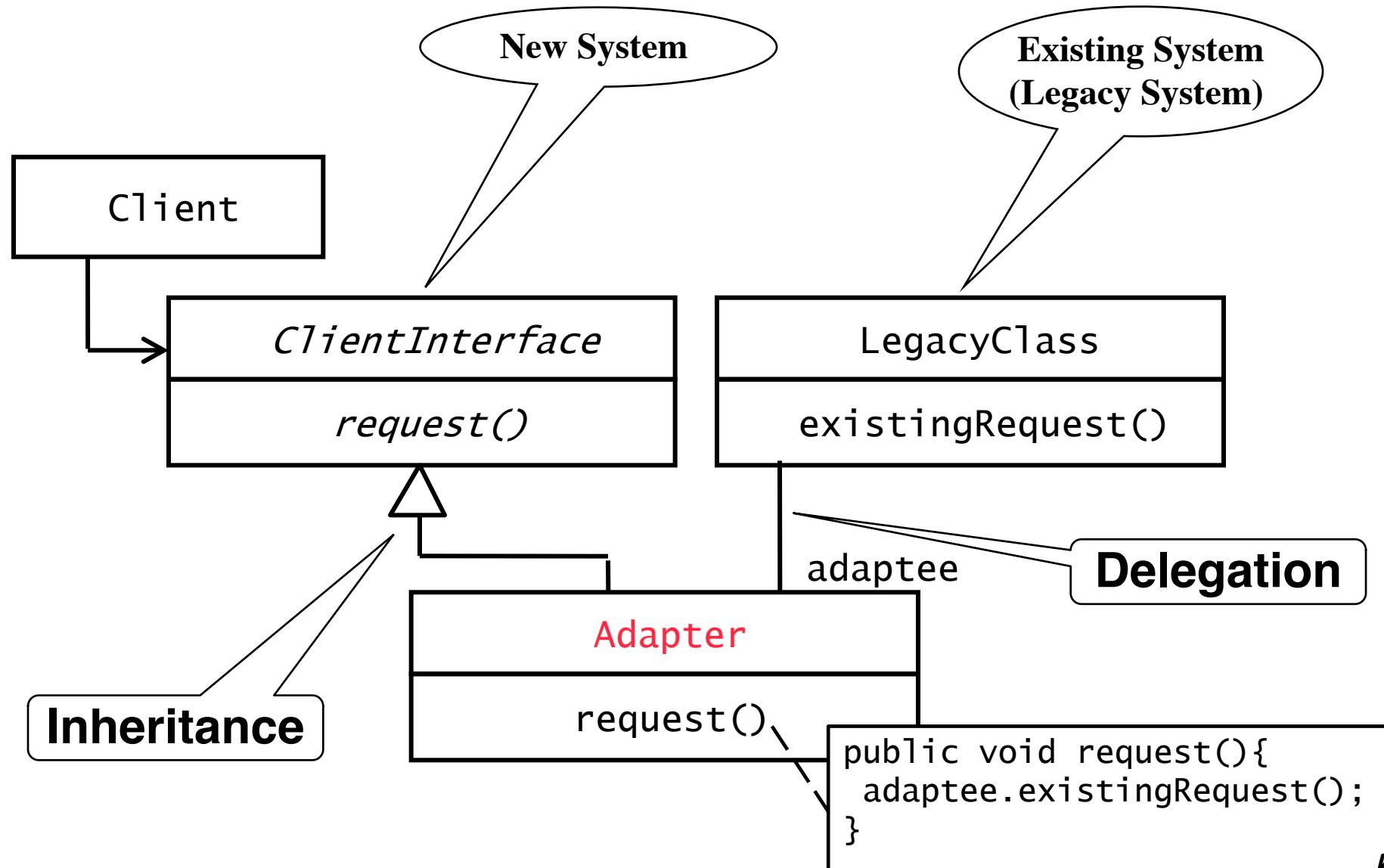
Taxonomy of Design Patterns (23 Patterns)



Adapter Pattern

- **Adapter Pattern:** Connects incompatible components
 - Allows the reuse of existing components
 - Converts the interface of the existing component into another interface expected by the calling component
 - Useful in interface engineering projects and in reengineering projects
 - Often used to provide a new interface for a legacy system
- Also known as a wrapper.

Adapter Pattern



Definition Legacy System

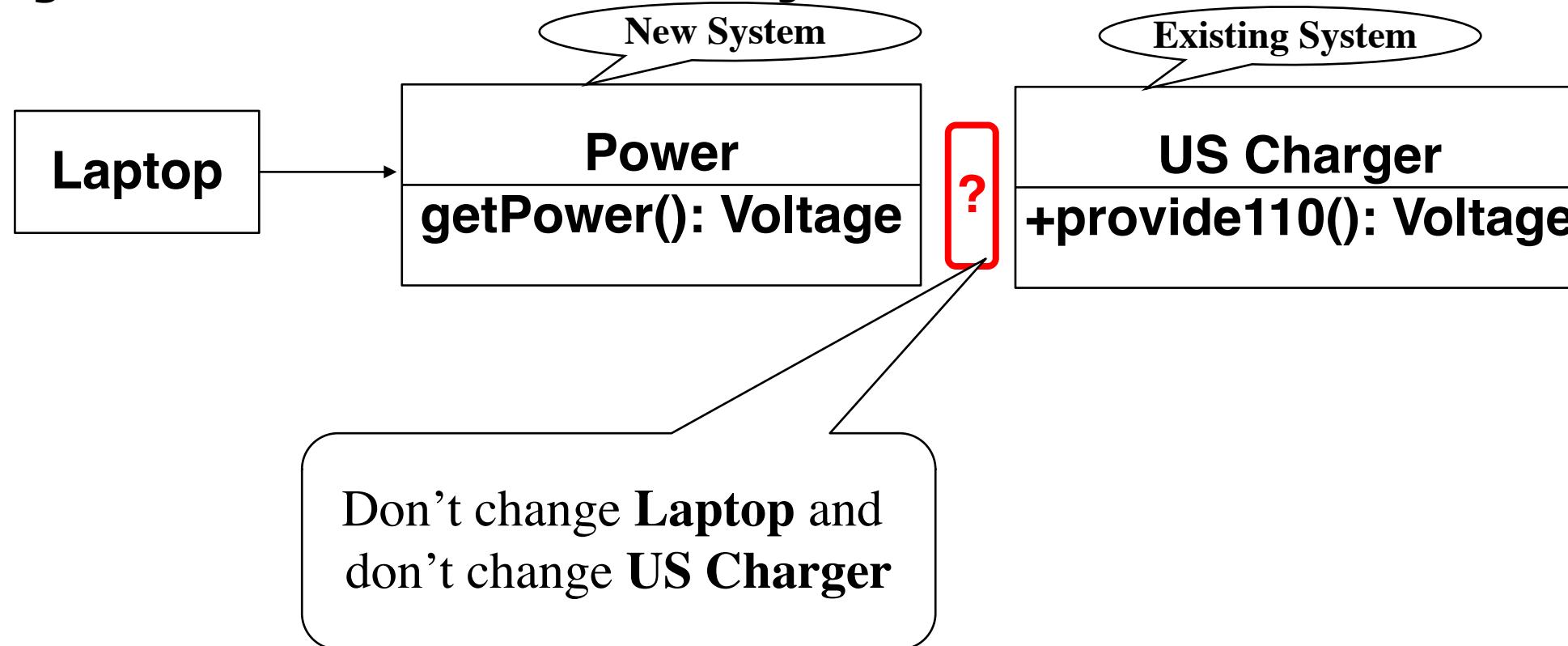
- A system that has evolved over the last 10 - 30 years
- It is still actively used in a production environment
- It is considered irreplaceable because re-implementation is too expensive or impossible
- It has a very high maintenance cost
- It was designed without modern software design methodologies or the design has been "lost"
- Change to the system is required due to new functional requirements, nonfunctional requirements or pseudo requirements.

Adapter Example: Accessing a Power Charger

Scenario: Joe is using a Laptop that provides Power by the `getPower Method()`.

Problem: Joe's laptop battery is empty. He has access to a US Charger that offers 110 Volt charging via the `provide110 Method()`.

Challenge: Provide access to the US Charger class from the Power class.



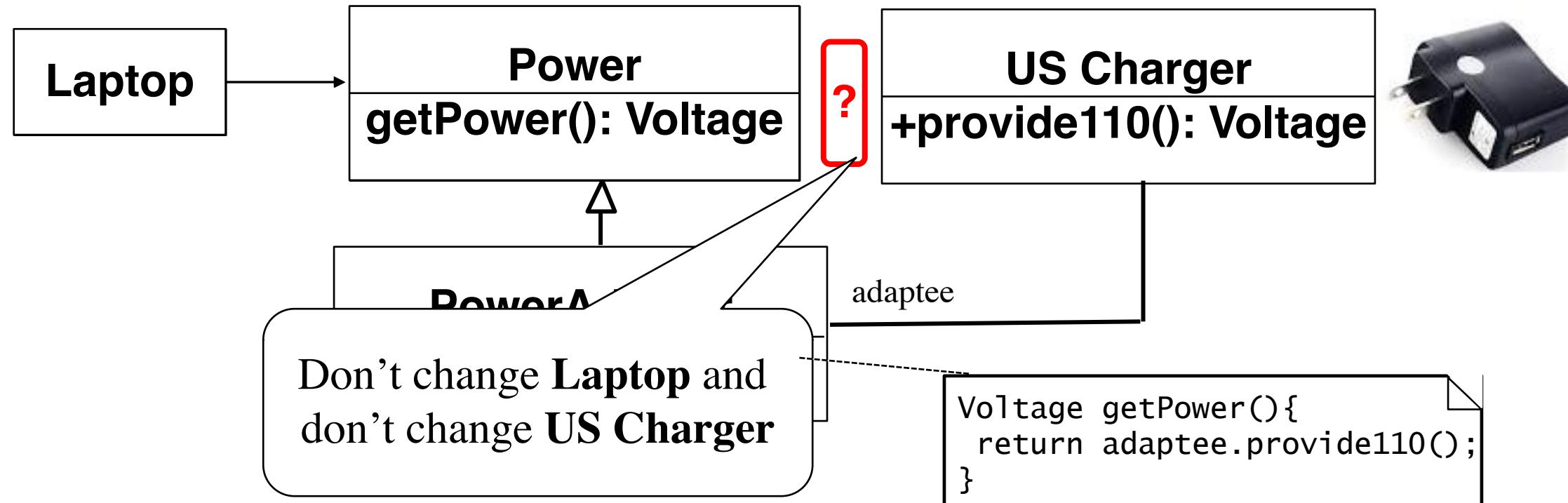
Example: Accessing a Power Charger (2)

Scenario: Joe is using a Laptop that provides Power by the getPower Method().

Problem: Joe's laptop battery is empty. He has access to a US Charger that offers 110 Volt charging via the provide110 Method().

Challenge: Provide access to the US Charger class from the Power class

- **Use the adapter pattern to connect to the US Charger**



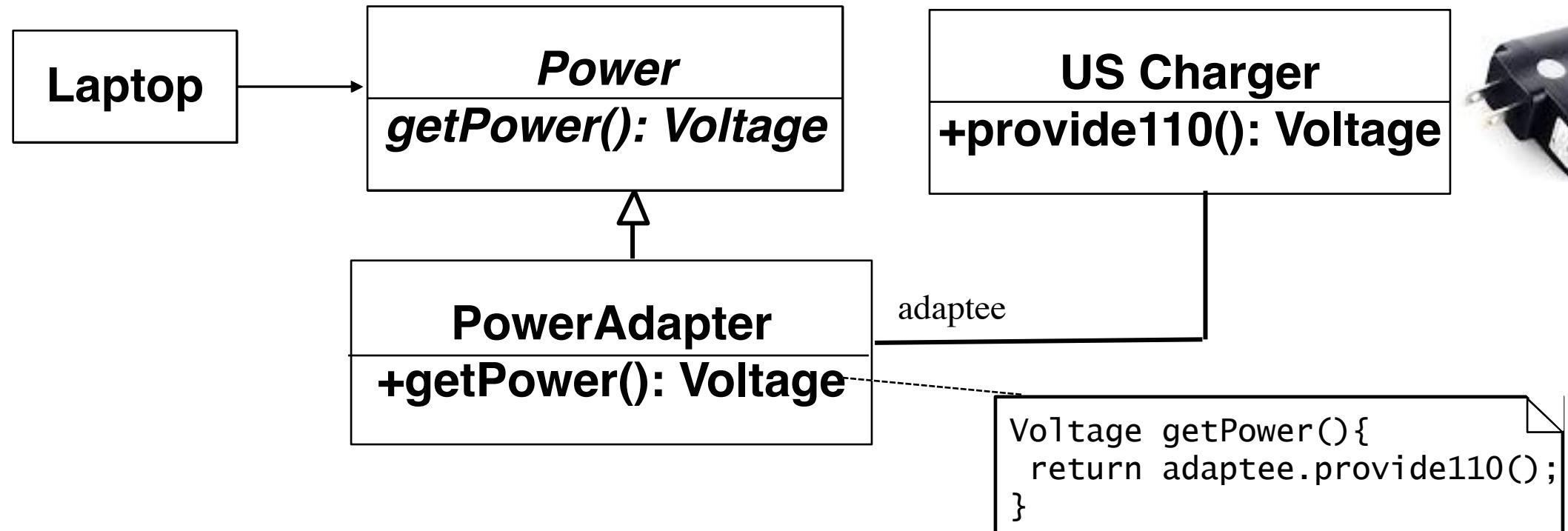
Example: Accessing a Power Charger (3)

Scenario: Joe is using a Laptop that provides Power by the `getPower` Method().

Problem: Joe's laptop battery is empty. He has access to a US Charger that offers 110 Volt charging via the `provide110` Method().

Modeling Task: Provide access to the US Charger class from the Power class

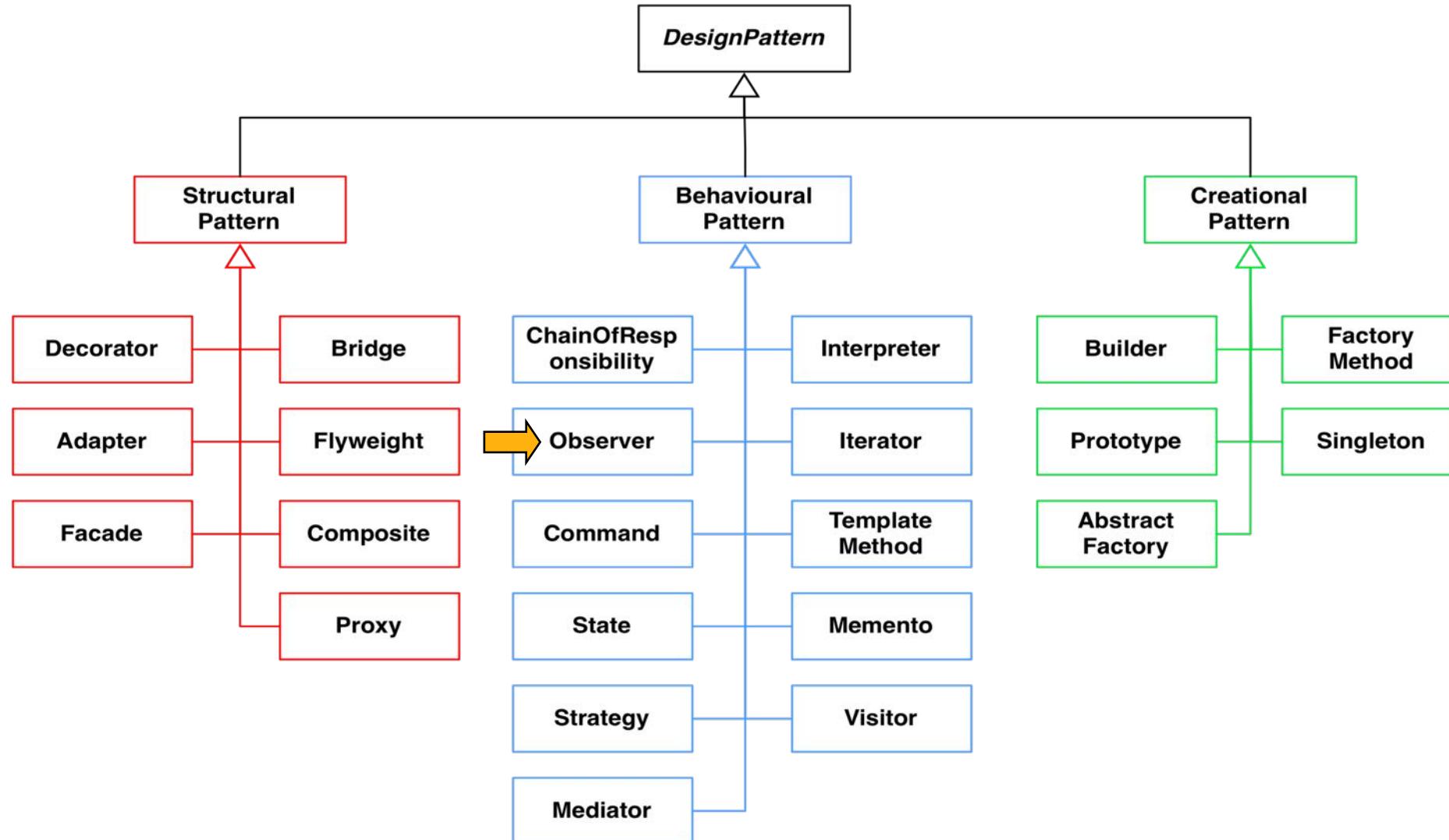
- **Use the adapter pattern to connect to the US Charger**



Comparison: Adapter vs Bridge

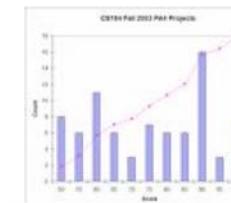
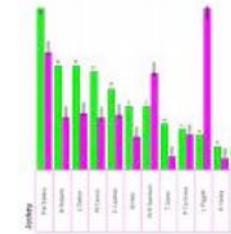
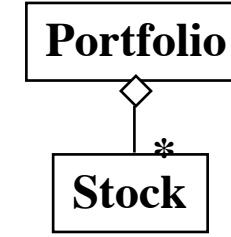
- Similarities:
 - Both hide the details of the underlying implementation
- Difference:
 - The adapter pattern is geared towards making unrelated components work together
 - Applied to systems that are already designed (reengineering, interface engineering projects)
 - “**Inheritance followed by delegation**”
 - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently
 - Green field engineering of an “extensible system”
 - New “beasts” can be added to the “zoo” (“application and solution domain zoo”, even if these are not known at analysis or system design time)
 - “**Delegation followed by inheritance**”.

Taxonomy of Design Patterns (23 Patterns)



Observer Pattern Motivation

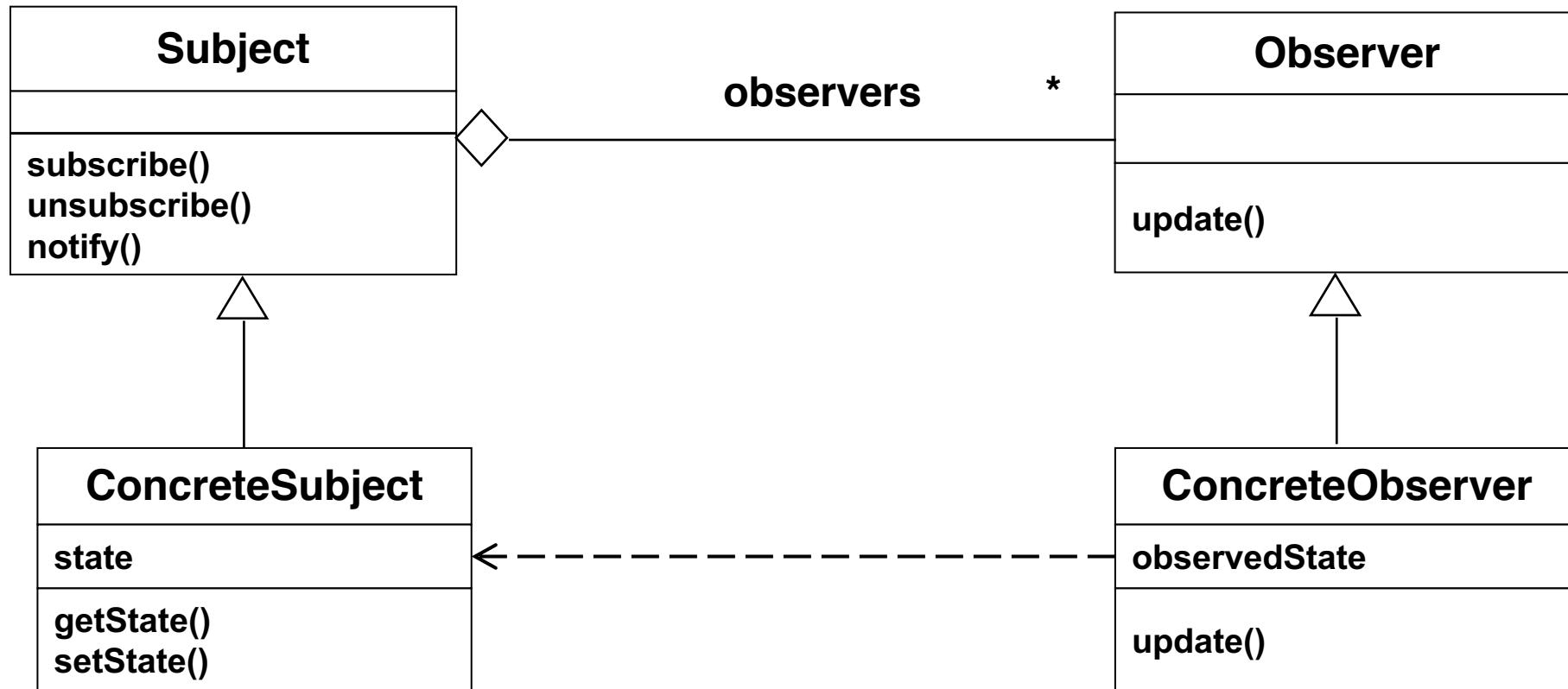
- Problem:
 - We have an object that changes its state quite often
 - Example: A Portfolio of stocks
 - We want to provide multiple views of the current state of the portfolio
 - Example: Histogram view, pie chart view, time line view
- Requirements:
 - The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes
 - The system design should be highly extensible
 - It should be possible to add new views - for example an alarm - without having to recompile the observed object or the existing views.



Observer Pattern

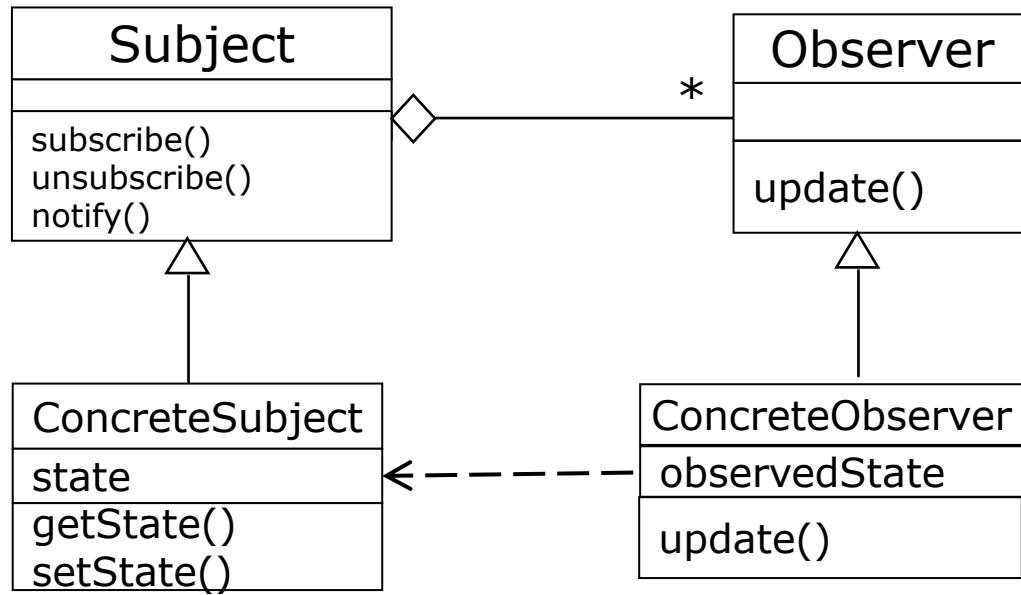
- Models a 1-to-many dependency between objects
 - Connects the state of an observed object, the **subject** with many observing objects, the **observers**
- Usage:
 - Maintaining consistency across redundant states
 - Optimizing a batch of changes to maintain consistency
- Also called **Publish and Subscribe.**

The Observer Pattern decouples a Subject from its Views



- The **Subject** represents the entity object
 - The **state** is contained in the subclass **ConcreteSubject**
- **Observers** attach to the Subject by calling **subscribe()**
- Each Observer has a different view of the state of the ConcreteSubject
 - The state can be **obtained and set** by the subclasses of type **ConcreteObserver**.

Variants of the Observer Pattern



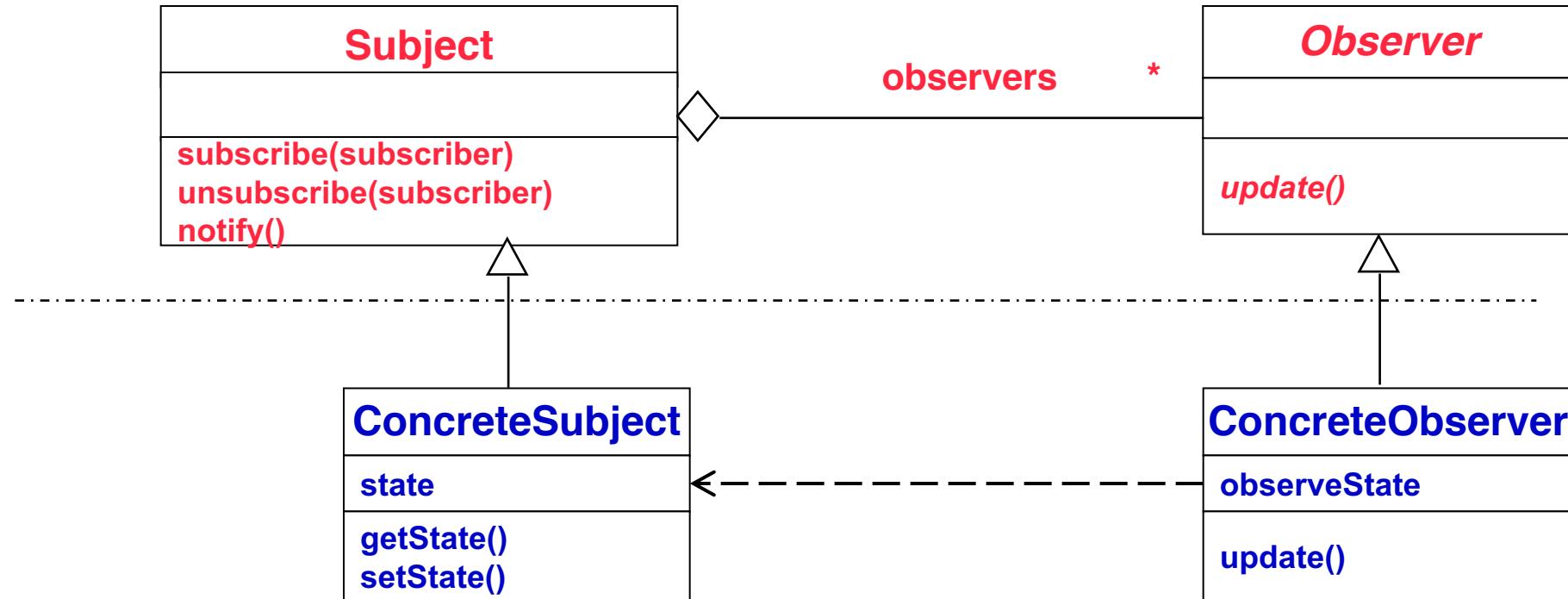
- Three variants for maintaining the consistency:
 - **Push Notification**: Every time the **state** of the **Subject** changes, `notify()` is called which calls `update()` in each **Observer**
 - **Push-Update Notification**: The **Subject** also includes the **state** that has been changed in each `update()` call
 - **Pull Notification**: An **Observer** inquires about the **state** of the **Subject** by calling `getState()`.

There are two possibilities for the **Subject** to change its **state**:

- An **Observer** calls `setState()`
- An external event occurs that changes the **state** of the Subject

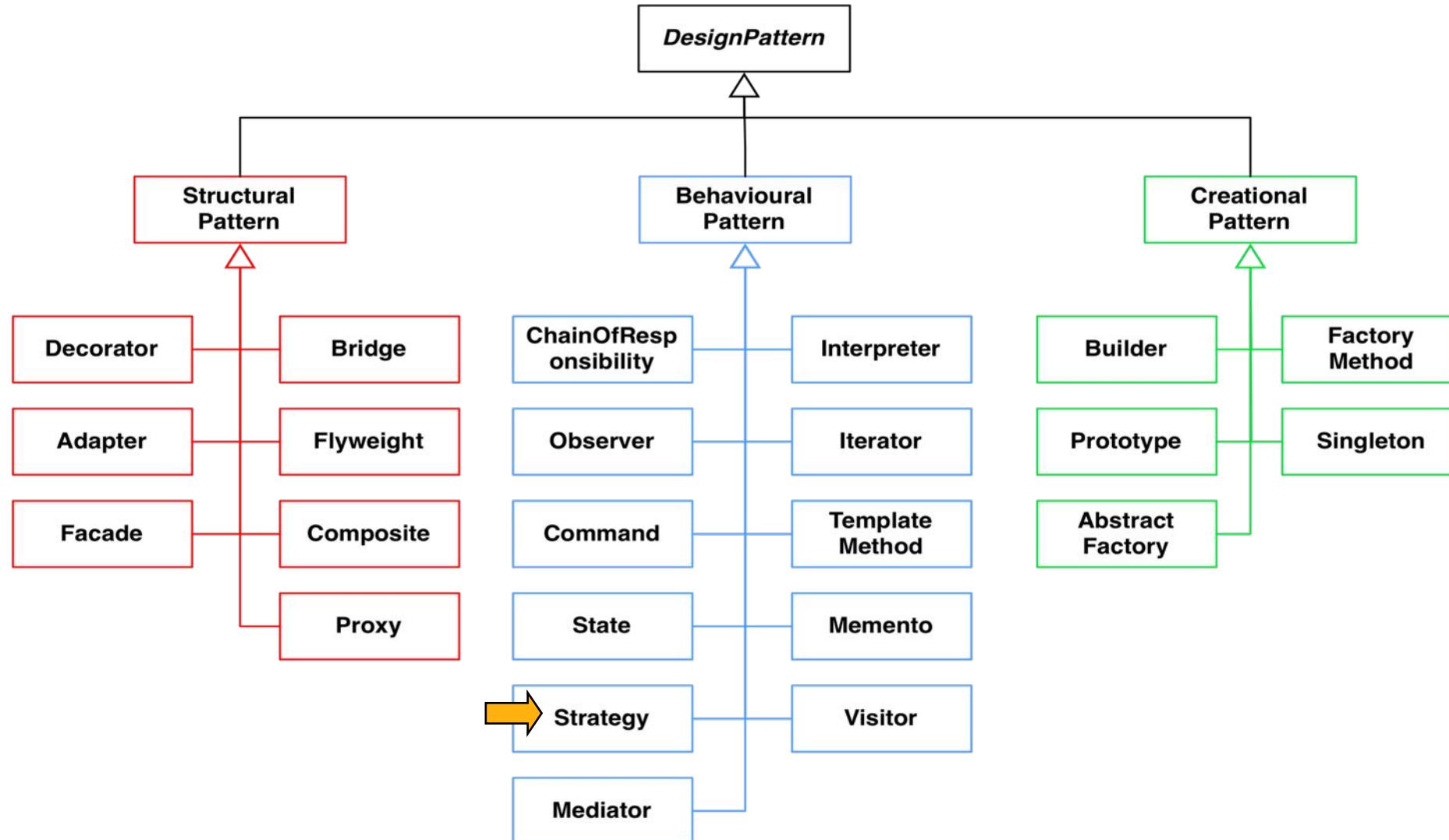
Review: Application Domain vs Solution Domain Objects

Requirements Analysis (Language of Application Domain)



Object Design (Language of Solution Domain)

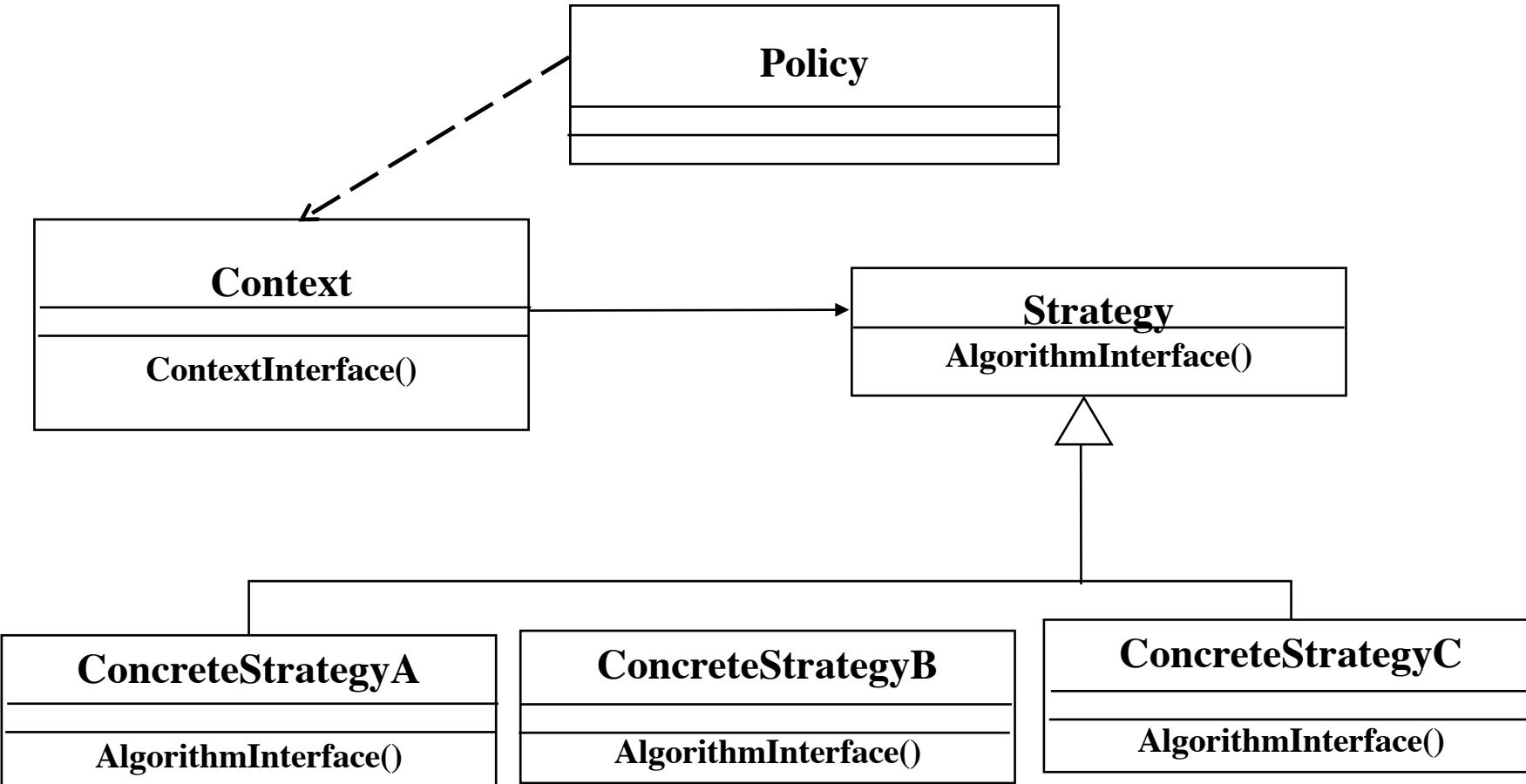
Taxonomy of Design Patterns (23 Patterns)



Strategy Pattern

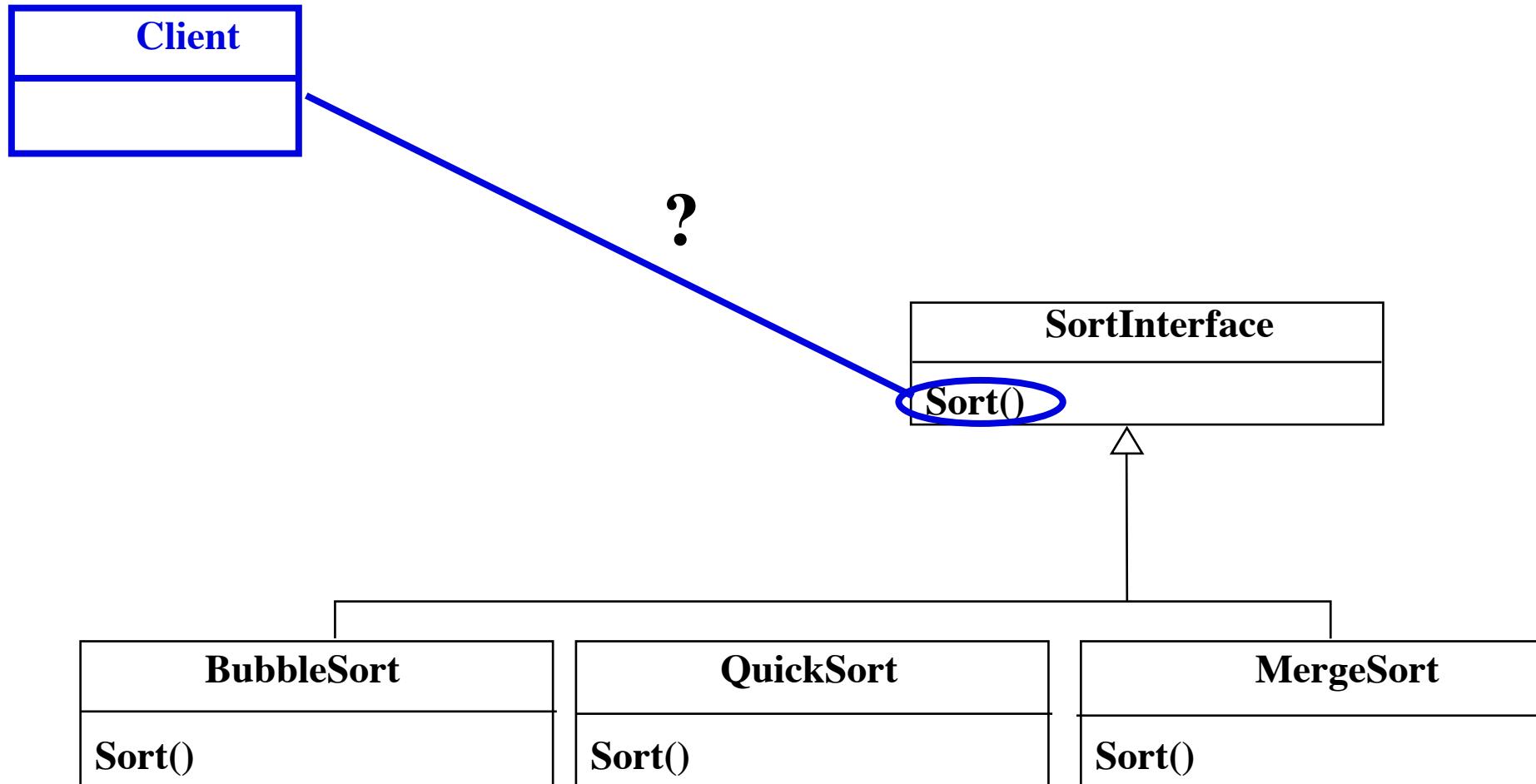
- Problem: Different algorithms exists for a specific task
 - We want to switch between these algorithms at run time
- Examples of specific tasks:
 - Different ways to sort a list (bubblesort, mergesort, quicksort)
 - Different collision strategies for objects in video games
 - Different ways to parse a set of tokens into an abstract syntax tree (Bottom up, top down)
 - Different development times require different algorithms
 - Prototype testing
 - System integration testing
 - System testing
- If we need a new algorithm, we want to add it without disturbing the application or the other algorithms.

Strategy Pattern

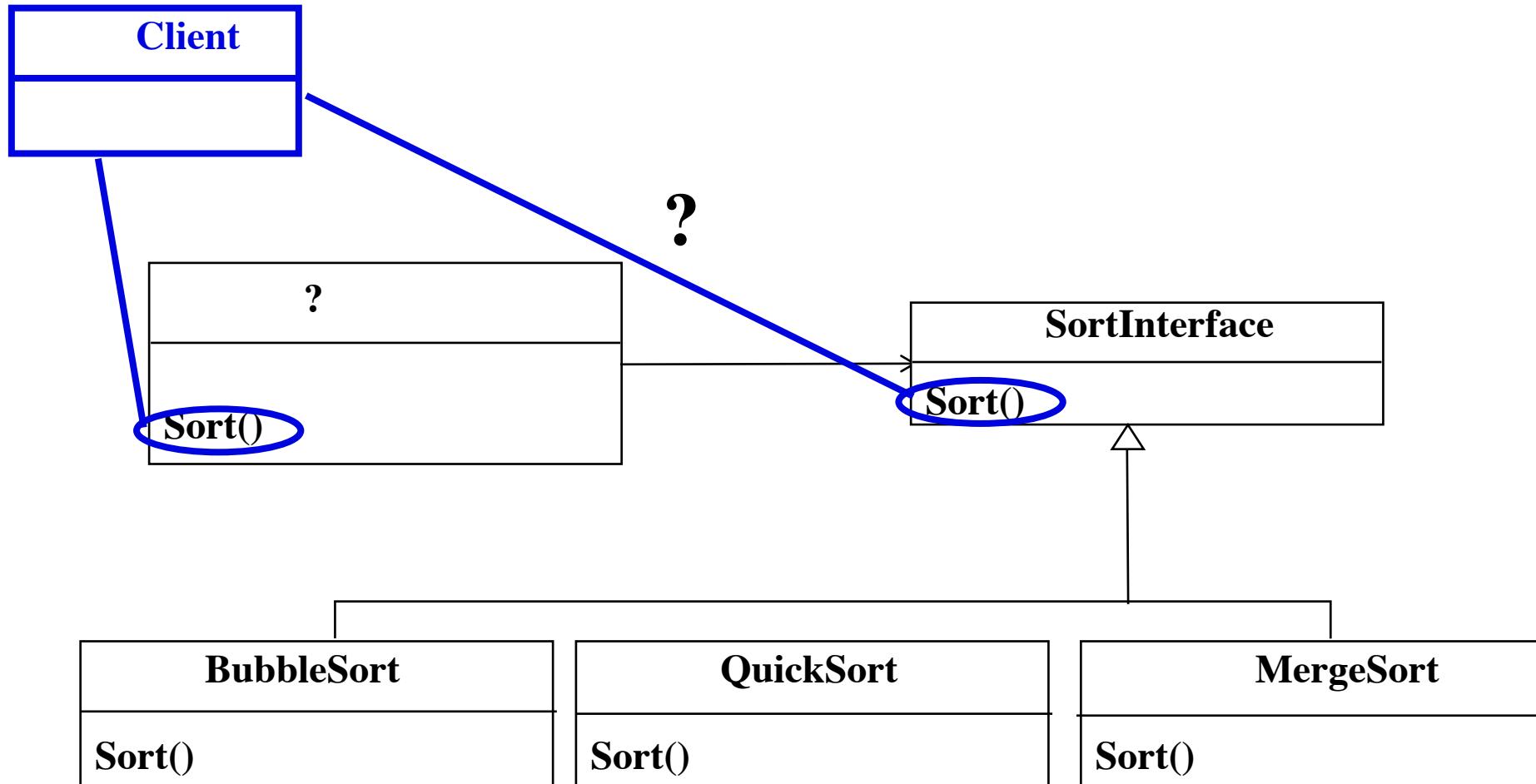


The Policy decides which ConcreteStrategy is best in a given Context.

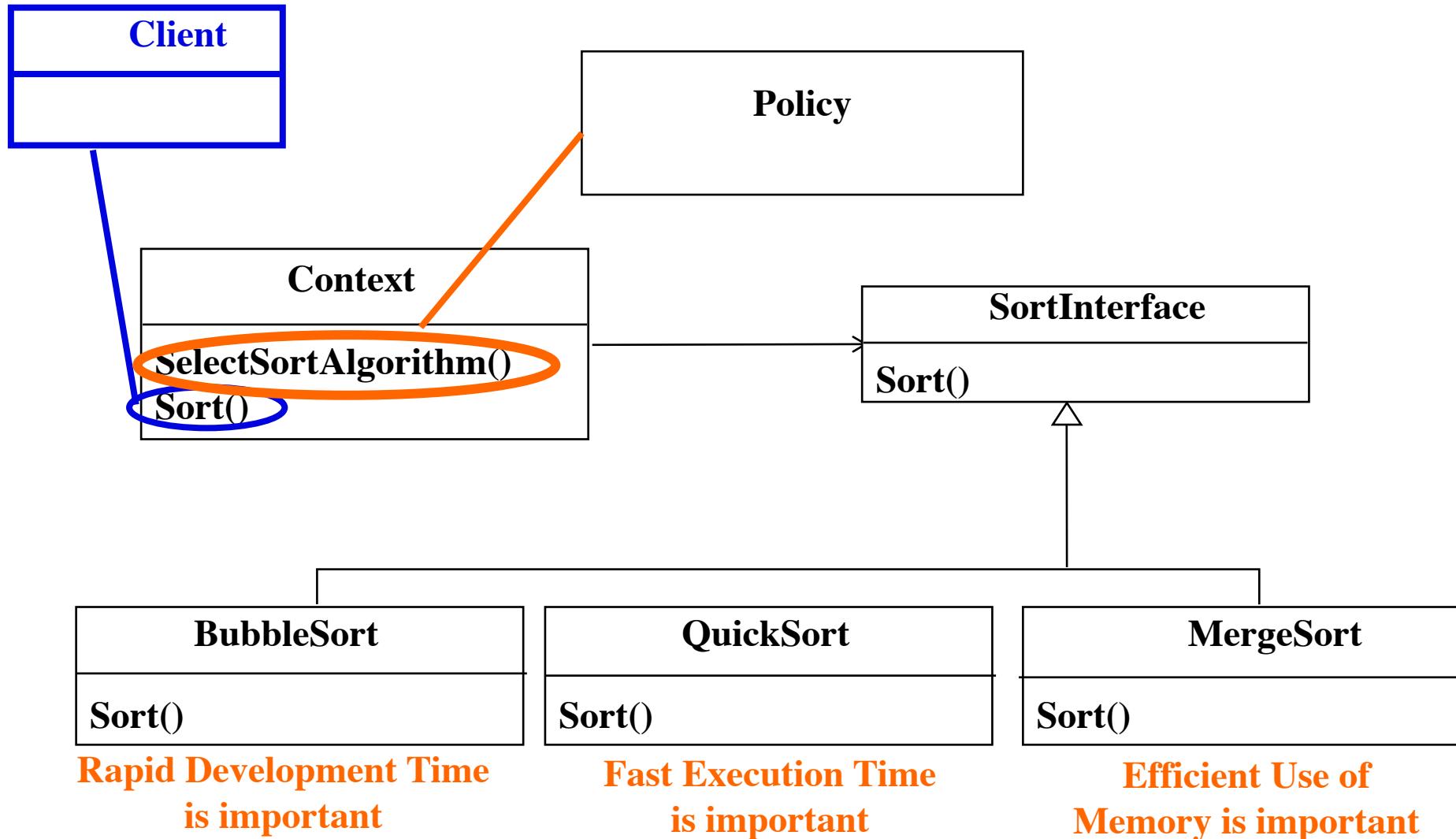
Example: Using the Strategy Pattern to switch between 3 different Sort Algorithms



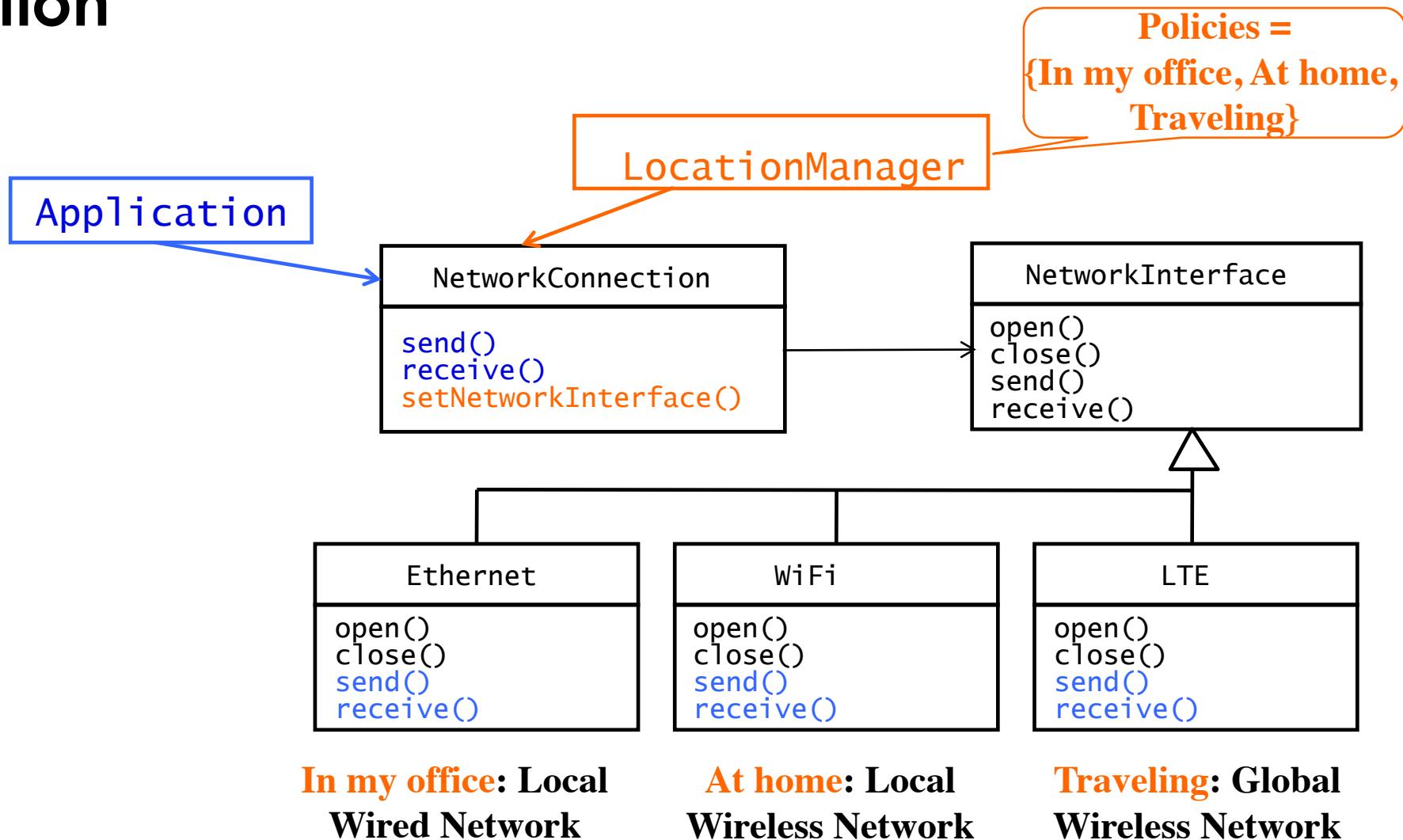
Example: Using the Strategy Pattern to switch between 3 different Sort Algorithms



Using a Strategy Pattern to switch between 3 different Sort Algorithms



Supporting Multiple implementations of a Network Connection



Summary

- Inheritance can be used in Analysis as well as in Object Design
 - During Analysis: Inheritance is used to describe taxonomies
 - During Object Design: Inheritance is used for interface specification and reuse.
- Blackbox vs Whitebox Reuse: Composition vs Inheritance
- Interface Specification
 - Implementation Inheritance
 - Delegation
 - Specification Inheritance
- Discovering Inheritance: Generalization and Specialization
- Design patterns
 - provide solutions to common problems
 - lead to extensible models and reusable code
 - Structural patterns, behavioral patterns, creational patterns
- My favorites: Observer, Proxy, Strategy.

Morning Quiz 07

- Start Time: **8:00**
- End Time: **8:10**
- To participate in the quiz, use ArTEMiS
- Click on Open Quiz or Start Quiz
- The Lecture starts at 8:10

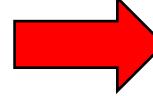
Remaining Time: **46 s**

Saved: never

● Connected

Submit

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	Start exercise
Good Morning Quiz 07		Open Quiz 

Only click on Submit when you have entered all answers!

Model Transformations and Refactorings

Bernd Bruegge, Stephan Krusche
Applied Software Engineering
Technische Universitaet Muenchen
7 June 2018

Roadmap for Today's Lecture

- **Context and Assumptions**

- We completed Chapter 1 to 9 in the OOSE text book by Bruegge and Dutoit

- **Content of this lecture**

- We introduce how to map models to code and to tables
- We complete Chapter 10

- **Objective:** At the end of this lecture you are able to

- Distinguish between forward and reverse engineering
- Refactor models and source code
- Map models to source code, contracts and tables in relational databases
- Understand the state pattern
- Transform objects into JSON
- Map inheritance to database tables

Overview of Today's Lecture

→ Model-Driven Engineering

- Model Transformation: Model Refactoring
- Forward Engineering: Mapping Models to Code
 - Mapping Models to Classes
 - Mapping Models to State Charts
 - Mapping Models to Contracts
 - Mapping Models to Tables
- Source Code Transformation: Code Refactoring
 - Object Transformation: Mapping Objects to JSON
- Reverse Engineering: Mapping Source Code to Models.

Model-Driven Engineering (MDE)

- Model-driven engineering is a methodology which focuses on the creation of domain models
 - Domain models are abstract representations of the knowledge and activities for a particular application domain
- The aim is to increase of productivity by
 - Reuse of standardized models
 - Reuse of knowledge (patterns in the application domain)
 - Collaboration between developers working on the systems
 - Standardized terminology
 - Identification of best practices in the application domain.

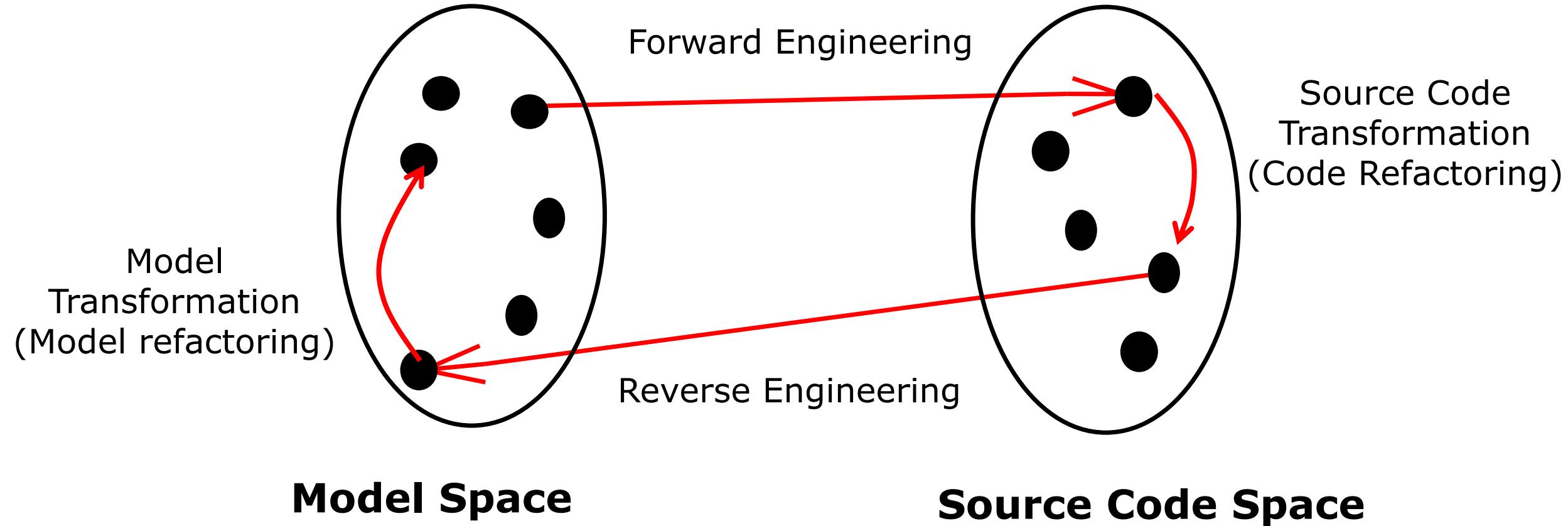
Model Based Software Engineering (MBSE)

- Application of modeling to support requirements, design, analysis, verification, validation activities
- Advantages
 - Better communications and knowledge management
 - Impact analysis of requirements and design changes
 - More complete representation
- A system engineering model contains several models addressing all aspects of the participating systems, hardware as well as software:
 - Functional model
 - Behavioral model (Dynamic model)
 - Structural model (Object model)
 - Cost model, organizational model, development environment, target environment.

Model Transformation

- The goal in model-driven engineering is to **automate** model transformations
 - Saves effort and reduce errors
- One of the characteristics of a model transformation is that the transformation is also a model, that means, it conforms to a given meta-model
- **Model Transformation:**
 - **Input:** A model conforming to a given meta-model
 - **Output:** Another model conforming to a given meta-model.

4 Types of Model Transformations



Refactoring: Definition

- **Refactoring** (noun): A change made to the internal structure of source code to make it...
 - ... Easier to understand and
 - ... Cheaper to modify
 - ... Without changing its observable behavior
- **Refactor** (verb): To restructure source code by applying a series of refactorings.
- **Note:** Models can also be refactored

Notations for the exchange and the communication between models

- **XMI** (XML metadata interchange): International standard for the exchange of models between modeling tools and MOF-based repositories
 - Unfortunately there are many different XMI implementations by different tool vendors ☹
- Used for integration
 - Tools, applications, repositories, data warehouses
 - Typically used as interchange format for UML tools
- Defines rules for schema definition
 - Schema production — how is a metamodel mapped onto a grammar?
 - Definition of schema from any valid Meta Object Facility (MOF) model

XMI Example

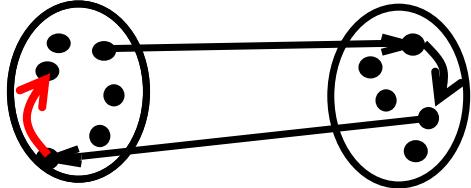
```
<?xml version="1.0"?>
<XMI xmi.version="1.2" xmlns:UML="org.omg/UML/1.4">
  <XMI.header>...</XMI.header>
  <XMI.content>
    <UML:Model xmi.id="M.1" name="address" visibility="public" isSpecification="false" isRoot="false"
               isLeaf="false" isAbstract="false">
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id="C.1" name="Address" visibility="public" isSpecification="false" namespace="M.1"
                   isRoot="true" isLeaf="true" isAbstract="false" isActive="false">
          <UML:Classifier.feature>
            <UML:Attribute xmi.id="A.1" name="name" visibility="private" isSpecification="false" ownerScope="instance"/>
            <UML:Attribute xmi.id="A.2" name="street" visibility="private" isSpecification="false" ownerScope="instance"/>
            <UML:Attribute xmi.id="A.3" name="zip" visibility="private" isSpecification="false" ownerScope="instance"/>
            <UML:Attribute xmi.id="A.4" name="region" visibility="private" isSpecification="false" ownerScope="instance"/>
            <UML:Attribute xmi.id="A.5" name="city" visibility="private" isSpecification="false" ownerScope="instance"/>
            <UML:Attribute xmi.id="A.6" name="country" visibility="private" isSpecification="false" ownerScope="instance"/>
          </UML:Classifier.feature>
        </UML:Class>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```

Address
name
street
zip
region
city
country

Overview of Today's Lecture

- Model-Driven Engineering
- Model Transformation: Model Refactoring
- Forward Engineering: Mapping Models to Code
 - Mapping Models to Classes
 - Mapping Models to State Charts
 - Mapping Models to Contracts
 - Mapping Models to Tables
- Source Code Transformation: Code Refactoring
 - Object Transformation: Mapping Objects to JSON
- Reverse Engineering: Mapping Source Code to Models.

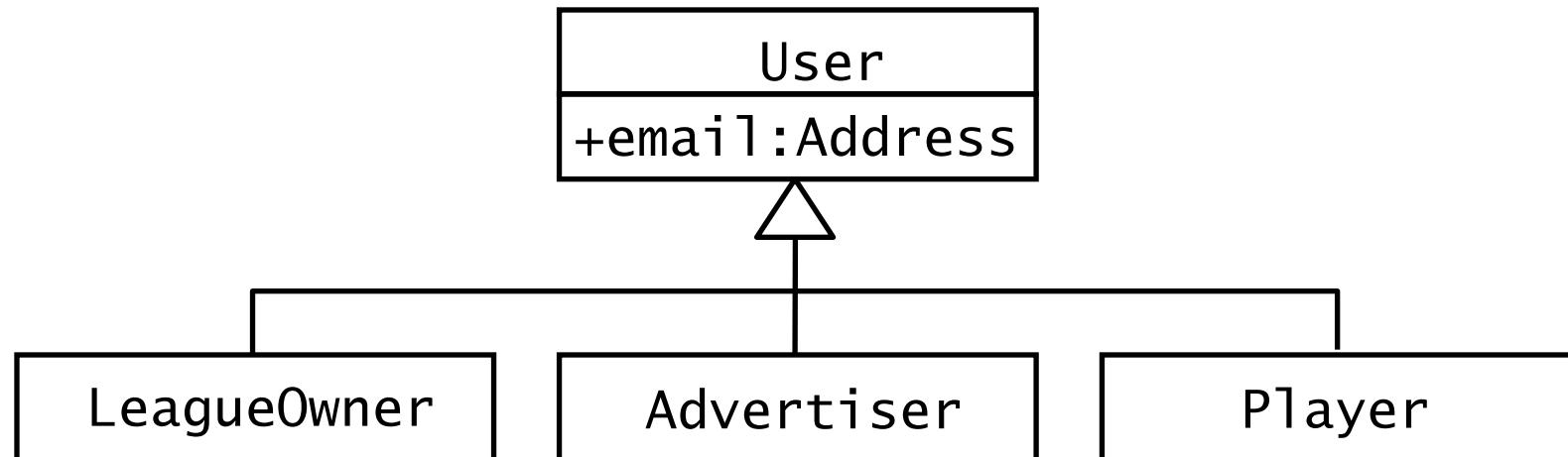
Model Transformation: Model Refactoring



Object model before model refactoring:



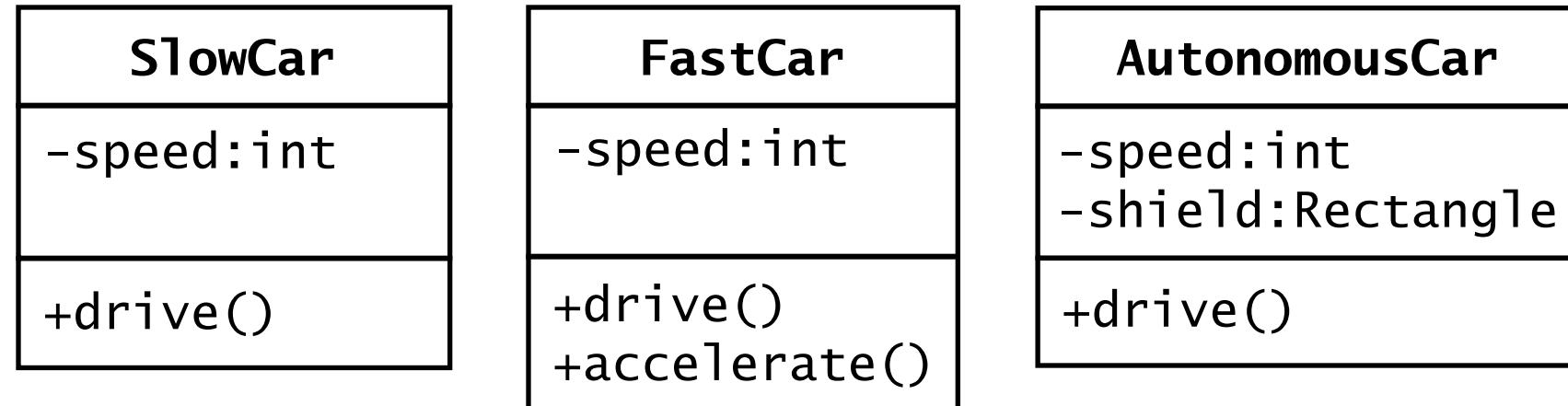
Object model after model refactoring:



In-Class Exercise 01: Model Refactoring

Duration
5 min

Object model before model refactoring:

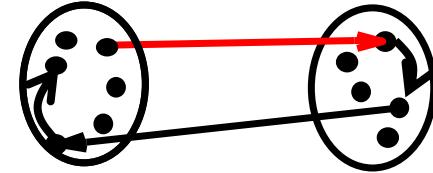


- **Task:** Move all common attributes and methods into a new abstract super class **Car**
- Create a new UML diagram (on paper or using your preferred tool) with your solution
- Take a photo or export your diagram and upload a PNG, JPG or PDF to Moodle: <https://www.moodle.tum.de/mod/assign/view.php?id=755880>

Overview of Today's Lecture

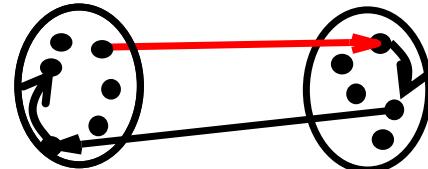
- Model-Driven Engineering
- Model Transformation: Model Refactoring
- Forward Engineering: Mapping Models to Code
-  Mapping Models to Classes
 - Mapping Models to State Charts
 - Mapping Models to Contracts
 - Mapping Models to Tables
- Source Code Transformation: Code Refactoring
 - Object Transformation: Mapping Objects to JSON
- Reverse Engineering: Mapping Source Code to Models.

Forward Engineering: Model to Java



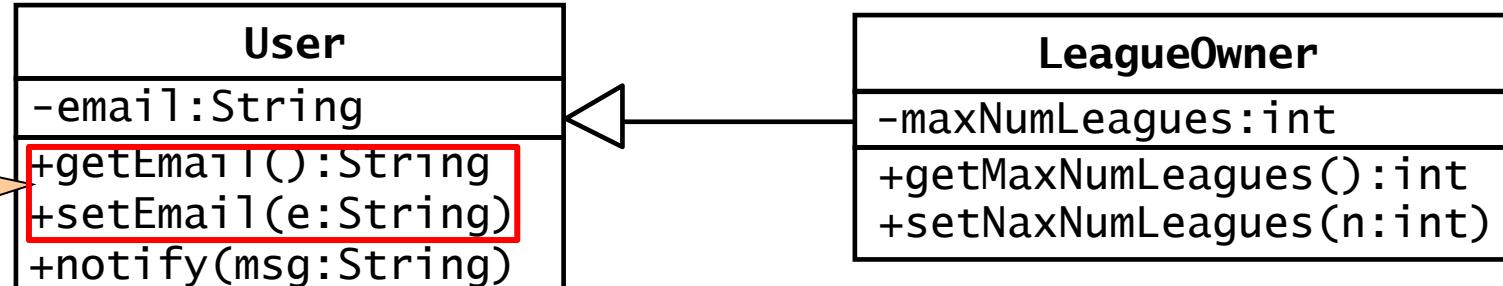
- Every UML Class should be split into a Java interface and an implementation class
- Attributes should be non-public instance variables
 - Generate getters and setters with appropriate visibility
- Signatures of UML operations are mapped straightforward to signatures of Java methods
- UML Inheritance can be mapped to
 - Inheritance in Java with super classes (**extends**)
 - Subtyping without inheritance in Java with interfaces (**implements**)

Mapping Inheritance



Object design model before transformation:

Note that getters and setters are typically not shown in UML classes

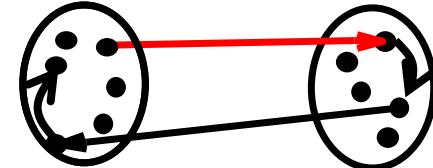


Source code after transformation:

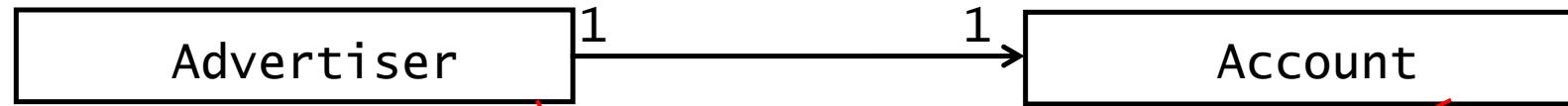
```
public class User {
    private String email;
    public String getEmail() {
        return email;
    }
    public void setEmail(String value) {
        email = value;
    }
    public void notify(String msg) {
        // ....
    }
}
```

```
public class LeagueOwner extends User {
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
        return maxNumLeagues;
    }
    public void setMaxNumLeagues (int value) {
        maxNumLeagues = value;
    }
}
```

Mapping Unidirectional 1-to-1 Associations



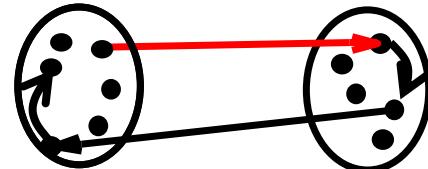
Object design model before transformation:



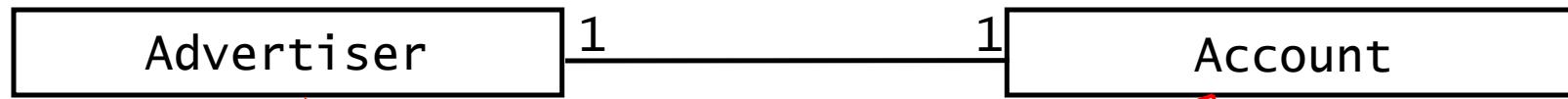
Source code after transformation:

```
public class Advertiser {  
    private Account account;  
    public Advertiser() {  
        account = new Account();  
    }  
    public Account getAccount() {  
        return account;  
    }  
}
```

Mapping Bidirectional 1-to-1 Associations



Object design model before transformation:

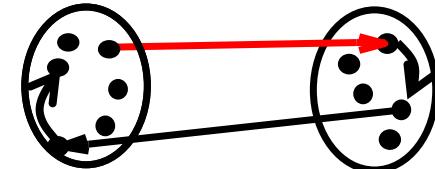


Source code after transformation:

```
public class Advertiser {  
    /* account is initialized in the  
     * constructor and never modified. */  
    private Account account;  
    public Advertiser() {  
        account = new Account(this);  
    }  
    public Account getAccount() {  
        return account;  
    }  
}
```

```
public class Account {  
    /* owner is initialize in the  
     * constructor and never modified. */  
    private Advertiser owner;  
    public Account(Advertiser owner) {  
        this.owner = owner;  
    }  
    public Advertiser getOwner() {  
        return owner;  
    }  
}
```

Mapping Bidirectional 1-to-Many Associations



Object design model before transformation:

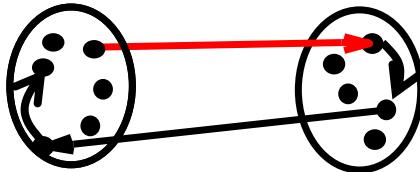


Source code after transformation:

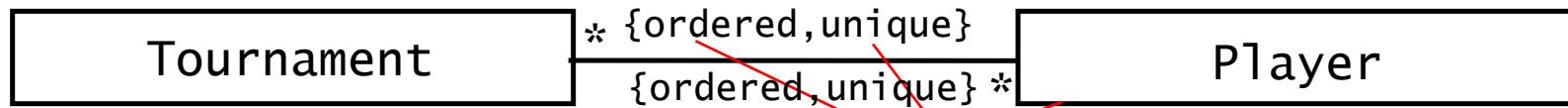
```
public class Advertiser {  
    private Set accounts;  
    public Advertiser() {  
        accounts = new HashSet();  
    }  
    public void addAccount(Account a) {  
        accounts.add(a);  
        a.setOwner(this);  
    }  
    public void removeAccount(Account a) {  
        accounts.remove(a);  
        a.setOwner(null);  
    }  
}
```

```
public class Account {  
    private Advertiser owner;  
    public void setOwner(Advertiser newOwner) {  
        if (owner != newOwner) {  
            Advertiser old = owner;  
            owner = newOwner;  
            if (newOwner != null) {  
                newOwner.addAccount(this);  
            }  
            if (oldOwner != null) {  
                old.removeAccount(this);  
            }  
        }  
    }  
}
```

Mapping Bidirectional Many-to-Many Associations



Object design model before transformation:



Source code after transformation:

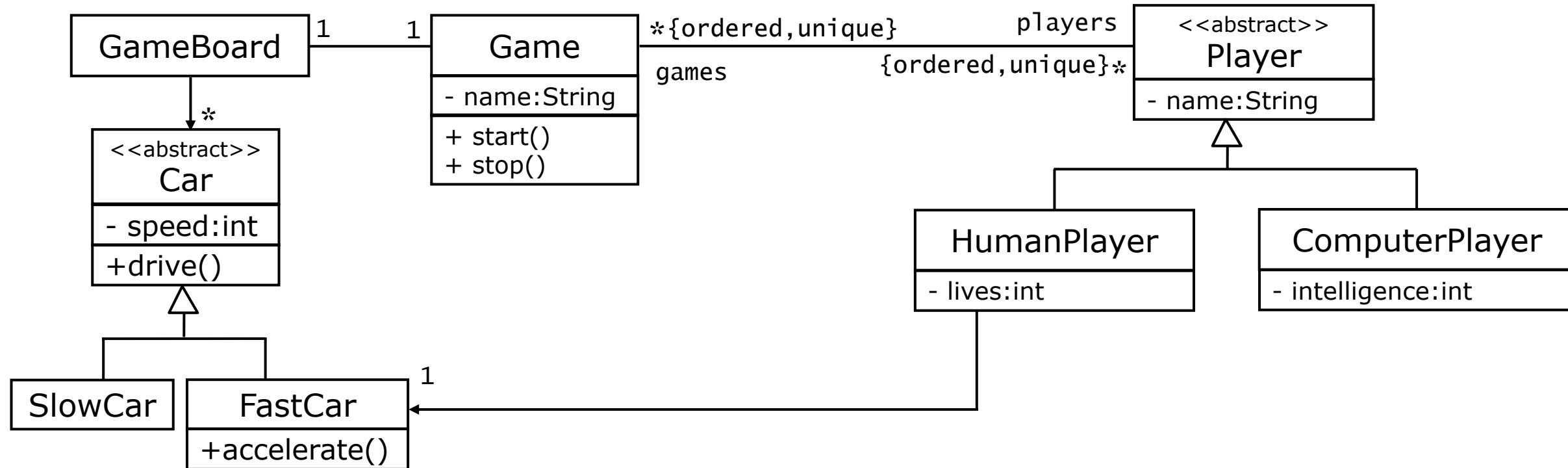
```
public class Tournament {  
    private List players;  
    public Tournament() {  
        players = new ArrayList();  
    }  
    public void addPlayer(Player p) {  
        if (!players.contains(p)) {  
            players.add(p);  
            p.addTournament(this);  
        }  
    }  
}
```

```
public class Player {  
    private List tournaments;  
    public Player() {  
        tournaments = new ArrayList();  
    }  
    public void addTournament(Tournament t) {  
        if (!tournaments.contains(t)) {  
            tournaments.add(t);  
            t.addPlayer(this);  
        }  
    }  
}
```

In-Class Exercise 02: Mapping UML to Java

Duration
10 min

Object design model before transformation:

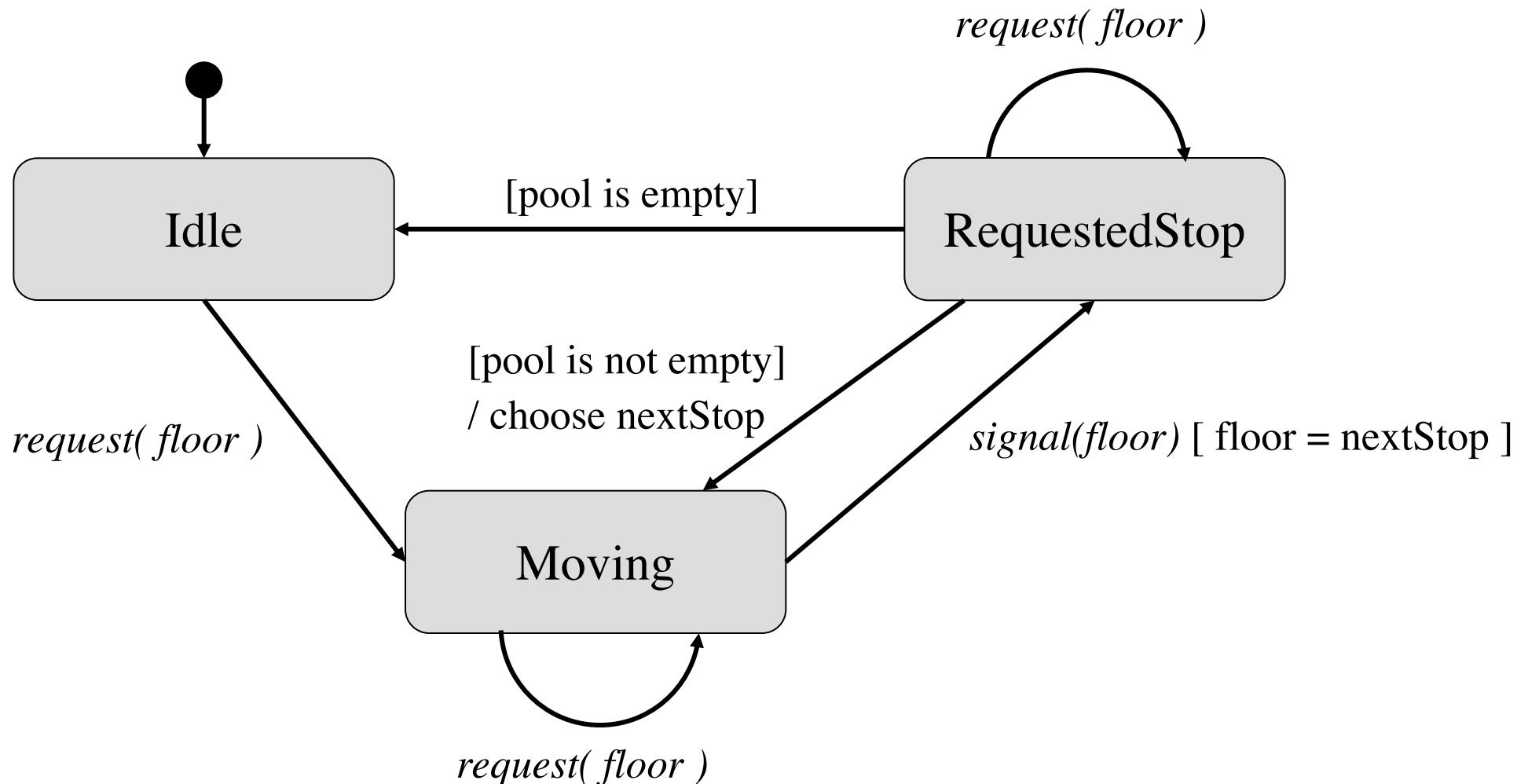


- **Task:** Map the classes **Game**, **Player** and **HumanPlayer** into Java Code
- Start the exercise on ArTEMiS
- Clone the repository, import the project to Eclipse, solve the tasks, commit and push the code and review the result on ArTEMiS

Overview of Today's Lecture

- Model-Driven Engineering
- Model Transformation: Model Refactoring
- Forward Engineering: Mapping Models to Code
 - Mapping Models to Classes
-  Mapping Models to State Charts
 - Mapping Models to Contracts
 - Mapping Models to Tables
- Source Code Transformation: Code Refactoring
 - Object Transformation: Mapping Objects to JSON
- Reverse Engineering: Mapping Source Code to Models.

UML State Diagram for an Elevator

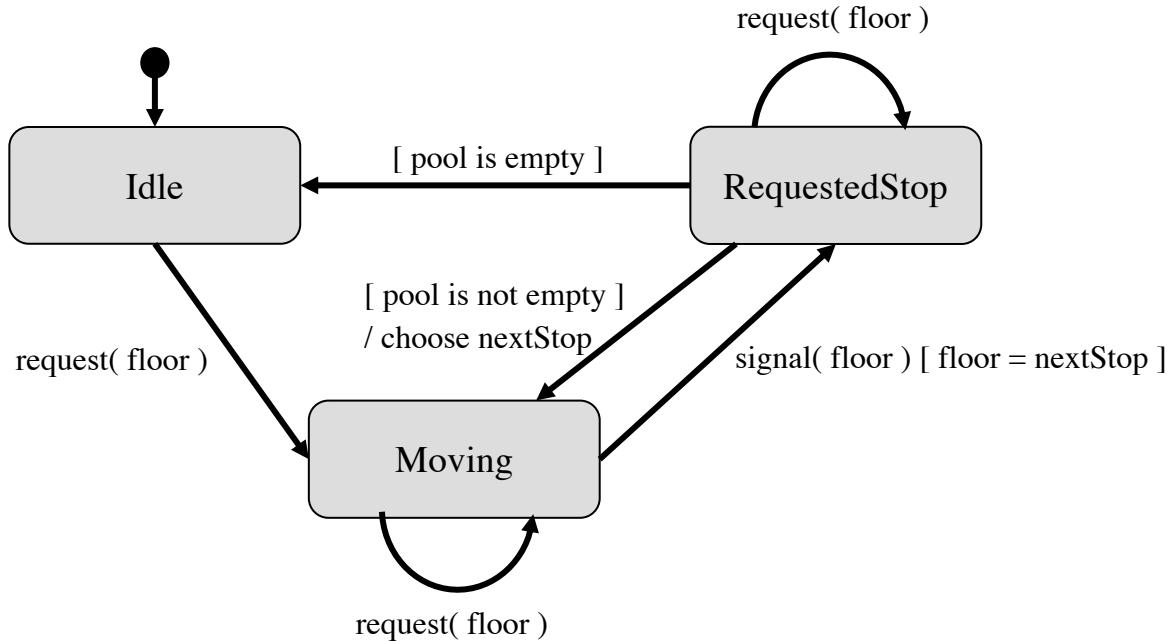


Mapping UML State Diagrams to Java Code

Basic Idea:

- Write a **public method** for each event
- Define a variable state
- Write a **switch statement** with cases for each of the states in the UML state diagram
- Write an **if statement** for each guard predicate on the transitions to check if it has become true
- Update the state variable accordingly.

Implementation of the request Event



```
public void request( int floor ) throws ... {  
    switch( state ) {  
        case Idle: state = Moving; break;  
        case Moving: break;  
        case RequestedStop:  
            if ( pool.IsEmpty( ) ) state = Idle;  
            else {  
                nextStop = pool.Choose( );  
                state = Moving;  
            }  
            break;  
        default: throw new UnexpectedStateException();  
    }  
}
```

Implementation of the request Event (2)

```
public void request( int floor ) throws ... {  
    switch( state ) {  
        case Idle: state = Moving; break;  
        case Moving: break;  
        case RequestedStop:  
            if ( pool.IsEmpty( ) ) state = Idle;  
            else {  
                nextStop = pool.Choose( );  
                state = Moving;  
            }  
            break;  
        default: throw new UnexpectedStateException( );  
    }  
}
```

Introduce state variable for current state

Transition

Check condition of transition

Perform action

Spontaneous transition from RequestedStop

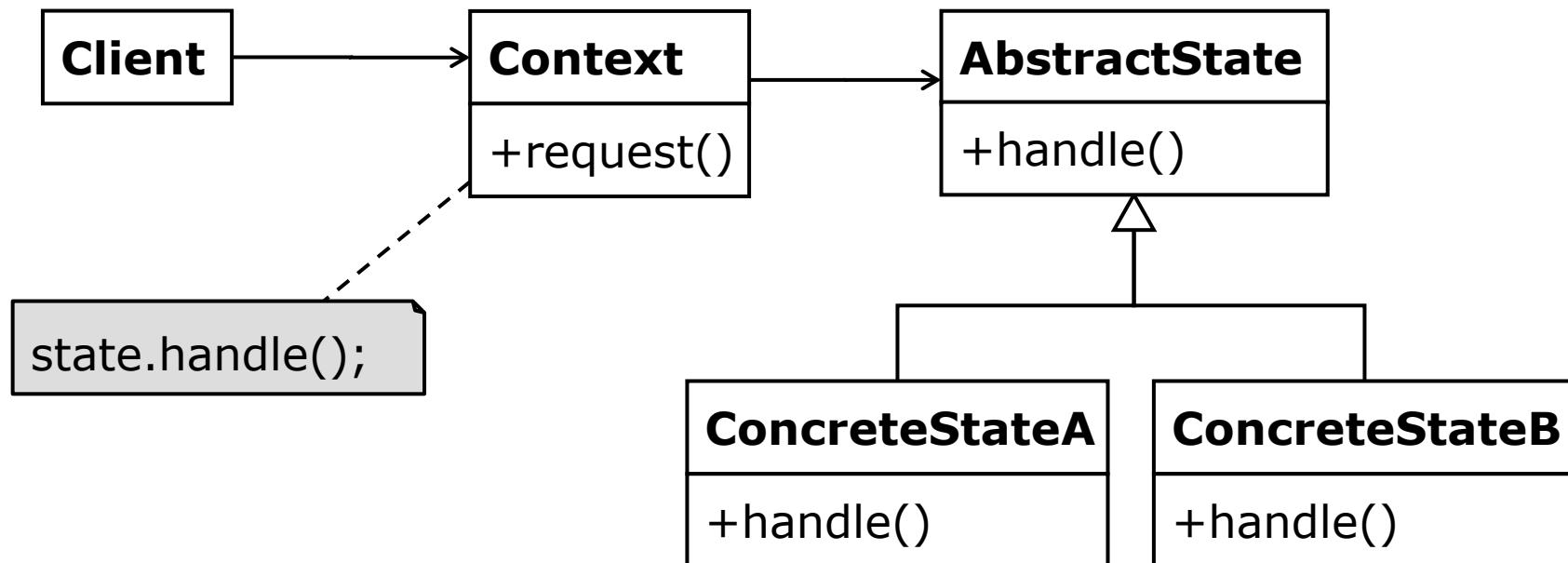
Illegal state or message

State Pattern

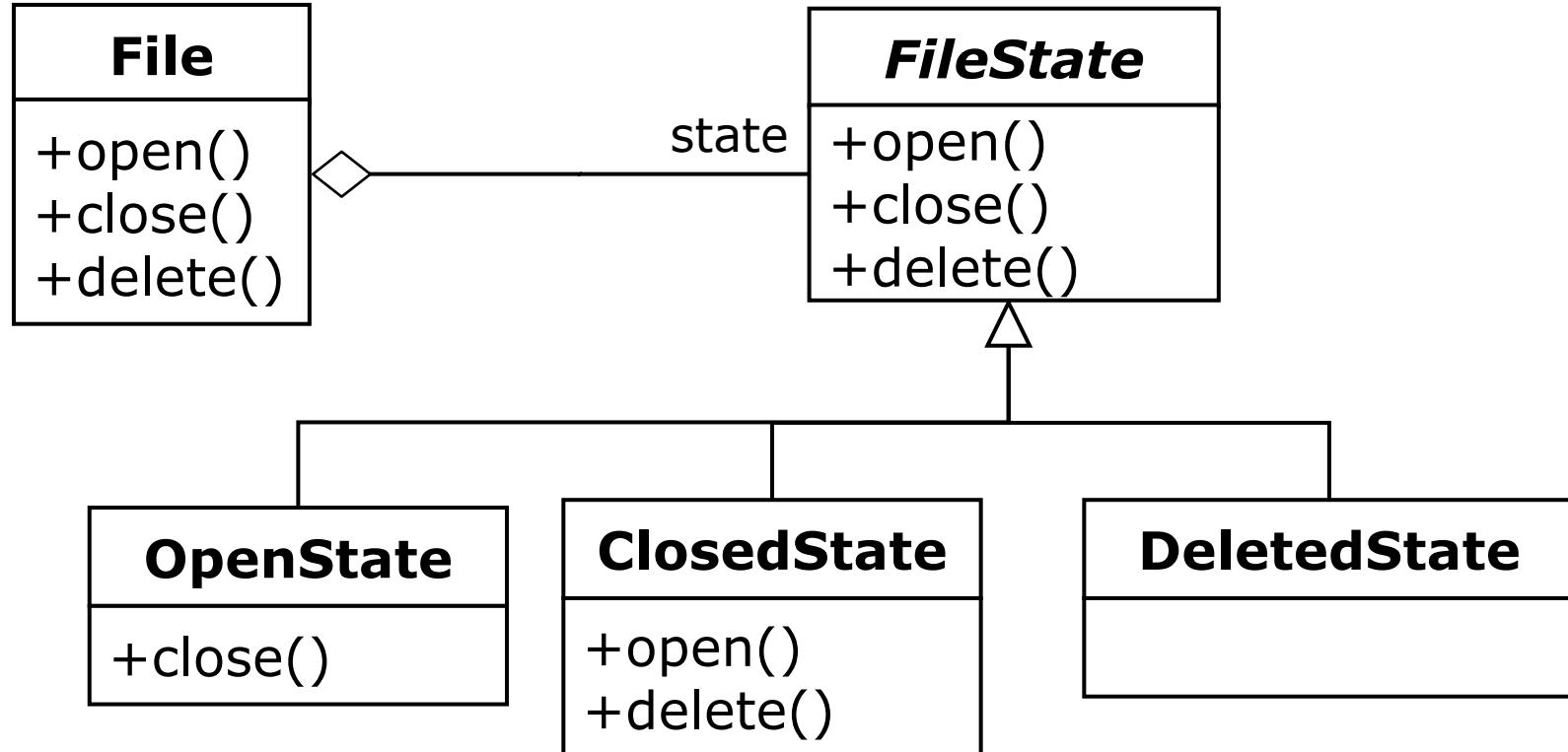
Motivation:

- You have an object that can be in several states with different behavior in each of these states
- The object behavior depends on the state

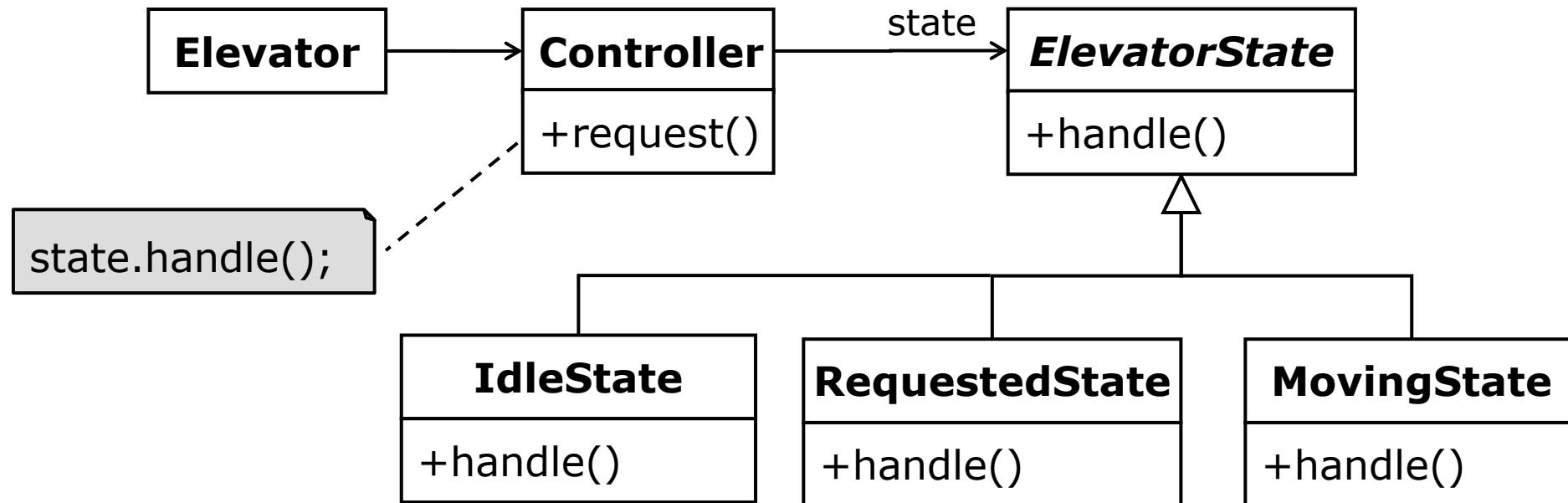
Structure:



Example: File



State Pattern applied to the Elevator: UML Class Diagram



State Pattern applied to the Elevator: Java Code

```
class IdleState extends ElevatorState {  
    public void handle(int floor, Controller controller) {  
        controller.state = new MovingState();  
    }  
}
```

switch statements
replaced by dynamic
method binding

```
class RequestedStopState extends ElevatorState {  
    public void handle(int floor, Controller controller) {  
        if (controller.pool.isEmpty()) {  
            controller.state = new IdleState();  
        }  
        else {  
            controller.nextStop = controller.pool.choose();  
            controller.state = new MovingState();  
        }  
    }  
}
```

Transition

State Pattern: Properties

- Useful for objects which change their state at runtime
- Uses polymorphism to define different behaviors for different states of an object
- The selection of the subclass depends on the state of the object
- Comparison with Bridge pattern:
 - In the bridge pattern, the selection of the subclass is done at **system initialization time**
- Comparison with Strategy pattern:
 - In the strategy pattern, the selection of the subclass **depends on an external policy**.

20 Minute Break



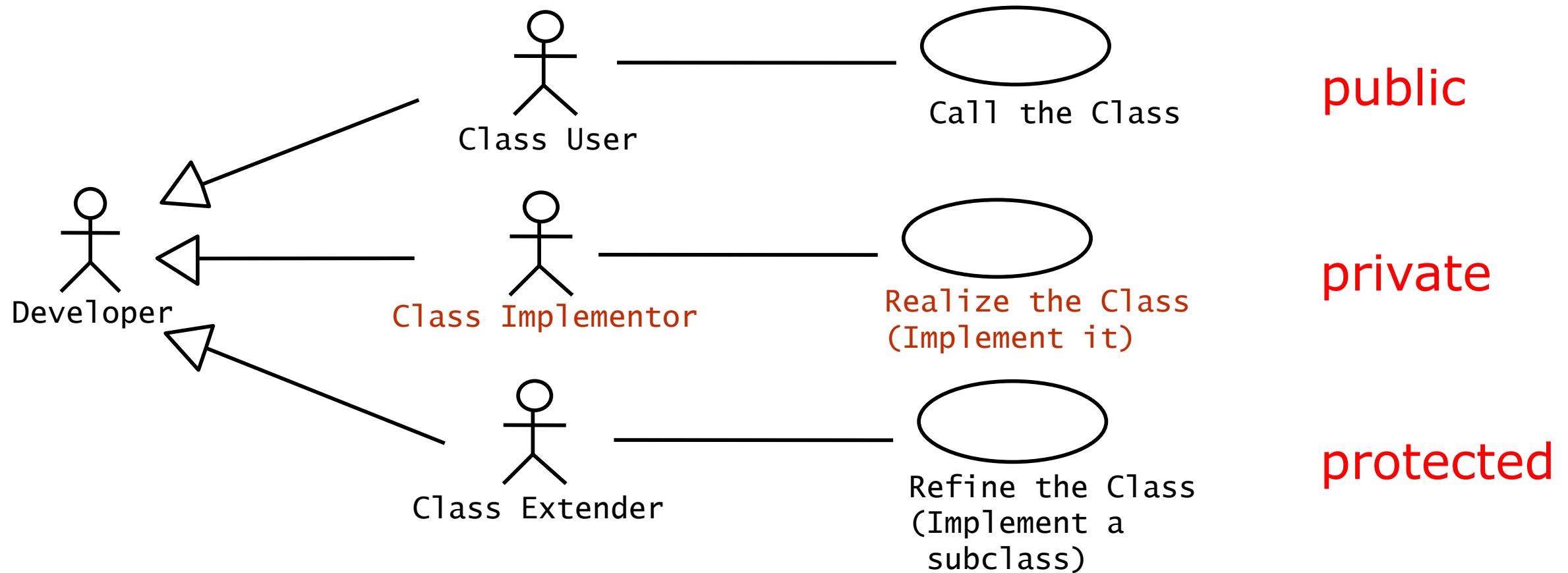
Overview of Today's Lecture

- Model-Driven Engineering
- Model Transformation: Model Refactoring
- Forward Engineering: Mapping Models to Code
 - Mapping Models to Classes
 - Mapping Models to State Charts
- Mapping Models to Contracts
 - Mapping Models to Tables
- Source Code Transformation: Code Refactoring
 - Object Transformation: Mapping Objects to JSON
- Reverse Engineering: Mapping Source Code to Models.

Definition Contract

- **Contracts** are constraints on a class that enable **class users**, **class implementors**, and **class extenders** to share the same assumptions about the class
- A contract specifies
 - Constraints that the class user must meet before using the class
 - Constraints that are ensured by the class implementor and the class extender when used.
- Contracts include three types of constraints:
 - Precondition
 - Postcondition
 - Invariant
- There is a language called OCL (Object Constraint Language) to define contracts for UML models
 - More details about OCL in the OOSE book in Chapter 9.3 and 9.4

During Object Design: Developers play 3 different Roles



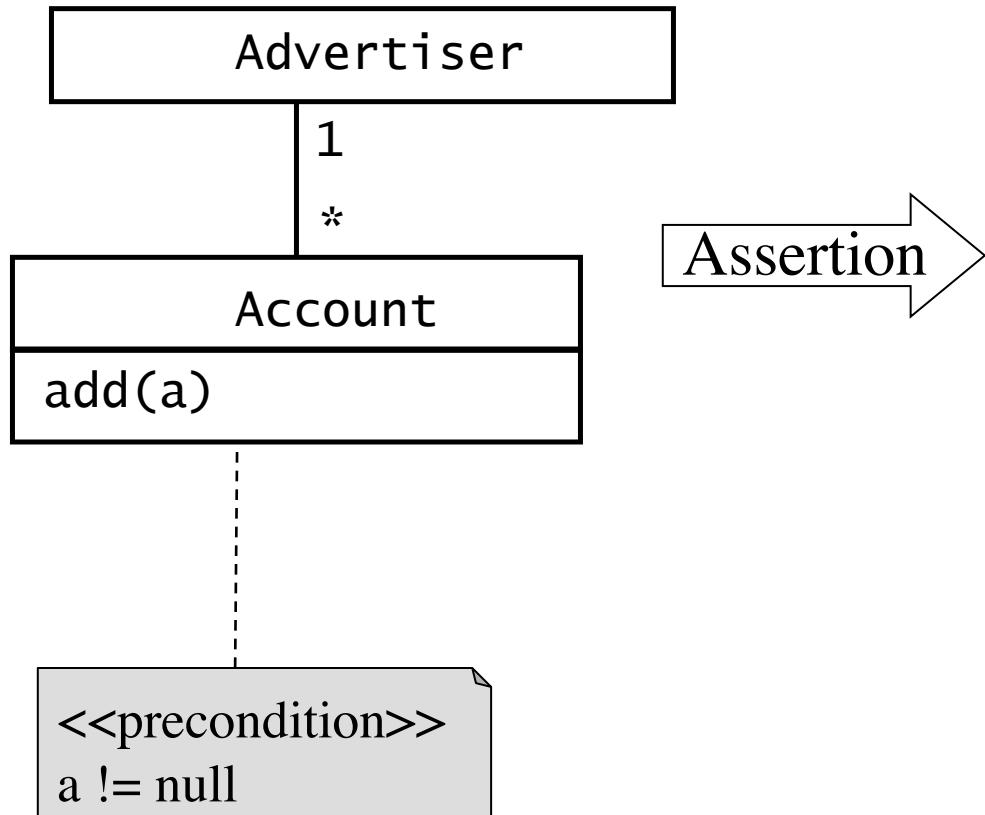
Mapping Contracts

- Many object-oriented languages do not include built-in support for contracts
 - Contracts are available in Eiffel and .NET
 - There are research compilers for JML, Spec#, etc
- Possible workaround mechanisms (“Informal design-by-contract programming”)
 - Assert statements (e.g., Java 5)
 - Exception handling

Mapping Contracts to Java

- **Check precondition:**
 - Assert the predicate before the beginning of the method
 - Or throw an exception if the precondition is false
- **Check postcondition:**
 - Assert the predicate at the end of the method
 - Or throw an exception if the predicate evaluates to false
 - If more than one postcondition is not satisfied, throw an exception only for the first violation
- **Check invariant:**
 - Check invariants at the same time as postconditions
 - Of course this is only an approximation
- Deal with Inheritance:
 - Encapsulate the checking code into methods that can be called from subclasses.

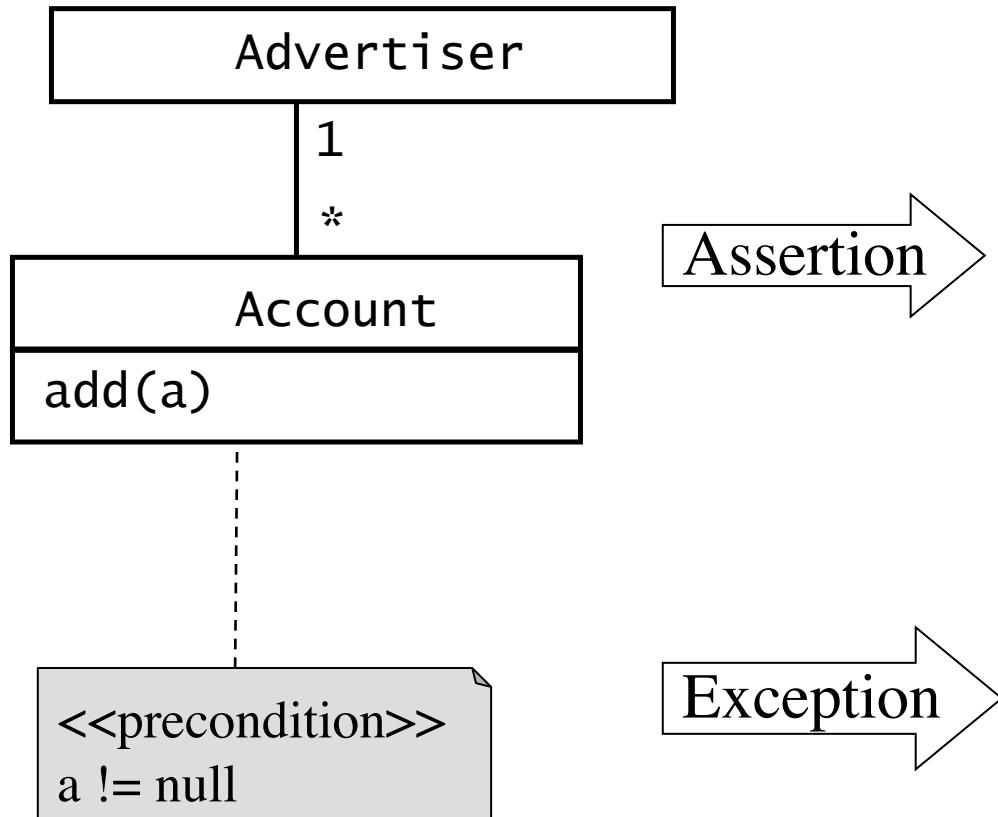
Example: Mapping a Precondition to an Assertion



```
void add(Account a) {  
    assertTrue (a != null);  
    accounts.add(a);  
    a.setOwner(this);  
}
```

Note is written in OCL

Example: Mapping a Precondition to an Exception



```
void add(Account a) {  
    assertTrue (a != null);  
    accounts.add(a);  
    a.setOwner(this);  
}
```

```
void add(Account a) {  
    if (a == null) {  
        throw new IllegalArgumentException();  
    }  
    accounts.add(a);  
    a.setOwner(this);  
}
```

Note is written in OCL

Heuristics for Implementing Contracts

Be pragmatic, if you don't have enough time, change your constraints in the following order:

1. Omit checking code for postconditions and invariants
 - Often redundant, if you are confident that the code accomplishes the functionality of the method
 - Not likely to detect many bugs unless written by a separate developer
2. Omit the checking code for private and protected methods
 - If you are the class implementor or class extender, you can trust these classes more than classes written by others.

Overview of Today's Lecture

- Model-Driven Engineering
- Model Transformation: Model Refactoring
- Forward Engineering: Mapping Models to Code
 - Mapping Models to Classes
 - Mapping Models to State Charts
 - Mapping Models to Contracts
- Mapping Models to Tables
- Source Code Transformation: Code Refactoring
 - Object Transformation: Mapping Objects to JSON
- Reverse Engineering: Mapping Source Code to Models.

Motivation: Object Relational Mapping

- Persisting data in a relational database involves writing and executing SQL statements
- SQL can be difficult to read and write, error prone and hard to debug
- To make programming easier and reduce the chance of making mistakes:
 - Many developers prefer not to execute SQL statements directly ...
 - ... But to build an object model that reflects the data structure
 - Data will be retrieved from the database and filled into the object model automatically or using a higher level query language
 - Developers can then work entirely with objects, without writing any SQL statements (e.g. using Hibernate)
- The technique to convert data between an object model and a relational database is known as **object relational mapping (ORM)**

Mapping an Object Model to a Table

- UML class diagrams can be mapped to relational databases:
 - Some degradation occurs because all UML constructs must be mapped to a single relational database construct - the **table**
- Mapping of classes, attributes and associations
 - Each **class** is mapped to a **table**
 - Each class **attribute** is mapped onto a **column** in the table
 - An **instance** of a class (an object) represents a **row** in the table
 - A *many-to-many association* is mapped into its own table
 - A *one-to-many association* is implemented as buried foreign key
- UML Operations (methods) are not mapped.

SQL Mini Primer

- SQL - Structured Query Language: domain-specific language for managing data in relational database management systems (RDBMS)
- Initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce after learning about the relational model from Ted Codd in the early 1970s.
- Typical syntax to retrieve data with a statement:
SELECT column1, column2 **FROM** table1, table2 **WHERE** condition
 - **SELECT** filters for specific columns (*class attributes*)
 - **FROM** filters for specific tables and allows to join them (*classes*)
 - **WHERE** filters for specific rows using a condition statement (*instances*)
- Statements can be nested and can become very complex
- Great interactive tutorial: Learn SQL in 20 minutes:
<https://tutorialzine.com/2016/01/learn-sql-in-20-minutes>

Types of Databases

- **Object-oriented Database**

- Information is represented in the form of objects as used in object-oriented programming.
- The database is typically integrated with the programming language.

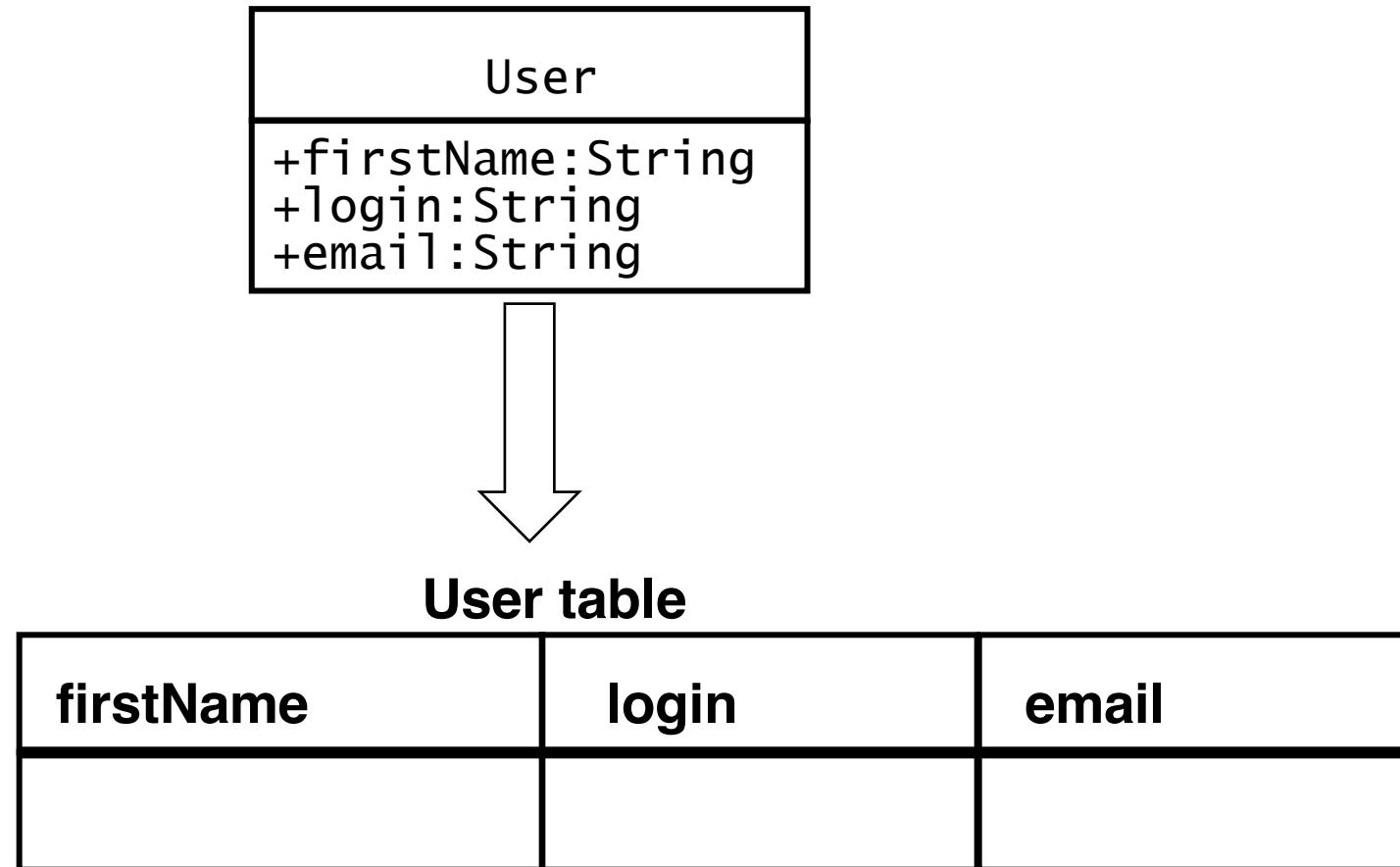
- **Relational Database:**

- Uses a structure that allows to identify and access data in relation to another piece of data in the database.
- Typically support transactions which are atomic, consistent, isolated and durable (ACID)
- **Example:** MySQL, SQLite, PostgreSQL, Oracle DB, ...

- **NoSQL database:**

- Addresses distributed systems and performance and scalability concerns by avoiding the relational model
- Most NoSQL stores lack true ACID transactions and use a key-value and/or document-based approach
- **Example:** MongoDB, CouchDB, Cassandra

Mapping a UML Class to a Table



Primary and Foreign Keys

- Any set of attributes that could be used to uniquely identify any data record in a relational table is called a **candidate key**
- The actual candidate key that is used in the application to identify the records is called the **primary key**
 - The primary key of a table is a set of attributes whose values uniquely identify the data records in the table
- A **foreign key** is an attribute (or a set of attributes) that references the primary key of another table.

Example for Primary and Foreign Keys

User table

firstName	login	email
“alice”	“am384”	“am384@mail.org”
“john”	“js289”	“john@mail.de”
“bob”	“bd”	“bobd@mail.ch”

Primary key

League table

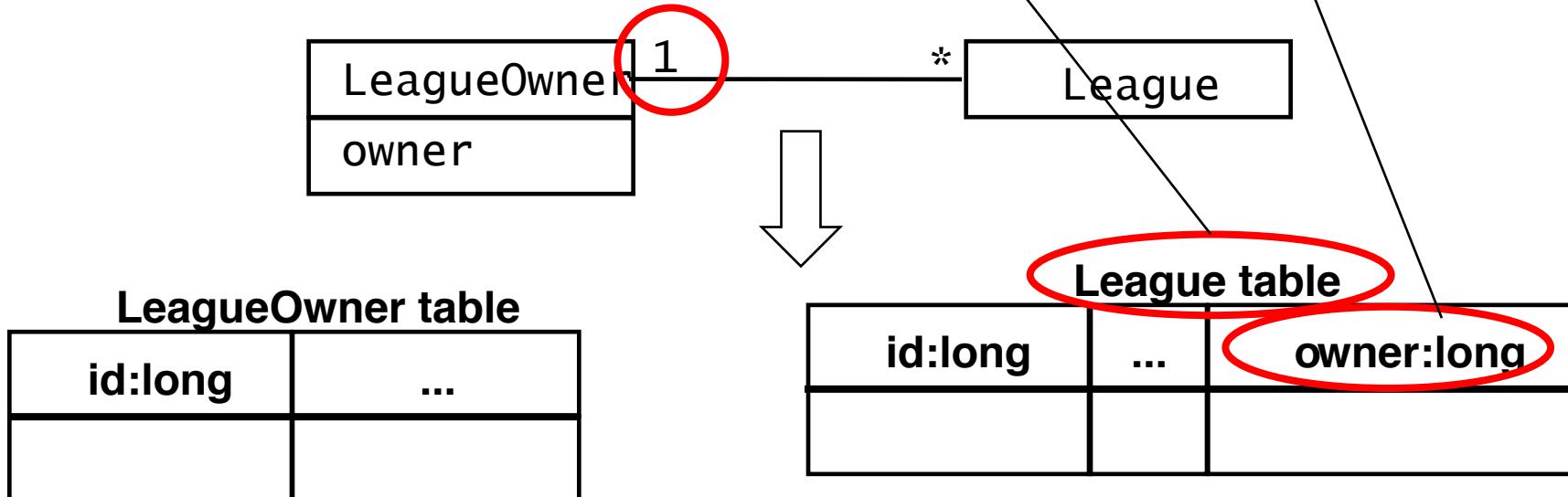
name	login
“tictactoeNovice”	“am384”
“tictactoeExpert”	“bd”
“chessNovice”	“js289”

Candidate key **Candidate key**

Foreign key referencing User table

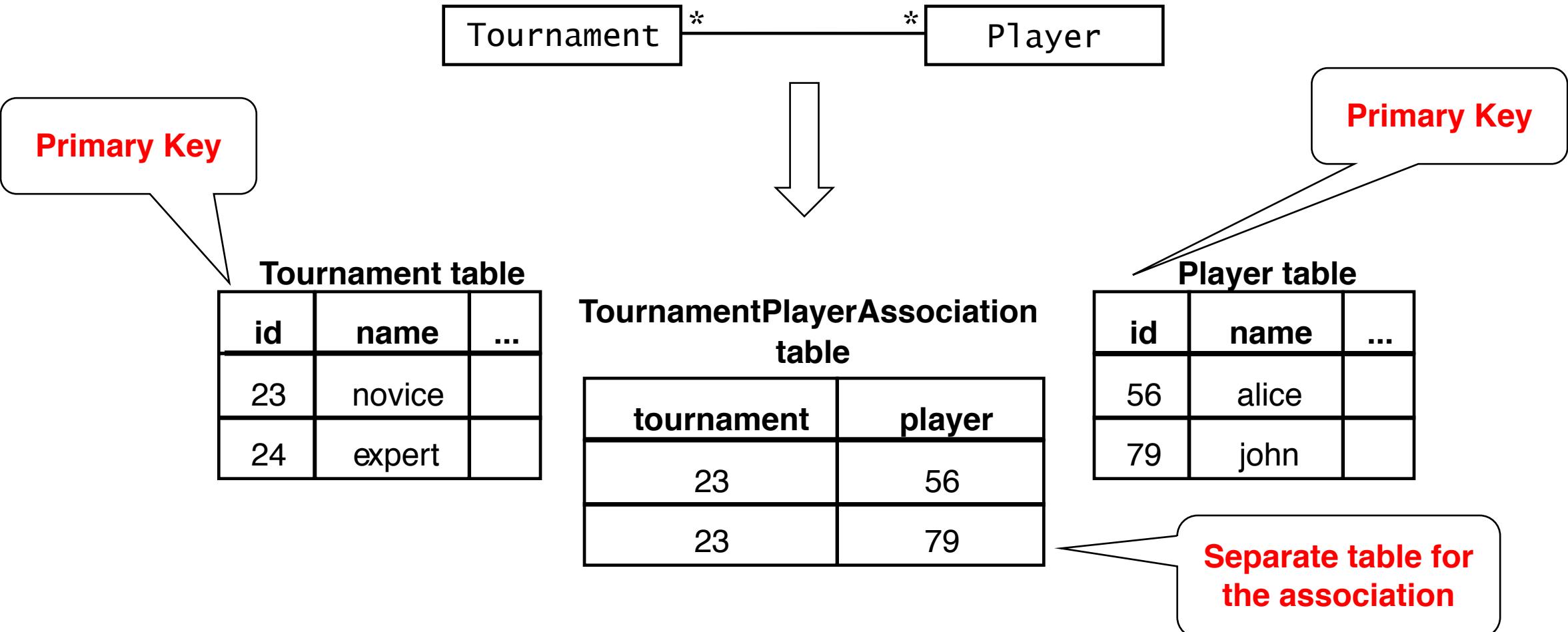
Buried Association

- Associations with multiplicity “one” can be implemented using a foreign key
- For one-to-many associations we add the foreign key to the table representing the class on the “many” end
- For all other associations we can select either class at the end of the association.



Many-to-Many Association Mapping

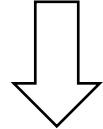
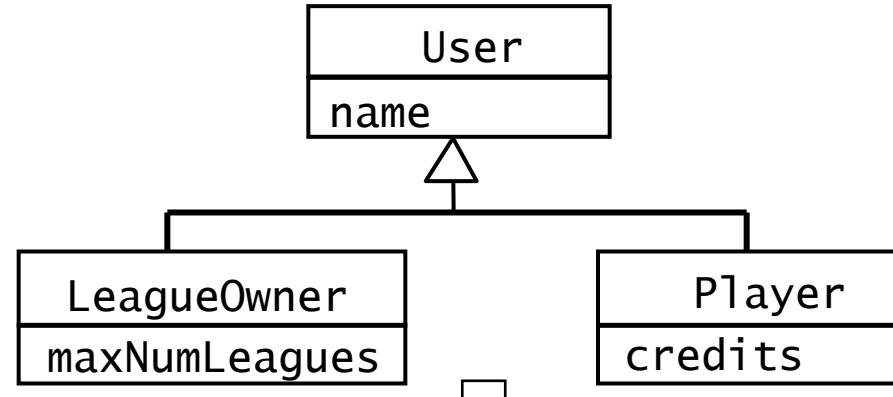
We need the Tournament/Player association as a separate table



Realizing Inheritance

- Relational databases do not support inheritance
- Two possibilities to map an inheritance association to a database schema
 - With a separate table ("vertical mapping")
 - The attributes of the superclass and the subclasses are mapped to different tables
 - By duplicating columns ("horizontal mapping")
 - There is no table for the superclass
 - Each subclass is mapped to a table containing the attributes of the subclass and the attributes of the superclass

Realizing Inheritance with Vertical Mapping



User table

id	name	...
56	zoe	
79	john	

Separate table for each class

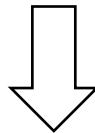
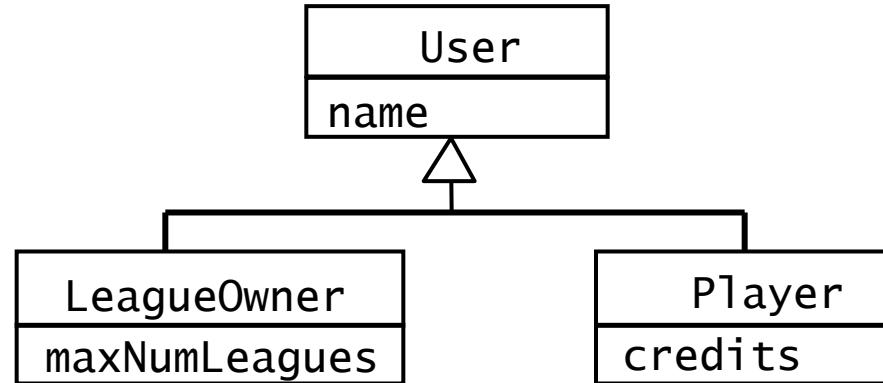
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Realizing Inheritance with the Horizontal Mapping



Only subclasses
are mapped to tables

The diagram shows two database tables. The **LeagueOwner table** has columns **id**, **name**, **maxNumLeagues**, and **...**. It contains one row with values 56, zoe, 12, and an empty cell respectively. The **Player table** has columns **id**, **name**, **credits**, and **...**. It contains one row with values 79, john, 126, and an empty cell respectively.

id	name	maxNumLeagues	...
56	zoe	12	

id	name	credits	...
79	john	126	

The column name
appears twice

Comparison: Separate Tables vs Duplicated Columns

- The trade-off is between modifiability and response time
 - How likely is a change of the super class?
 - What are the performance requirements for queries?
- Separate table mapping (Vertical mapping)
 - + We can add attributes to the super class easily by adding a column to the super class table
 - Searching for the attributes of an object requires a join operation
- Duplicated columns (Horizontal Mapping)
 - Modifying the database schema is more complex and error-prone
 - + Individual objects are not fragmented across a number of tables, resulting in faster queries.

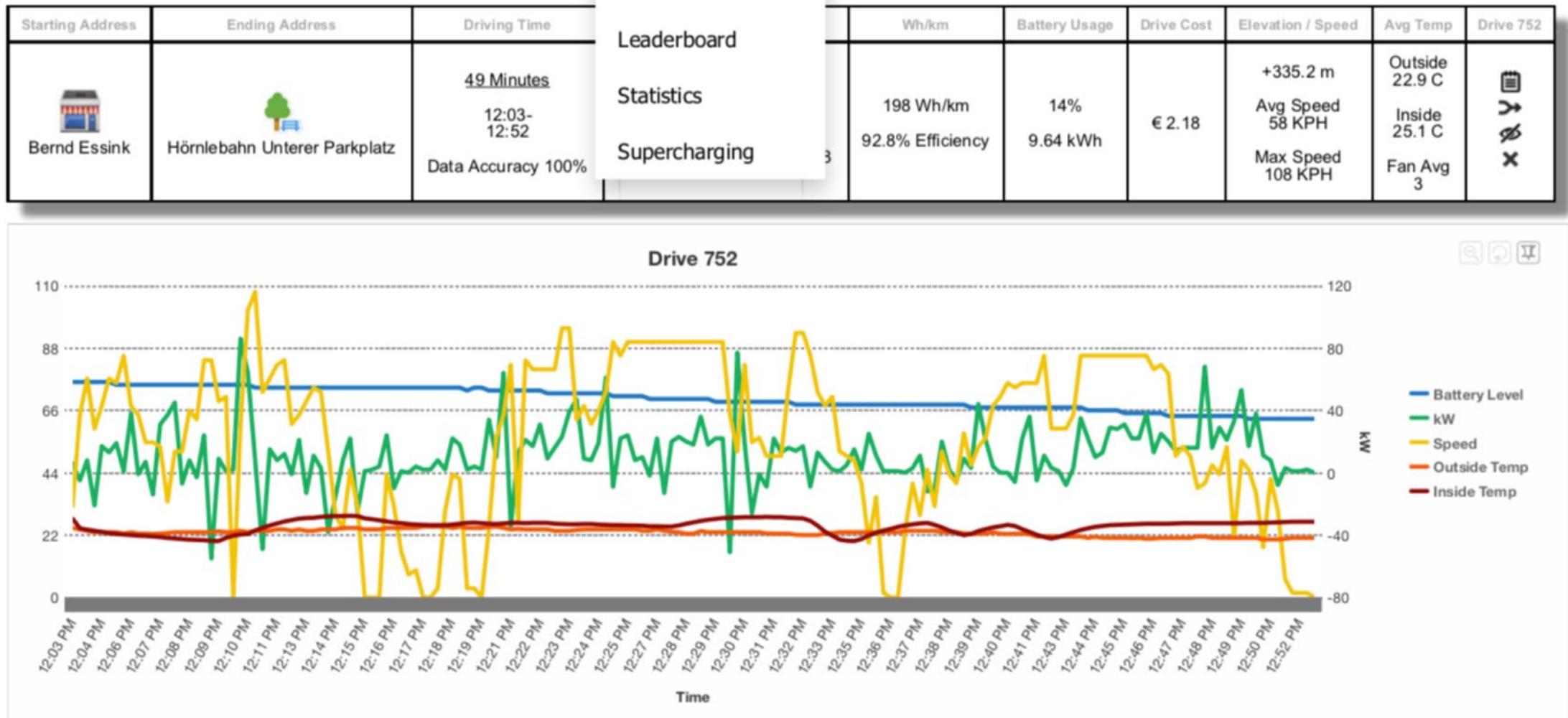
Example:



- TeslaFi provides an API for real-time tracking for Tesla car drivers
- The collected data is recorded and visualized on a website so that drivers can analyze their driving behavior and track their routes
- Prof. Brügge is a Tesla driver and likes trips into the mountains
- He recently had a trip to the mountain Hörnle



Prof. Brügge's drive to Hörnle (visualized data on TeslaFI)



Prof. Brügge's drive to Hörnle (raw data on TeslaFi)

Time	Connection State	Shift State	Speed	Latitude	Longitude	Odometer	Battery Current	Battery Level	Battery Range	Outside Temp	Inside Temp	Driver Temp Setting	Fan Status	Locked
05/12/2018 11:58	online	None	0	47.909637	11.280294	41444.056		77	359.95	None	None	15.0	0	True
05/12/2018 11:59	online		0			0			0					
05/12/2018 12:00	online		0			0			0					
05/12/2018 12:01	online		0			0			0					
05/12/2018 12:02	online		0			0			0					
05/12/2018 12:03	online	D	32.2	47.903503	11.26681	41445.517		76	356.52	24.5	27.8	15.0	8	True
05/12/2018 12:04	online	D	64.4	47.897247	11.260844	41446.36		76	354.46	24.0	24.4	15.0	8	True

Goal: Store this drive in a database and visualize the data

- **Task 1 (Demo):** Convert a Tesla log file into a database table.
 - Design the table. Each entry consists of attributes that need to be mapped to columns in the table.
 - Each entry is a row in the table
 - Import the [CSV file on Moodle](#) into a database
- **Task 2 (Demo):** Execute SQL Queries on the table
 - What was the top speed?
 - What was the average speed?
- **Task 3 (Homework):** Refactor the table
 - Reverse engineer an analysis object model (UML class diagram) from the table where you logically group related attributes into objects (e.g. car, location, temperature, battery, etc.)
- **Task 4 (Homework):** Visualization (Optional Challenge)
 - Create a small application to visualize the route with its source and destination (e.g. using Google Maps)

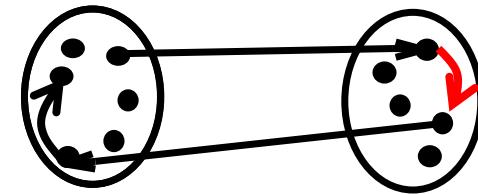
Overview of Today's Lecture

- Model-Driven Engineering
- Model Transformation: Model Refactoring
- Forward Engineering: Mapping Models to Code
 - Mapping Models to Classes
 - Mapping Models to State Charts
 - Mapping Models to Contracts
 - Mapping Models to Tables

→ Source Code Transformation: Code Refactoring

- Object Transformation: Mapping Objects to JSON
- Reverse Engineering: Mapping Source Code to Models.

Source Code Transformation: Code Refactoring



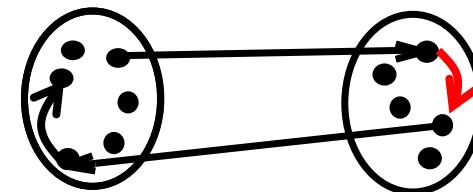
Source code before transformation:

```
public class Player {  
    private String email;  
    //...  
}  
  
public class LeagueOwner {  
    private String eMail;  
    //...  
}  
  
public class Advertiser {  
    private String email_address;  
    //...  
}
```

Source code after transformation:

```
public class User {  
    private String email;  
}  
  
public class Player extends User {  
    //...  
}  
  
public class LeagueOwner extends User {  
    //...  
}  
  
public class Advertiser extends User {  
    //...  
}
```

Another Example: Pull Up Constructor Body



Source code before transformation:

```
public class User {  
    protected String email;  
}  
  
public class Player extends User {  
    public Player(String email) {  
        this.email = email;  
    }  
}  
  
public class LeagueOwner extends User {  
    public LeagueOwner(String email) {  
        this.email = email;  
    }  
}  
  
public class Advertiser extends User {  
    public Advertiser(String email) {  
        this.email = email;  
    }  
}
```

Source code after transformation:

```
public class User {  
    private String email;  
    public User(String email) {  
        this.email = email;  
    }  
}  
  
public class Player extends User {  
    public Player(String email) {  
        super(email);  
    }  
}  
  
public class LeagueOwner extends User {  
    public LeagueOwner(String email) {  
        super(email);  
    }  
}  
  
public class Advertiser extends User {  
    public Advertiser(String email) {  
        super(email);  
    }  
}
```

Tool Support

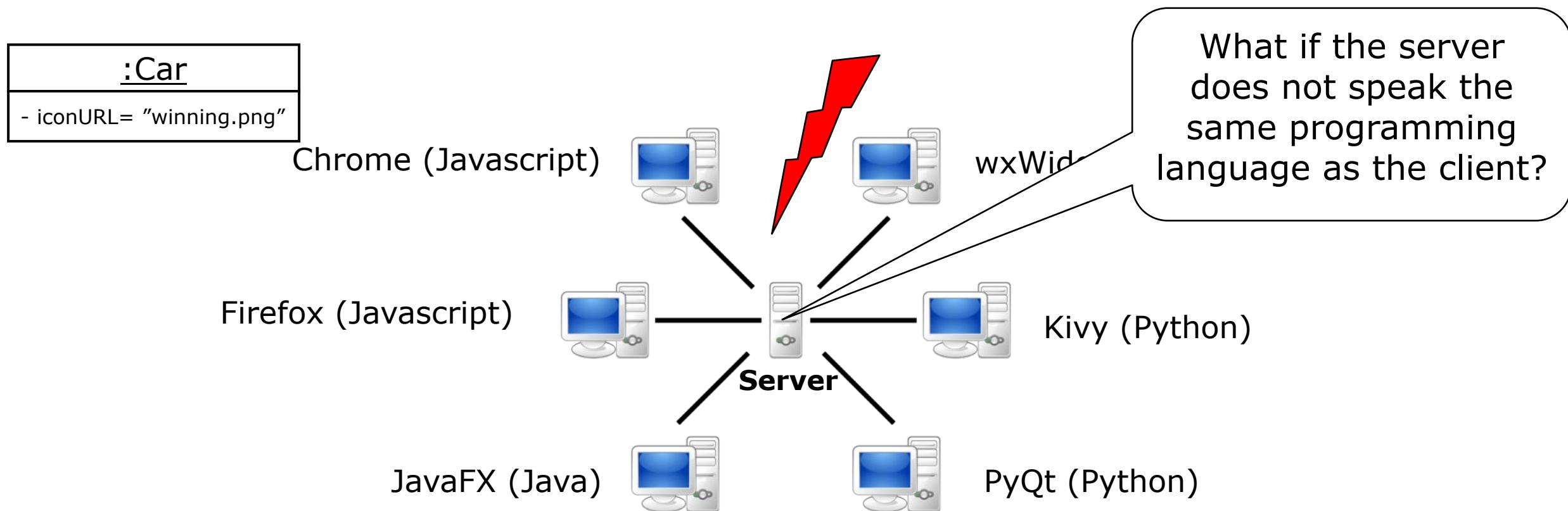
- Refactorings can be applied to models and to code
- IDEs (such as Eclipse, IntelliJ, Xcode) provide support for refactorings
- However, the support is limited
 - Only simple source code refactorings
 - Preservation of behavior is not always guaranteed
 - Sometimes, not even syntactic correctness is guaranteed

Refactoring Principles

- Why do we refactor?
 - To **improve the design** of software
 - To make model and source code **easier to understand**
- When should we refactor?
 - Refactor when you **add functionality**
 - Refactor when you need to **fix a bug**
 - Refactor as you do **code reviews**
 - Refactor when the code **starts to smell**
- What about performance?
 - Worry about performance **only when you have identified a performance problem.**

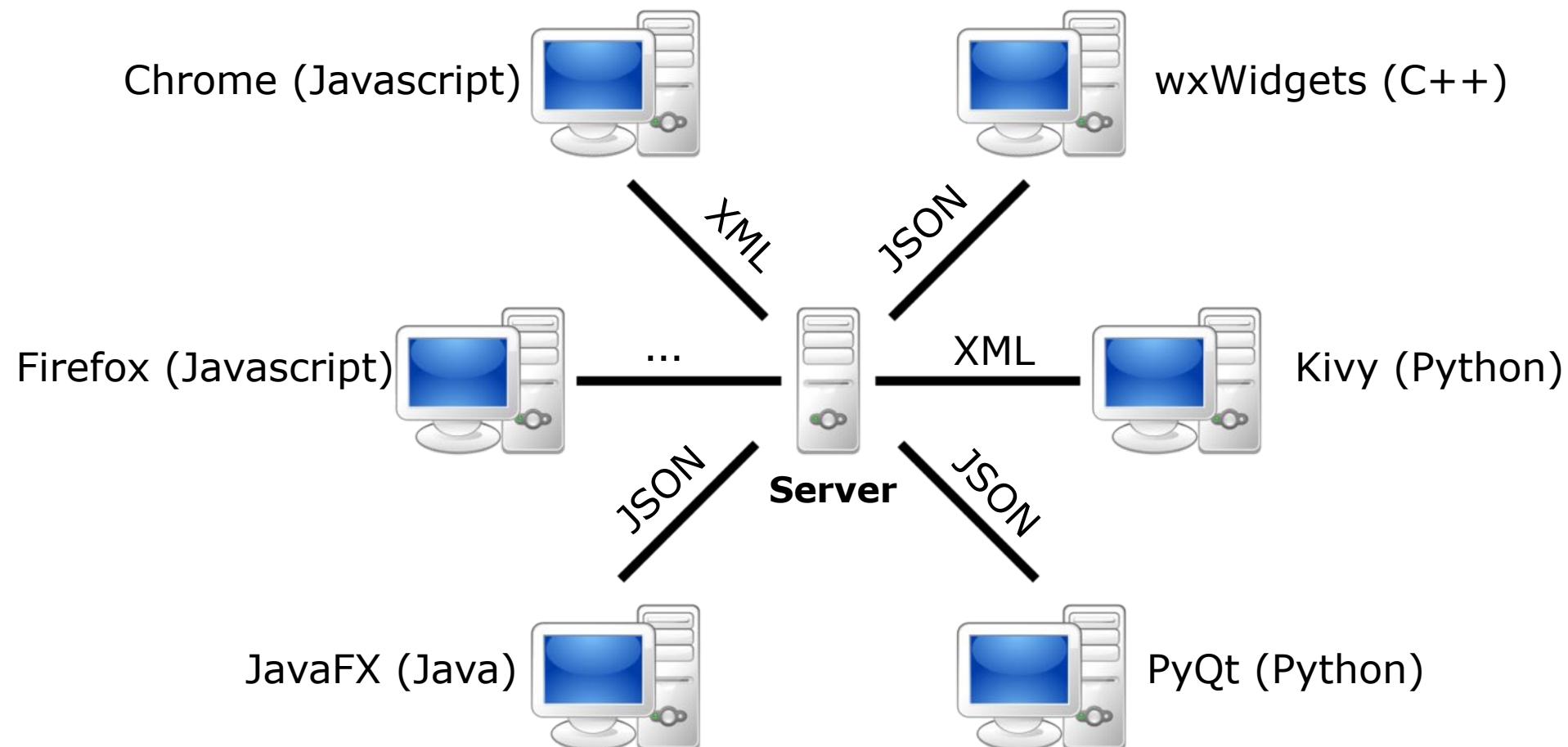
Object Transformation

Dealing with data exchange often requires transformations:



Object Transformation: Standardized data exchange formats

We need an intermediate representation of data



Example: JSON

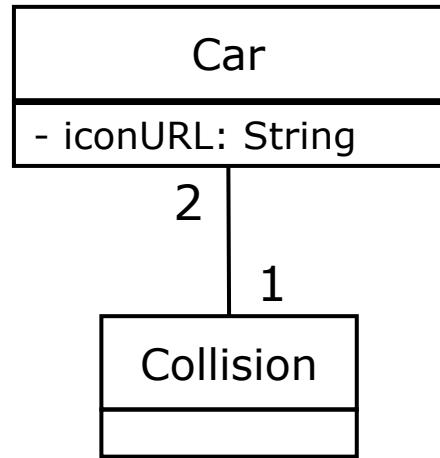
Javascript Object Notation (JSON)

- Interchangeable and standardized format that can be read by machines (e.g. web-browsers using Javascript)
- Also a human readable format

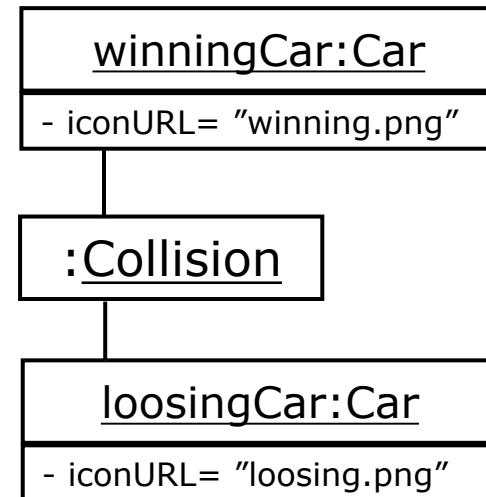
```
{  
    "Collision" : {  
        "winningCar" : {  
            "iconUrl" : "winner.png"  
        },  
        "loosingCar" : {  
            "iconUrl" : "looser.png"  
        }  
    }  
}
```

Model Transformation: Serialization / Deserialization

UML Class Diagram



UML Object Diagram



Java Classes

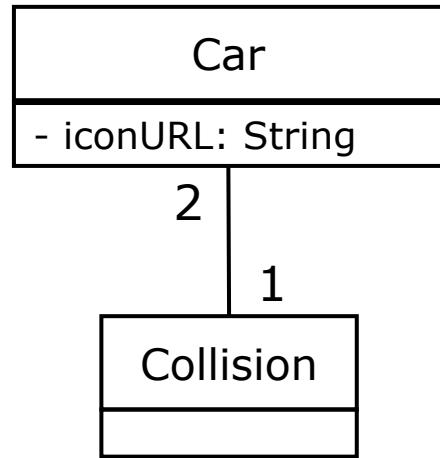
```
public class Collision {  
    private Car winningCar;  
    private Car loosingCar;  
}  
  
public class Car {  
    private String iconUrl;  
}
```

Java Objects

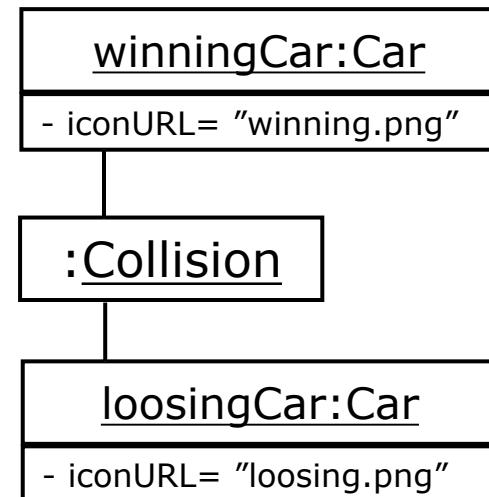
```
public static void main(String[] args) {  
    Car winningCar = new Car("winner.png");  
    Car loosingCar = new Car("loser.png");  
    Collision col = new Collision(winningCar, loosingCar);  
}
```

Model Transformation: Serialization / Deserialization

UML Class Diagram

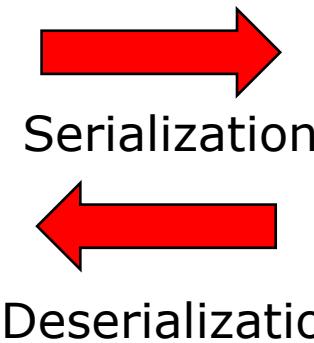


UML Object Diagram



JSON

```
{
    "Collision": {
        "winningCar": {
            "iconUrl": "winner.png"
        },
        "loosingCar": {
            "iconUrl": "looser.png"
        }
}
```



Java Classes

```
public class Collision {
    private Car winningCar;
    private Car loosingCar;
}
public class Car {
    private String iconUrl;
}
```

Java Objects

```
public static void main(String[] args) {
    Car winningCar = new Car("winner.png");
    Car loosingCar = new Car("looser.png");
    Collision col = new Collision(winningCar, loosingCar);
}
```

Demo: Serialization / Deserialization

The requirements for Bumpers have changed:

- We need to store our car collisions on a server.
- To put new data onto the Server the data needs to be transformed to JSON Entities to allow the data exchange
- To send the corresponding objects to the server, we need to serialize them
- This includes the serialization of the Collision object with the two involved car objects.
- We send these objects to the server so that the server can calculate the high-score
- You can download the demo project from Moodle:
<https://www.moodle.tum.de/mod/resource/view.php?id=755911>

Source Code for Car and Collision

```
public class Car {  
    private String iconUrl;  
  
    public Car(String iconUrl) {  
        this.iconUrl = iconUrl;  
    }  
  
    public String getIconUrl() {  
        return iconUrl;  
    }  
  
    public void setIconUrl(String iconUrl) {  
        this.iconUrl = iconUrl;  
    }  
}  
  
public class Collision {  
    private Car winningCar;  
    private Car loosingCar;  
  
    public Collision(Car winningCar, Car loosingCar) {  
        this.winningCar = winningCar;  
        this.loosingCar = loosingCar;  
    }  
  
    public Car getWinningCar() { return winningCar; }  
  
    public void setWinningCar(Car winningCar) {  
        this.winningCar = winningCar;  
    }  
  
    public Car getLoosingCar() { return loosingCar; }  
  
    public void setLoosingCar(Car loosingCar) {  
        this.loosingCar = loosingCar;  
    }  
}
```

Source Code for Serialization

```
public static void main(String[] args) throws JsonGenerationException, JsonMappingException, IOException {  
  
    Car winnerCar = new Car("winner.png");  
    Car looserCar = new Car("looser.png");  
    Collision collision = new Collision(winnerCar, looserCar);  
  
    String json = serializeObject(collision);  
    System.out.println(json);  
}  
  
private static <T> String serializeObject(T object) throws IOException, JsonGenerationException, JsonMappingException  
{  
    ObjectMapper mapper = new ObjectMapper();  
    mapper.setVisibility(PropertyAccessor.FIELD, Visibility.ANY);  
    mapper.enable(SerializationFeature.INDENT_OUTPUT);  
    mapper.enable(SerializationFeature.WRAP_ROOT_VALUE);  
    File file = new File("./" + object.getClass().getSimpleName() + ".json");  
    if (!file.exists()) {  
        file.createNewFile();  
    }  
    mapper.writeValue(file, object);  
    return mapper.writeValueAsString(object);  
}
```

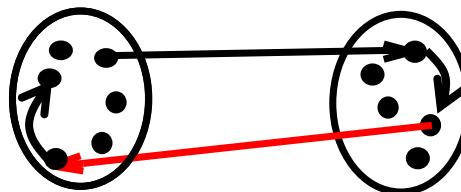
Source Code for Deserialization

```
public static void main(String[] args) throws JsonParseException, JsonMappingException, IOException {  
  
    Collision collision = deserializeObject(Collision.class);  
    System.out.println(collision);  
}  
  
  
private static <T> T deserializeObject(Class<T> type) throws JsonParseException, JsonMappingException, IOException {  
    ObjectMapper mapper = new ObjectMapper();  
    File file = new File("./" + type.getSimpleName() + ".json");  
    mapper.enable(DeserializationFeature.UNWRAP_ROOT_VALUE);  
    T object = (T) mapper.readValue(file, type);  
    return object;  
}
```

Overview of Today's Lecture

- Model-Driven Engineering
 - Model Transformation: Model Refactoring
 - Forward Engineering: Mapping Models to Code
 - Mapping Models to Classes
 - Mapping Models to State Charts
 - Mapping Models to Contracts
 - Mapping Models to Tables
 - Source Code Transformation: Code Refactoring
 - Object Transformation: Mapping Objects to JSON
- Reverse Engineering: Mapping Source Code to Models.

Reverse Engineering: Mapping Source Code to Models

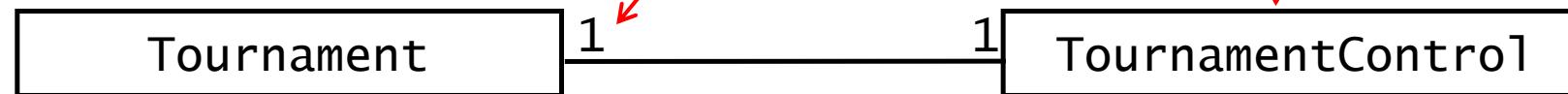


Source code before transformation:

```
public class Tournament {  
    private TournamentControl control;  
    public Tournament() {  
        control = new TournamentControl(this);  
    }  
    public TournamentControl getControl() {  
        return control;  
    }  
}
```

```
public class TournamentControl {  
    private Tournament tournament;  
    public TournamentControl(Tournament tournament) {  
        this.tournament = tournament;  
    }  
    public Tournament getTournament() {  
        return tournament;  
    }  
}
```

Object design model after transformation:



In-Class Exercise 03: Mapping Java to UML



Source code before transformation:

```
public class Earth {  
    Ocean ocean;  
    List<Mountain> mountains;  
  
    public Earth() {  
        this.ocean = new Ocean("Atlantic", 82440000);  
        this.mountains = new ArrayList<Mountain>();  
        addMountain();  
    }  
  
    public void addMountain() {  
        Mountain m = new Mountain("Zugspitze", 2962);  
        this.mountains.add(m);  
    }  
}
```

```
public class Ocean {  
    String name;  
    double size;  
  
    public Ocean(String name, double size) {  
        this.name = name;  
        this.size = size;  
    }  
}  
  
public class Mountain {  
    String name;  
    double height;  
  
    public Mountain(String name, double height) {  
        this.name = name;  
        this.height = height;  
    }  
}
```

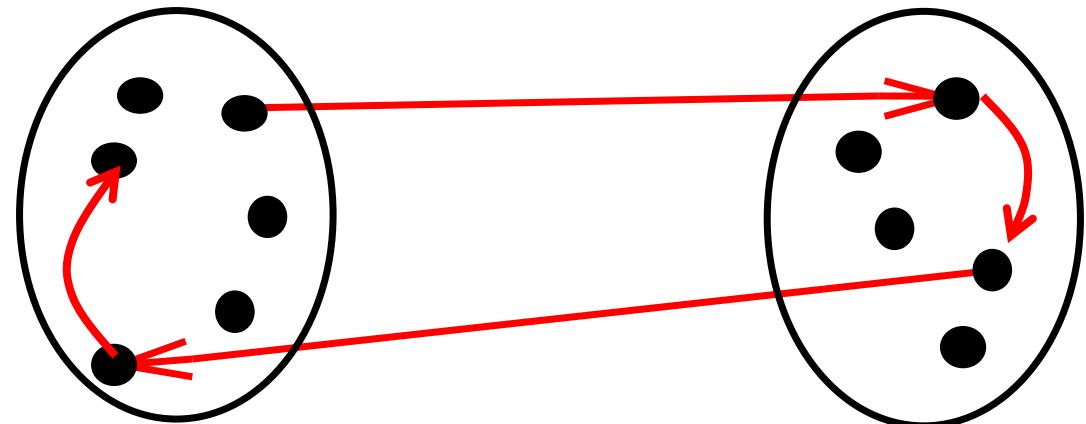
- **Task:** Given the above Java Code, reverse engineer the corresponding UML class diagram
- Create a new UML diagram (on paper or using your preferred tool) with your solution
- Take a photo or export your diagram and upload a PNG, JPG or PDF to Moodle:
<https://www.moodle.tum.de/mod/assign/view.php?id=755882>

When will MDE be Reality in the Profession?

- Tools that cover a single aspect of the software lifecycle: Market is full of them
- Lifecycle tools for MDE: Not very soon 😞
- Metrics need to be defined that measures for
 - Completeness, Consistency, Ambiguity
- Formal methods need to be included to check for
 - Verification
- Tools need to be developed with capabilities for
 - Real-time collaboration, user interface generation
- Other problem areas
 - Graphical expressiveness (end users and other stakeholders must be able to read models)
 - Gap between traditional developers and model-based developers (education!).

Summary

- Four mapping concepts:
 1. Model transformation improves the compliance of the object design model with a design goal
 2. Forward engineering improves the consistency of the code with respect to the object design model
 3. Refactoring improves code readability/modifiability
 4. Reverse engineering discovers the design from the code
- Forward engineering techniques:
 - Optimizing the class model
 - Mapping associations to collections
 - Mapping contracts to exceptions
 - Mapping class model to tables.



Readings

- D.C. Schmidt, Model-Driven Engineering
 - IEEE Computer Nr. 39 Vol. 2, 2006
- J.D. Pool, Model-Driven Architecture, 2001
 - http://www.omg.org/mda/mda_files/Model-Driven_Architecture.pdf
- J.A Estefan, Survey of MBSE Methodologies
 - INCOSE MBSE Focus group, 24 May 2007.
- *Modeling Language Portal*
 - <http://modeling-languages.com>
- Object Relational Mapping
 - https://www.visual-paradigm.com/support/documents/vpuserguide/3563/3581/85424_whatisobject.html

More Readings

- Martin Fowler and Kent Beck
 - Refactoring: Improving the Design of Existing Code, Object Technology Series, Addison-Wesley, 1999
- Kent Beck and Martin Fowler, Bad Smells in Code
 - <http://sourcemaking.com/refactoring/bad-smells-in-code>
- Code Smells
 - http://en.wikipedia.org/wiki/Code_smell
- State Pattern
 - http://en.wikipedia.org/wiki/State_pattern
- Eric Armstrong, How to implement state-dependent behavior
 - <http://www.javaworld.com/javaworld/jw-08-1997/jw-08-stated.html>
- Programming with Assertions
 - <http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html>
- Round-trip Code Engineering
 - Tutorial for over 10 programming languages
 - <http://www.visual-paradigm.com/product/vpuml/provides/roundtripcodeengine.jsp>

Pattern-Based Development

Bernd Bruegge

Chair for Applied Software Engineering
Technische Universität München

June 14, 2018



EIST

University Course Evaluation

- EIST with 1650 students is a real challenge!
- We put a lot of effort and passion to create a great learning atmosphere in EIST and to provide you with the newest concepts, tools, and workflows
- We hope you appreciate our effort ☺
- **Your feedback is valuable to us and to the university!**
- You should have received an email by the Department Student Council MPI ("Fachschaft") to evaluate EIST
- **You now have 15 minutes to fill out the anonymous online survey**
- The Lecture starts at 9:15 am

University Course Evaluation (15 min)

Search your emails by subject:
“Onlineumfrage zur Veranstaltung: ...”

Dear student,

You are registered as participant of the course

Introduction to Software Engineering

The evaluation is online. The evaluation is possible until 20.06.2018 23:59:00.

Please follow this link to the online-survey:

<https://evasys.zv.tum.de/evasys/online.php?pswd=xxx>

Click on the link and fill out the anonymous survey in your browser

Although the evaluation is done online, the lecturer should still grant you some time during the lecture or the exercise class to fill out the forms.

Please note that the evaluation is anonymous. The forms cannot be matched back to you. Transfer of the data is encrypted.

Your participation in the evaluation is very important in order to continuously improve the quality of teaching. We are very thankful for your prompt and complete participation!

Best regards

Evaluation unit of the Departmental Student Council MPI

Roadmap for Today's Lecture

- **Context and Assumptions**

- We completed Chapter 1 to 9 in the OOSE textbook by Bruegge and Dutoit

- **Content of this lecture**

→ We introduce pattern-based development

- You apply pattern-based development using UML
- You implement additional requirements for Bumpers using Java

- **Objective:** At the end of this lecture you are able to

- Understand pattern-based development
- Understand the use of patterns in models and code
- Implement the strategy pattern
- Implement the observer pattern
- Implement the adapter pattern

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)
 - System Design (Chapter 6 and 7)
 - Object Design (Chapter 8)
 - Model Transformations (Chapter 9)
- **Today: Pattern-Based Development**
- Next Lectures:
 - June 21, 2018: Lifecycle Modeling
 - June 28, 2018: Configuration-, Build- and Release Management
 - July 05, 2018: Testing
 - July 12, 2018: Project Management
 - July 19, 2018: Repetitorium

Outline of the Lecture

Goal: Appreciate the Beauty of Pattern-Based Development

- Introduction to Pattern-Based Development
- Review of the 3 finished Sprints in Bumpers
 - Updated Product Backlog
 - Analysis Object Model
 - Object Design
- 3 more sprints including in-class exercises and homework
 - **Sprint 4:** Application of the Strategy Pattern (In-Class Exercise)
 - **Sprint 5:** Application of the Observer Pattern (In-Class Exercise)
 - **Sprint 6:** Application of the Adapter Pattern (Homework)

Overview of Today's Lecture

- Introduction to Pattern-Based Development
- Review of the 3 finished Sprints in Bumpers
 - Sprint 4: Application of the Strategy Pattern
 - Sprint 5: Application of the Observer Pattern
 - Sprint 6: Application of the Adapter Pattern

Recap: Algorithm vs. Pattern

- **Algorithm**

- A method for solving a problem using a finite sequence of well-defined instructions for solving a problem
- Starting from an initial state, the algorithm proceeds through a series of successive states, eventually terminating a final state



- **Pattern**

"A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice" - Christopher Alexander, A Pattern Language.



Original Definition of a Pattern

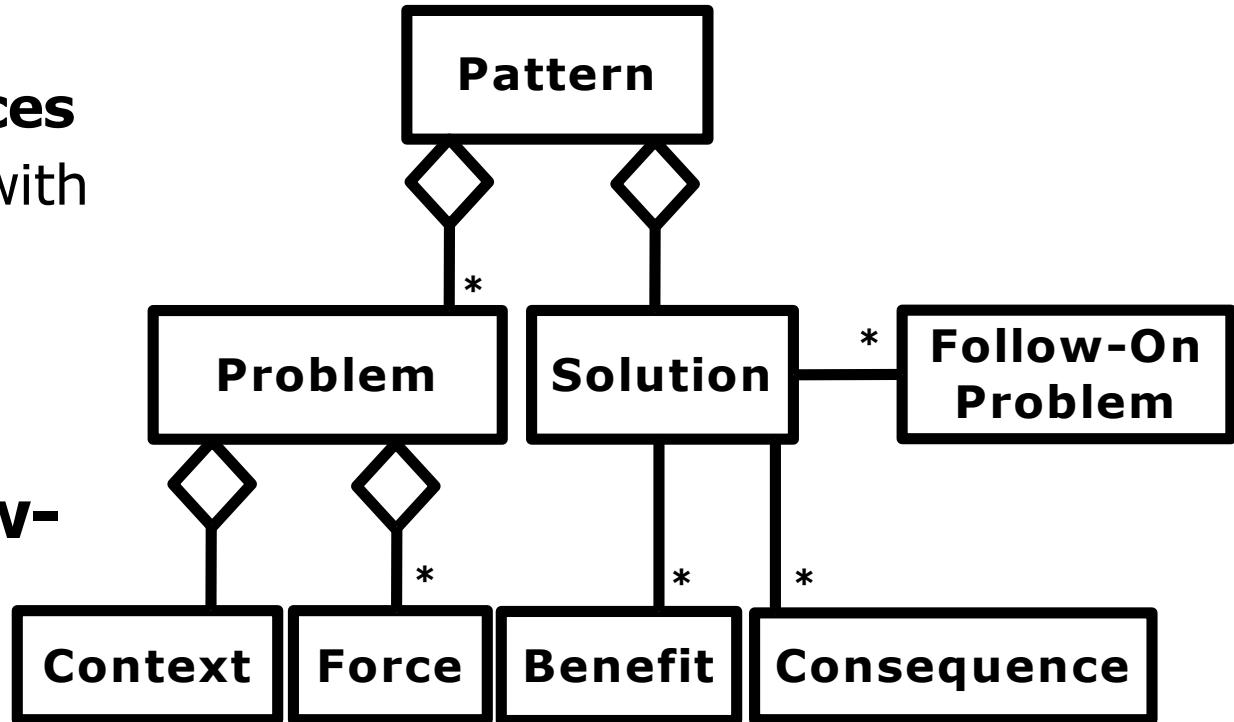


Original definition (Christopher Alexander):

“A pattern is a three-part rule, which expresses a relation between a certain **context**, a **problem**, and a **solution**. ”

Modeling a Pattern in UML

- A pattern has a **problem** and a **solution**
 - The **problem** class is elaborated in terms of a **context** and a set of **forces**
 - The **solution** resolves these forces with **benefits** and **consequences**
 - To be considered as a pattern, the solution must be applicable to more than one specific problem
- Solutions usually generate **follow-on problems**
 - Follow-on problems can again be elaborated in terms of context and forces which may lead to the applicability of other patterns



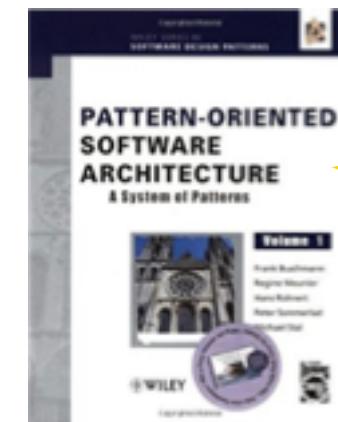
Schemata for Describing Patterns

1. Alexandrian Form (Architecture)



Erich Gamma et al.

2. **Gang of Four:** name, intent, also known as, motivation, applicability, ...



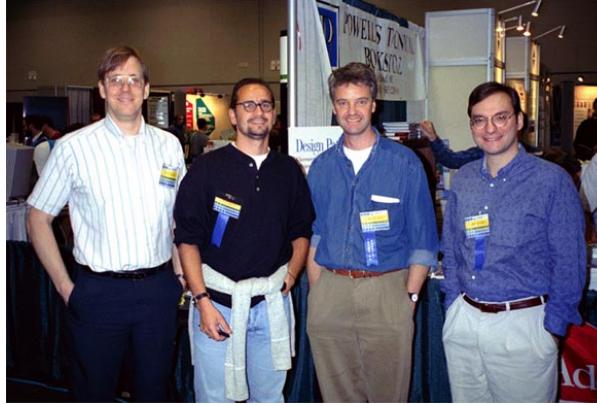
Frank Buschmann
et al.

Gang of Four Scheme

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

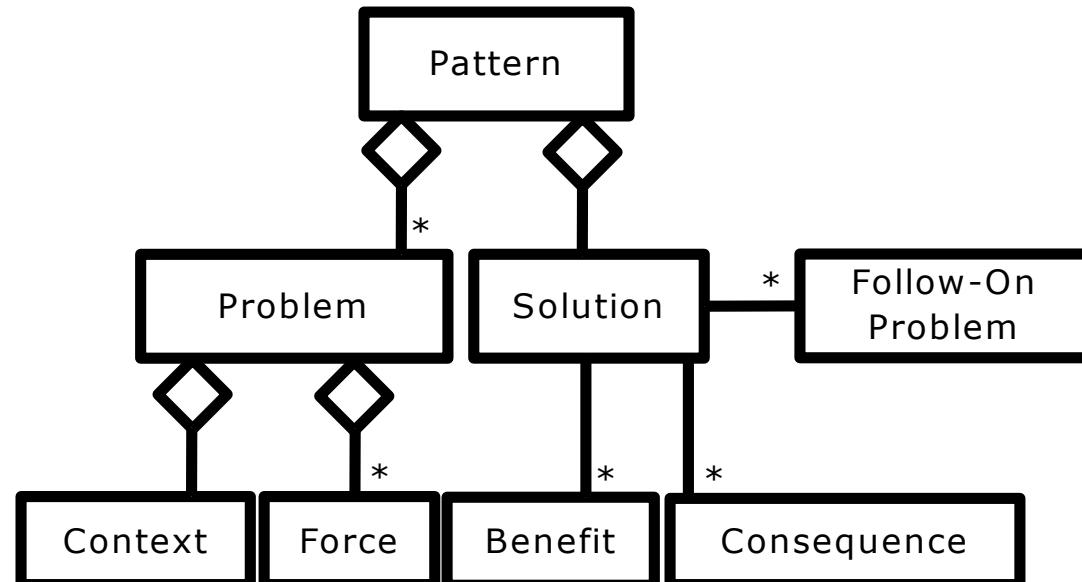
- Pattern Name & Classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

See pages 6-7
[Gamma et al. 95]



The Pattern Model used in EIST

- The **Problem** explains the actual situation in form of context and forces
 - The **Context** sets the stage where the pattern takes place
 - **Forces** describe why the problem is difficult to solve
- The **Solution** resolves these forces with benefits and consequences
 - **Benefits** describe positive outcomes of the solution
 - **Consequences** explain effects, results, and other outcomes of the application of the pattern
- **Follow-On Problems** can occur when you apply the solution



Pattern Categorization

- **Patterns for development activities**
 - Analysis
 - System Design, e.g. Architectural Styles
 - Object Design, e.g. Design Patterns
 - Testing
- **Patterns for cross-functional activities**
 - Process
 - Agile
 - Build and Release Management
- **Antipatterns**
 - Smells & Refactoring

**Curious about Patterns?
Check out the PSE Lecture
in WS1819!**



Recap: 3 Types of Design Patterns

- **Structural Patterns**

- Reduce coupling between two or more classes
- Introduce an abstract class to enable future extensions
- Encapsulate complex structures

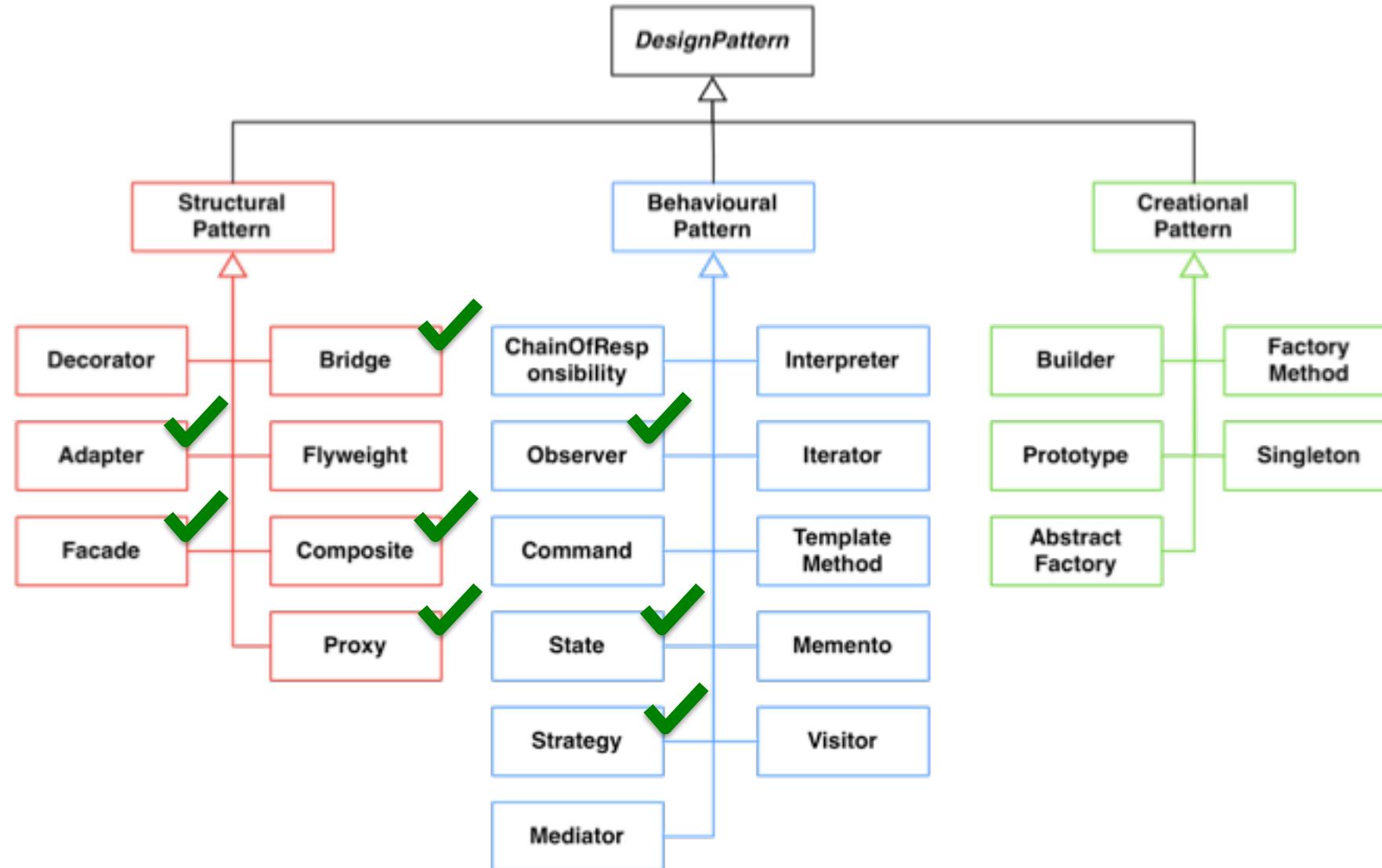
- **Behavioral Patterns**

- Allow a choice between algorithms
- Allow the assignment of responsibilities to objects ("Who does what?")
- Model complex control flows that are difficult to follow at runtime

- **Creational Patterns**

- Allow to abstract from complex instantiation processes
- Make systems independent from the way its objects are created, composed and represented

Recap: Which Design Patterns do we know already?



Pattern-Based Development: Definition

- **Pattern-Based Development**

- Model-based development that focuses on reuse and extensive use of patterns during analysis, system design, object design and testing
- Goal: Manage Complexity, reduce cost and time-to-market
- Desirable: Applying patterns throughout the software lifecycle

- **Pattern Coverage**

- Every element in the UML model and every element in the source code is covered by a pattern
- Desirable: 100% Coverage

Pattern-Based Development: Approach

→ We start with a problem statement

- The in-class exercises and homework are organized in sprints
 - Sprint planning meeting: sprint backlog definition
 - Sprint: in-class exercise, tutorial or homework
 - Sprint review
- We are using a reduced Software Process Lifecycle
 - Object-Oriented Analysis, System Design, Object Design & Implementation
- Modeling in UML & Implementation in JAVA

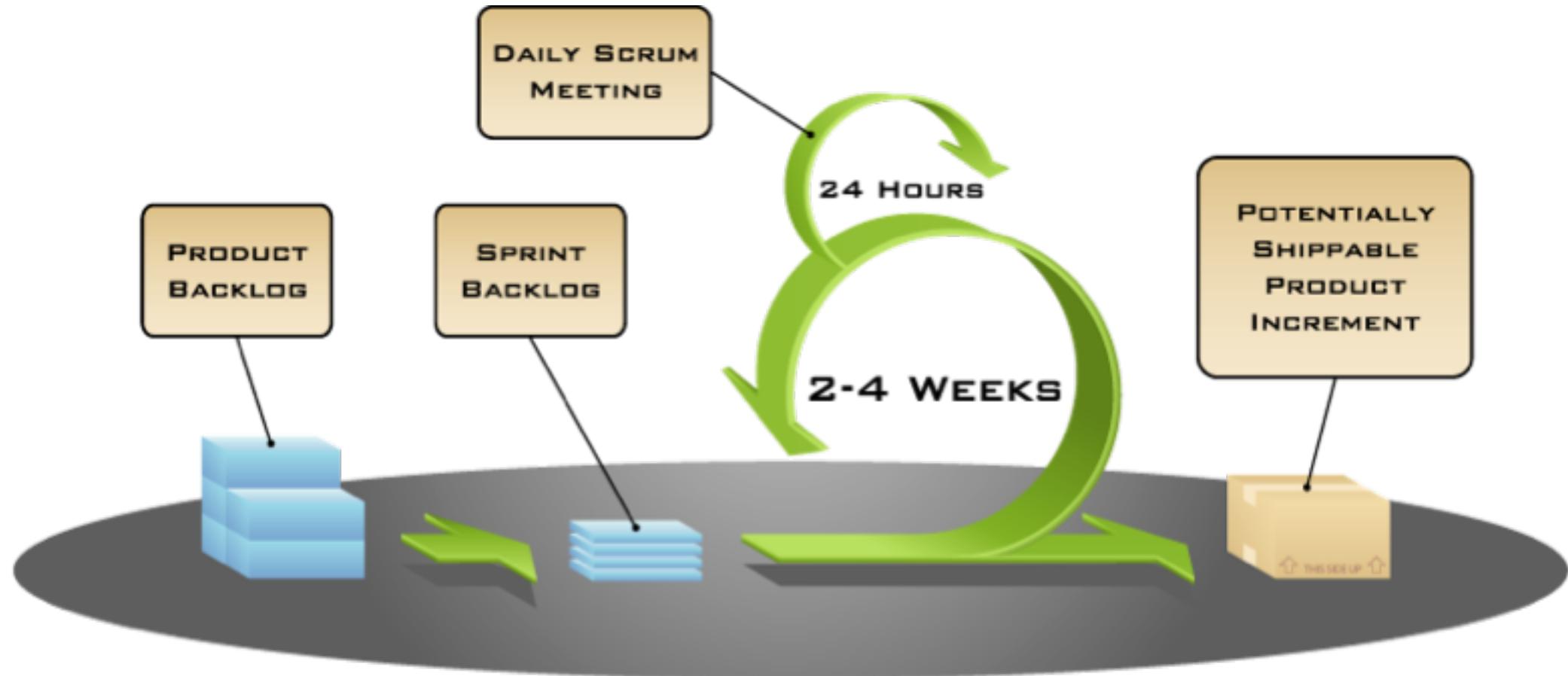
The Problem Statement provides Clues for the Use of Design Patterns (Examples 1/2)

- *Text*: “must interface with an existing object”
⇒ **Adapter Pattern**
- *Text*: “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”, “must provide backward compatibility”
⇒ **Bridge Pattern**
- *Text*: “must interface to existing set of objects”, “must interface to existing API”, “must interface to existing service”
⇒ **Façade Pattern**
- *Text*: “must be manufacturer independent”, “device independent”, “must support a family of products”
⇒ **Abstract Factory Pattern (not discussed in class, optional reading)**

The Problem Statement provides Clues for the Use of Design Patterns (Examples 2/2)

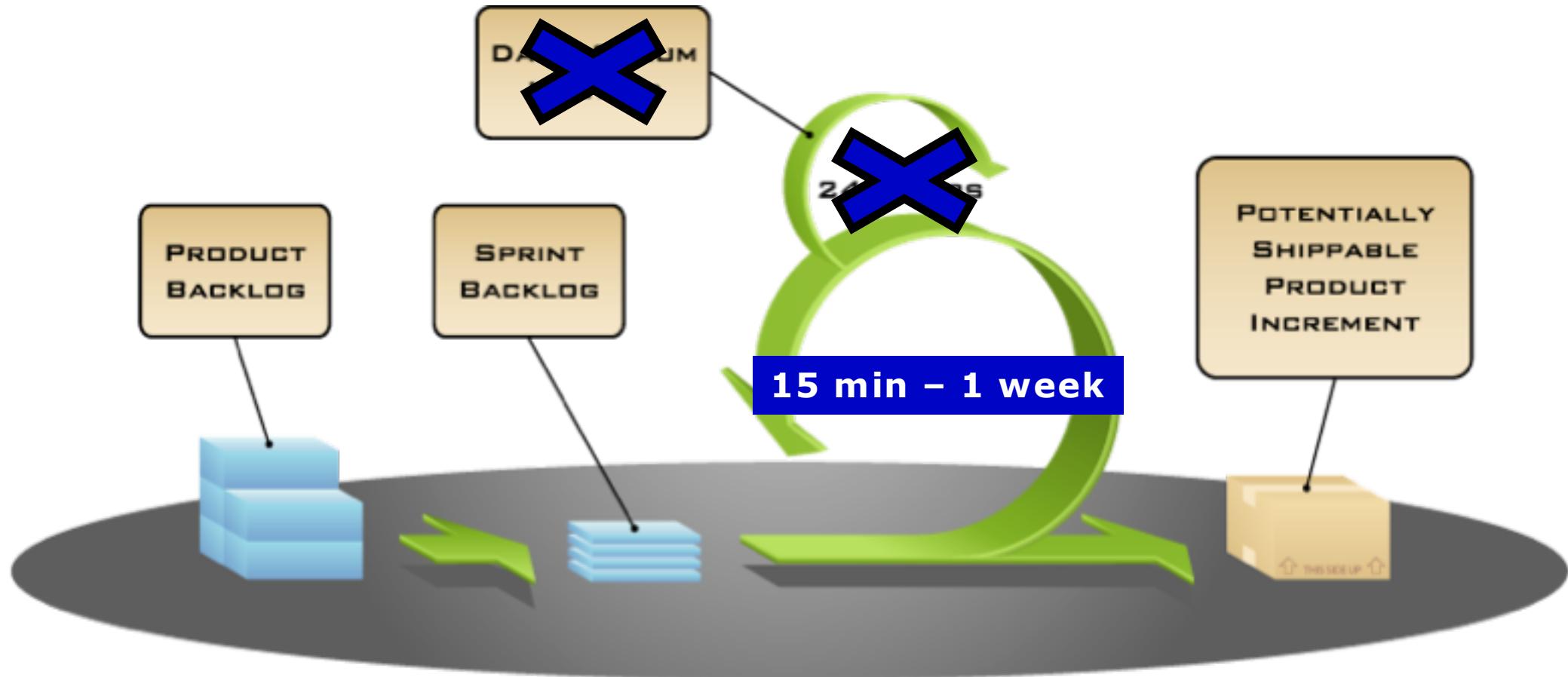
- *Text*: “complex structure”, “must have variable depth and width”
⇒ **Composite Pattern**
- *Text*: “must provide a policy independent from the mechanism”, “must allow to change algorithms at runtime”
⇒ **Strategy Pattern**
- *Text*: “must be location transparent”
⇒ **Proxy Pattern**
- *Text*: “must be extensible”, “must be scalable”
⇒ **Observer Pattern** (MVC Architectural Pattern)

Recap: Our Development Approach is based on Scrum



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

Recap: We use a Modified Form of Scrum



COPYRIGHT © 2005, MOUNTAIN GOAT SOFTWARE

Modified Scrum Development Process for Bumpers



- Each sprint is an in-class exercise
- After each sprint there is a review (our sample solution or a solution from you)
- Interaction with the customer (the instructors are your customers)
- Tutors are domain-specific experts and are eager to help you
- Each sprint has a sprint backlog which is a subset of the product backlog
 - The instructors can also change the product backlog after any sprint
 - Remember: Change is the only constant 😊

Overview of Today's Lecture

- Introduction to Pattern-Based Development
- Review of the 3 finished Sprints in Bumpers
 - Sprint 4: Application of the Strategy Pattern
 - Sprint 5: Application of the Observer Pattern
 - Sprint 6: Application of the Adapter Pattern

Recap: Bumpers – Problem Statement

Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change it's speed.

The game should be platform independent, and visualizes different parameters of the car, e.g. the speed, consumption, and location of the car. When cars crash, there has to be a sound effect. The game should support different collisions and the determination of the collision winner should be changeable during gameplay.

Update: Bumpers – Problem Statement

Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change it's speed.

The game should be platform independent, and visualizes different parameters of the car, e.g. the speed, consumption, and location of the car. **Different visualization types can be easily added. Bumpers also supports backward compatibility.**

When cars crash, there has to be a sound effect. The game should support different collisions and the determination of the collision winner should be changeable during gameplay.

Update: Bumpers – Problem Statement – Spot Patterns

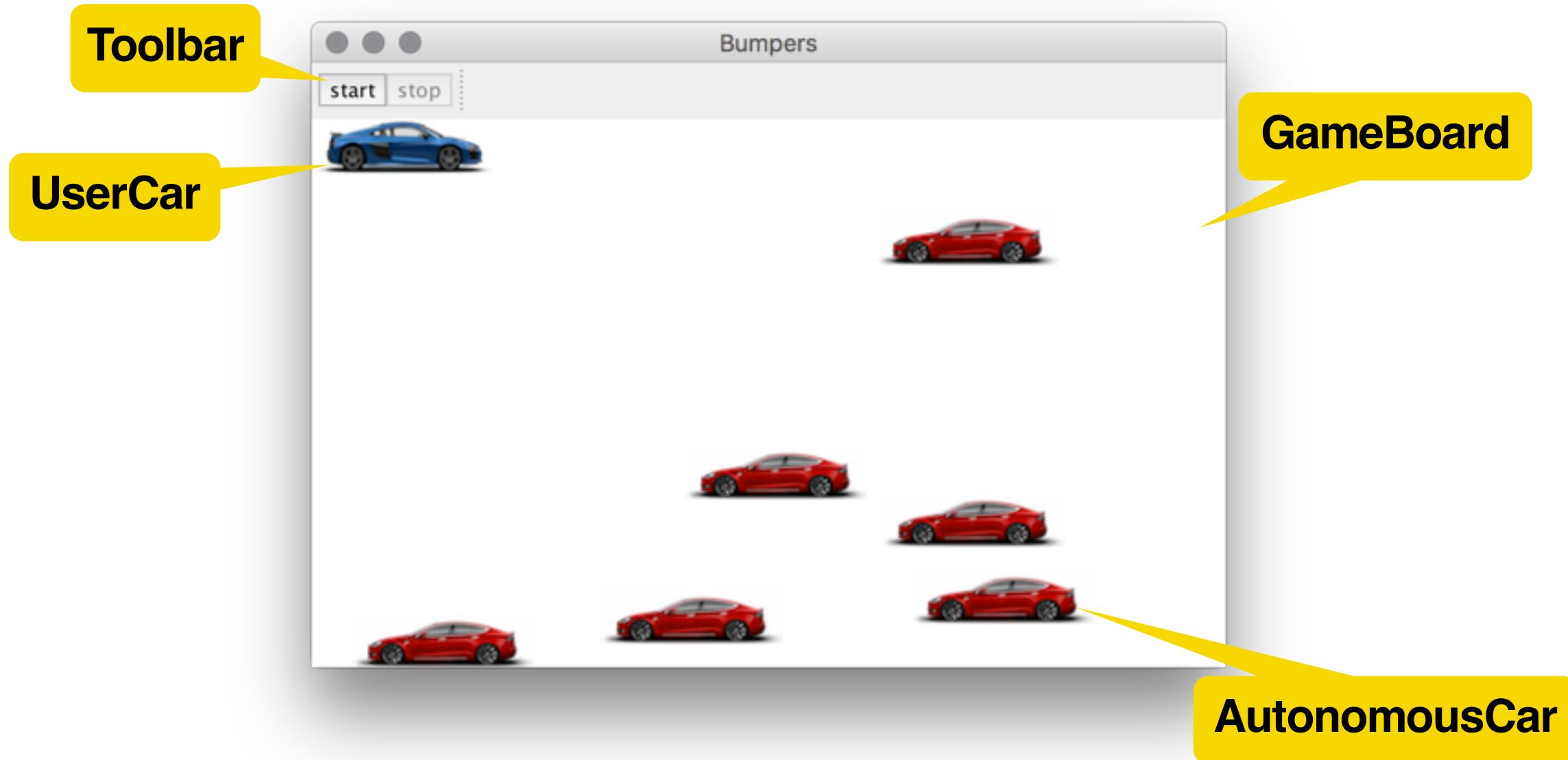
Bumpers is a game where cars drive on a game board and can crash each other. In each collision, there is a winning car. The car that wins all collisions is the winner of the game. The player can start and stop the game. When the game is started, music is played.

A car can be either fast or slow. There is one car controlled by the player. The player can steer the direction of their car with the mouse and change its speed.

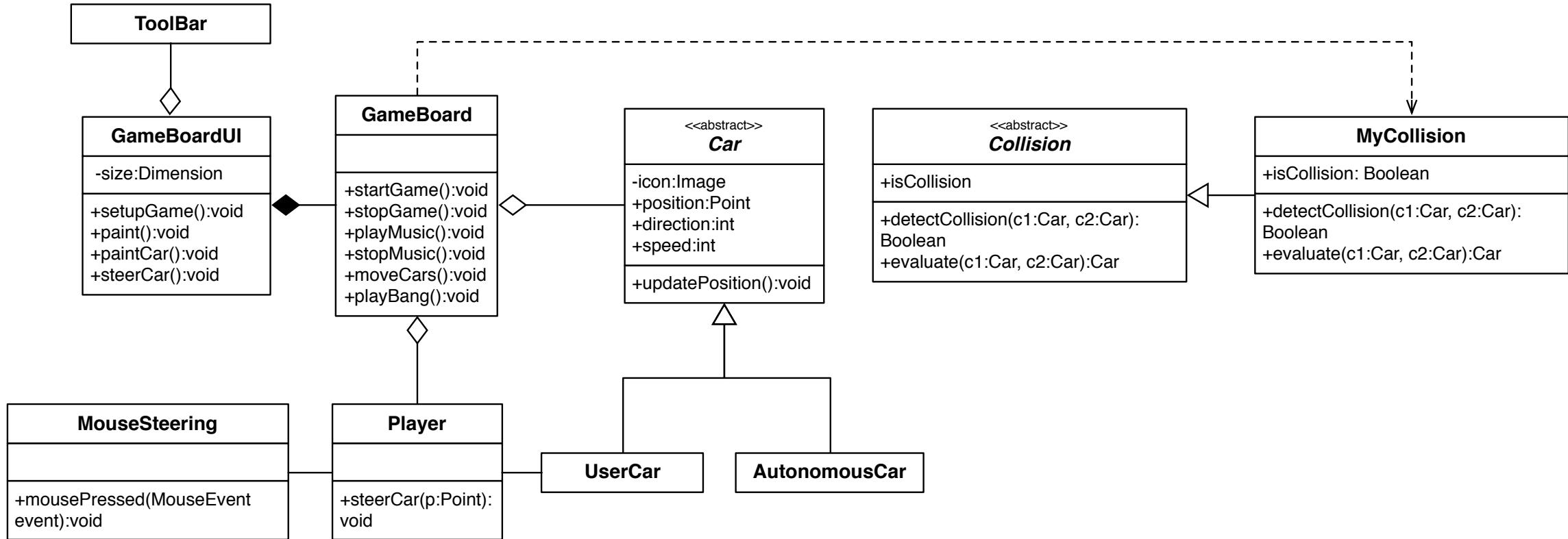
The game should be **platform independent**, and visualizes different parameters of the car, e.g. the speed, consumption, and location of the car. Different visualization types **can be easily added**. Bumpers also **supports backward compatibility**.

When cars crash, there has to be a sound effect. The game should **support different collisions** and the determination of the collision winner should **be changeable during gameplay**.

User Interface Design of Bumpers after Sprint 3



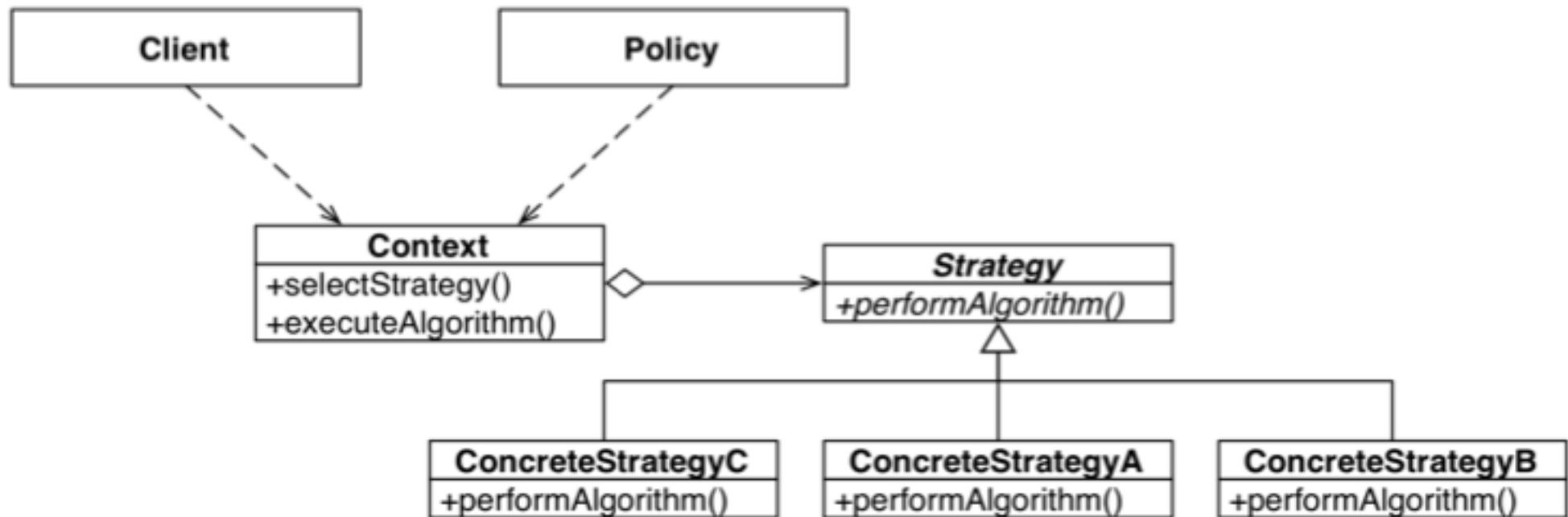
Transformed Object Model



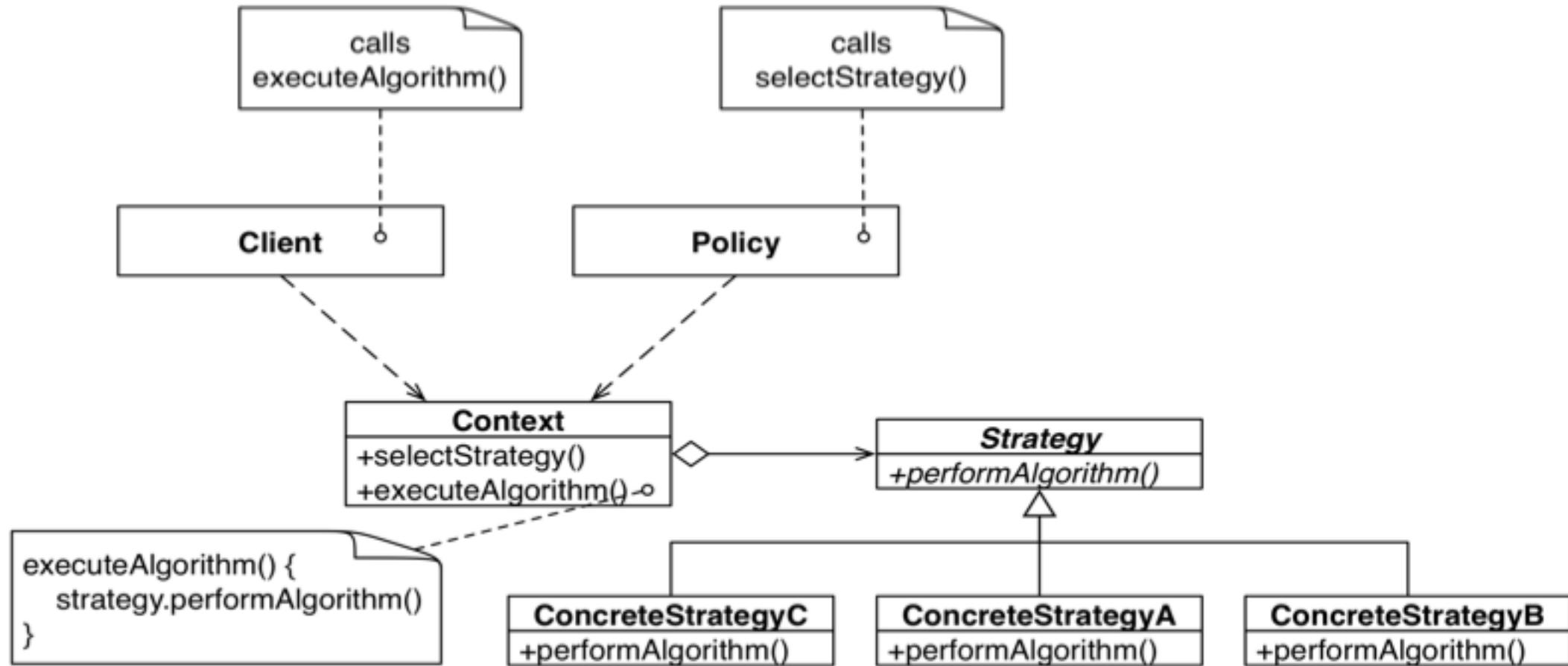
Overview of Today's Lecture

- Introduction to Pattern-Based Development
- Review of the 3 finished Sprints in Bumpers
- Sprint 4: Application of the Strategy Pattern
- Sprint 5: Application of the Observer Pattern
- Sprint 6: Application of the Adapter Pattern

Strategy Pattern: UML Class Diagram

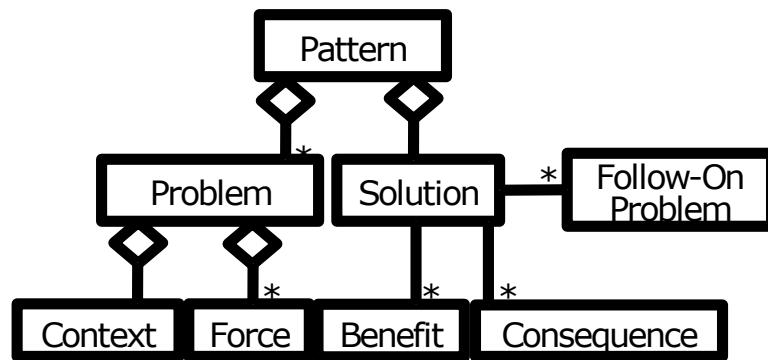


Strategy Pattern: UML Class Diagram

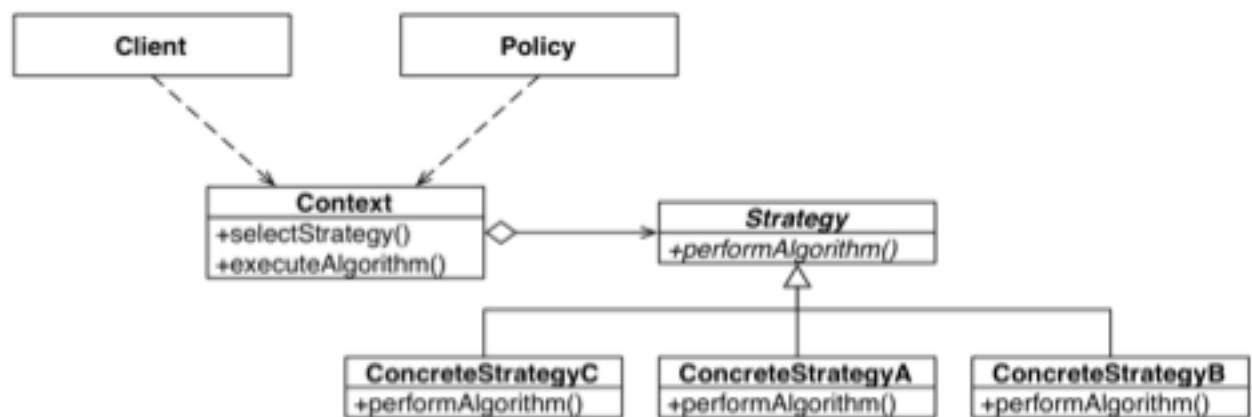


EIST Pattern Model applied to the Strategy Pattern

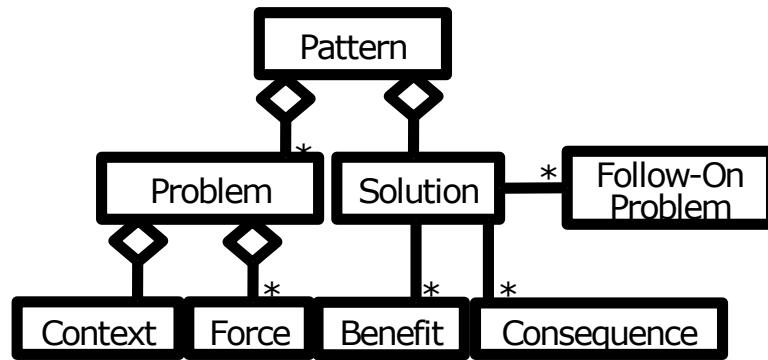
Problem	Switch between different algorithms at run-time
Context	Different algorithms exist for a specific task
Force	<ul style="list-style-type: none">• Add a new algorithm without disturbing the application or other algorithms• A client needs to choose from multiple algorithms
Solution	<ul style="list-style-type: none">• Extract different algorithms into separate classes: ConcreteStrategy• The Client accesses services provided by a Context: <code>performAlgorithm()</code>• The abstract Strategy class describes the interface that is common to all mechanisms that the Context can use• The Policy decides what ConcreteStrategy to use given the Context



Continued on next slide



EIST Pattern Model applied to the Strategy Pattern (ctd)



Benefit

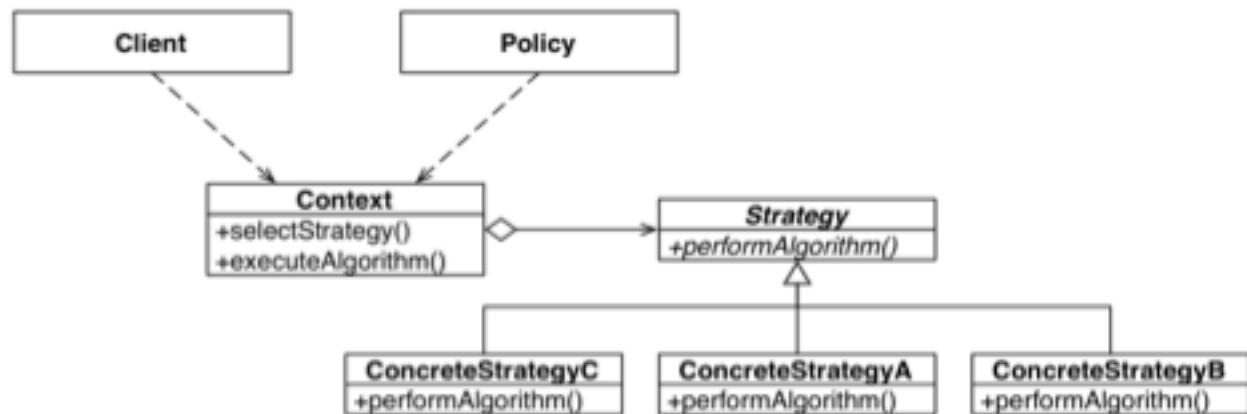
- New algorithms can be added without modifying **Context** or **Client**
- **ConcreteStrategies** can be substituted transparently from *Context*

Consequence

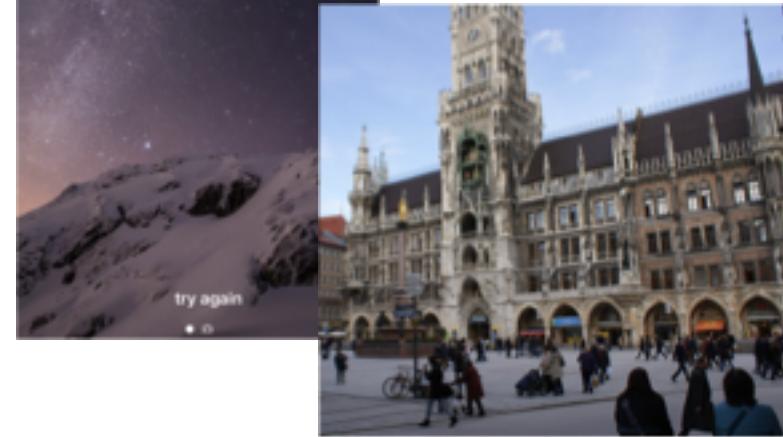
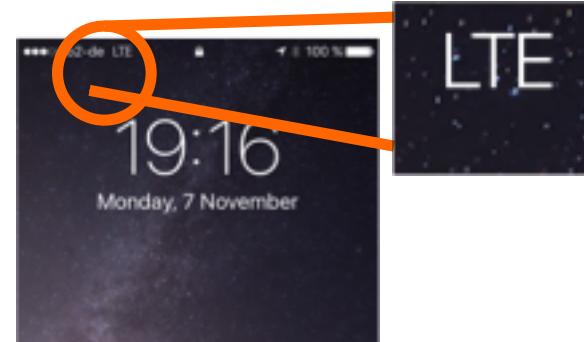
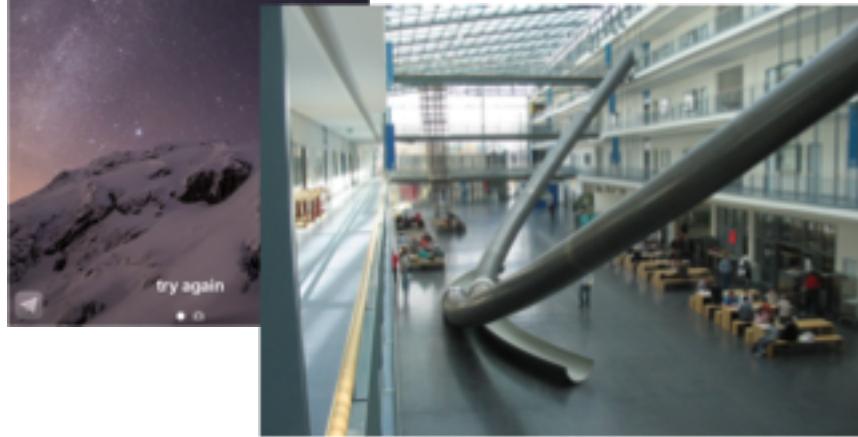
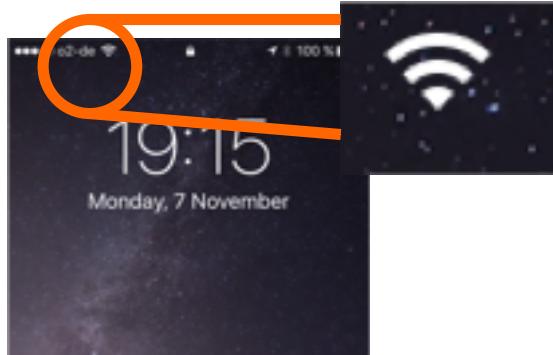
- **Policy** decides which **Strategy** is best, given the current circumstances (e.g. type of collision)
- The **Client** is not aware of the **ConcreteStrategies**

Follow-On Problem

Code becomes more complex the more **ConcreteStrategies** are added



Strategy Pattern Real Life Application Example



If WiFi available, use WiFi ...

... otherwise, use mobile data

Bumper's Backlog before Sprint 4

1. User Interface design of the game board
2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. The player starts and stops the game
6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play
15. Implement a new type of collision

The customer can change the product backlog after any sprint

In-Class Exercise 01: Sprint 4

Duration
15 min

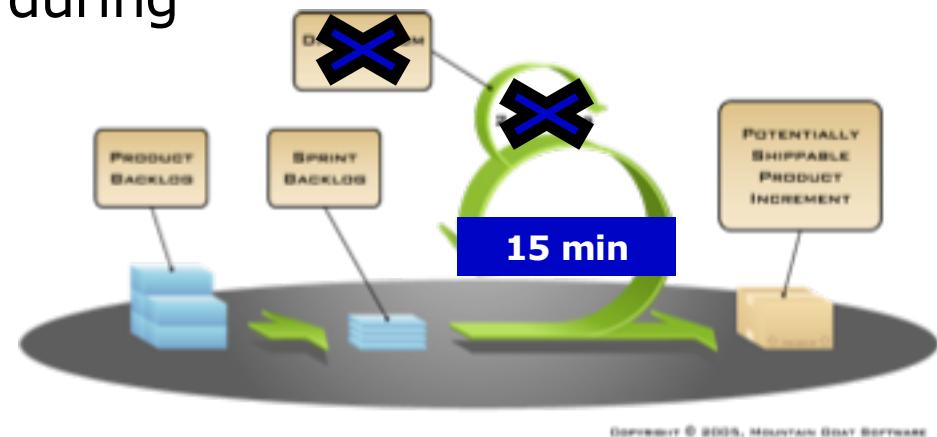
Sprint Backlog:

14. The game supports different collisions, the determination of the collision winner is changeable during game play

Additional Information:

- DefaultCollision is selected when the game starts
- With the DefaultCollision the UserCar always wins
- You can change the collision type to MyCollision during game play using the ComboBox in the ToolBar
- Find TODOs in the template code

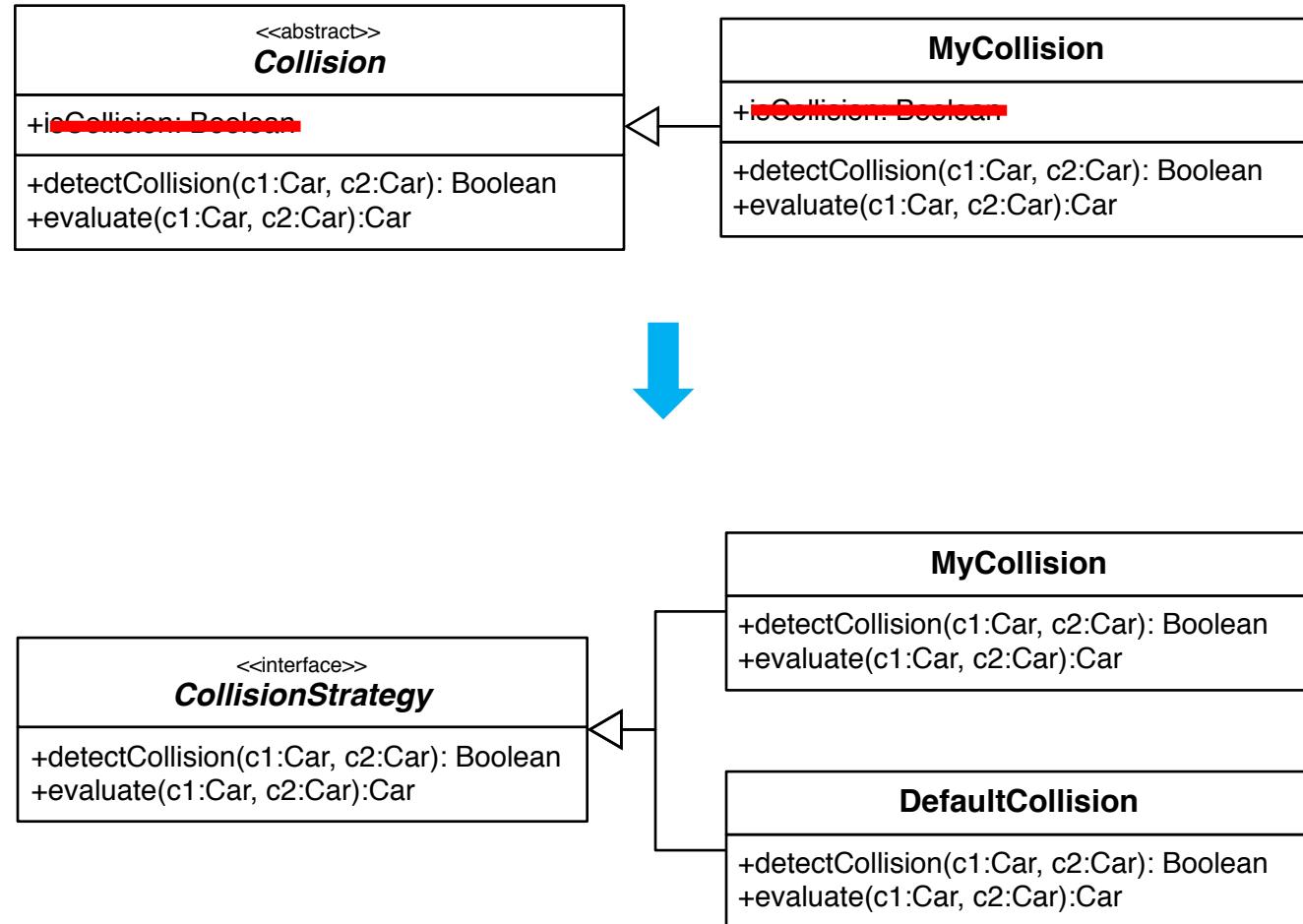
Start the exercise on ArTEMiS



User Interface Design of Bumpers after Sprint 4



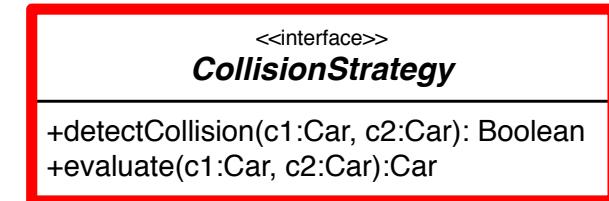
Hint: Use the Strategy Pattern



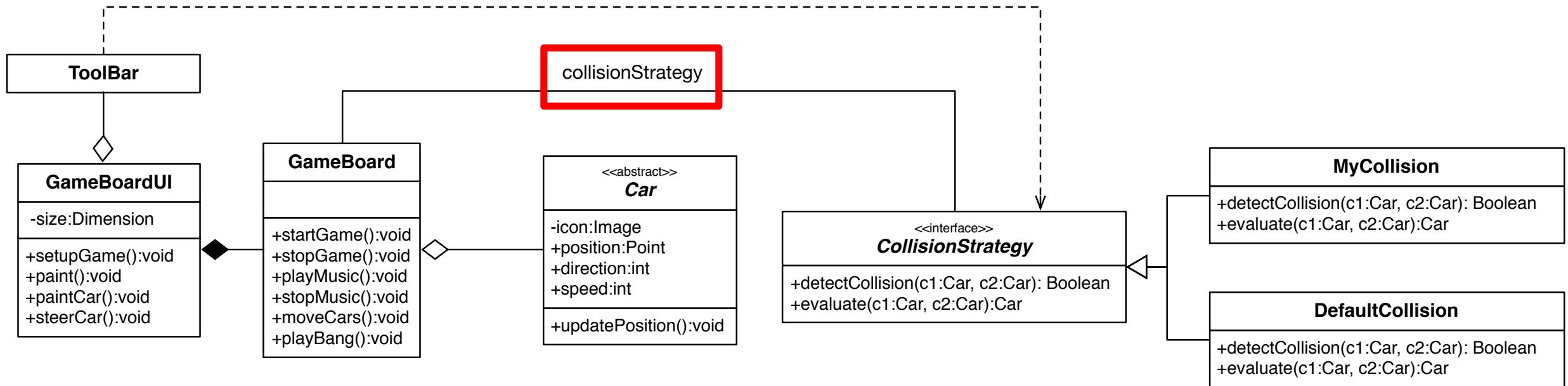
Hint: CollisionStrategy Interface

```
package edu.tum.cs.i1.eist;  
  
import edu.tum.cs.i1.eist.car.Car;
```

```
public interface CollisionStrategy {  
  
    public boolean detectCollision(Car car1, Car car2);  
    public Car evaluate(Car car1, Car car2);  
}
```



Hint: Setting the DefaultCollision Strategy - Model

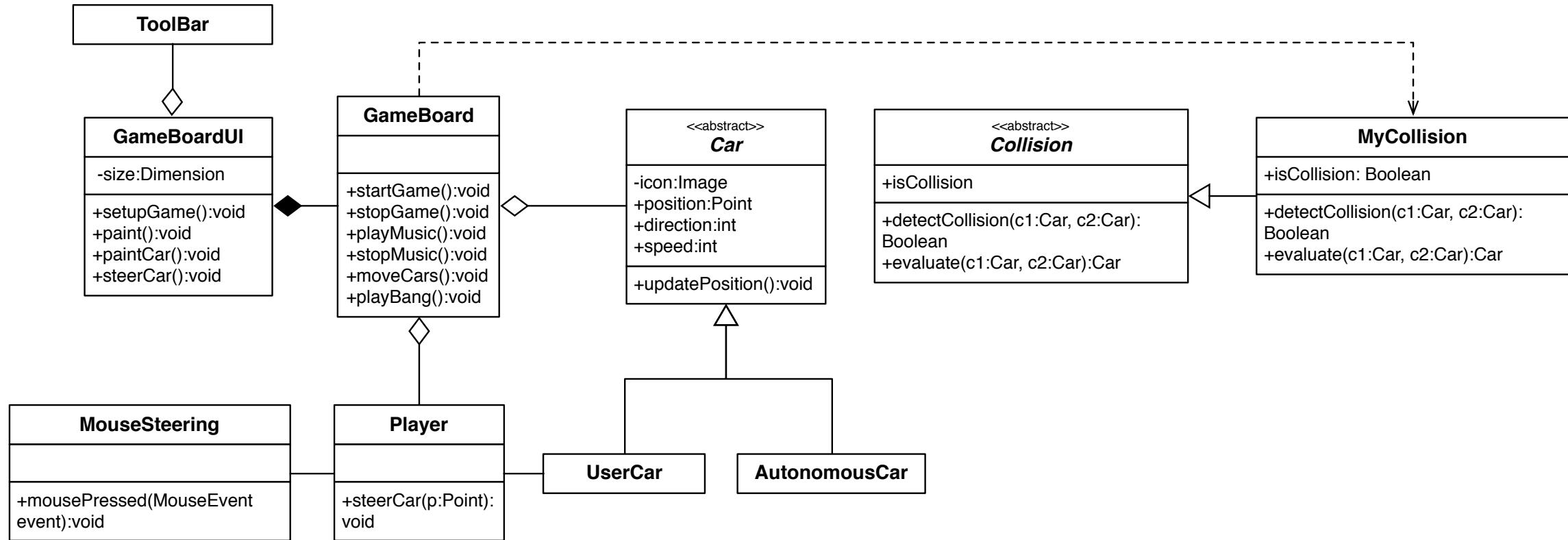


Hint: Setting the DefaultCollision Strategy - Code

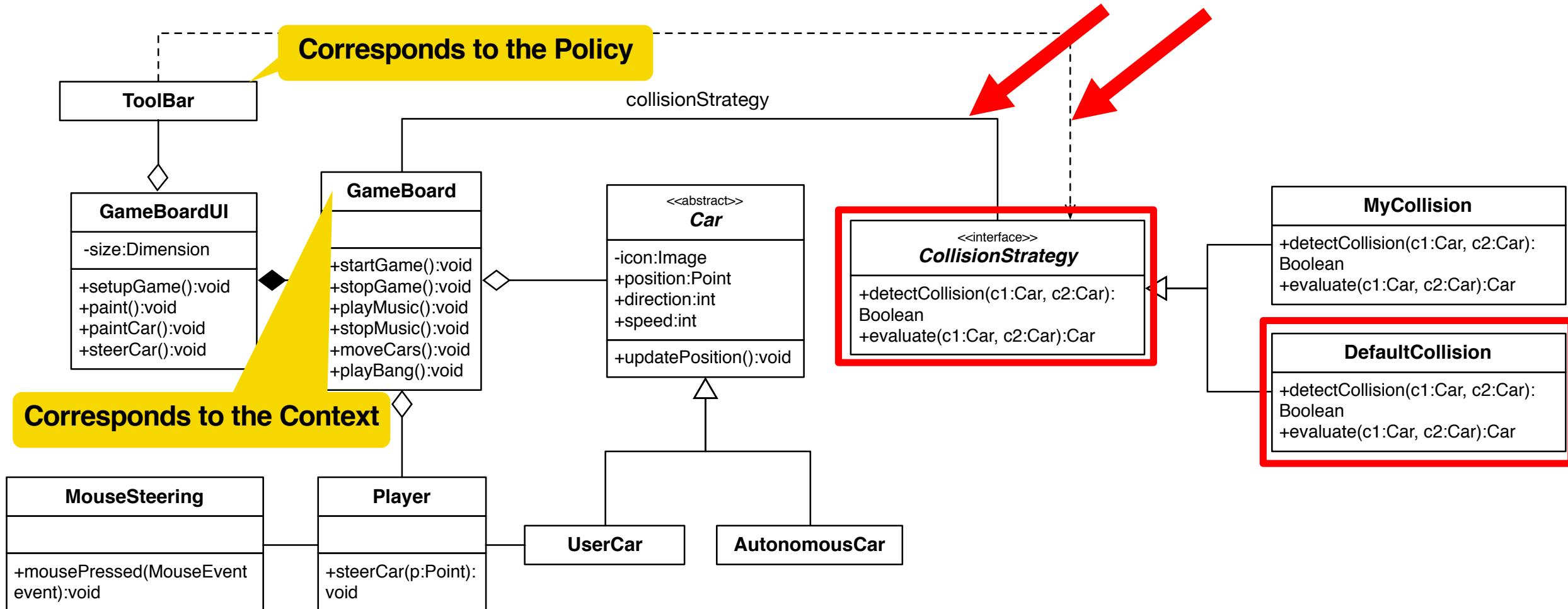
```
public class GameBoard {  
    ...  
    private CollisionStrategy collisionStrategy = new DefaultCollision();  
    ...  
    public void moveCars() {  
        ...  
        for (Car car : cars) {  
            ...  
            if(collisionStrategy.detectCollision(getPlayerCar(), car)) {  
                collisionStrategy.evaluate(getPlayerCar(), car);  
                audioPlayer.playBangAudio();  
            }  
        }  
    }  
  
    public CollisionStrategy getCollisionStrategy() {  
        return collisionStrategy;  
    }  
  
    public void setCollisionStrategy(CollisionStrategy collisionStrategy) {  
        this.collisionStrategy = collisionStrategy;  
    }  
}
```

GameBoard
+startGame():void
+stopGame():void
+playMusic():void
+stopMusic():void
+moveCars():void
+playBang():void

Object Model before Sprint 4



Sprint 4 Review: Object Model after Sprint 4



Bumper's Backlog after Sprint 4

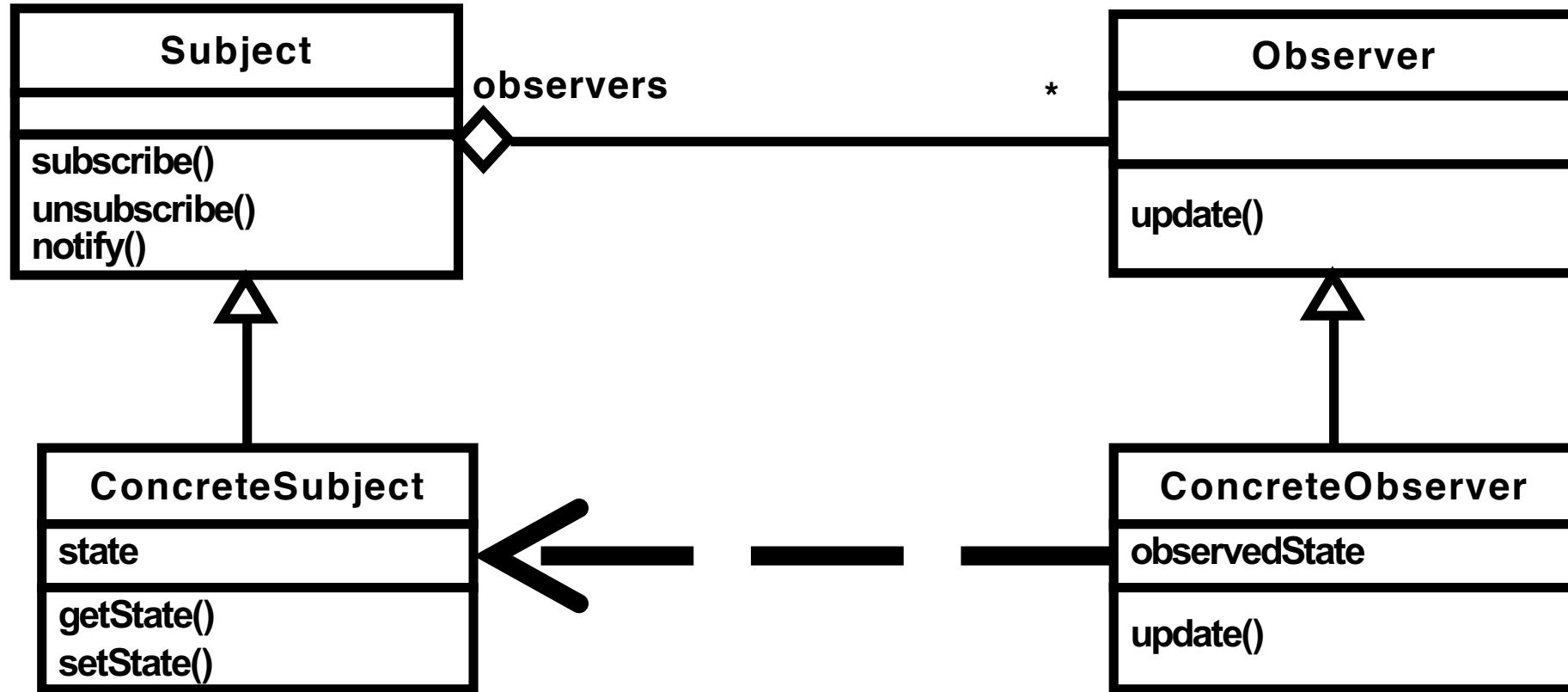
1. User Interface design of the game board
2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. The player starts and stops the game
6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14.  The game supports different collisions, the determination of the collision winner is changeable during game play
15. Implement a new type of collision

The customer can change the product backlog after any sprint

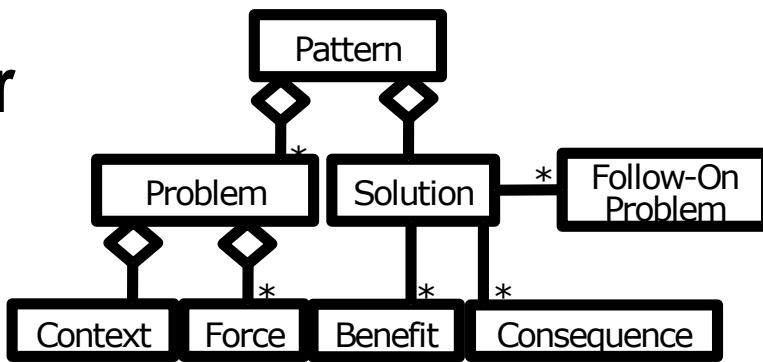
Overview of Today's Lecture

- Introduction to Pattern-Based Development
- Review of the 3 finished Sprints in Bumpers
- Sprint 4: Application of the Strategy Pattern
- Sprint 5: Application of the Observer Pattern
- Sprint 6: Application of the Adapter Pattern

Observer Pattern: UML Class Diagram



EIST Pattern Model applied to the Observer Pattern



Problem

An object changes its state quite often and it is hard to maintain consistency among dependent objects

Context

Maintain consistency across the states of one subject and many observers

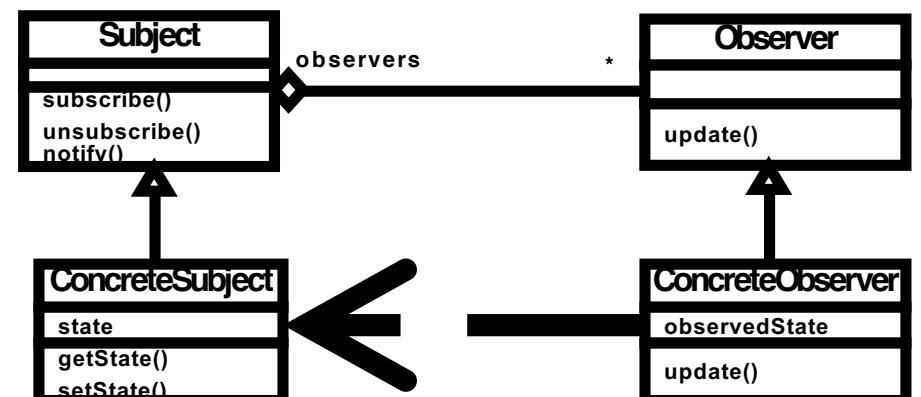
Force

Model a 1-to-many dependency between objects: when one object changes its state, the observing objects are notified

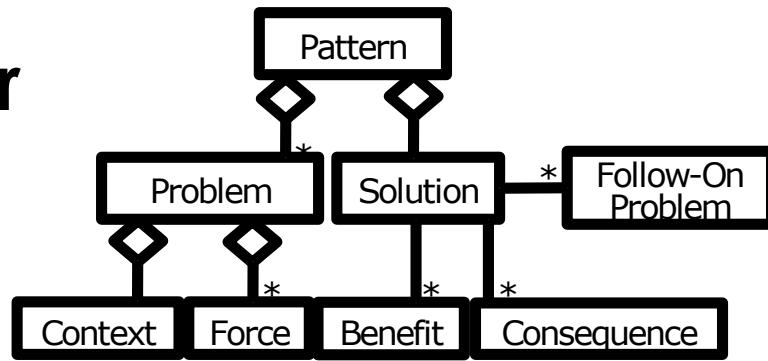
Solution

- **Subject** represents the object with a changing state
 - The state is contained in the subclass **ConcreteSubject**
- **Observers** attach to the **Subject** by calling `subscribe()`
- Each **Observer** has a different view of the state of the **ConcreteSubject**
 - The state can be obtained and set by the subclasses of type **ConcreteObserver**

Continued on next slide



EIST Pattern Model applied to the Observer Pattern (ctd)



Benefit

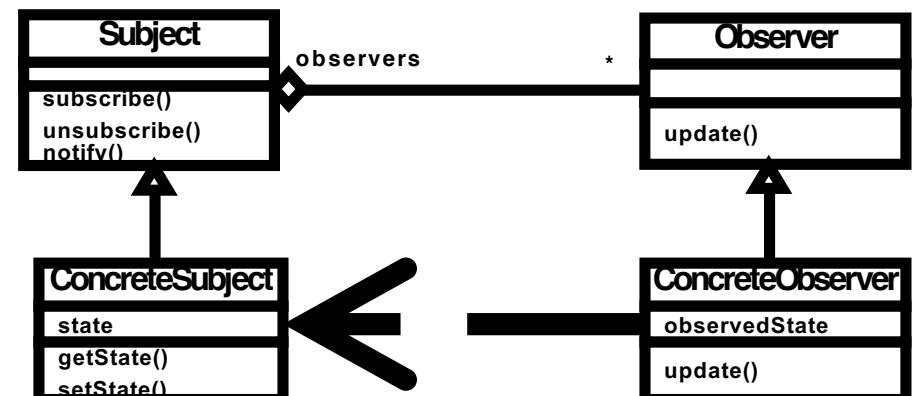
- Decouples the Subject from the *Observers*
- New *Observers* can be added easily

Consequence

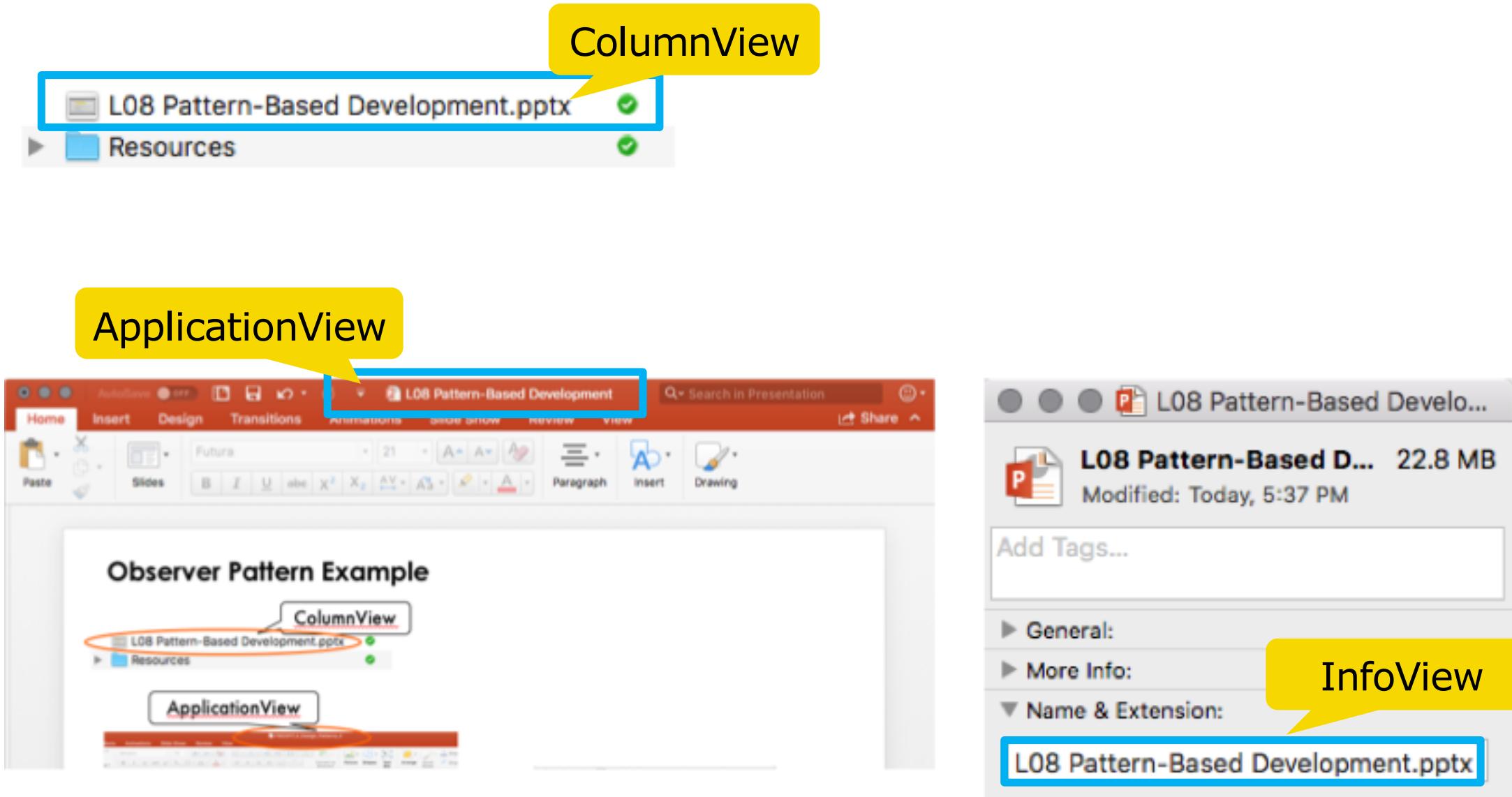
The *Observers* decide how to handle updates/notifications

Follow-On Problem

Can result in many spurious broadcasts when the state of the *Subject* changes



Observer Pattern Example



Bumper's Backlog before Sprint 5

1. User Interface design of the game board
2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. The player starts and stops the game
6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play
15. Implement a new type of collision

The customer can change the product backlog after any sprint

In-Class Exercise 02: Sprint 5

Duration
15 min

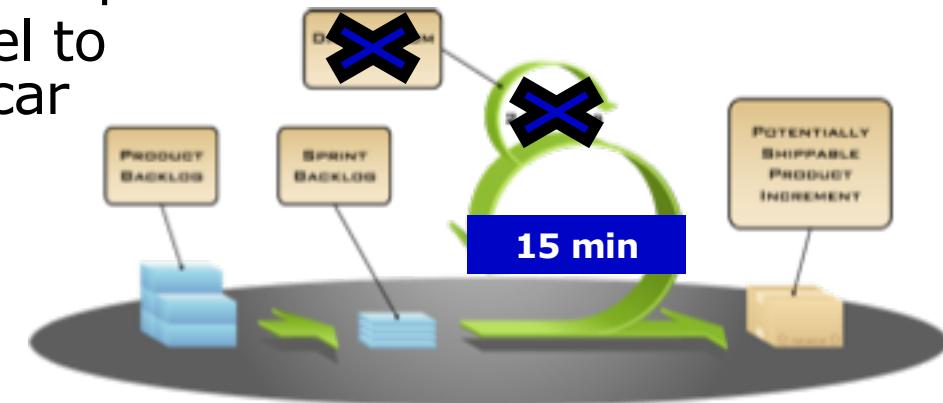
Sprint Backlog

9. The player can change their car's speed
11. The speed, consumption, and location of the player's car is visualized in an instrument panel

Additional Information:

- Add an instrument panel
- The instruments for the instrument panel will be provided
- Add a speed controller to the instrument panel to decrease and increase the speed of the user car

Start the exercise on ArTEMiS



Reuse Existing Code

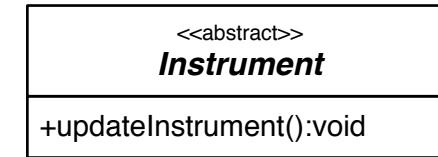
- You find the code for the instruments in the following repository:
<https://repobruegge.in.tum.de/projects/EIST2018L08BUMPERSS05/repos/eist2018-l08-bumpers-sprint05-instruments/browse>
(Short: <http://bit.ly/eistsprint04>)

 [GPS.java](#)

 [RotationsPerSecond.java](#)

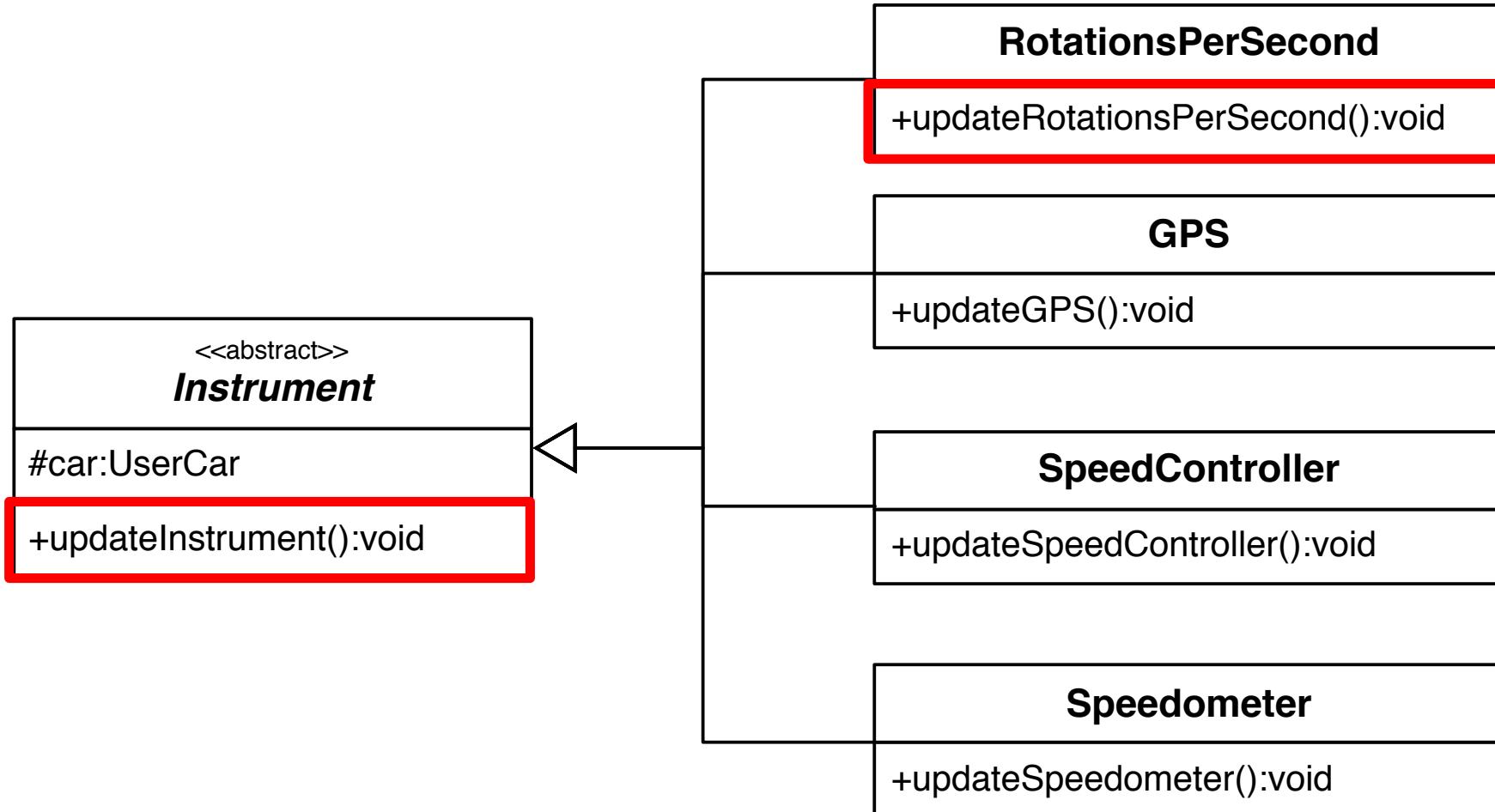
 [SpeedController.java](#)

 [Speedometer.java](#)

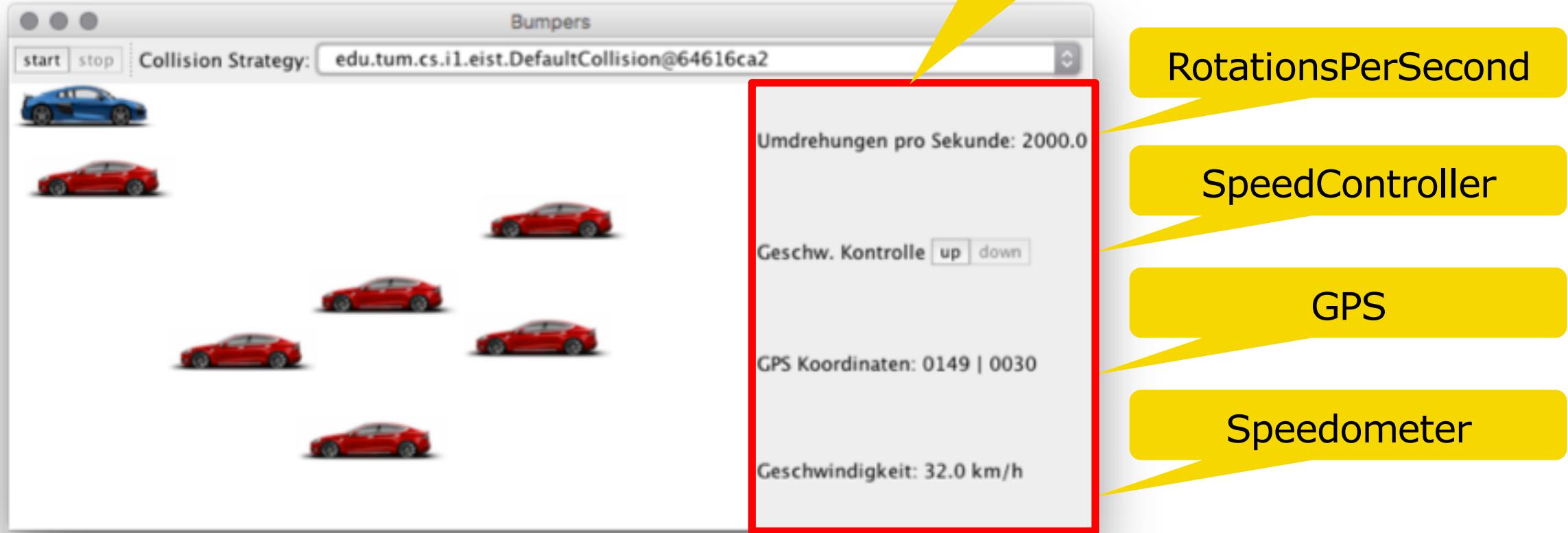


- Copy & paste the external code
- Change the name of the update methods to comply with the `updateInstrument()` Signature of an abstract class `Instrument`

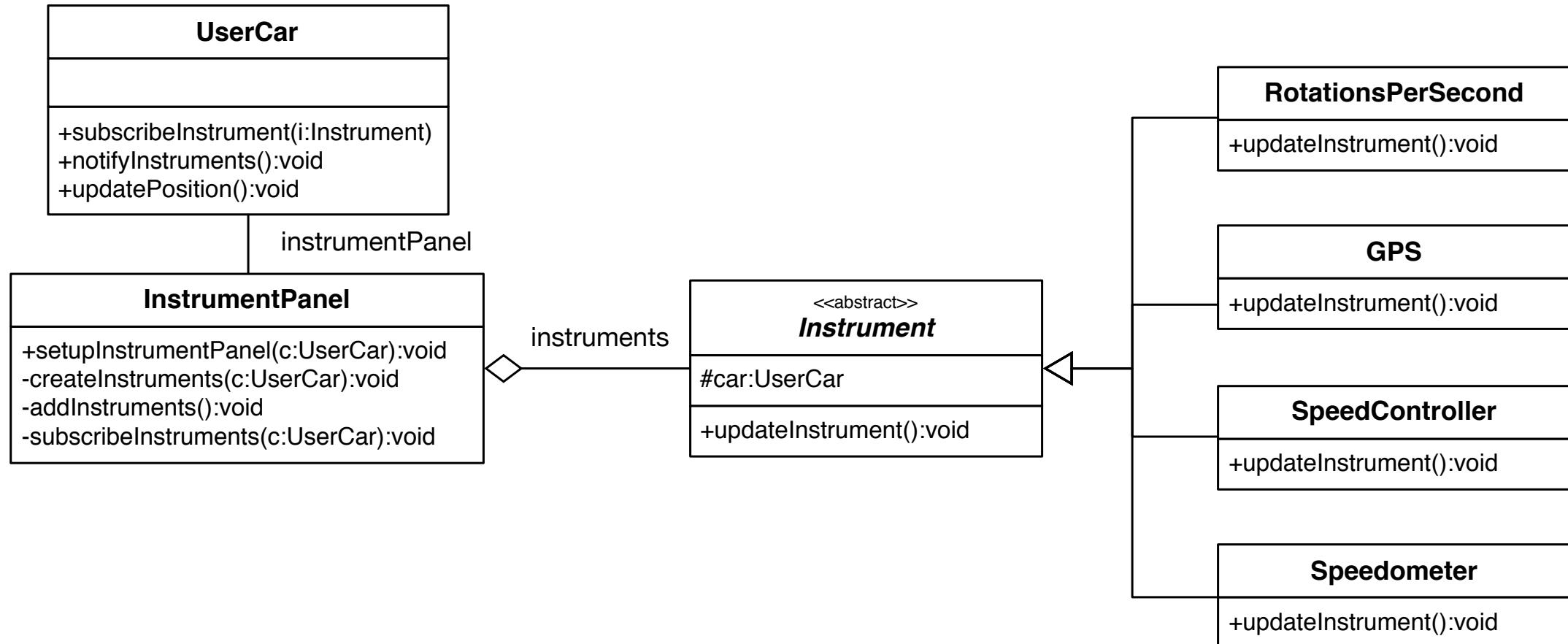
Compatibility Problem



User Interface after Sprint 5

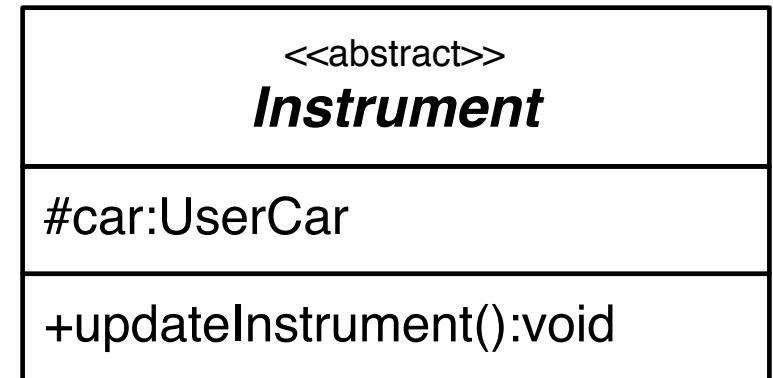


Hint: Methods for Instrument, InstrumentPanel & UserCar



Hint: Introduction of the abstract class Instrument

```
public abstract class Instrument extends JPanel {  
  
    protected UserCar car;  
  
    public Instrument(UserCar car) {  
        this.car = car;  
    }  
  
    public abstract void updateInstrument();  
  
}
```



Hint: The InstrumentPanel Pt. 1

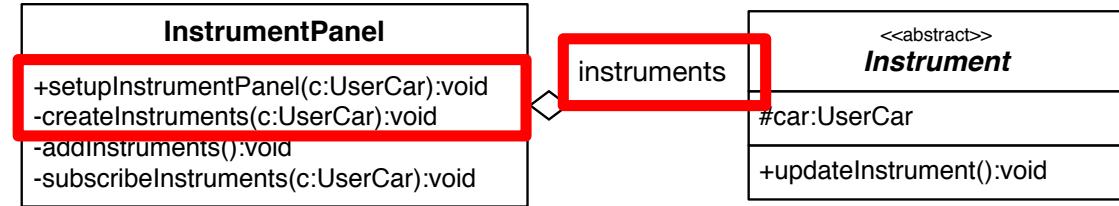
```
public class InstrumentPanel extends JToolBar {
```

```
    public List<Instrument> instruments = new ArrayList<Instrument>();
```

```
    public InstrumentPanel() {
        super(JToolBar.VERTICAL);
        setFloatable(false);
    }
```

```
    public void setupInstrumentPanel(UserCar userCar) {
        createInstruments(userCar);
        subscribeInstruments(userCar);
        addInstruments();
    }
```

```
    private void createInstruments(UserCar car) {
        if (instruments.size() == 0) {
            instruments.add(new RotationsPerSecond(car));
            instruments.add(new SpeedController(car));
            instruments.add(new GPS(car));
            instruments.add(new Speedometer(car));
        }
    }
```



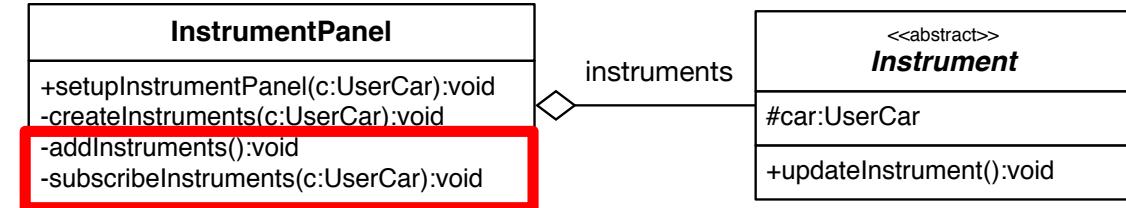
Hint: The InstrumentPanel Pt. 2

```
public class InstrumentPanel extends JToolBar {  
  
    public List<Instrument> instruments = new ArrayList<Instrument>();
```

...

```
private void addInstruments() {  
    for (Instrument instrument: instruments) {  
        add(instrument);  
    }  
}
```

```
private void subscribeInstruments(UserCar car) {  
    for (Instrument instrument: instruments) {  
        car.subscribeInstrument(instrument);  
    }  
}
```

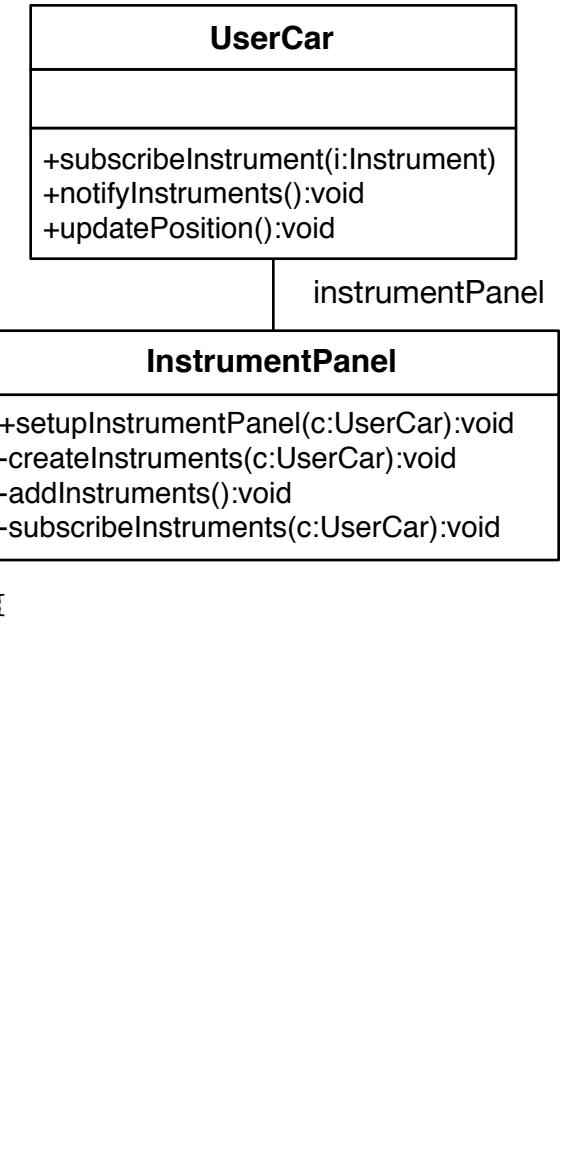


Hint: Adding the InstrumentPanel to the User Interface

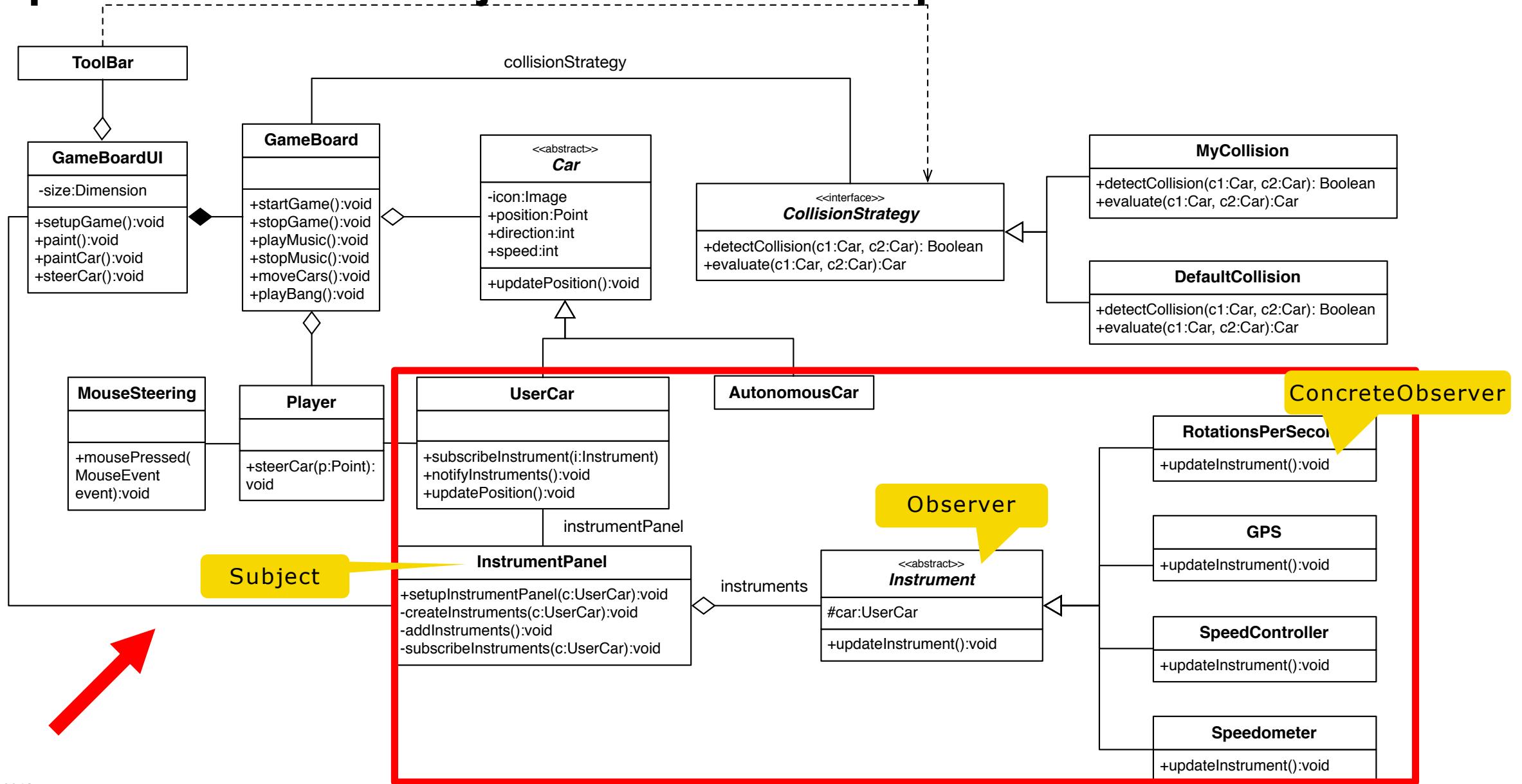
```
public class BumpersWindow extends JFrame {  
    ...  
  
    public BumpersWindow() {  
        super("Bumpers");  
  
        ...  
  
        getContentPane().setLayout(new BorderLayout());  
        content.add(toolBar, BorderLayout.NORTH);  
        content.add(gameBoardUI, BorderLayout.CENTER)  
        content.add(gameBoardUI.gameBoard.getPlayerCar().getInstrumentPanel(), BorderLayout.EAST);  
        getContentPane().add(content, BorderLayout.CENTER),  
        toolBar.setupStrategyBox();  
    }  
  
    ...  
}
```

Hint: Updating the Instruments

```
public class UserCar extends Car {  
    ...  
    private InstrumentPanel instrumentPanel;  
  
    public UserCar(int max_x, int max_y) {  
        super(max_x, max_y);  
        instrumentPanel = new InstrumentPanel();  
        instrumentPanel.setupInstrumentPanel(this);  
        setBody(DEFAULT_USER_CAR_IMAGE);  
    }  
  
    public void subscribeInstrument(Instrument instrument) {  
        if (instrument != null && !this.instrumentPanel.instruments.contains(instrument)) {  
            this.instrumentPanel.instruments.add(instrument);  
        }  
    }  
  
    public void notifyInstruments() {  
        for (Instrument instrument: this.instrumentPanel.instruments) {  
            instrument.updateInstrument();  
        }  
    }  
  
    @Override  
    public void updatePosition(int max_x, int max_y) {  
        super.updatePosition(max_x, max_y);  
        notifyInstruments();  
    }  
}
```



Sprint 5 Review: Object Model after Sprint 5



Bumper's Backlog after Sprint 5

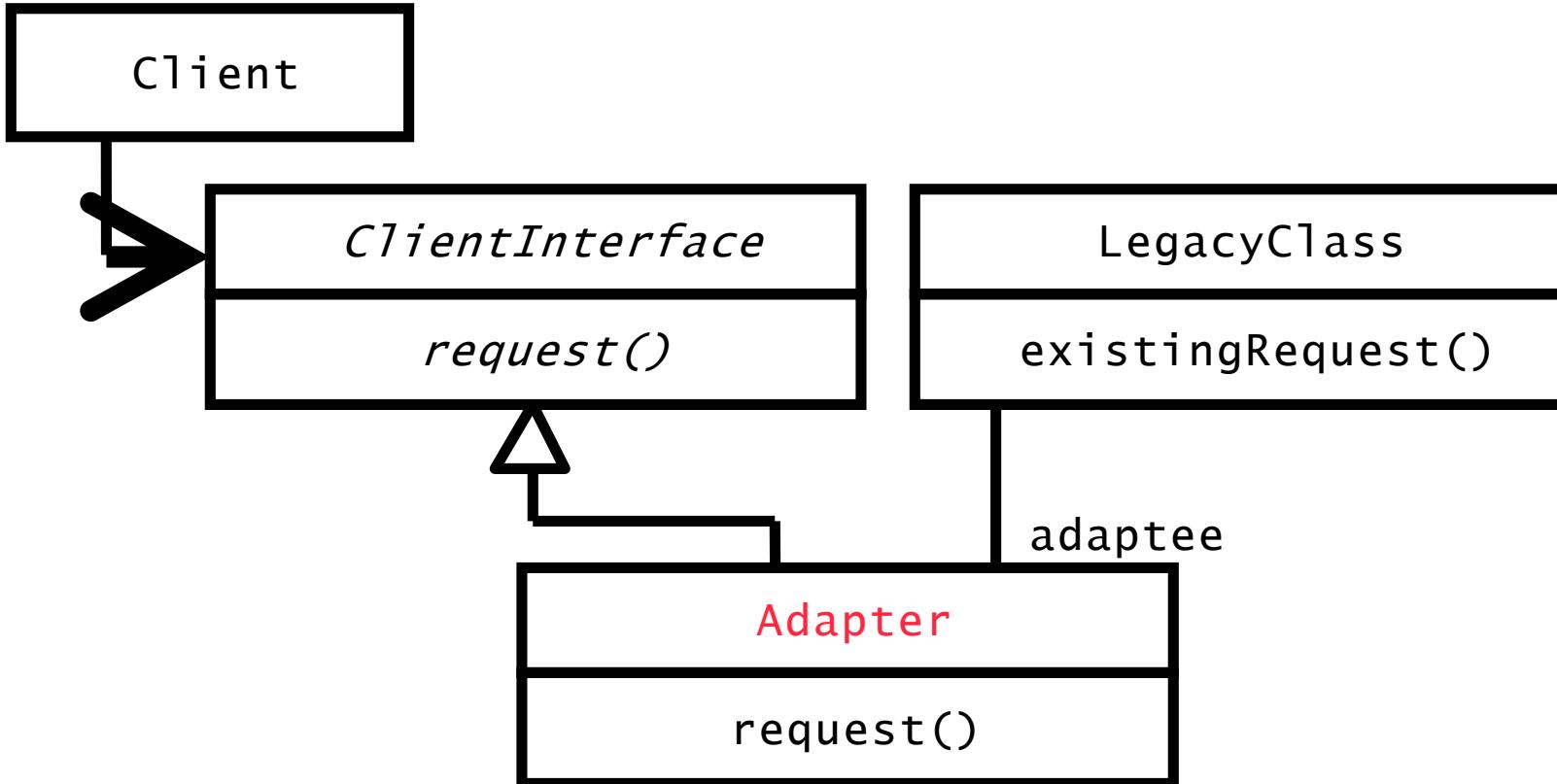
- 1. User Interface design of the game board
- 2. Cars drive on the game board
- 3. Cars collide with each other and each collision has a winning car
- 4. The winner of the game is the car that wins all the collisions
- 5. The player starts and stops the game
- 6. Music plays when the game begins and stops to play when the game ends
- 7. The game supports different car types
- 8. The player steers the car with the mouse
- 9. The player can change their car's speed
- 10. The game should be compatible with multiple platforms
- 11.  The speed, consumption, and location of the player's car is visualized in an instrument panel
- 12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
- 13. A crash sound and/or animation plays when two cars collide
- 14. The game supports different collisions, the determination of the collision winner is changeable during game play
- 15. Implement a new type of collision

The customer can change the product backlog after any sprint

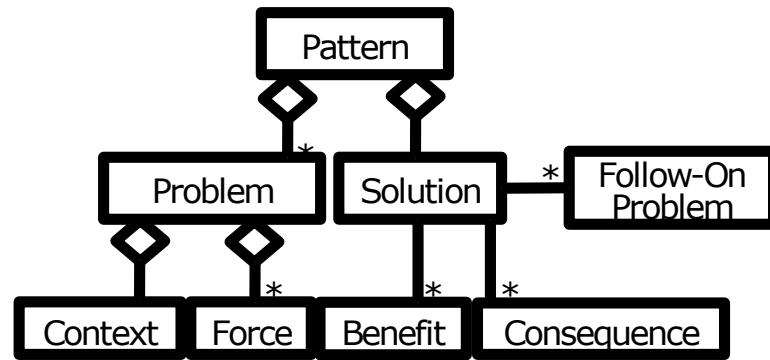
Overview of Today's Lecture

- Introduction to Pattern-Based Development
- Review of the 3 finished Sprints in Bumpers
- Sprint 4: Application of the Strategy Pattern
- Sprint 5: Application of the Observer Pattern
- Sprint 6: Application of the Adapter Pattern

Adapter Pattern: UML Class Diagram

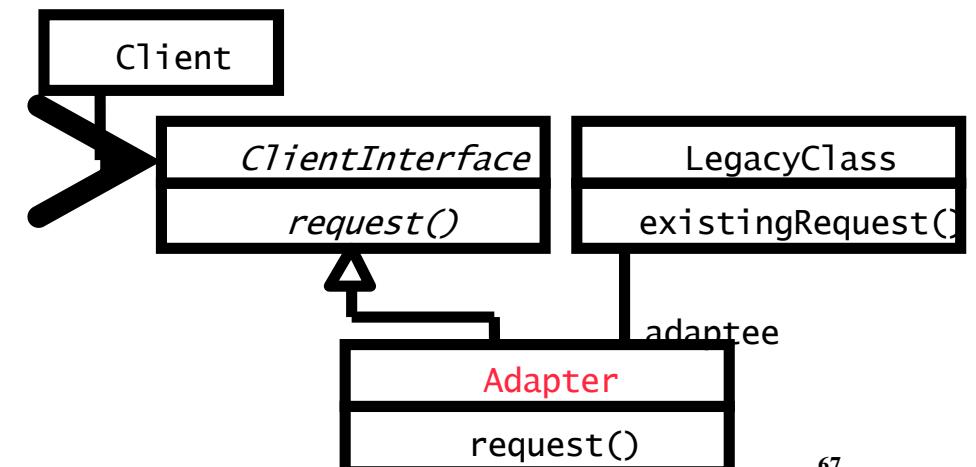


EIST Pattern Model applied to the Adapter Pattern

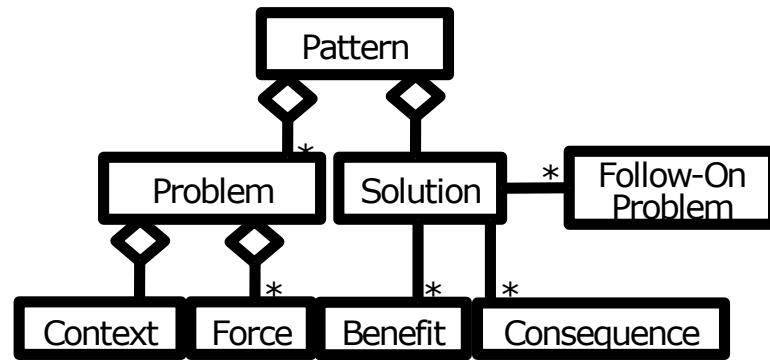


Pattern Name	Adapter
Problem	A legacy class/system exists but is hard to reuse due to incompatible interfaces
Context	Connect incompatible components
Force	Converts the interface of an existing component into another interface expected by the calling component
Solution	<ul style="list-style-type: none">An Adapter implements the ClientInterface expected by the ClientThe Adapter delegates requests from the client to the LegacyClass and performs any necessary conversion.

Continued on next slide



EIST Pattern Model applied to the Adapter Pattern (ctd)



Benefit

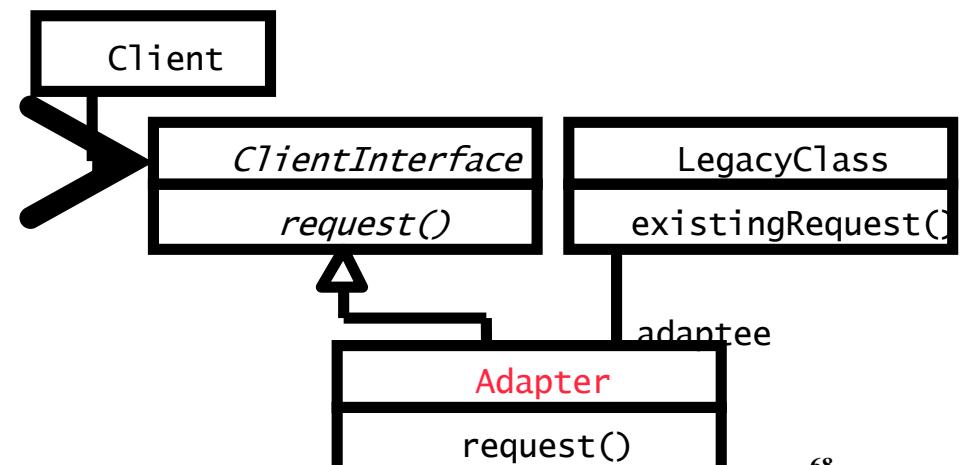
- Reuse existing components
- Adapter works with **LegacyClass** and all of its subclasses

Consequence

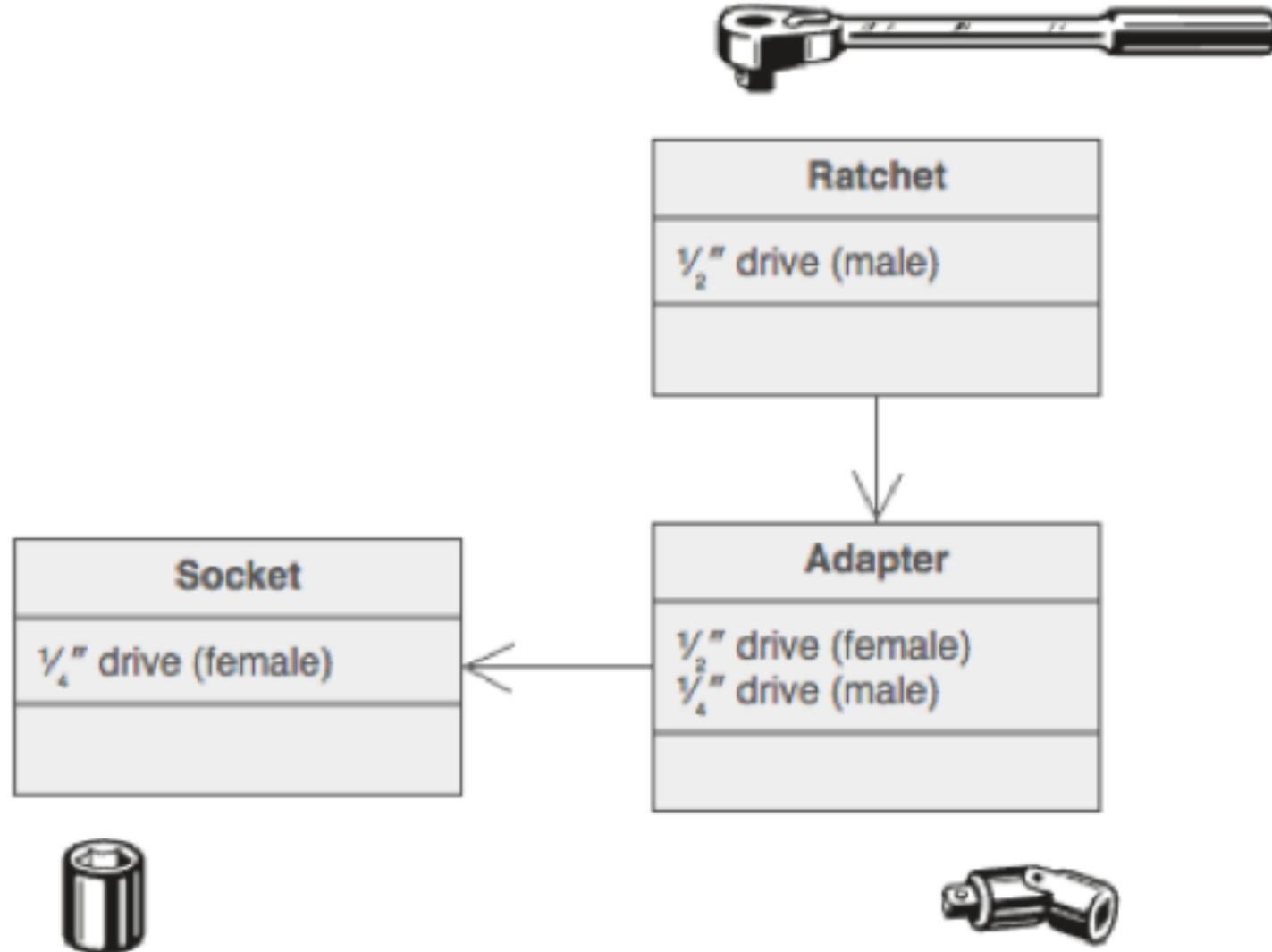
Client and **LegacyClass** work together without modification of either **Client** or **LegacyClass**

Follow-On Problem

A new **Adapter** needs to be written for each specialization (e.g., subclass) of **ClientInterface**

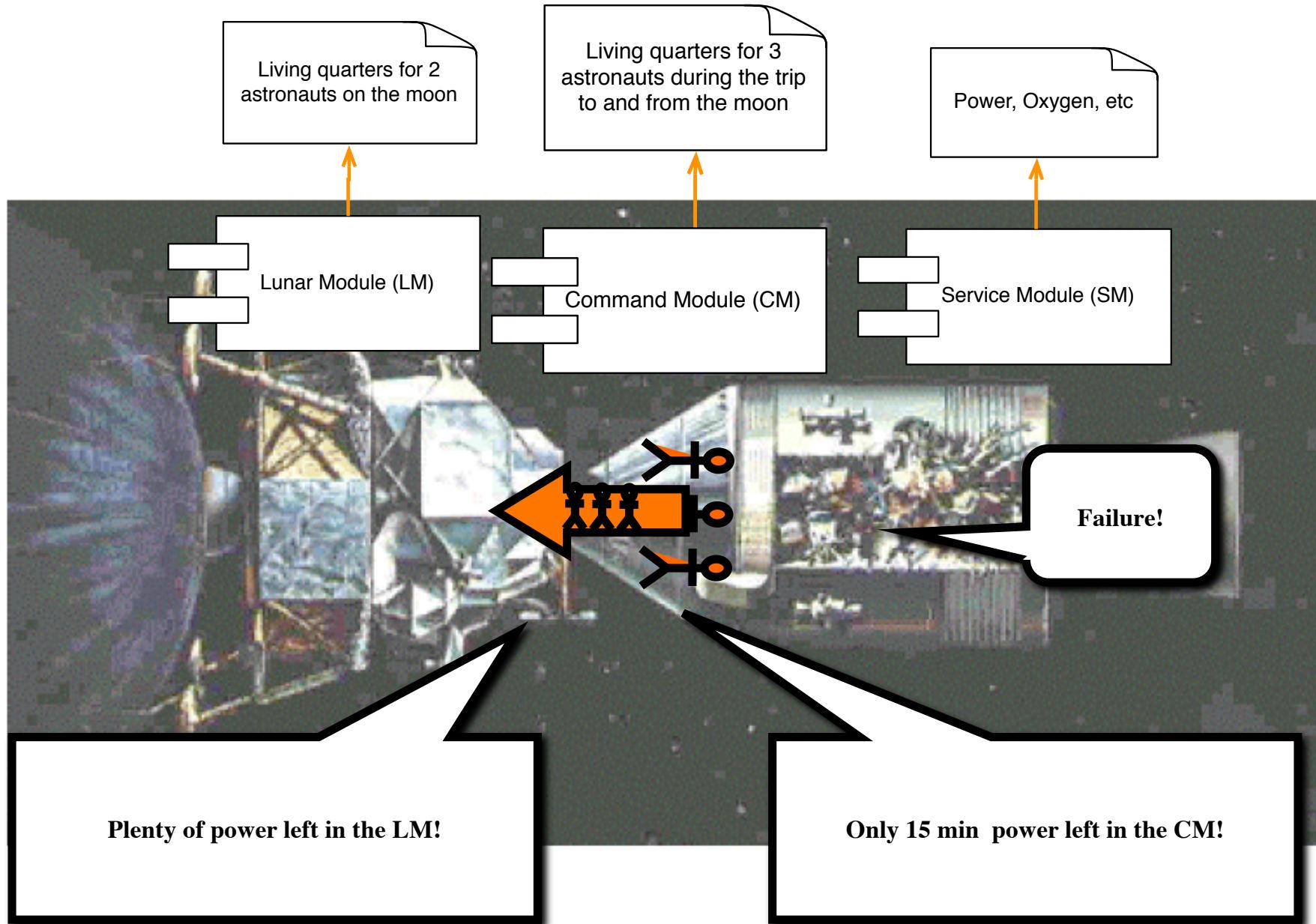


Adapter Pattern Example



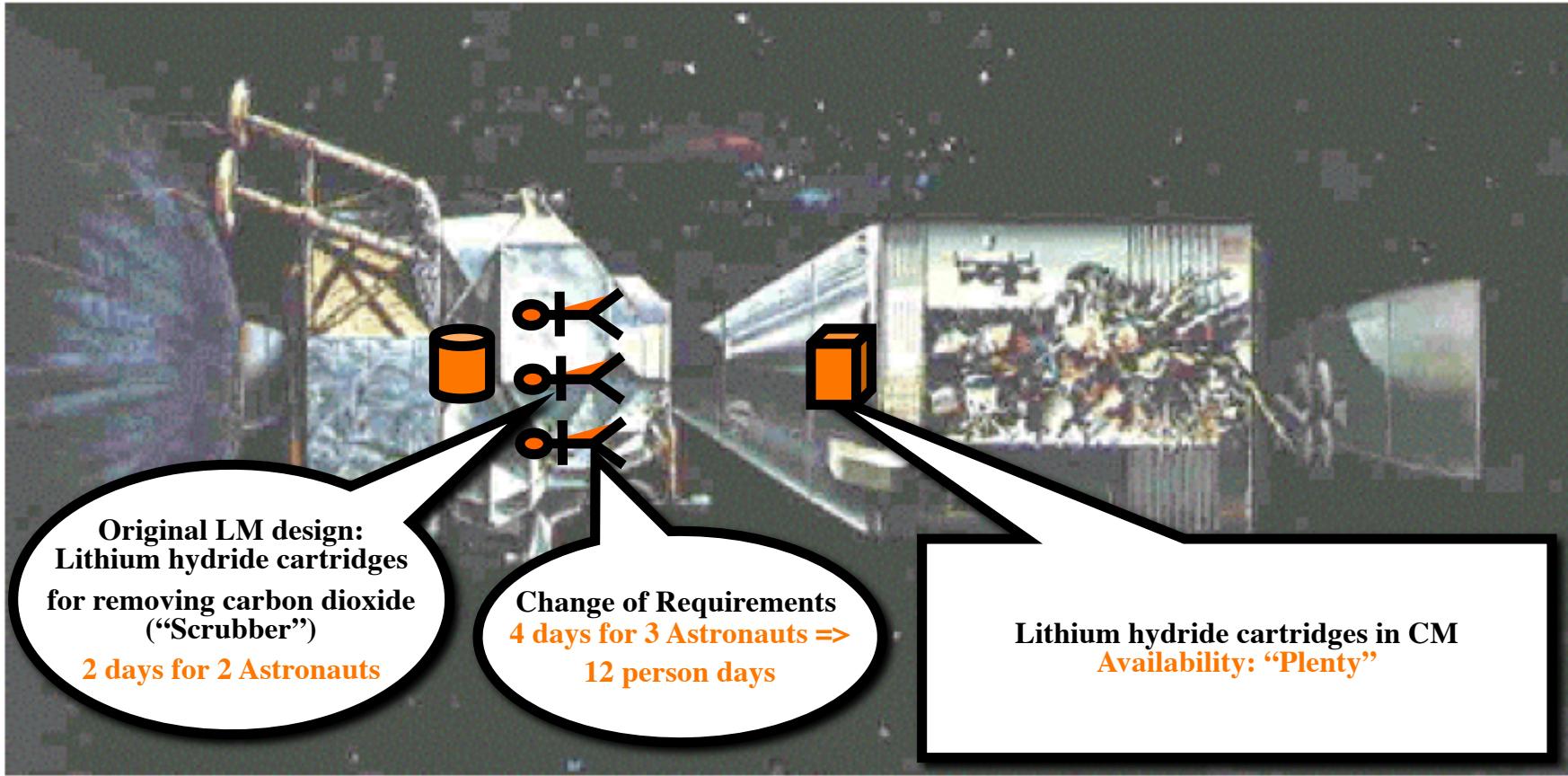
Another Adapter Pattern Example

“Houston, we’ve had a Problem!”



Subsystem Decomposition of the Apollo 13 Spacecraft

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



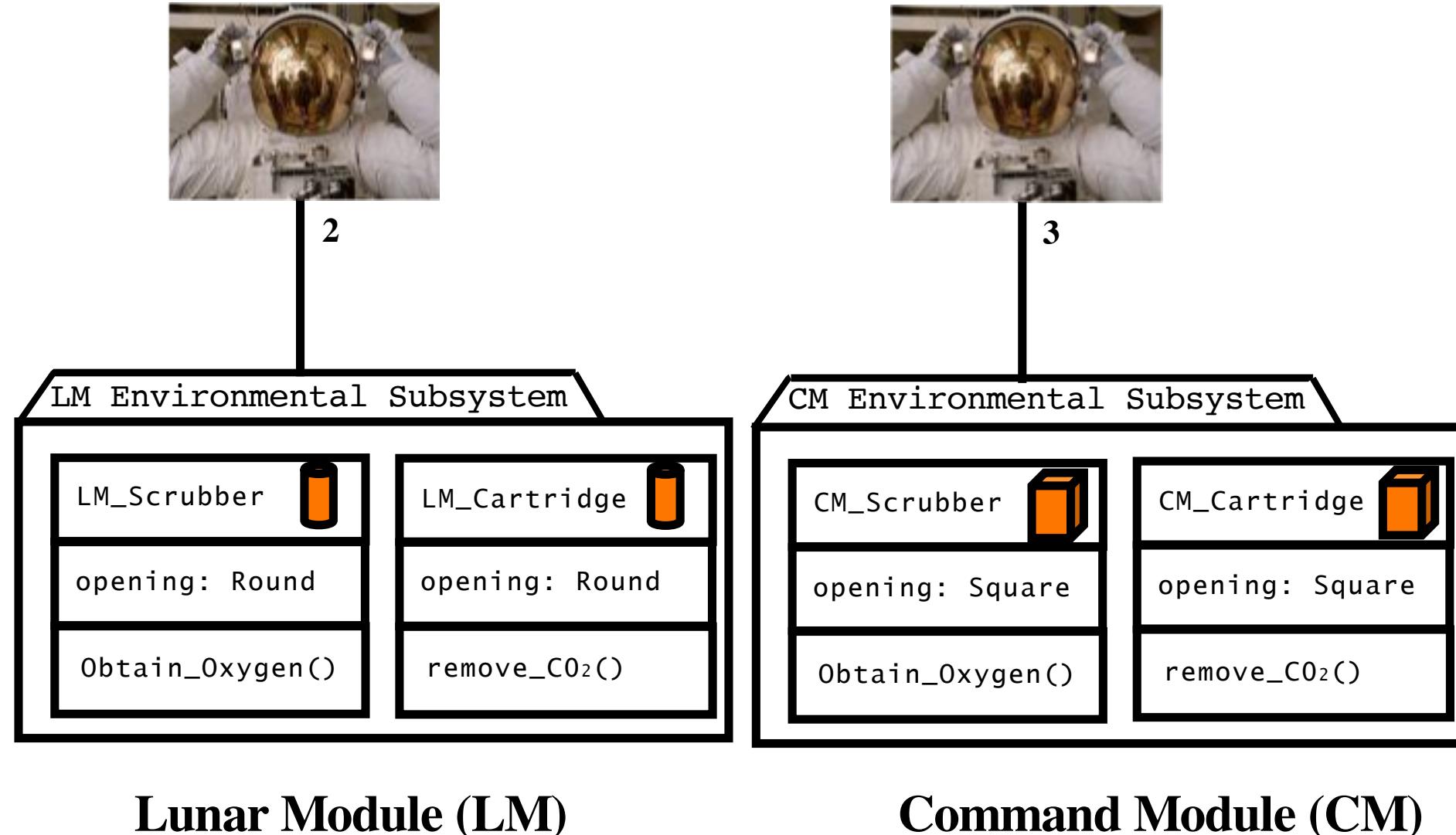
The LM was **designed** for 2 astronauts staying 2 days on the moon (4 person days)

Redesign challenge: How can the LM be used for 12 person days (reentry into Earth)?

Proposal from Mission Control : “Use the lithium hydride cartridges from the CM to extend life in LM”

Problem: Cartridges in CM are incompatible with the cartridges in the LM subsystem!

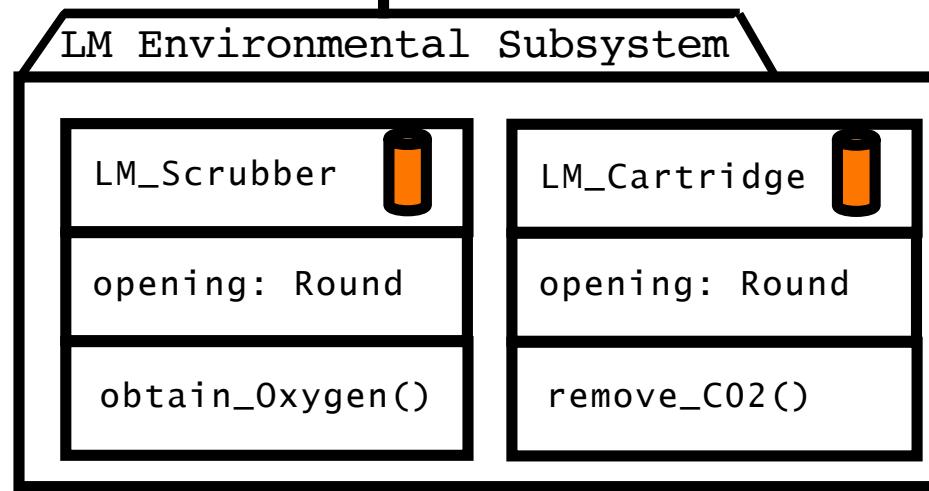
Original Design of the Apollo 13 Environmental System



Change!



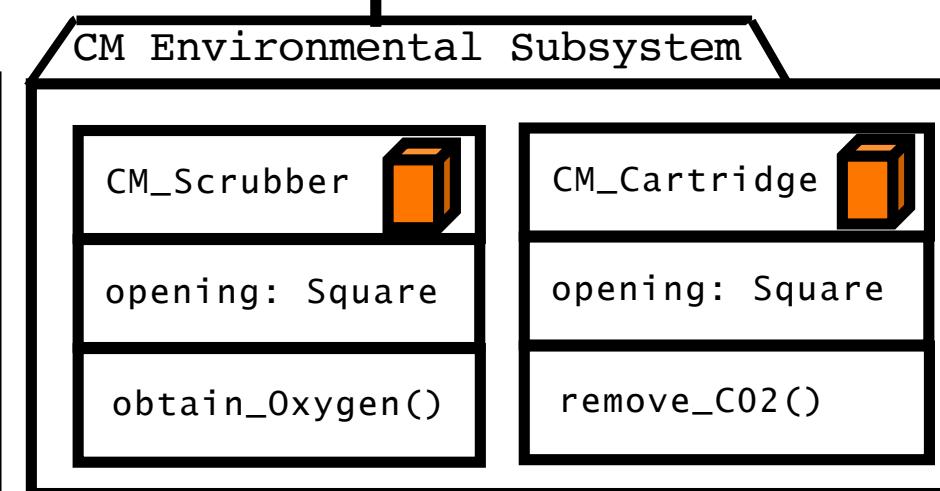
2



Lunar Module (LM)

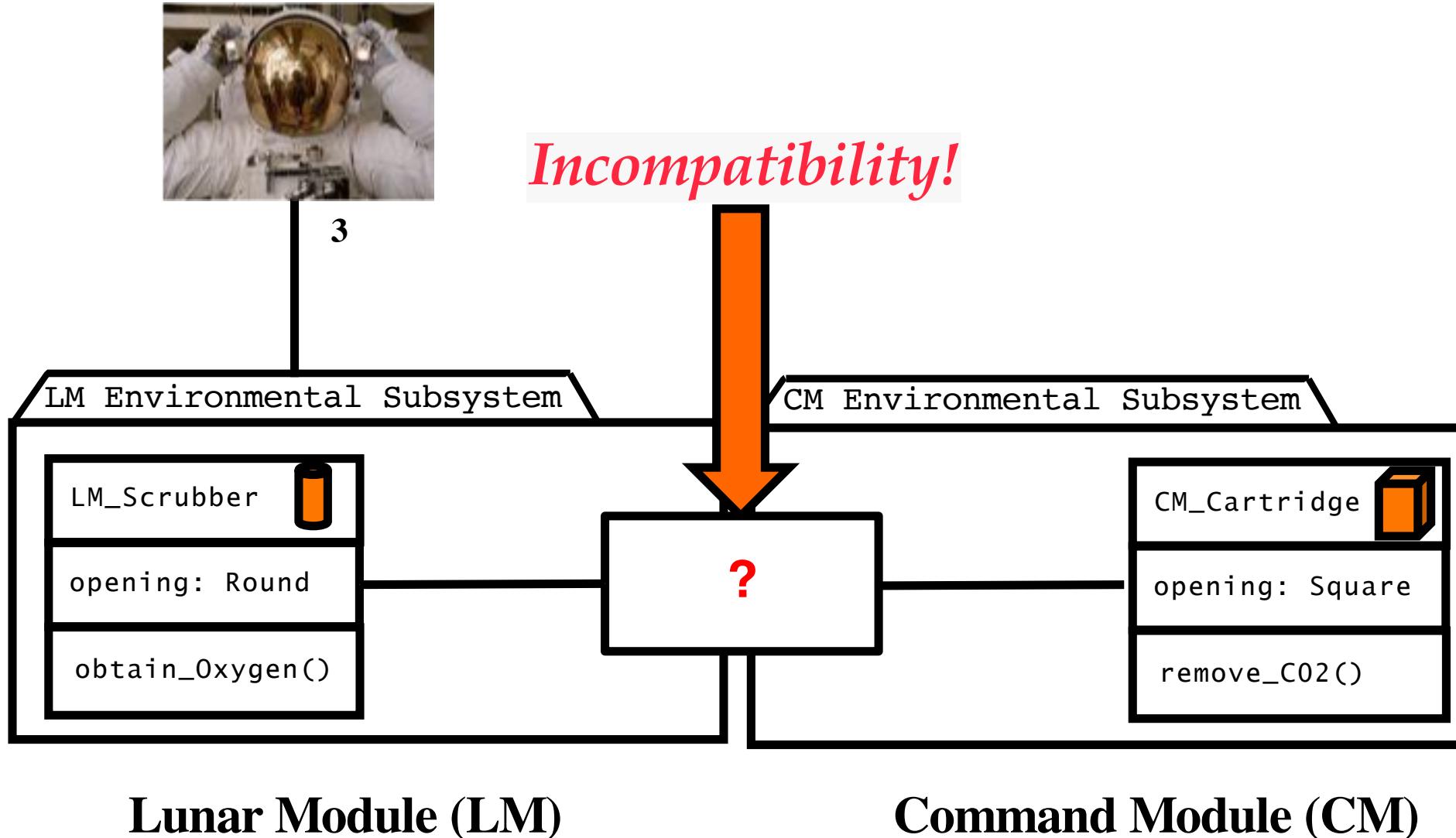


3



Command Module (CM)

Can we connect the LM Subsystem with the CM Subsystem?

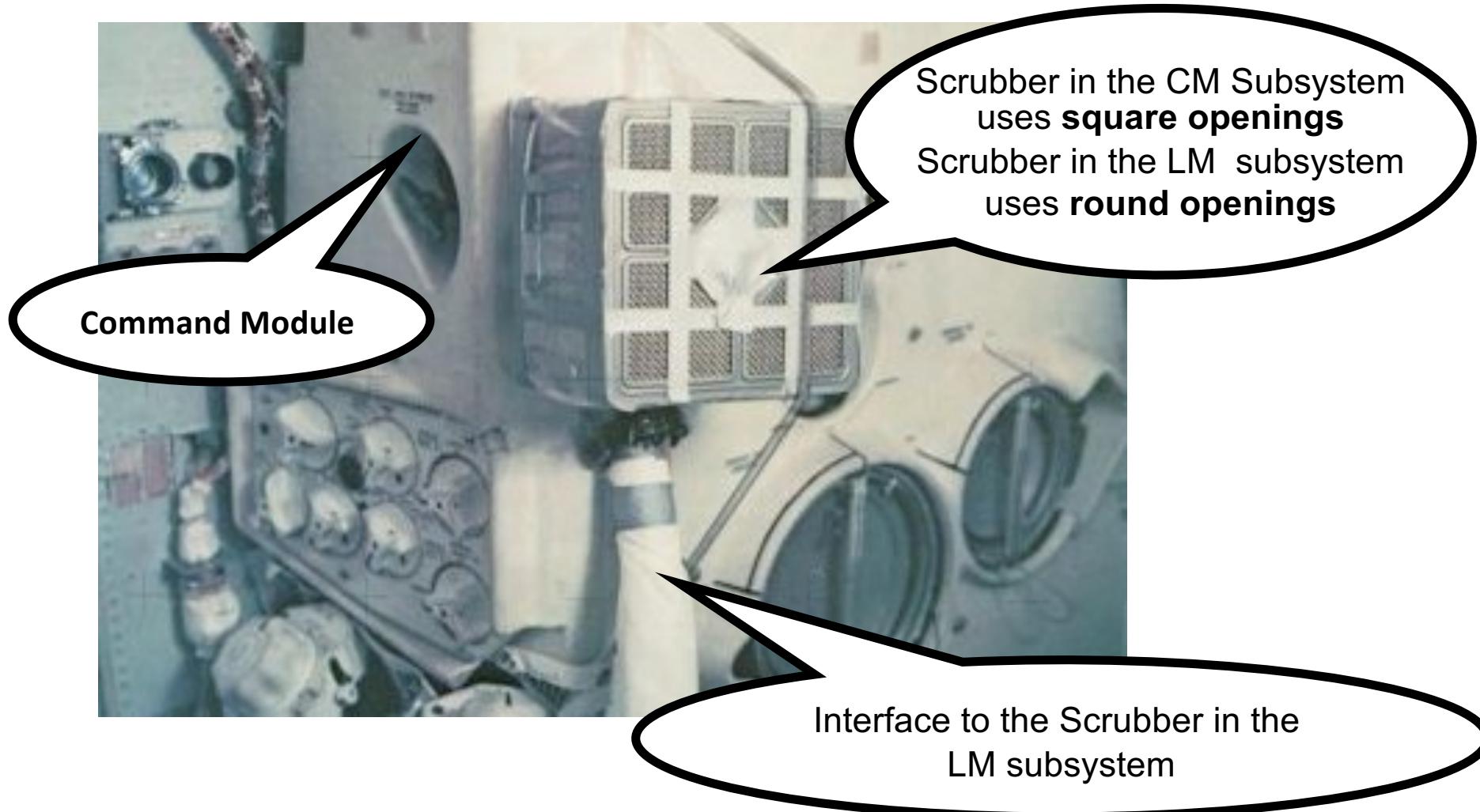


Apollo 13: “Fitting a square peg in a round hole”



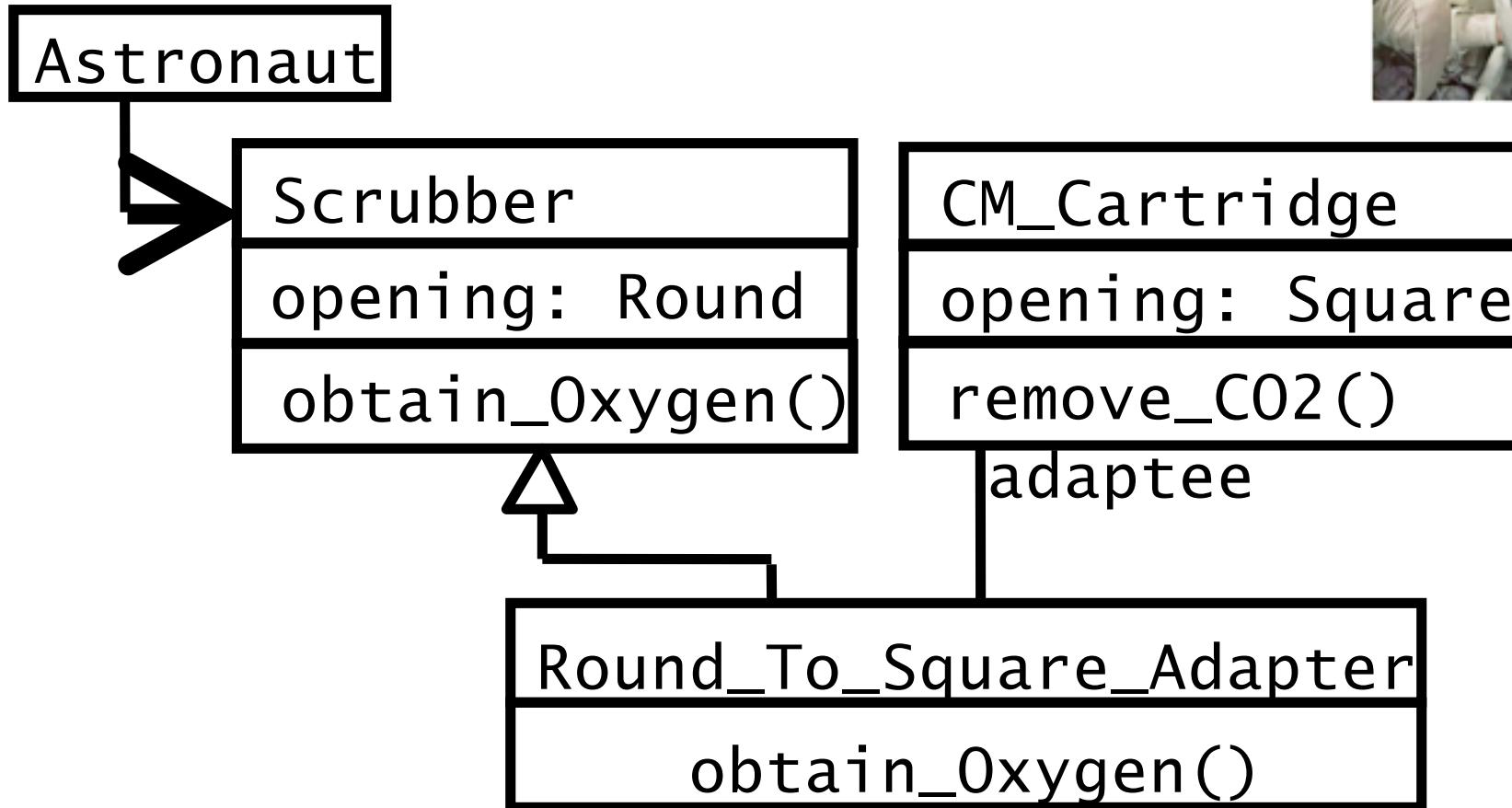
Source: <http://www.hq.nasa.gov/office/pao/History/SP-350/ch-13-4.html>

A Typical Object Design Challenge: Connecting Incompatible Components



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

Adapter for Scrubber in Lunar Module



- A carbon dioxide scrubber (round opening) in the lunar module using square cartridges from the command module (square opening).

Bumper's Backlog before Sprint 6

1. User Interface design of the game board
2. Cars drive on the game board
3. Cars collide with each other and each collision has a winning car
4. The winner of the game is the car that wins all the collisions
5. The player starts and stops the game
6. Music plays when the game begins and stops to play when the game ends
7. The game supports different car types
8. The player steers the car with the mouse
9. The player can change their car's speed
10. The game should be compatible with multiple platforms
11. The speed, consumption, and location of the player's car is visualized in an instrument panel
12. Collisions follow the "right before left" rule, and thus right-most cars win the collisions
13. A crash sound and/or animation plays when two cars collide
14. The game supports different collisions, the determination of the collision winner is changeable during game play
15. Implement a new type of collision
16. Extend the instrument panel with an analog speedometer

The customer can change the product backlog after any sprint

Duration
1 week

Homework Exercise 03: Sprint 6

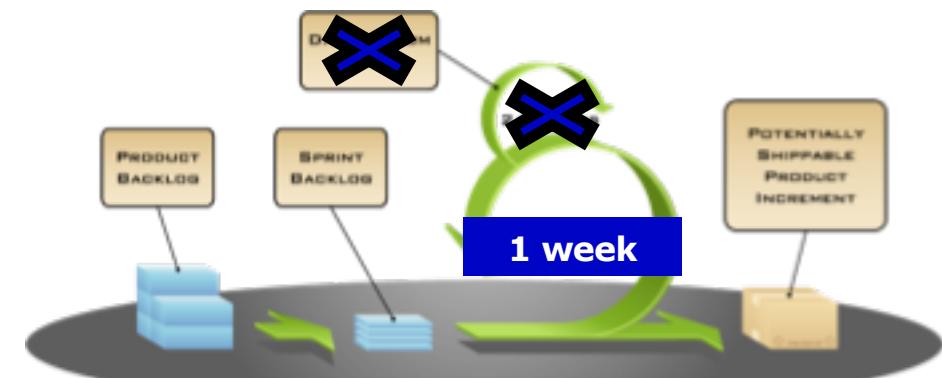
Sprint Backlog:

16. Extend the instrument panel with an analog speedometer

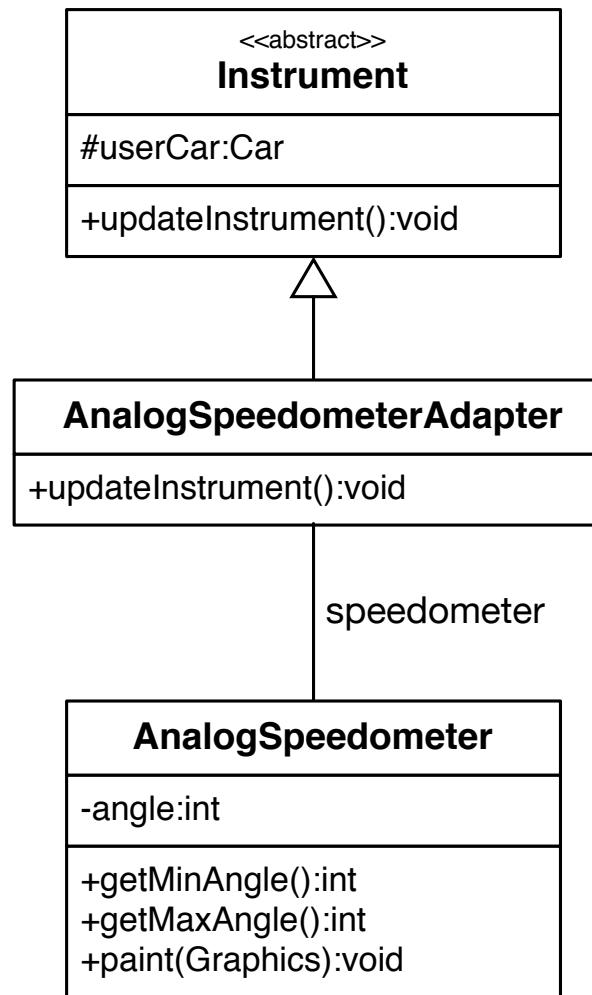
Additional Information:

- Bumpers Consulting provides you with an existing class AnalogSpeedometer - <https://repobruegge.in.tum.de/projects/FIST2018I02BUMPFRSS06/repos/eist2018-I08-bumpers-sprint06-speedometer/browse> (Short: <http://bit.ly/eistsprint05>)
- The analog speedometer is a legacy system and no longer be changed
- The interface of the analog speedometer does not fit into the actual design of Bumpers
- The analog speedometer does not follow the specification of the abstract Instrument class ("legacy code problem")

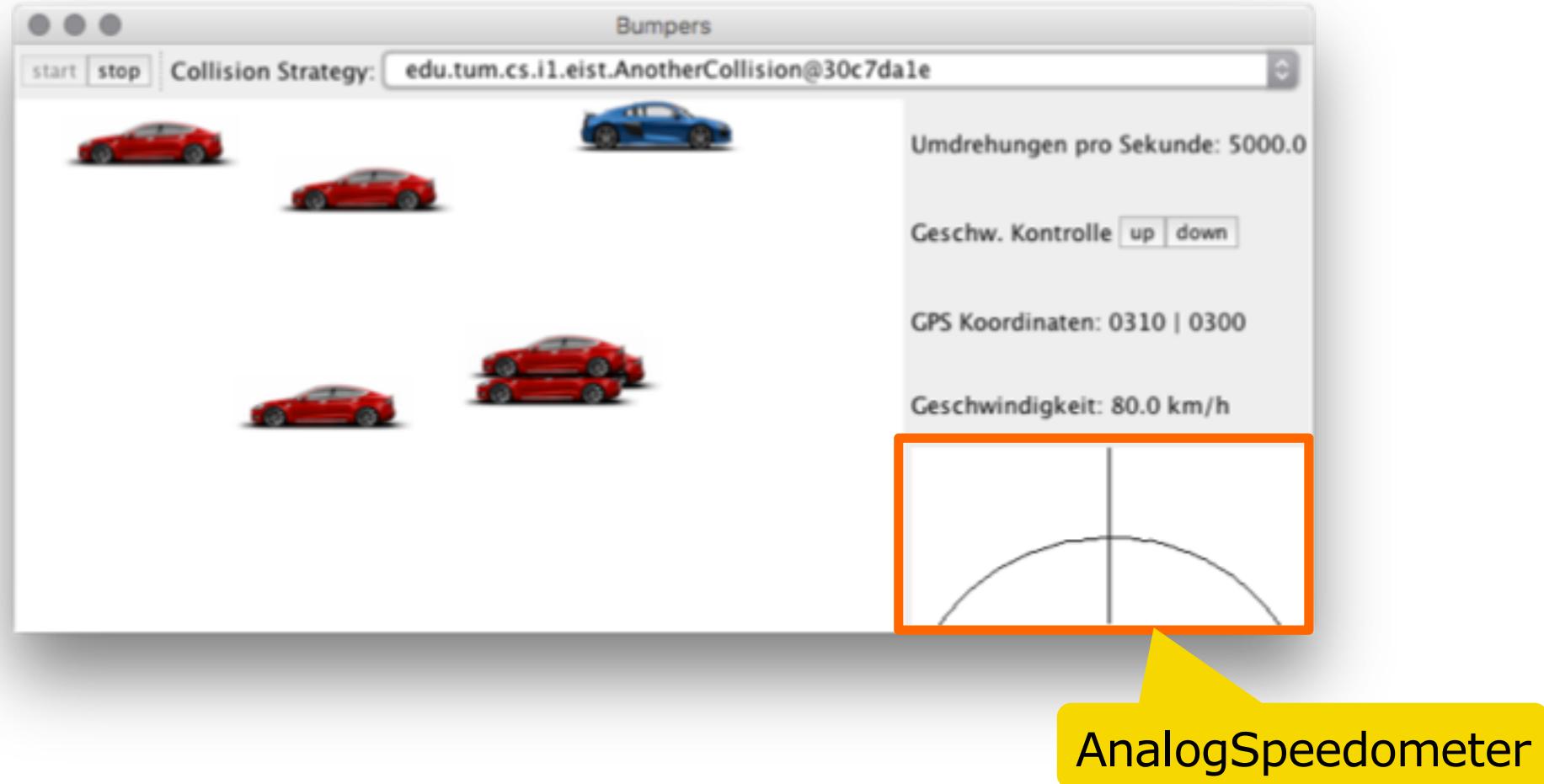
Start the exercise on ArTEMiS
starting Monday, June 18, noon



Hint: Using the Adapter Pattern for the Analog Speedometer

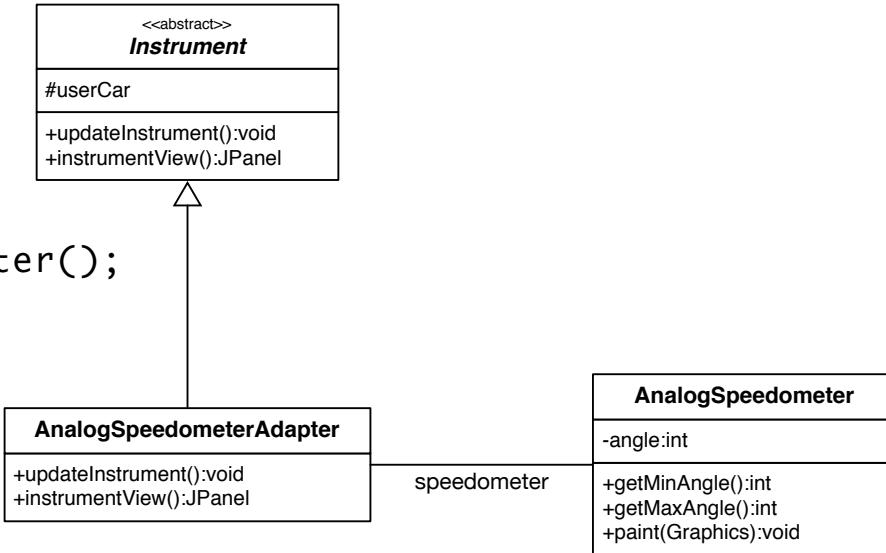


User Interface after Sprint 6

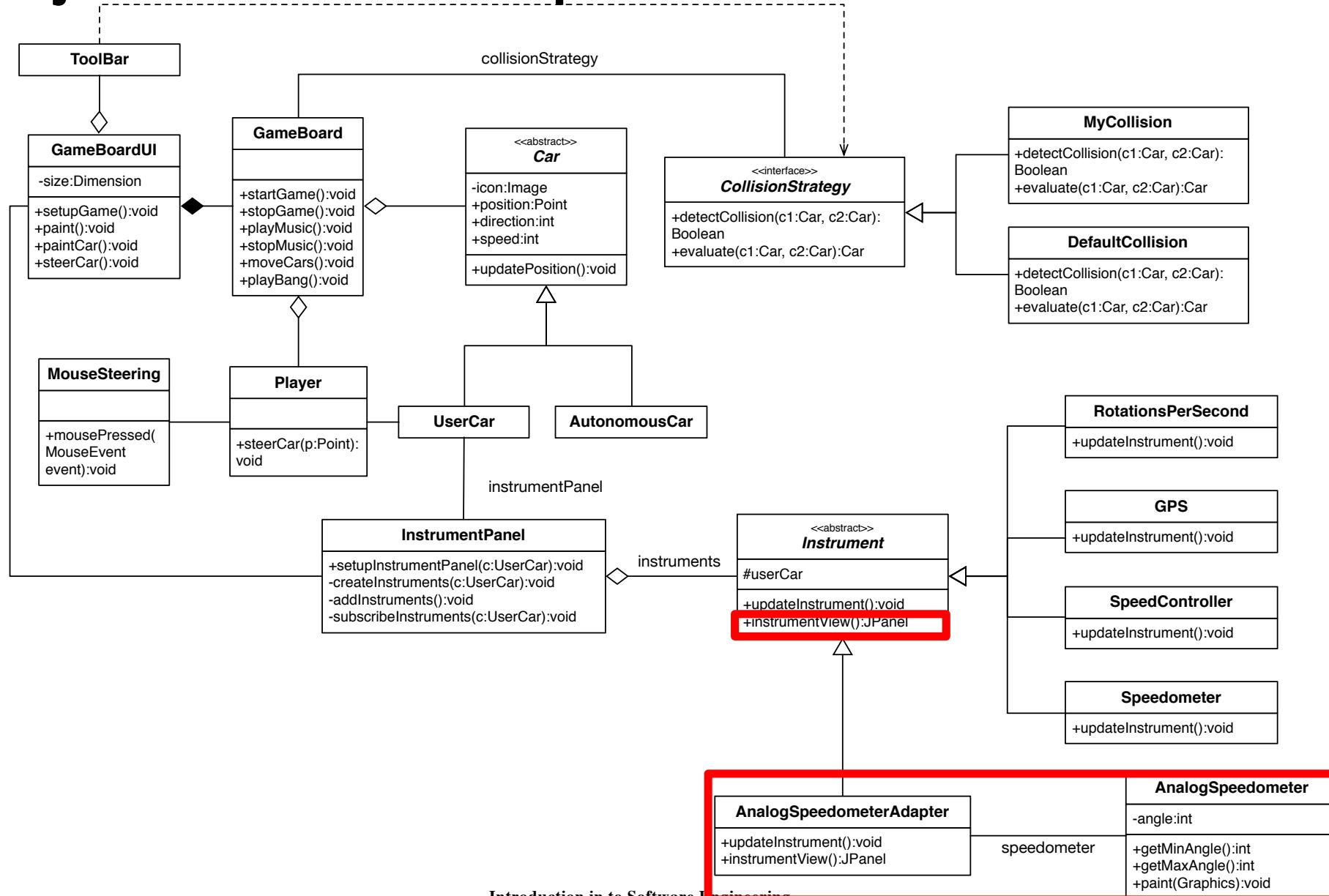


Hint: The AnalogSpeedometerAdapter

```
public class AnalogSpeedometerAdapter extends Instrument {  
    private AnalogSpeedometer speedometer = new AnalogSpeedometer();  
    private int speed;  
  
    public AnalogSpeedometerAdapter(UserCar userCar){  
        super(userCar);  
        updateInstrument();  
    }  
  
    @Override  
    public void updateInstrument() {  
        if(this.speed != car.getSpeed()){  
            this.speed = car.getSpeed();  
            double percent = (double)car.MAX_SPEED/100*this.speed;  
            int angle = (int)(speedometer.getMaxAngle()*percent);  
            this.speedometer.setAngle(angle);  
        }  
    }  
  
    @Override  
    public JPanel instrumentView() {  
        return speedometer;  
    }  
}
```



Hint: Object Model after Sprint 6



Summary

- We covered different pattern schemata
 - Alexander's original form, Gang of Four, Gang of Five
 - The EIST Pattern Scheme
- **Pattern-Based Development:** Applying Patterns throughout the Software Lifecycle. Model-Based Development that focuses on reuse and makes extensive use of patterns during analysis, system design and object design
 - Sprint 4: Application of the **Strategy Pattern**
 - Sprint 5: Application of the **Observer Pattern**
 - Sprint 6: Application of the **Adapter Pattern**
- **Pattern Coverage:** Every Model element and every element in the code is covered by a pattern. Desired: 100% Coverage

Readings

- **Design Patterns. Elements of Reusable Object-Oriented Software** – Gamma, Helm, Johnson & Vlissides
- **Pattern-Oriented Software Architecture, Volume 1, A System of Patterns** - Buschmann, Meunier, Rohnert, Sommerlad, Stal
- **Pattern-Oriented Analysis and Design** - Composing Patterns to Design Software Systems - Yacoub & Ammar
- <https://sourcemaking.com/>

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

Software Life Cycle Models

Bernd Bruegge
Chair for Applied Software Engineering
Technische Universität München

21 June 2018

Roadmap for Today's Lecture

- **Context and Assumptions**

We have completed Chapter 1 to 10 in the OOSE text book by Bruegge and Dutoit

- **Content of this lecture**

- Today we are jumping to Chapter 15: Software Life Cycle and also include concepts from Chapter 16: Methodology

- **Objective:** At the end of the lecture you understand

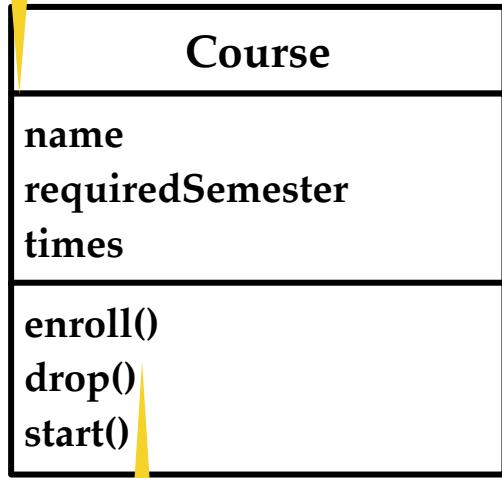
- Life cycle modeling
- Various models: waterfall model, spiral model, unified process
- The difference between activity- and entity-oriented models
- The difference between iterative, incremental and adaptive development
- The difference between defined and an empirical process control

Miscellaneous

- The distinction between Analysis Model vs Object Design Model
- Exam Information

Distinction between Analysis Object Model and Object Design Model

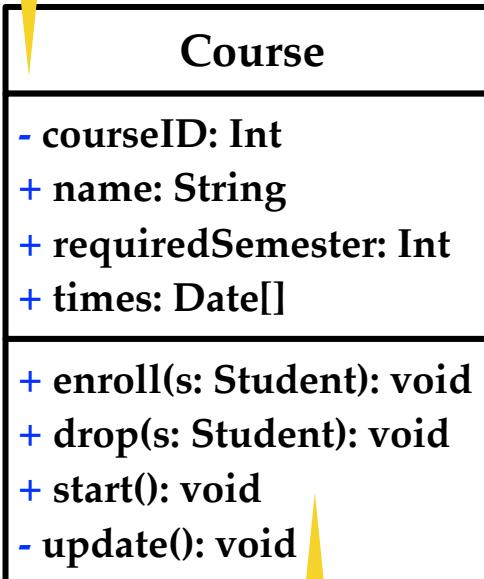
During analysis: attributes and methods without visibility information



During analysis: methods without signature

Analysis Object Model
(Application Domain Language)

During object design: we specify the visibility for each attribute and method



During object design: specify the signature of each method

Object Design Model
(Solution Domain Language)

```
public class Course {
    private Integer courseID;
    public String name;
    public Integer requiredSemester;
    public Date[] times;
    ...
    public void enroll(Student s)
    {...}
    public void drop(Student s) {...}
    public void start() {...}
    private void update() {...}
}
```

type

signature

visibility

Implementation
(Java)

Final Exam

- Registration via TUM Online until June 30 (strict deadline)
- You can bring one **handwritten** DIN A4 sheet of paper
 - The sheet has to be written by you (we will verify this during the exam)
 - Both sides can be used
- Bonus only counts for the final exam and **cannot** be applied to the repeat exam.

Morning Quiz 09

- Start Time: **8:10**
- End Time: **8:20**
- To participate in the quiz, use ArTEMiS
- Click on Open Quiz or Start Quiz
- The Lecture starts at 8:10

Remaining Time: **46 s**

Saved: never

● Connected

Submit

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	Start exercise
<u>Good Morning Quiz 09</u>		Open Quiz

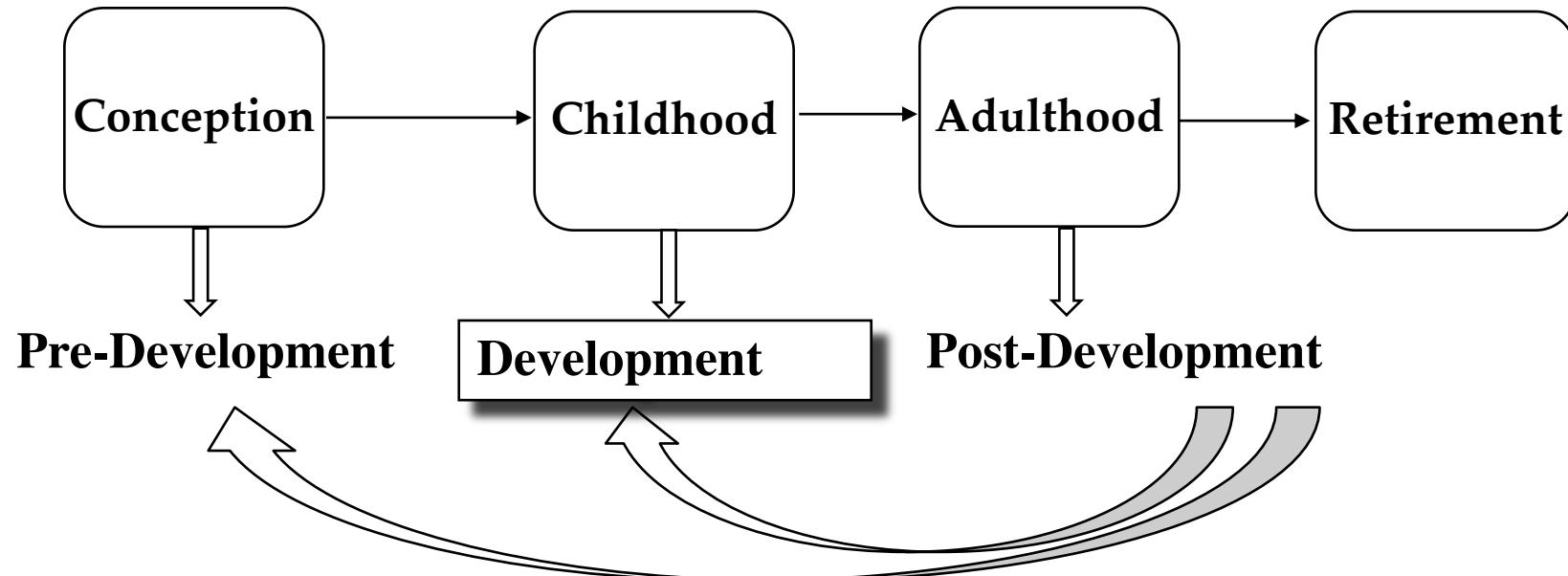
Only click on Submit when you have entered all answers!

Outline of the Lecture

- Software Development as Application Domain
 - Modeling the software lifecycle
- IEEE Standard 1074 for Software Lifecycles
- Software life cycle models
 - Sequential models
 - Waterfall model, V-model
 - Iterative models
 - Boehm's spiral model, Unified Process
 - Entity-oriented models
 - Scrum
- Process Control Models
 - Frequency of Change
 - Noise in communication
 - Methodology Issues
 - Defined process control model
 - Empirical process control model

Software Life Cycle

- The term “Lifecycle” is based on the metaphor of the life of a person:



Definitions

- **Software life cycle:**

- Set of activities and their relationships to each other to support the development of a software system

Examples of activities:

- Requirements Elicitation, Analysis, System Design, Implementation, Testing, Configuration Management, Delivery

Examples of relationships:

- Testing must be done before Implementation
- Analysis and System Design can be done in parallel

- **Software life cycle model:**

- An abstraction representing the development of software for the purpose of understanding, monitoring, or controlling the software.

Typical Software Life Cycle Questions



Which activities should we select?

- What are the *dependencies between activities*?
- How should we *schedule the activities*?

Software Development Activities – Example

Requirements Analysis

What is the problem?

Application
Domain

System Design

What is the solution?

Object Design

What are the best mechanisms
to implement the solution?

Implementation

How is the solution
constructed?

Solution
Domain

Testing

Is the problem solved?

Delivery

Can the customer use the solution?

Maintenance

Are enhancements needed?

Lifecycle Activities in EIST

Requirements
Elicitation

Analysis

System
Design

Object
Design

Implemen-
tation

Testing

Software Lifecycle Activities in EIST

...and their Models

Chapter 4

Requirements
Elicitation

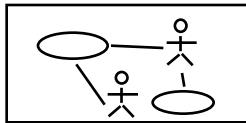
Analysis

System
Design

Object
Design

Implemen-
tation

Testing



Use Case
Model

Software Lifecycle Activities in EIST

...and their Models

Chapter 4

Chapter 5

Requirements
Elicitation

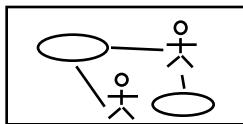
Analysis

System
Design

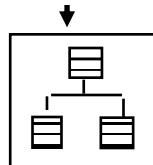
Object
Design

Implemen-
tation

Testing



Expressed in
terms of



Use Case
Model

Application
Domain
Objects

Software Lifecycle Activities in EIST

...and their Models

Chapter 4

Chapter 5

Chapter 6-7

Requirements
Elicitation

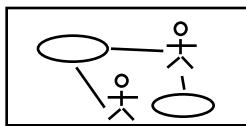
Analysis

System
Design

Object
Design

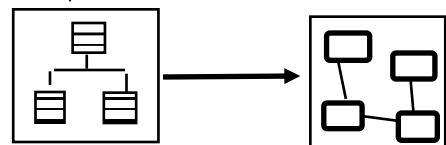
Implemen-
tation

Testing



Expressed in
terms of

Structured
by



Use Case
Model

Application
Domain
Objects

Sub-
systems

Software Lifecycle Activities in EIST

...and their Models

Chapter 4

Chapter 5

Chapter 6-9

Requirements
Elicitation

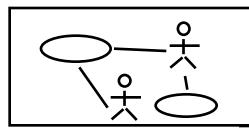
Analysis

System
Design

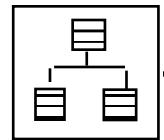
Object
Design

Implemen-
tation

Testing

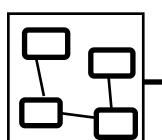


Expressed in
terms of



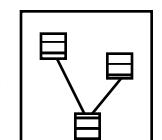
Use Case
Model

Structured
by



Application
Domain
Objects

Realized by



Sub-
systems

Solution
Domain
Objects

Software Lifecycle Activities in EIST

...and their Models

Chapter 4

Chapter 5

Chapter 6-9

Chapter 10

Requirements
Elicitation

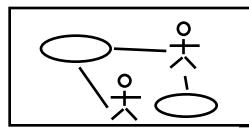
Analysis

System
Design

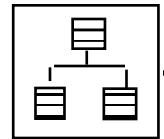
Object
Design

Implemen-
tation

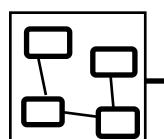
Testing



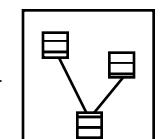
Expressed in
terms of



Structured
by



Realized by



Implemented by

class...
class...
class...

Use Case
Model

Application
Domain
Objects

Sub-
systems

Solution
Domain
Objects

Source
Code

Software Lifecycle Activities in EIST

...and their Models

Chapter 4

Chapter 5

Chapter 6-9

Chapter 10

Chapter 11

Requirements
Elicitation

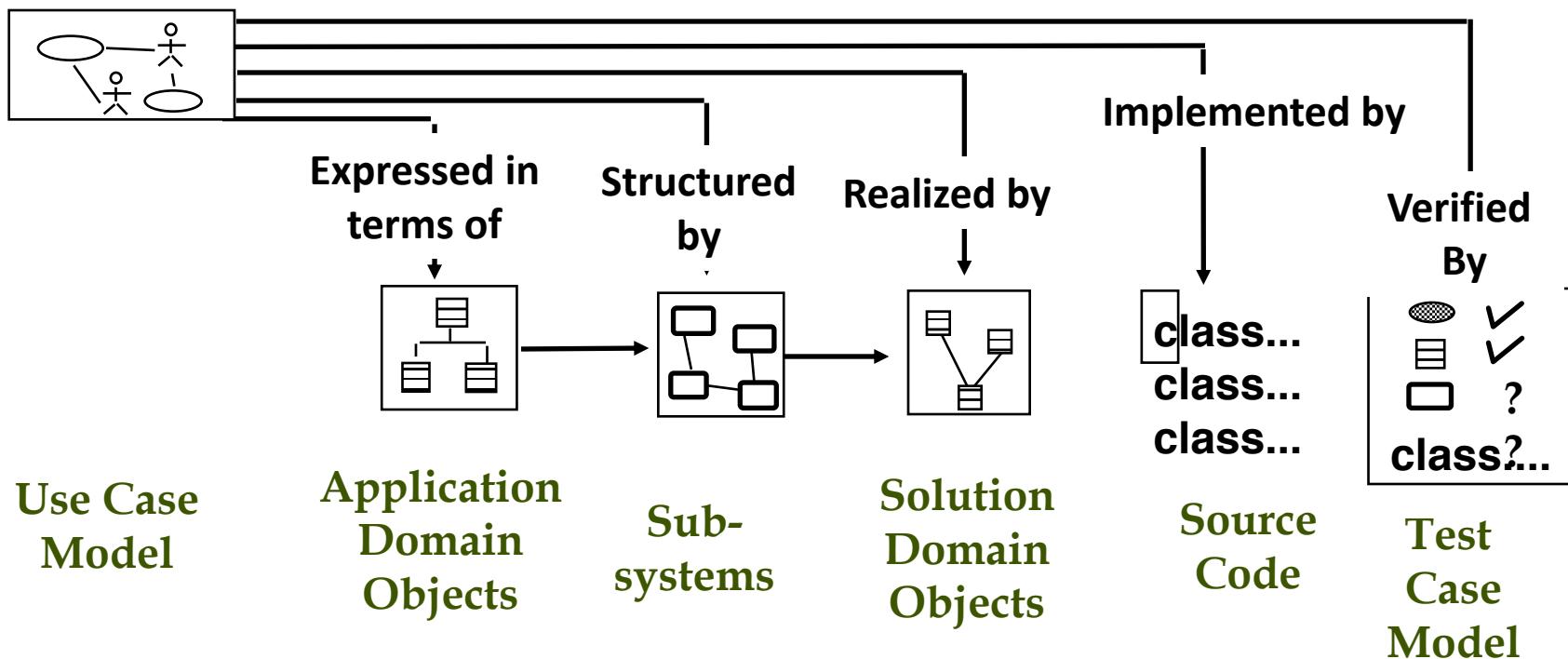
Analysis

System
Design

Object
Design

Implemen-
tation

Testing



Software Lifecycle Activities and their Models in EIST

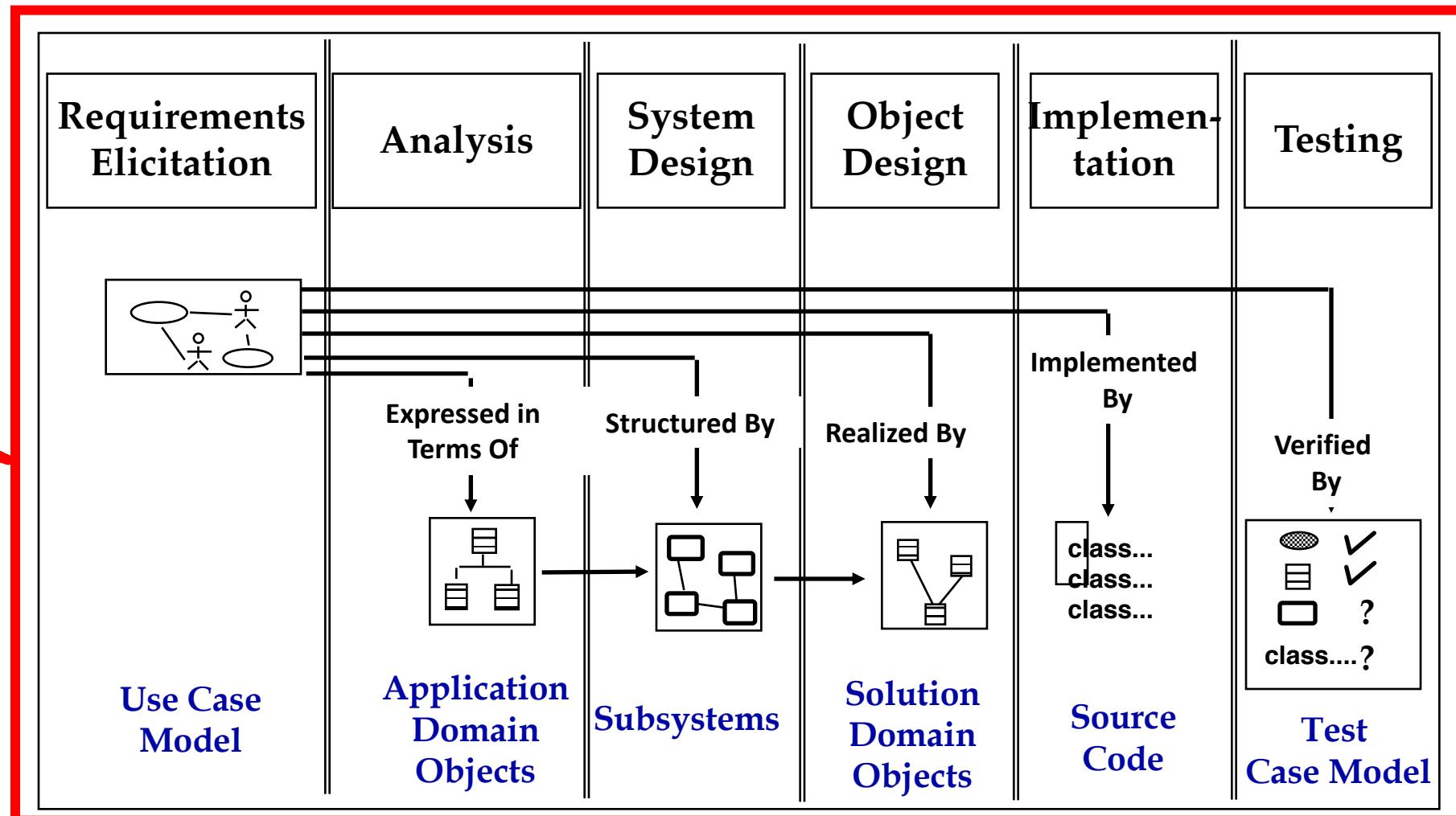
Chapter 4

Chapter 5

Chapter 6 to 9

Chapter 10

Chapter 11



Software Engineering as a System

- As Software engineers, we have developed methods and notations to model complex systems
- The discipline of Software Engineering itself can be seen as such a complex system
- We will now apply methods and notations to model software development
 - We use UML to illustrate software development as a complex system

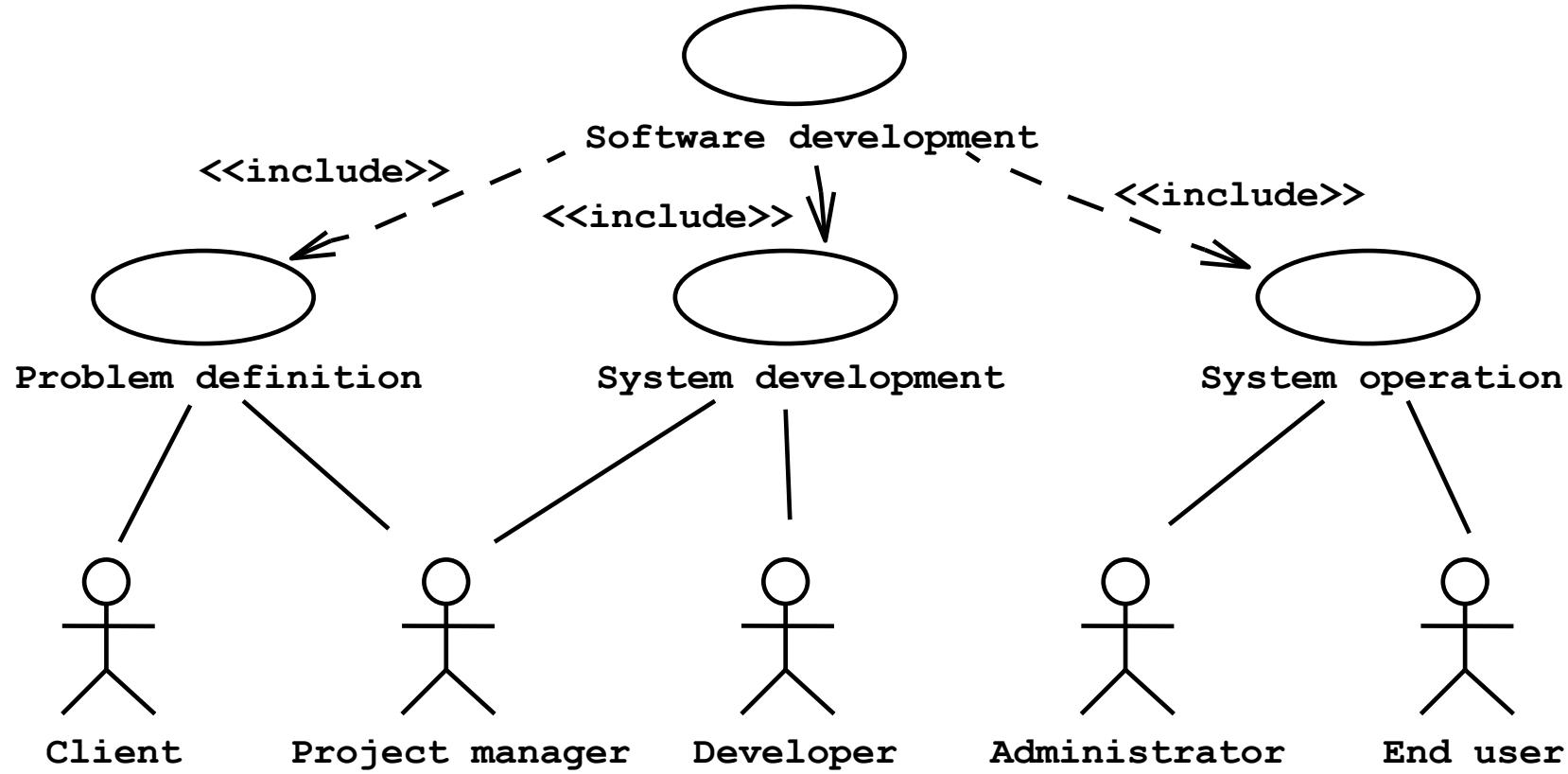


Norman Rockwell – “Triple Self Portrait”

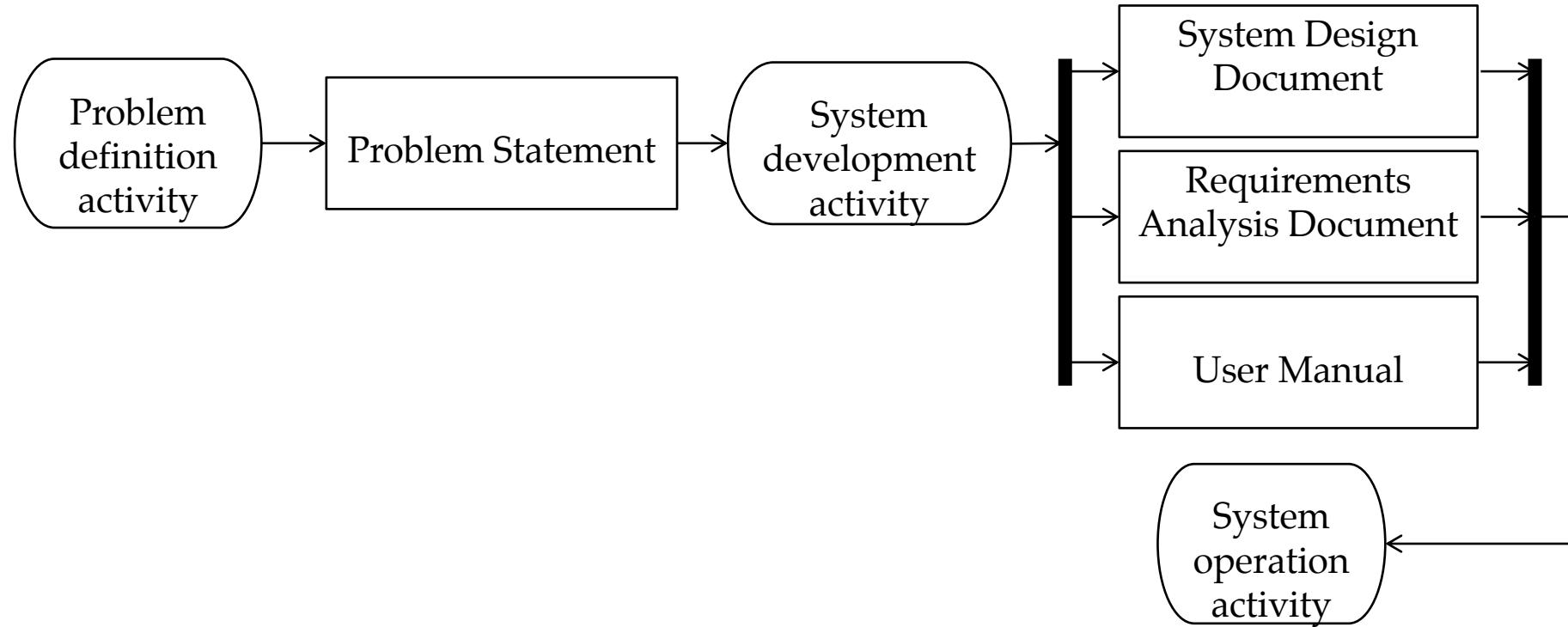
Modeling a Software Lifecycle

- We can use the same modeling techniques we use for software development:
 - Functional Model of a Software Lifecycle
 - Scenarios
 - Use case diagrams
 - Structural model of a Software Lifecycle
 - Class diagrams
 - Instance Diagrams
 - Dynamic Model of a Software Lifecycle
 - Sequence diagrams, statechart and activity diagrams

Use Case Diagram of a Simple Life Cycle Model



Activity Diagram for the same Life Cycle Model

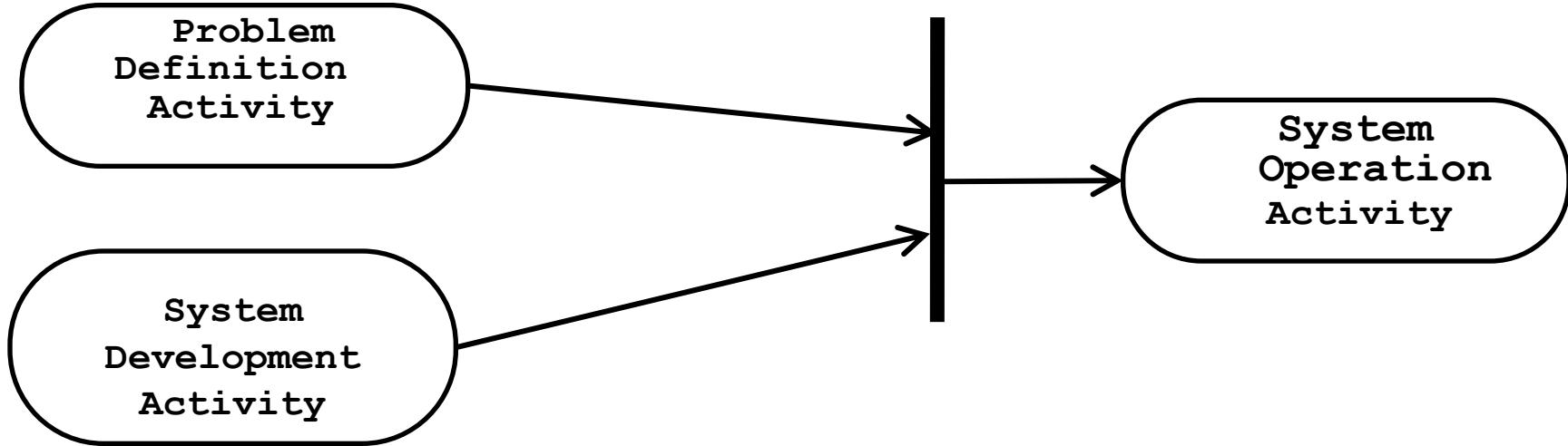


In this model, software development goes through a linear progression of states called software development activities.

Two Major Views of the Software Life Cycle

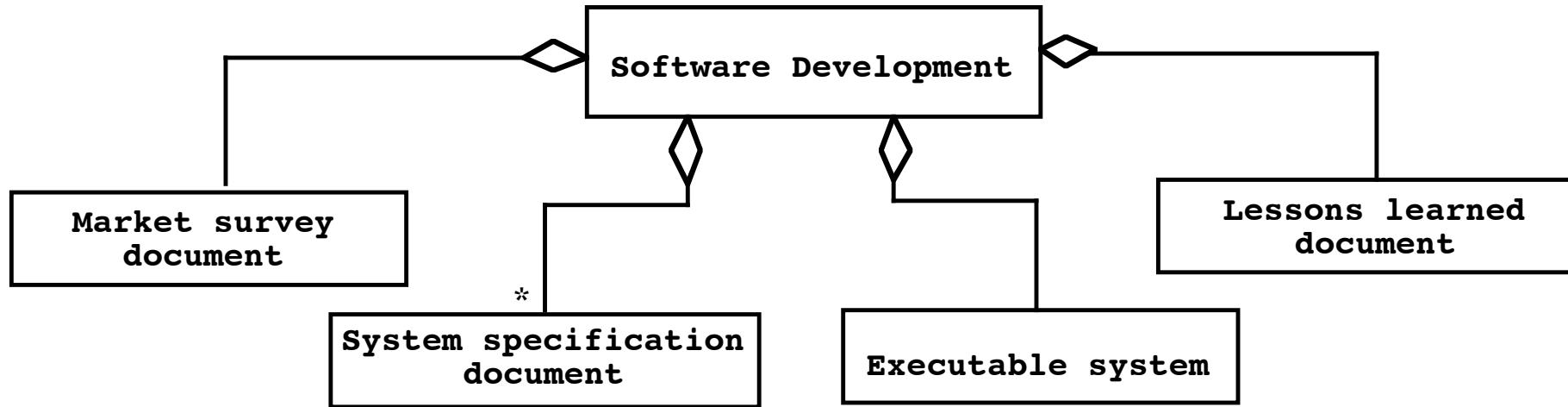
- **Activity-oriented view** of a software life cycle
 - Software development consists of a set of development activities
- **Entity-oriented view** of a software life cycle
 - Software development consists of the creation of a set of deliverables.

Activity-Oriented View of Software Development



Problem Definition and System Development can be done in parallel.
They must be done before the System Operation activity.

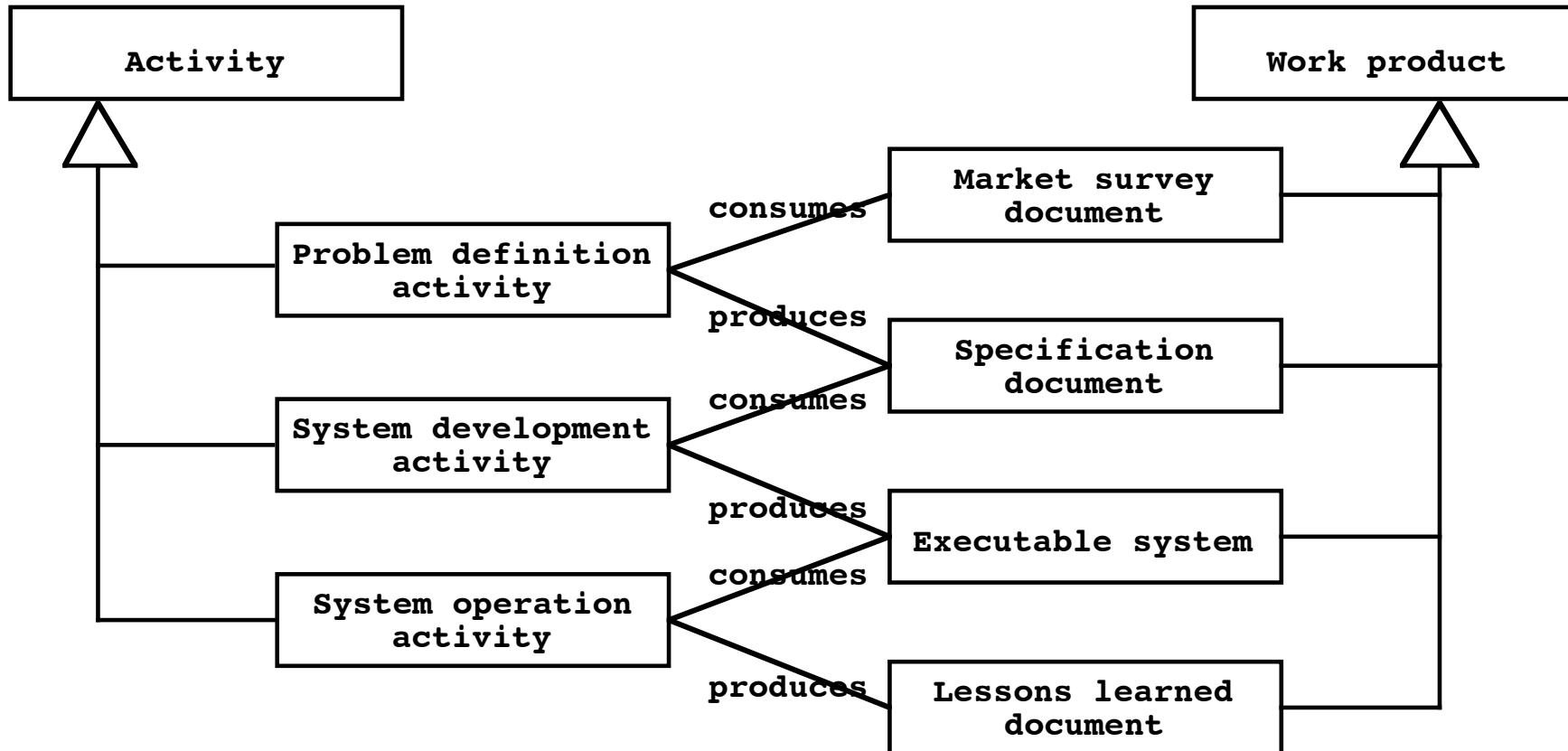
Entity-Oriented View of Software Development



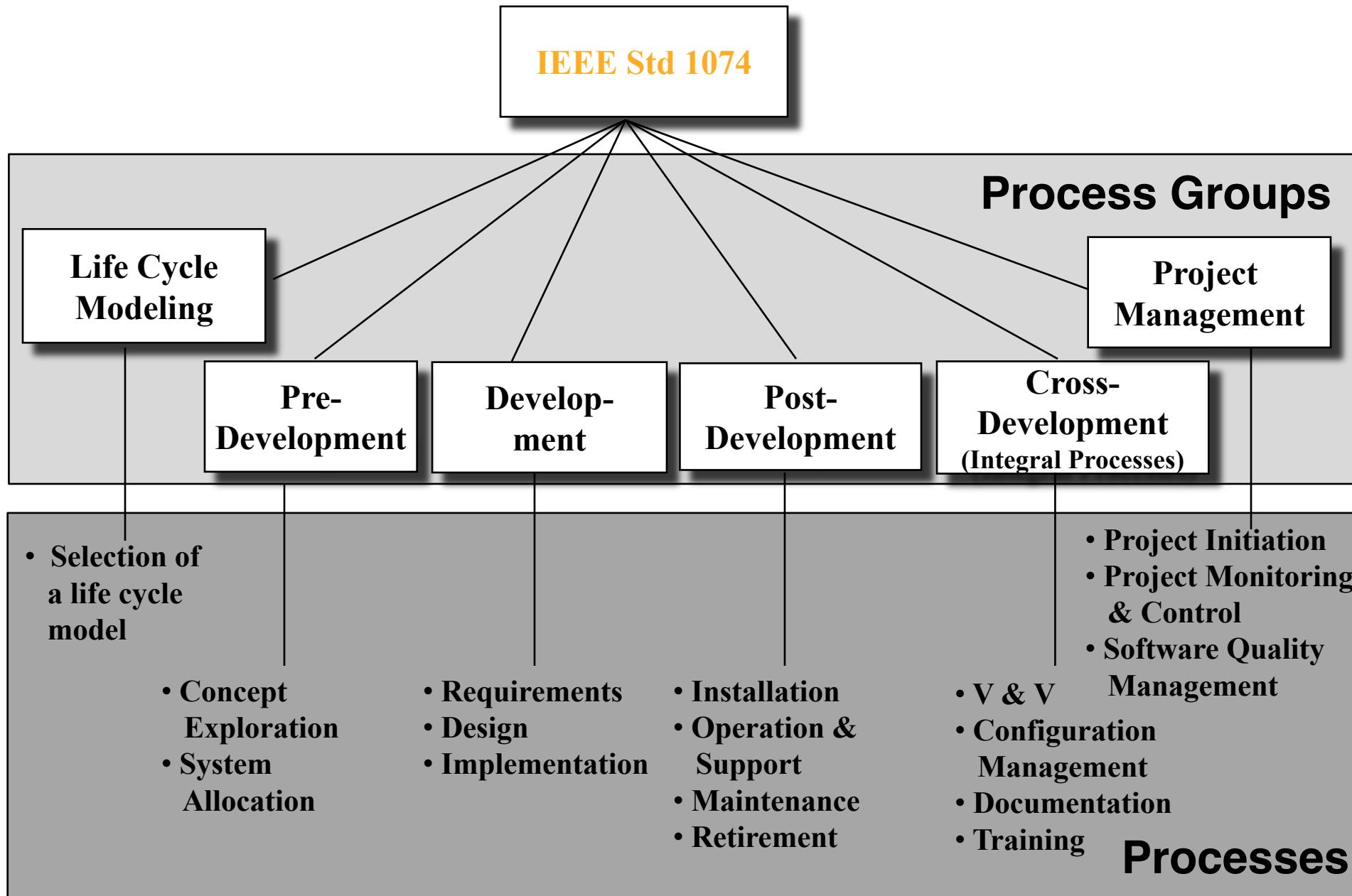
Software development consists of the creation of the following deliverables

- 1) Market survey
- 2) System specification documents
- 3) Executable system
- 4) Lessons learned document.

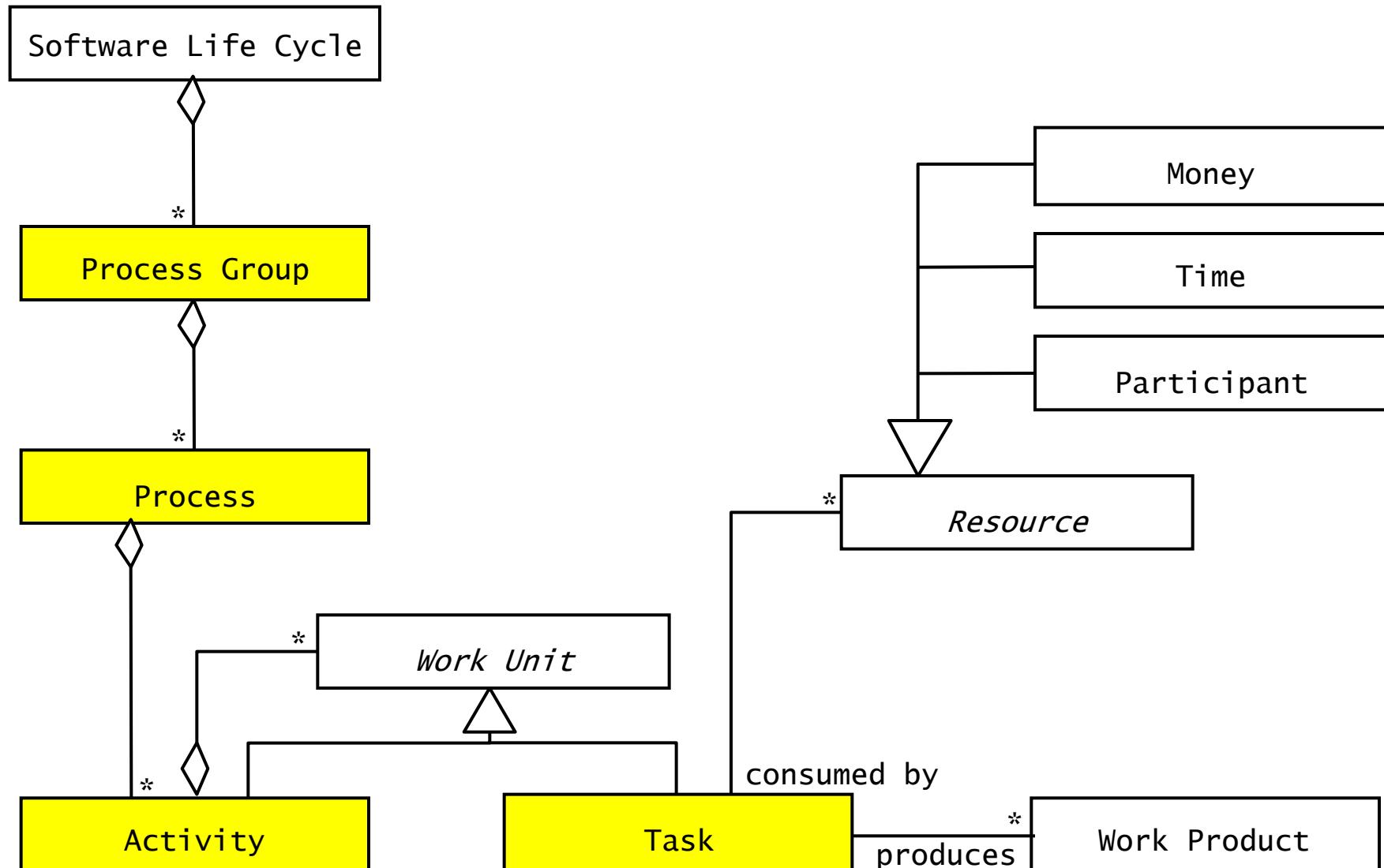
Combining Activities and Entities in a Class Diagram



IEEE Std 1074: Standard for Software Life Cycle Activities

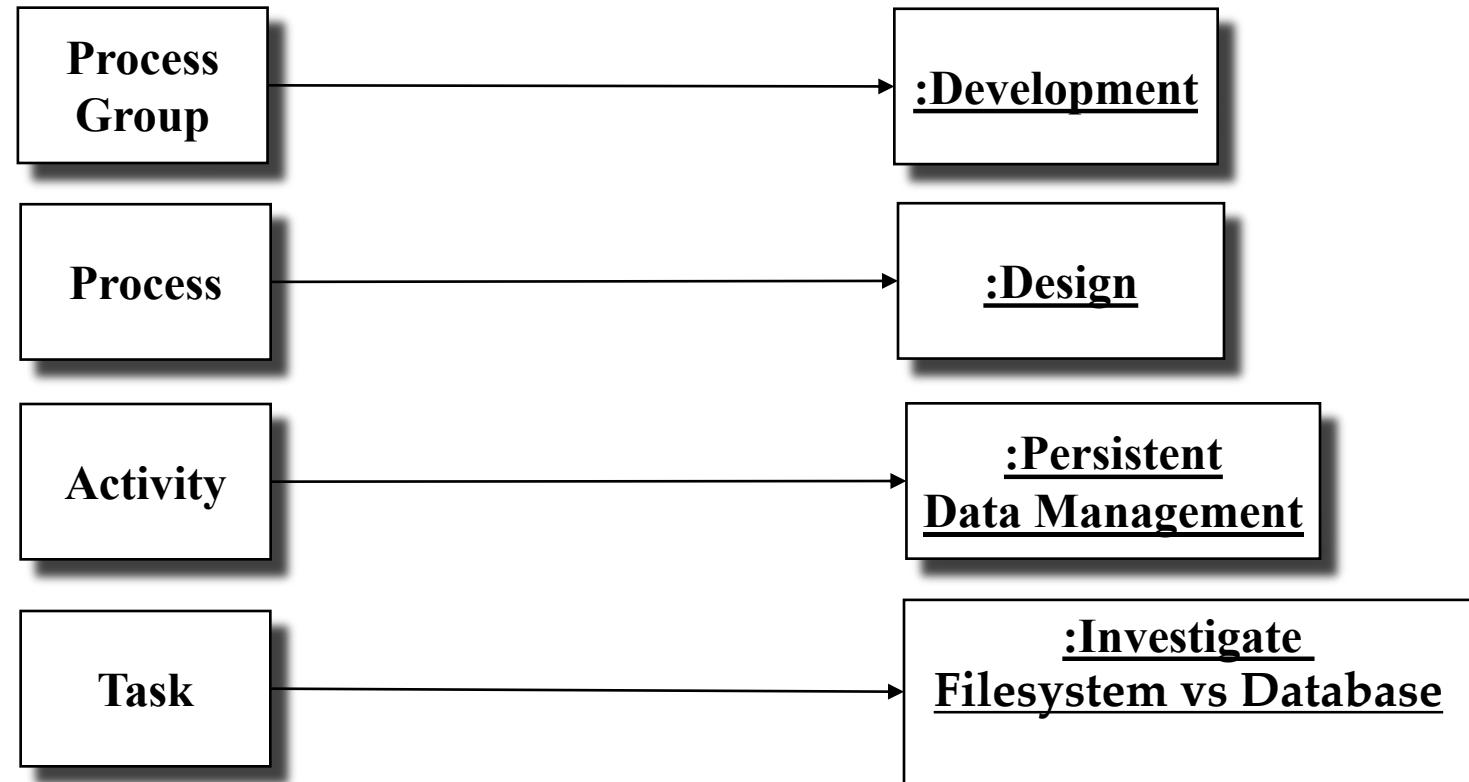


Object Model of the IEEE 1074 Standard

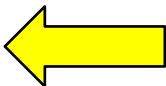


Processes, Activities and Tasks in IEEE 1074

- Process Group: Consists of a set of processes
- Process: Consists of activities
- Activity: Consists of sub activities and tasks



Development



Process Group

System Design



Process

1. Identify Design Goals

Additional NFRs
Trade-offs

2. Subsystem Decomposition

Layers vs Partitions
Architectural Style
Coherence & Coupling

3. Identify Concurrency

Identification of Parallelism
(Processes,
Threads)

4. Hardware/ Software Mapping

Identification of Nodes
Special Purpose Systems
Buy vs Build
Network Connectivity

Activity

5. Persistent Data Management

Storing Persistent
Objects
Filesystem vs
Database

8. Boundary Conditions

Initialization
Termination
Failure.

7. Software Control

Monolithic
Event-Driven
Conc. Processes

6. Global Resource Handling

Access Control
ACL vs Capabilities
Security



Task

Tailoring

- There is no “one size fits all” software lifecycle model that works for all possible software engineering projects
- Tailoring: Adjusting a lifecycle model to fit a project
 - Naming: Adjusting the naming of activities
 - Cutting: Removing activities not needed in the project
 - Ordering: Defining the order of the activities.

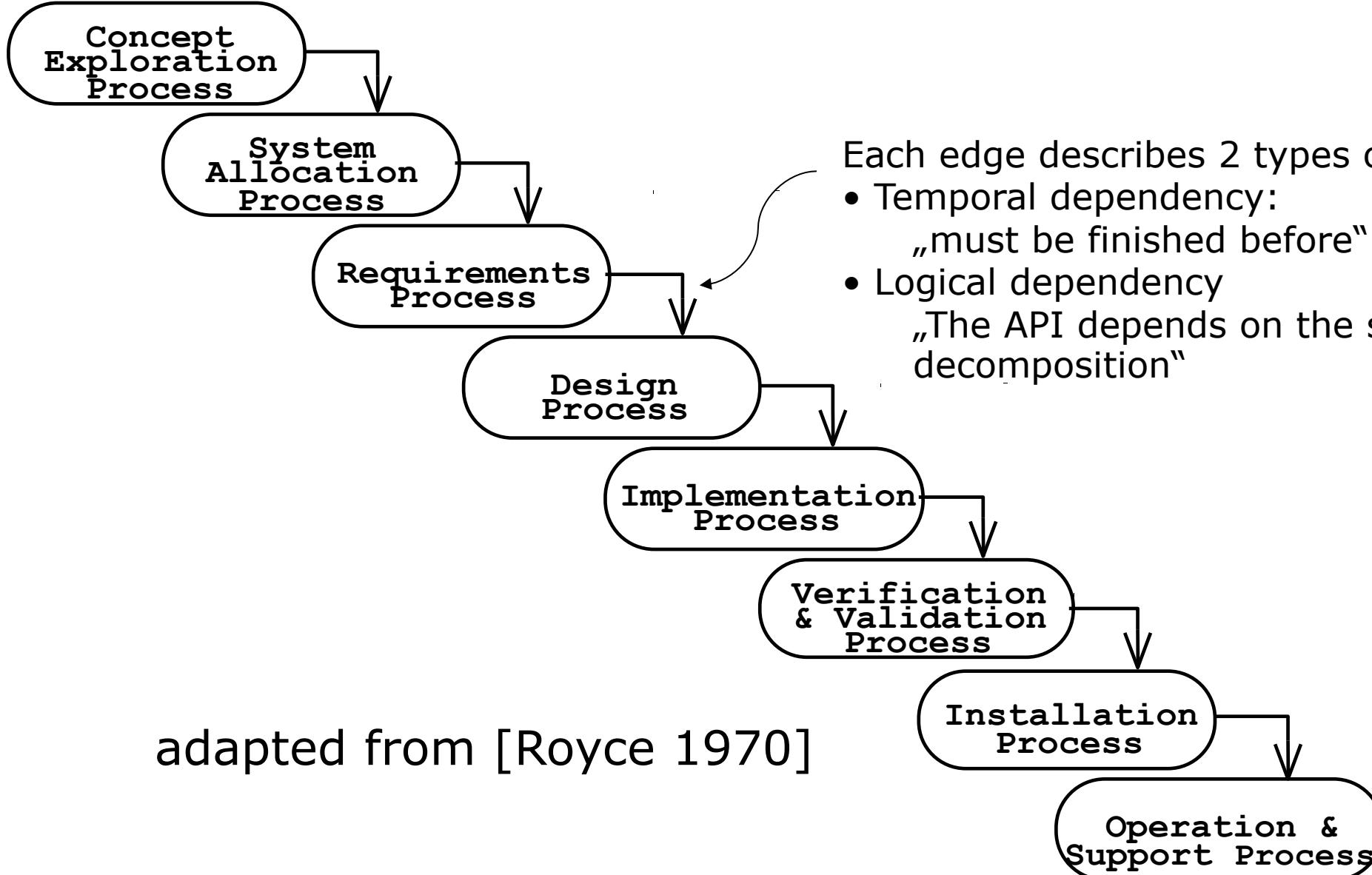
Lecture Roadmap

- Software Development as Application Domain
 - Modeling the software lifecycle
- IEEE Standard 1074 for Software Lifecycles
- Software life cycle models
 - Sequential models
 - Waterfall model, V-model
 - Iterative models
 - Boehm's spiral model, Unified Process
 - Entity-oriented models
 - Scrum
- Process Control Models
 - Noise in communication
 - Methodology Issues
 - Defined process control model
 - Empirical process control model

Waterfall Model

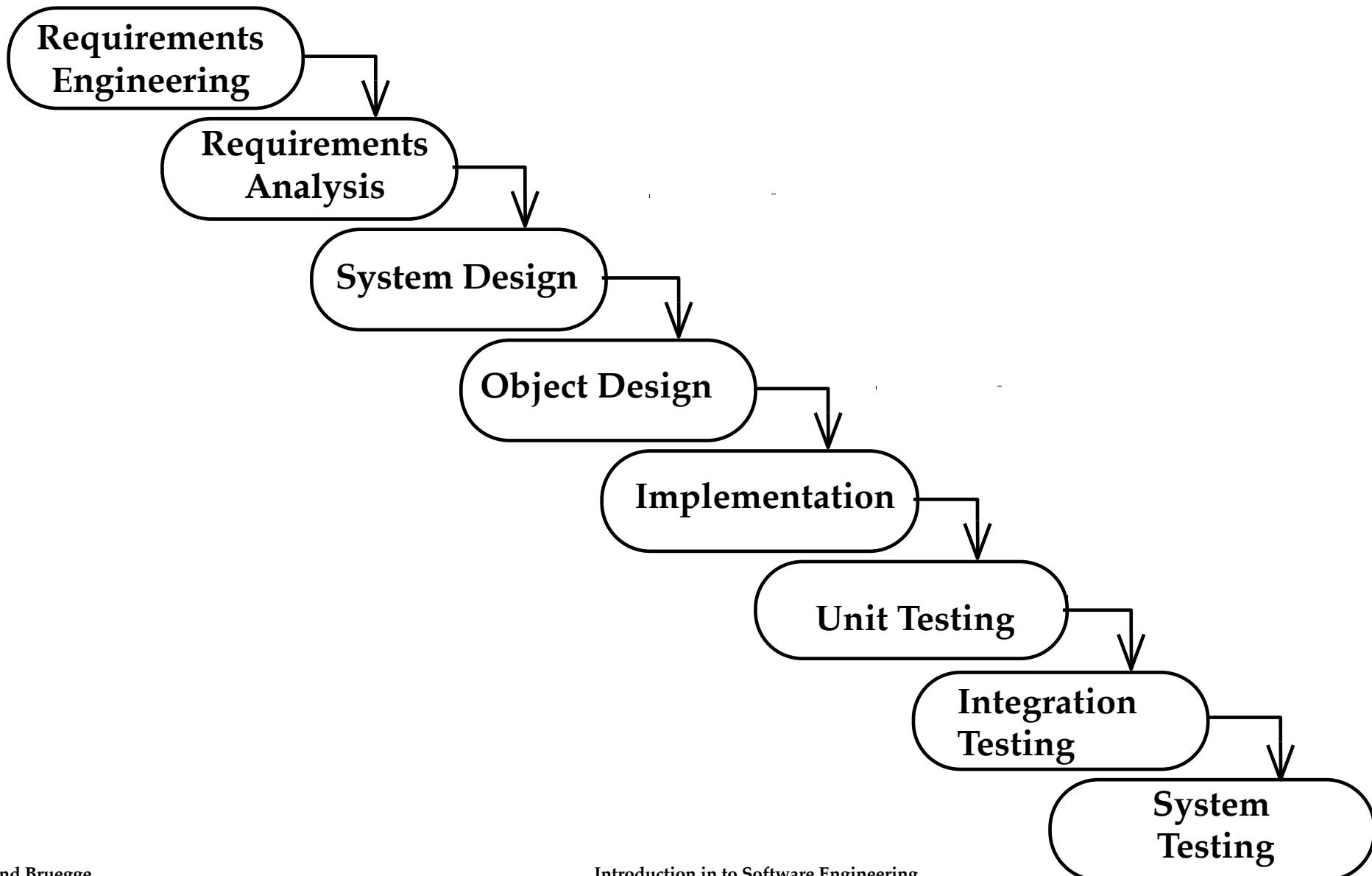
- The waterfall model was first described in 1970 [Royce, 1970].
- The waterfall model is an activity-centered life cycle model that prescribes a sequential execution of development activities and management activities
 - The model assumes that software development can be scheduled as a step-by-step process that transforms user needs into code
 - The goal is to never turn back once an activity is completed. The key feature of his model is the constant verification activity (called “verification step” by Royce) that ensures that each development activity does not introduce unwanted or deletes mandatory requirements.
- The waterfall model provides a *simplistic* view of software development that measures progress by the number of tasks that have been completed.
- The next slide shows UML activity diagram of the waterfall model adapted from [Royce, 1970] using IEEE 1074 names; project management and cross-development processes are omitted.

The Waterfall Model of the Software Life Cycle

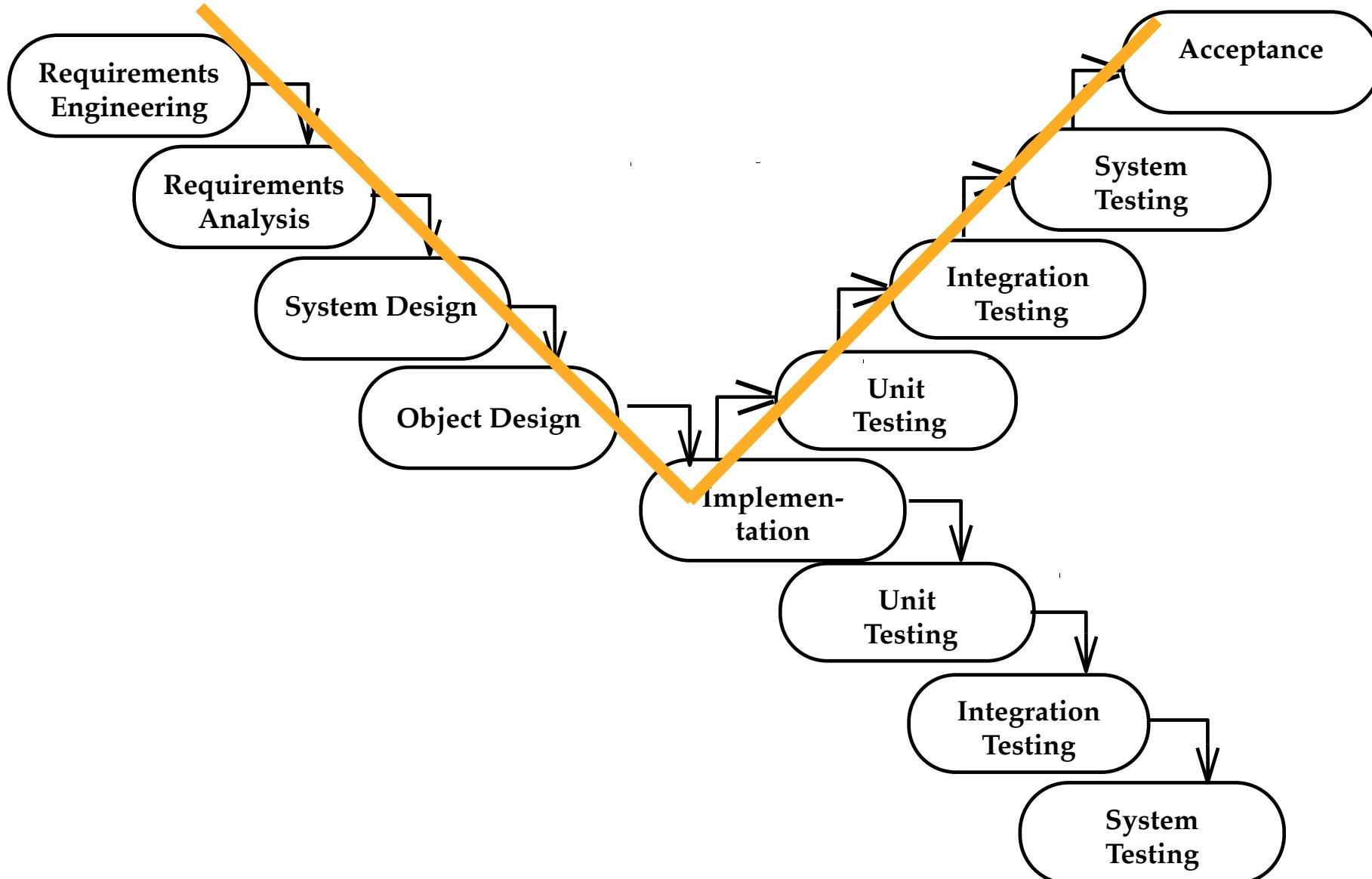


adapted from [Royce 1970]

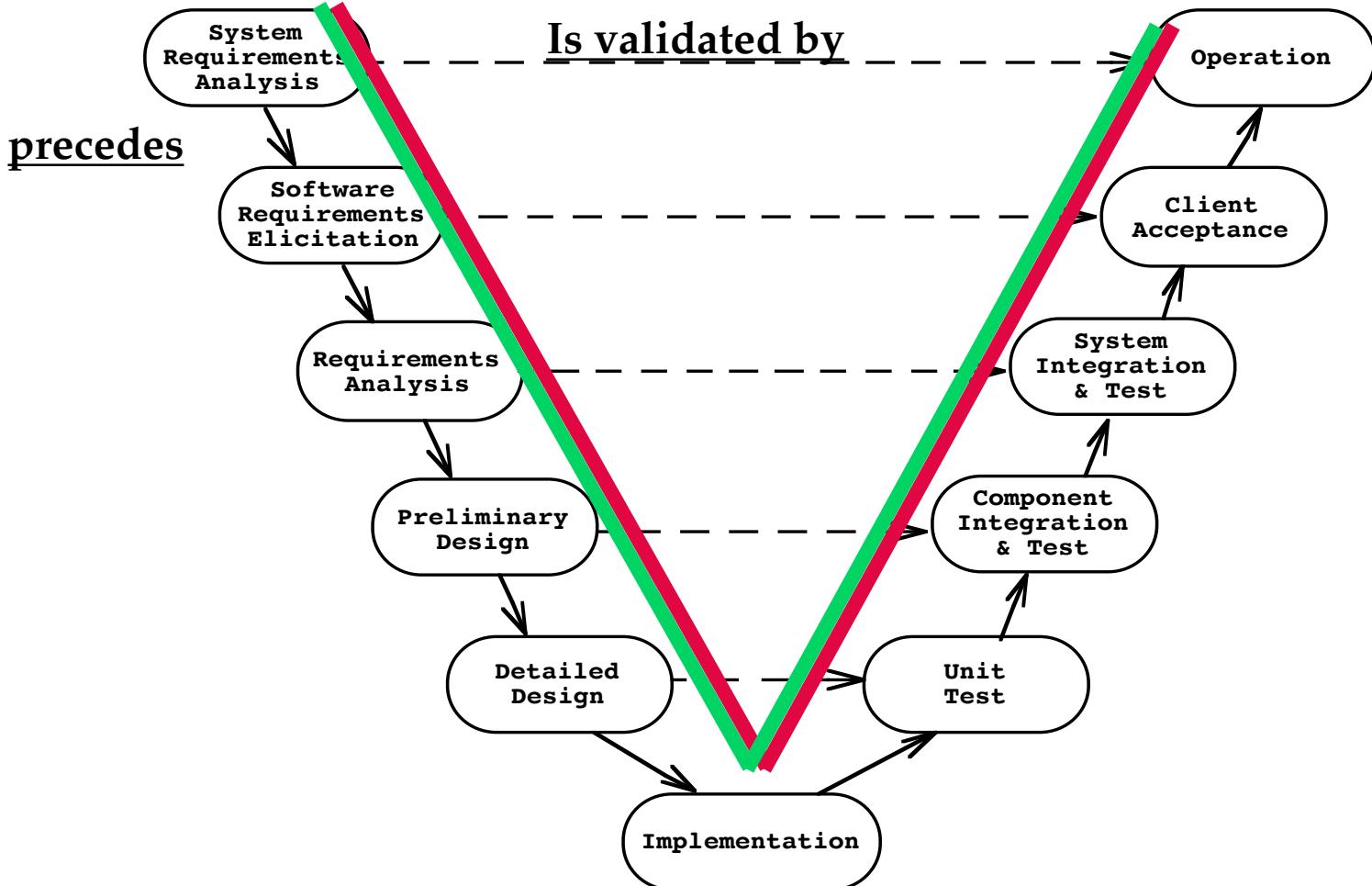
Waterfall Model with Activity Names from the Text Book



From the Waterfall Model to the V Model



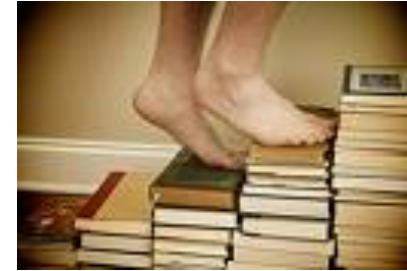
Activity Diagram of the V Model



Assumption of the V-Model: Developers Perception = User Perception

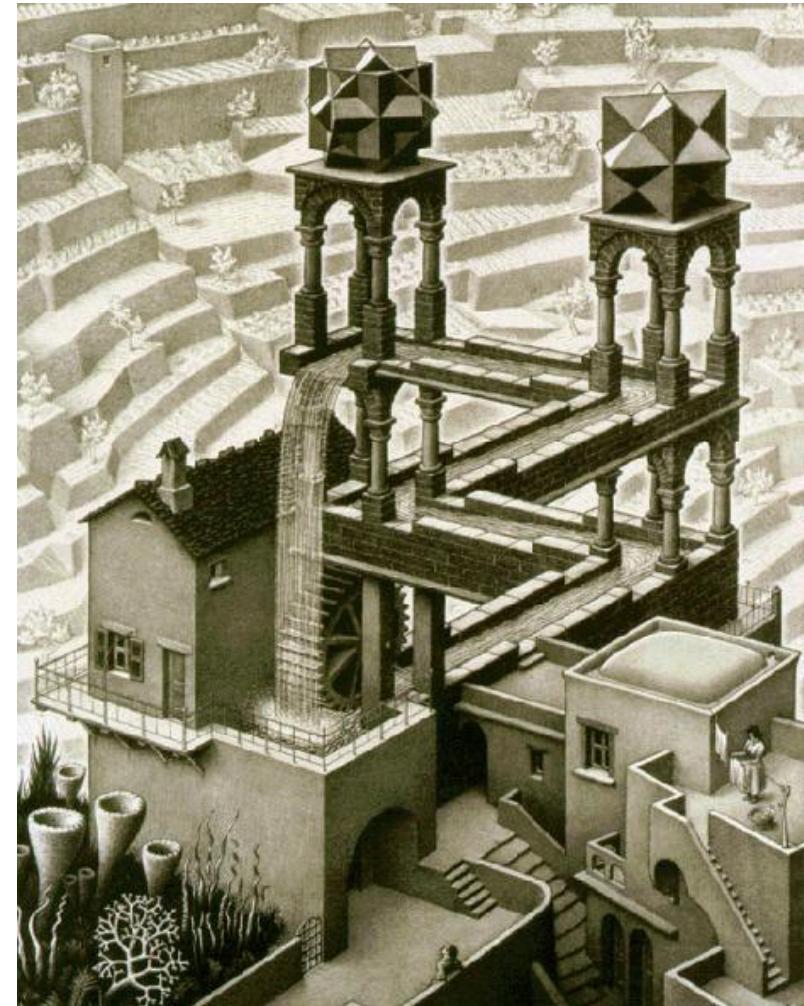
Properties of Waterfall-based Models

- Managers love waterfall models
 - Nice milestones
 - No need to look back (linear system)
 - Always one activity at a time
 - Easy to check progress during development: 90% coded, 20% tested
- However, software development is non-linear
 - While a design is being developed, problems with requirements are identified
 - While a program is being coded, design and requirement problems are found
 - While a program is tested, coding errors, design errors and requirement errors are found.



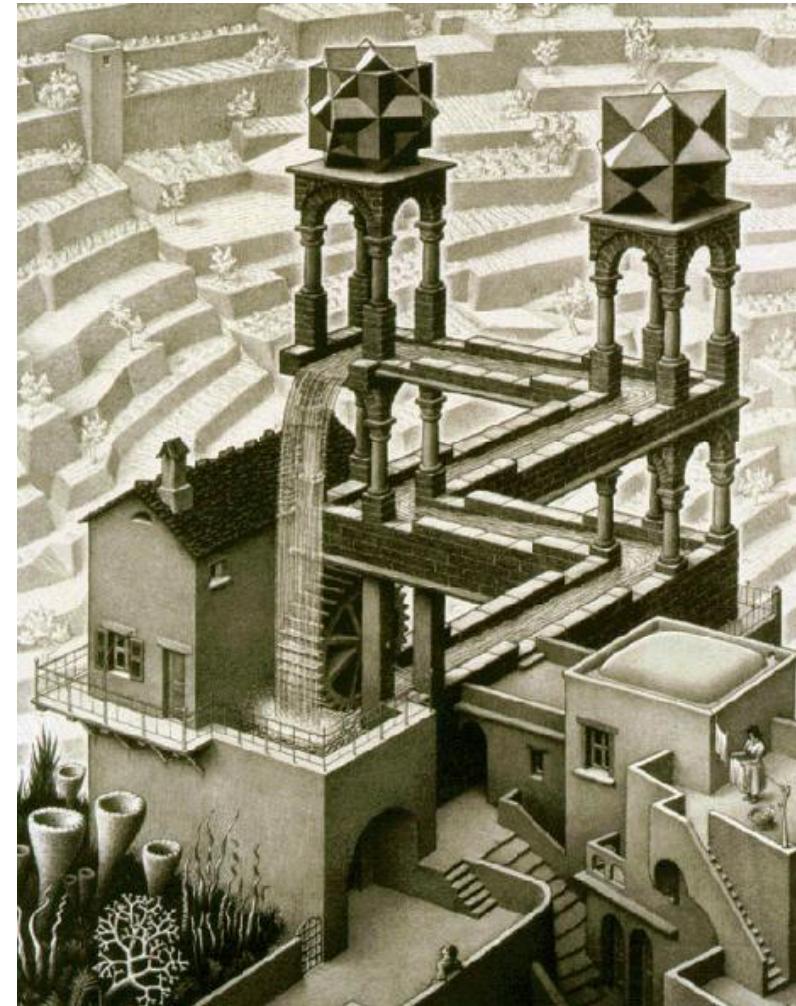
The Alternative: Allow Iteration

- Escher was the first :-)



Lecture Road Map

- Software Development as Application Domain
 - Modeling the software lifecycle
- IEEE Standard 1074 for Software Lifecycles
- Software life cycle models
 - Sequential models
 - Waterfall model, V-model
 - Iterative models
 - Boehm's spiral model, Unified Process
 - Entity-oriented models
 - Scrum
- Process Control Models
 - Noise in communication
 - Methodology Issues
 - Defined process control model
 - Empirical process control model



Spiral Model

The spiral model was proposed by Boehm to deal with the problems of the waterfall model

It is an iterative model with 4 major activities

1. Determine objectives and constraints
2. Evaluate alternatives and identify risks, resolve these risks by assigning priorities to them
3. Develop a series of prototypes for the identified risks starting with the highest risk. Use a waterfall model for each prototype development
4. If a risk has successfully been resolved, evaluate the results of the round and plan the next round

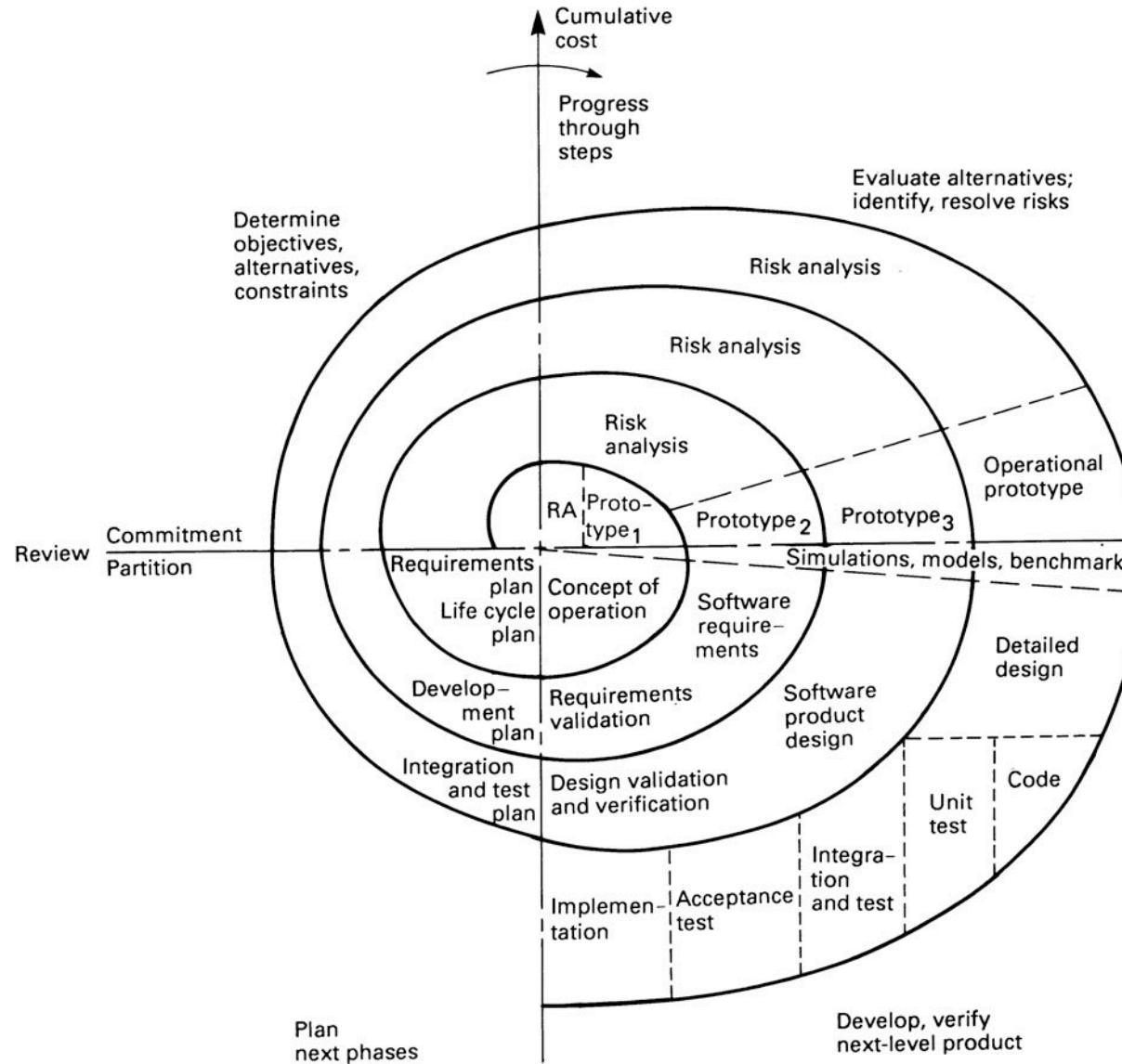
If a risk cannot be resolved, *terminate* the project immediately.

The 4 activities are applied iteratively in each of 9 rounds.

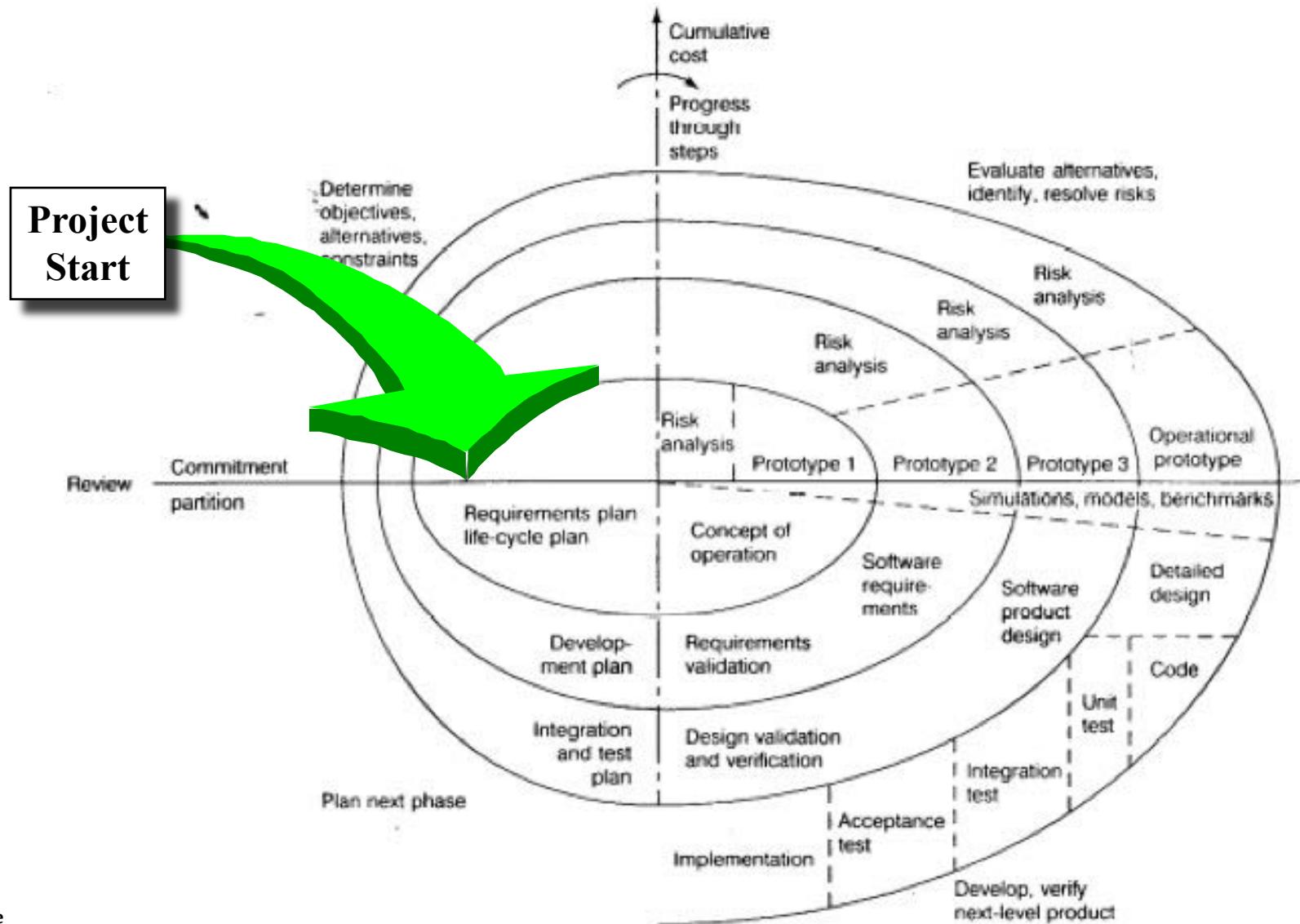
Rounds in the Spiral Model

- 1. Concept of Operations
 - 2. Software Requirements
 - 3. Software Product Design
 - 4. Detailed Design
 - 5. Code
 - 6. Unit Test
 - 7. Integration and Test
 - 8. Acceptance Test
 - 9. Implementation
- For each **round** go through these activities:
 1. Define objectives, alternatives, constraints
 2. Evaluate alternatives, identify and resolve risks
 3. Develop and verify a prototype
 4. Plan the next round.

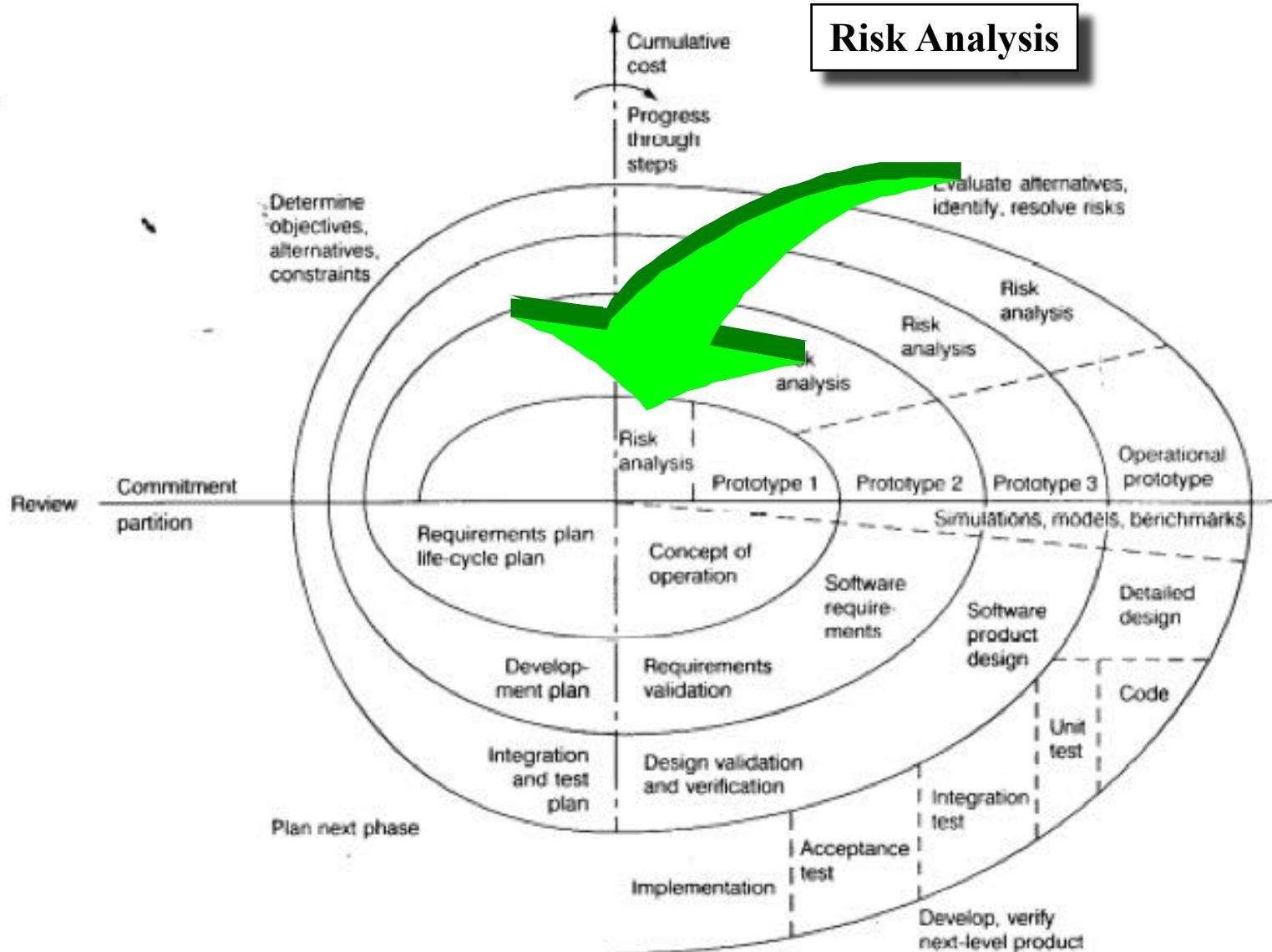
Spiral Model



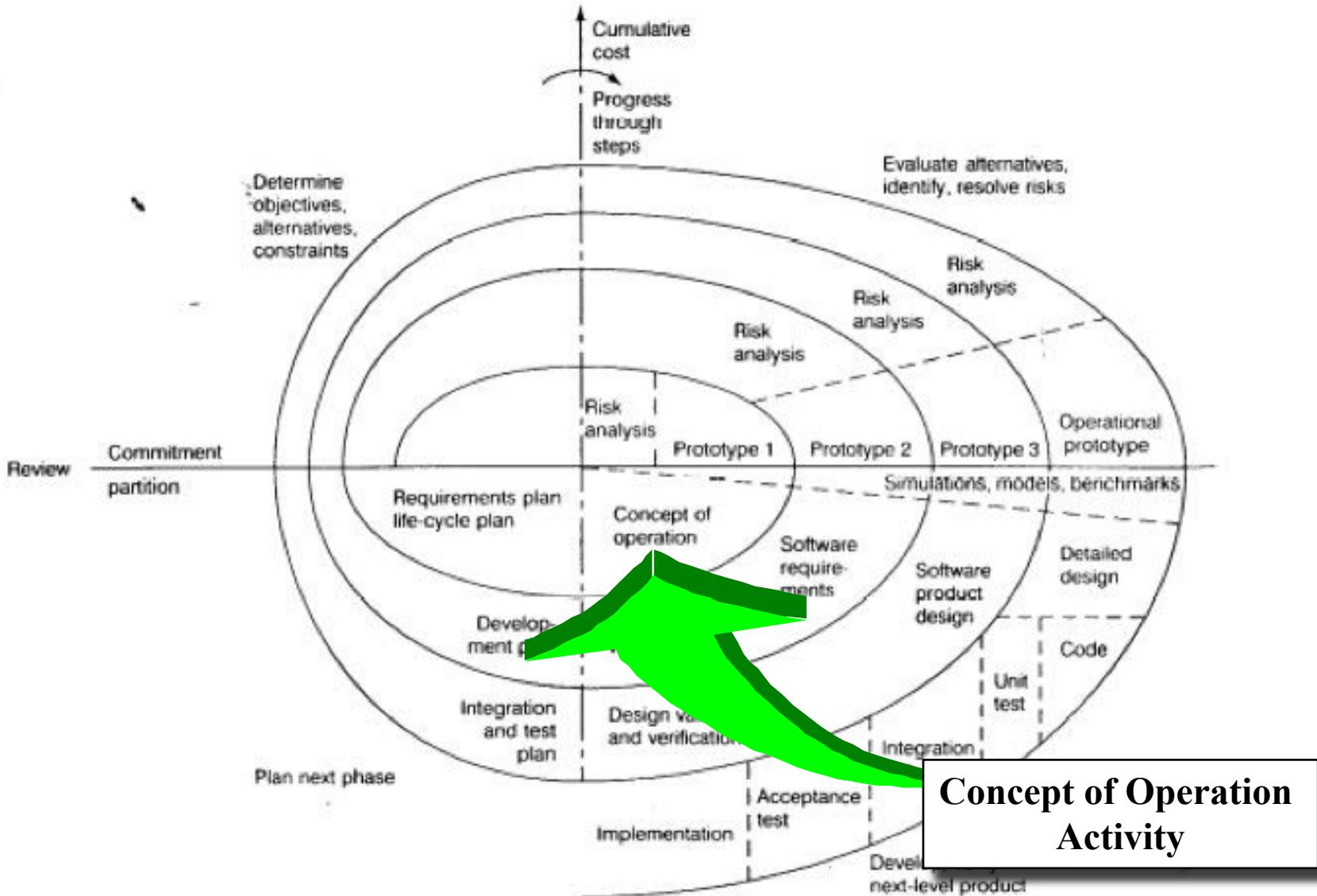
Round 1, Concept of Operations: Determine Objectives, Alternatives & Constraints



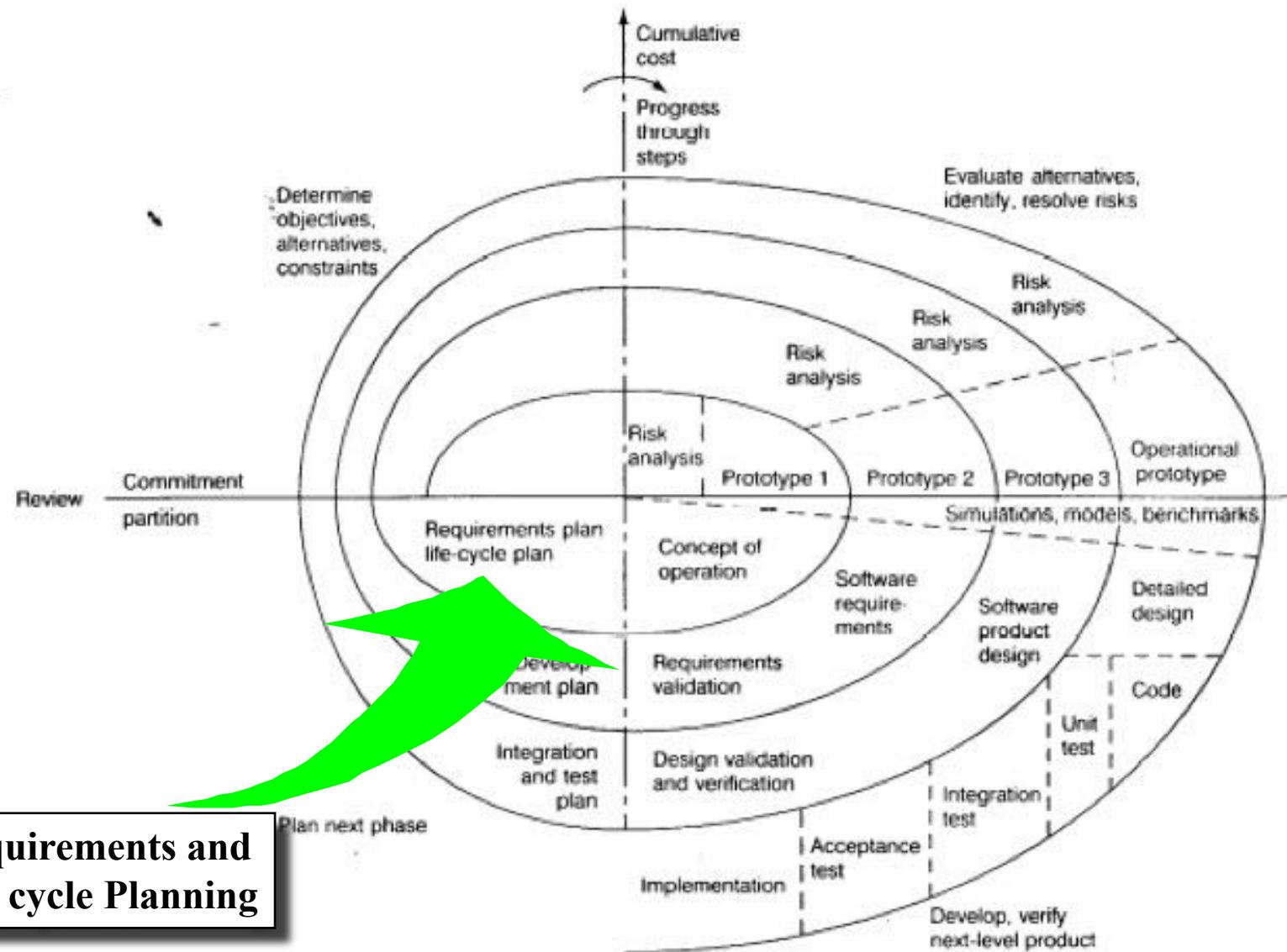
Round 1, Concept of Operations: Evaluate Alternatives, identify & resolve Risks



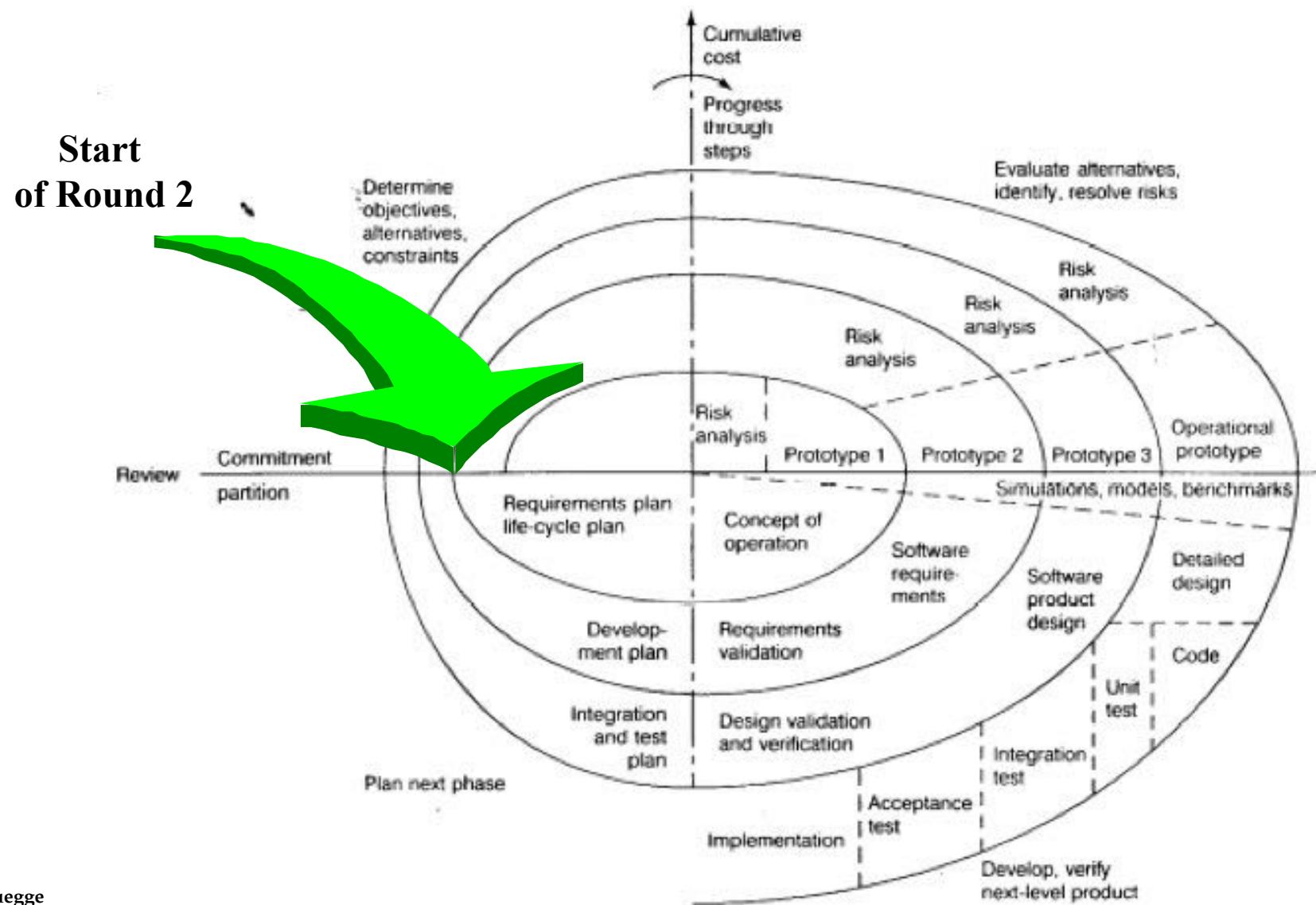
Round 1, Concept of Operations: Develop and Verify



Round 1, Concept of Operations: Prepare for Next Activity



Round 2, Software Requirements: Determine Objectives, Alternatives & Constraints



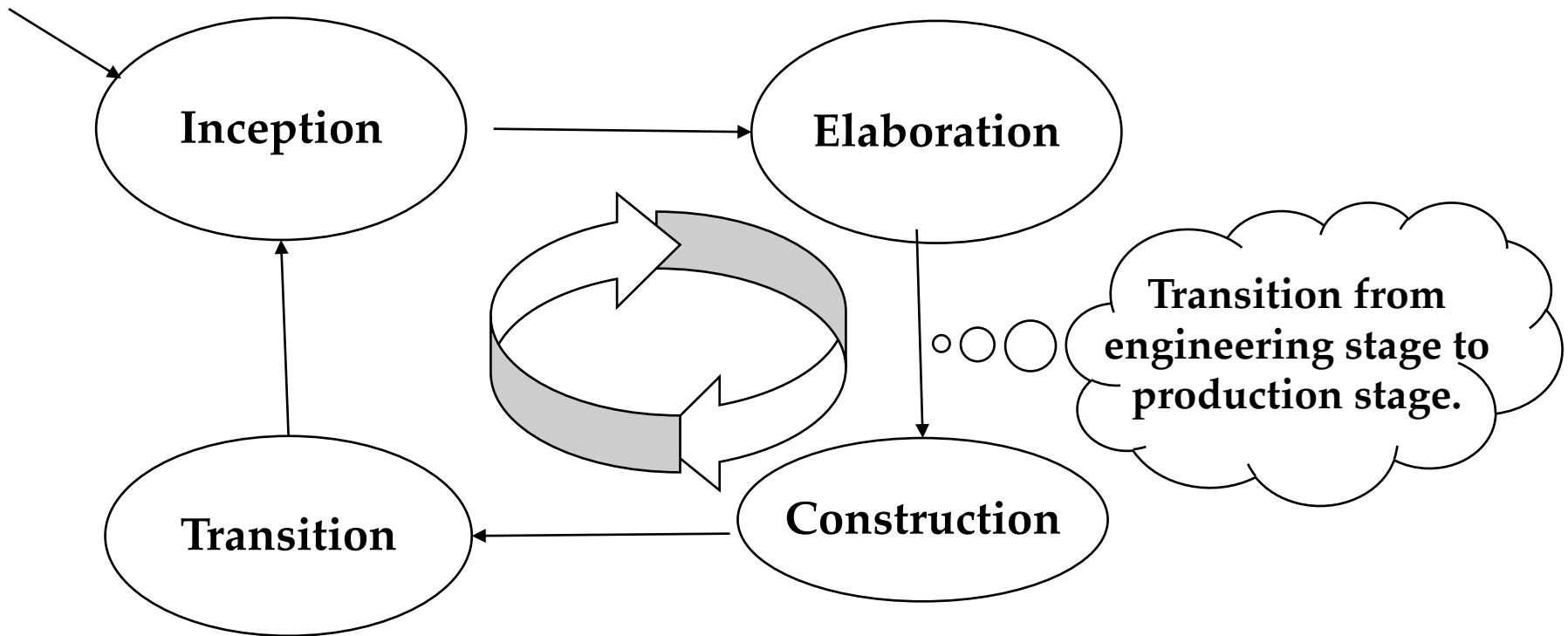
Another Iterative Model: Unified Process

- Developed by Booch, Jacobson, and Rumbaugh
 - Iterative and incremental lifecycle model built on the idea of *cycles* in the lifetime of a software system
- Each cycle consists of 2 **stages** and 4 **phases**
- Inception, Elaboration, Construction, Transition
 - Each phase can consist of several **iterations**
 - During the duration of each iteration, several **workflows** are performed in parallel
 - Management, Environment, Requirements, Design, Implementation, Assessment, Deployment.

The 2 Stages in the Unified Process

- **Engineering Stage**
 - Driven by less predictable but smaller teams, focusing on design and synthesis activities
 - The engineering stage consists of 2 phases
 - *Inception* phase
 - *Elaboration* phase
- **Production Stage**
 - Driven by more predictable but larger teams, focusing on construction, test and deployment activities.
 - The production stage also consists of 2 phases
 - *Construction* phase
 - *Transition* phase.

The 4 Phases in the Unified Process



Unified Process

- Developed by Booch, Jacobson, and Rumbaugh
 - Iterative and incremental lifecycle model built on the idea of *cycles* in the lifetime of a software system
 - Each cycle consists of 2 **stages** and 4 **phases**
 - Inception, Elaboration, Construction, Transition
- Each phase can consist of several **iterations**
- During the duration of each iteration, several **workflows** are performed in parallel
 - Management, Environment, Requirements, Design, Implementation, Assessment, Deployment.

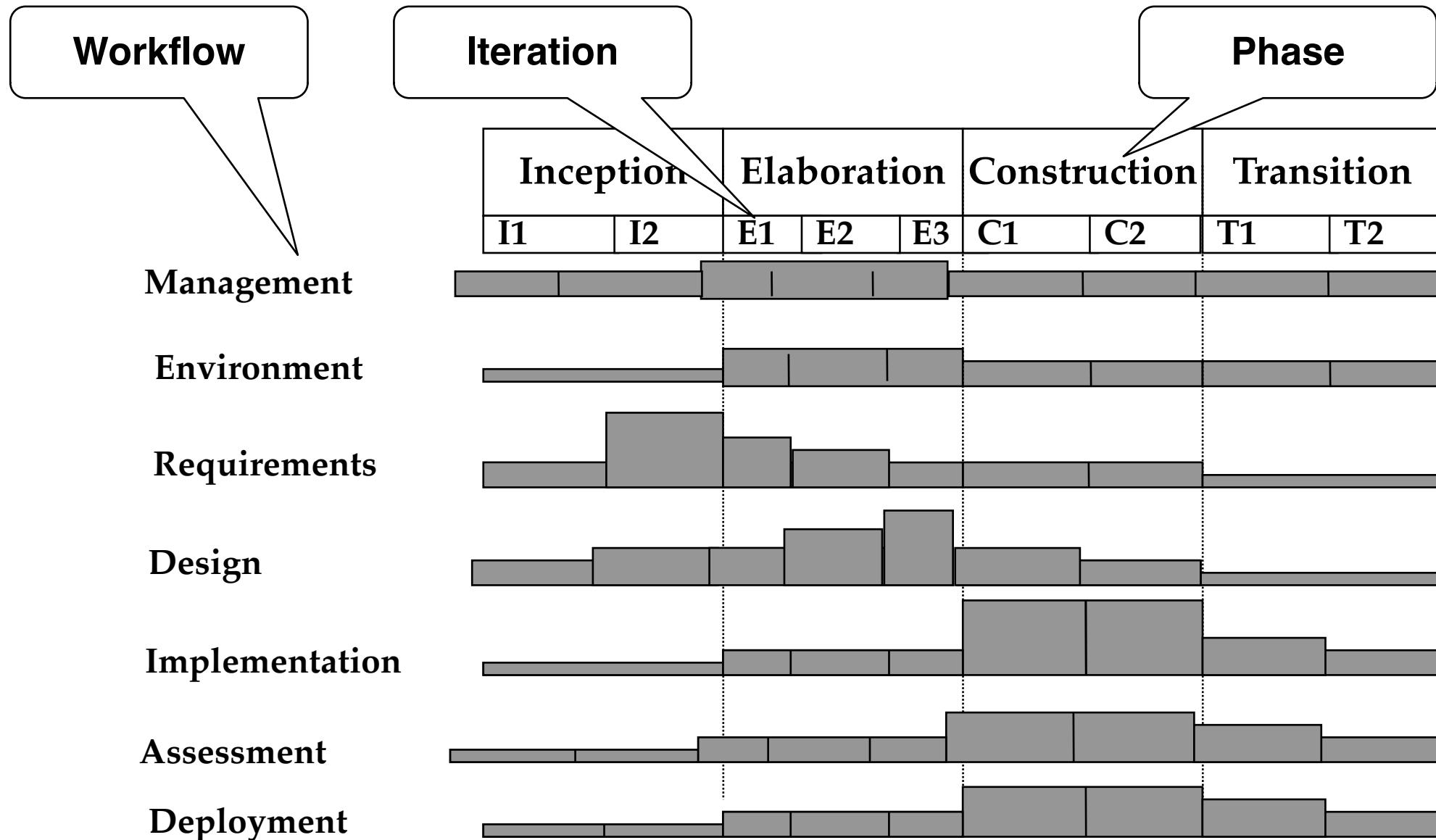
Iteration

- An **iteration** creates an informal, internally controlled version of artifacts
 - It leads to a “minor milestone”
 - Iteration to iteration transition:
 - Triggered by a specific software development activity.
- A **phase** creates a formal, stake-holder approved version of artifacts
 - It leads to a “major milestone”
 - Phase to phase transition:
 - triggered by a significant business decision (not by the completion of a software development activity)

Unified Process

- Developed by Booch, Jacobson, and Rumbaugh
 - Iterative and incremental lifecycle model built on the idea of *cycles* in the lifetime of a software system
 - Each cycle consists of 2 **stages** and 4 **phases**
 - Inception, Elaboration, Construction, Transition
 - Each phase can consist of several **iterations**
- During the duration of each iteration, several **workflows** are performed in parallel
- Management, Environment, Requirements, Design, Implementation, Assessment, Deployment.

Workflows work across Phases and Iterations



Linear and Spiral Models have Limitations

- Neither of these models deal well with frequent change
 - The Waterfall model assumes that once you are done with a phase, all issues covered in that phase are closed and cannot be reopened
 - The Spiral model can deal with change between phases, but does not allow change within a phase
- What do you do if change is happening more frequently?

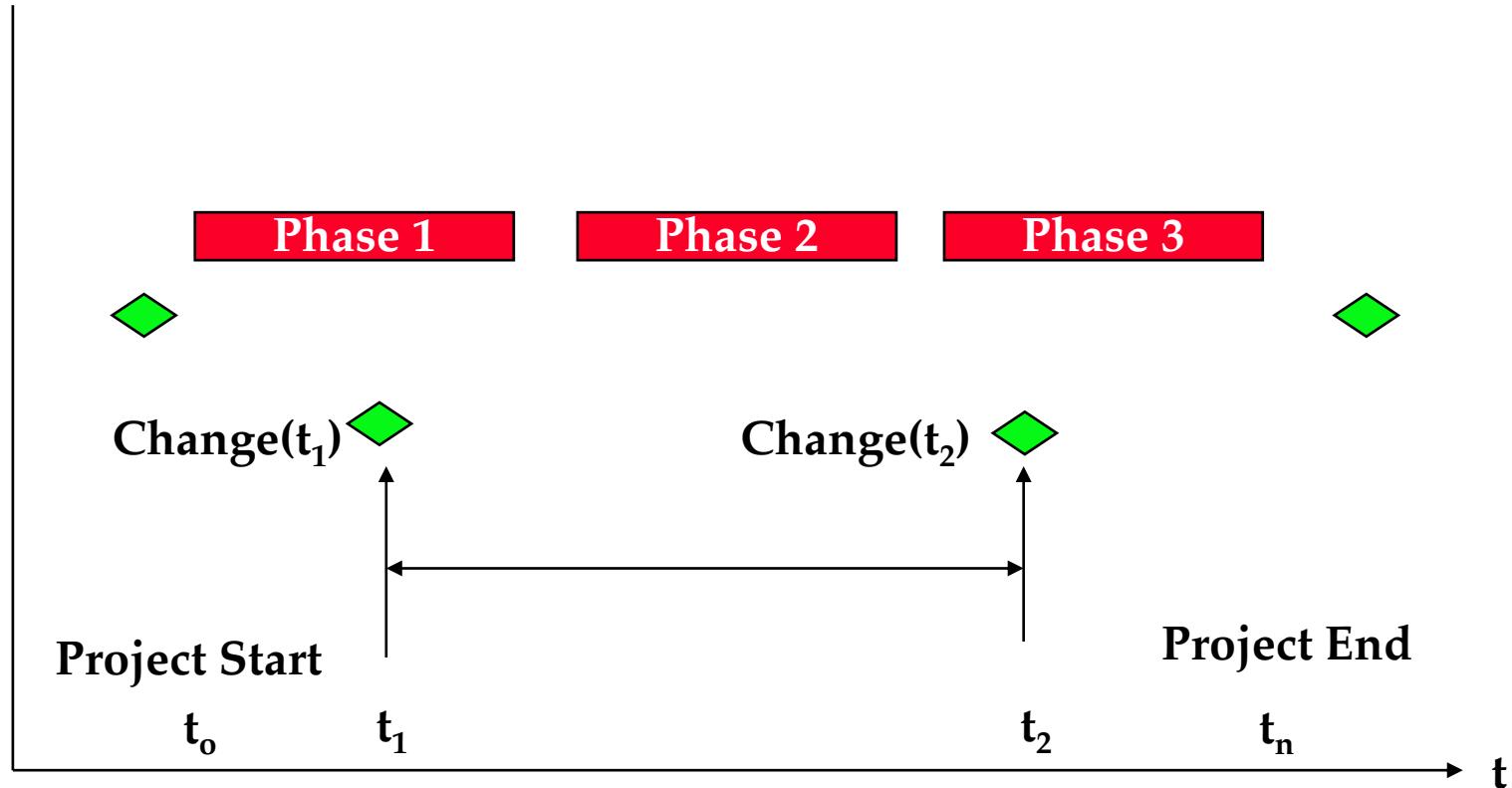
20 Minute Break



Lecture Roadmap

- ✓ Software Development as Application Domain
 - ✓ Modeling the software lifecycle
 - ✓ IEEE Standard 1074 for Software Lifecycles
 - ✓ Software life cycle models
 - ✓ Sequential models
 - ✓ Waterfall model
 - ✓ V-model
 - ✓ Iterative models
 - ✓ Boehm's spiral model
 - ✓ Unified Process
 - Entity-oriented models
 - Scrum
- Process Control Models
 - ➡ Frequency of Change
 - Noise in communication
 - Methodology Issues
 - Defined process control model
 - Empirical process control model

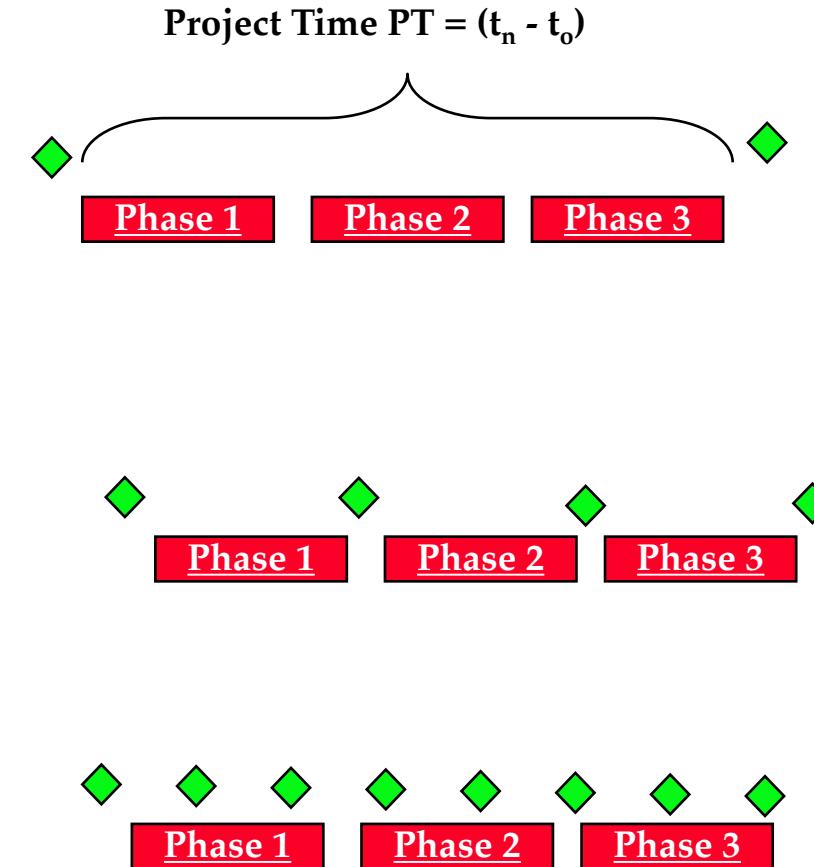
Frequency of Change



- **Project Time PT = ($t_n - t_0$)**
- **TBC = Time Between Changes: $\text{Change}(t_2) - \text{Change}(t_1)$**
- **MTBC = Mean Time Between Changes**

Change influences Choice of the Lifecycle Model

- No change during project
 - MTBC » Project Time PT
 - Linear Model: Waterfall model, V-model
- Infrequent changes during the project
 - MTBC ≈ Duration of Phase
 - Iterative Model: Spiral model, Unified Process
- Changes are frequent
 - MTBC « Project Time PT
 - Agile Model: Scrum.



Frequency of Change -> Software Lifecycle Model Choice

- Change rarely occurs
 - Linear model: Open issues are closed before moving to next phase
- Change occurs sometimes
 - Iterative model: Change of technology may lead to iteration of a previous phase or cancellation of the project
- Change is frequent
 - Agile model: Change of the requirements may lead to a deletion of a phase or artifact.

Incremental vs Iterative vs Adaptive

- **Incremental** means to “**add onto something**”
 - Incremental development improves your **process**
- **Iterative** means to “**re-do something**”
 - Iterative development improves your **product**
- **Adaptive** means “**react to changing requirements**”
 - Adaptive development improves the reaction to changing customer needs.

Incremental, Iterative, Adaptive Development

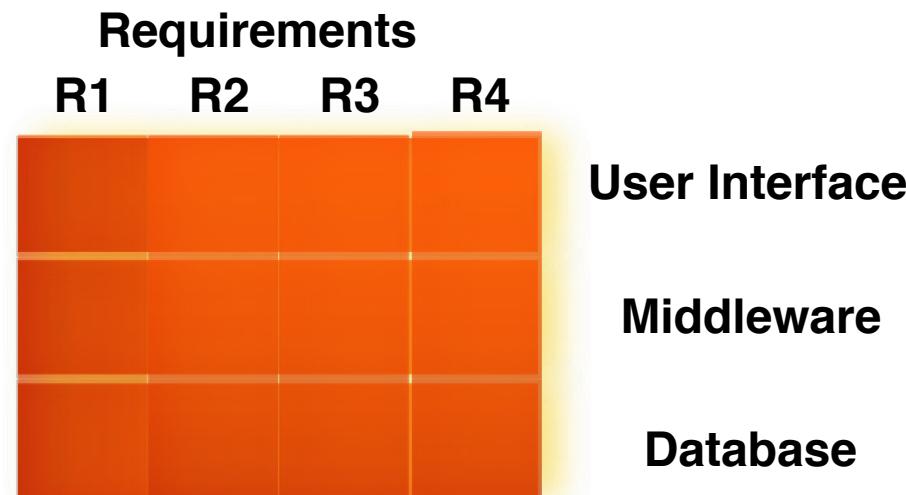
- **Incremental** means to “**add onto something**”
 - Incremental development improves your **process**

System Integration will be covered in more detail in Lecture 11 on Testing (July 5)

Bottom-up integration

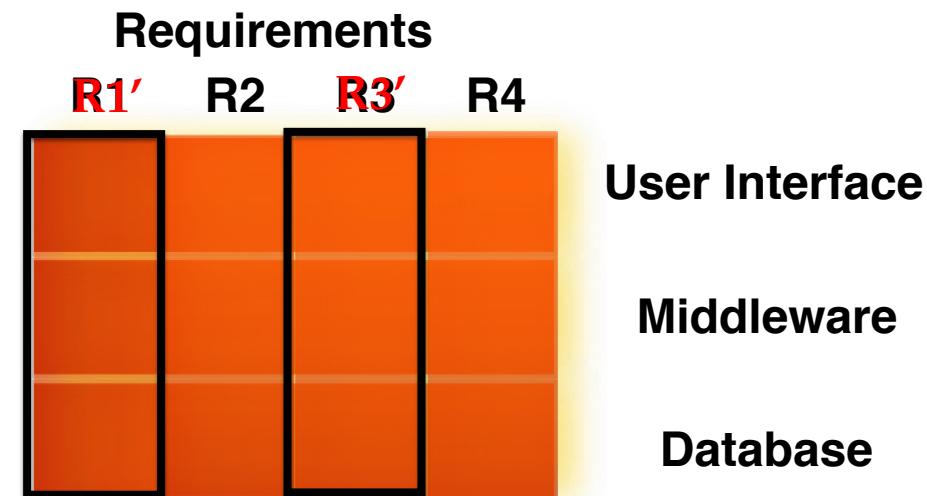
Top-down integration

Vertical integration



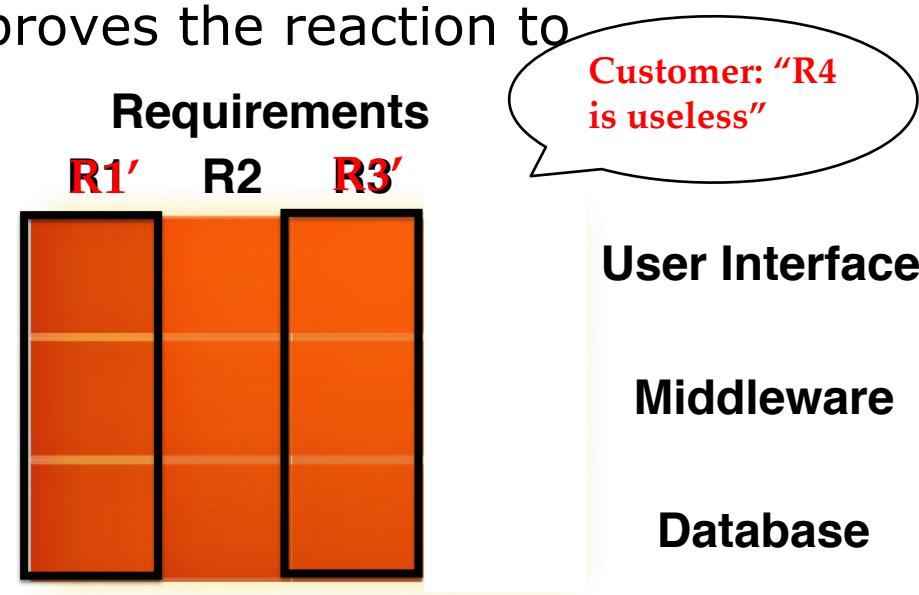
Incremental, Iterative, Adaptive Development

- **Incremental** means to “**add onto something**”
 - Incremental development improves your **process**
- **Iterative** means to “**re-do something**”
 - Iterative development debugs and improves your **product**



Incremental, Iterative, Adaptive Development

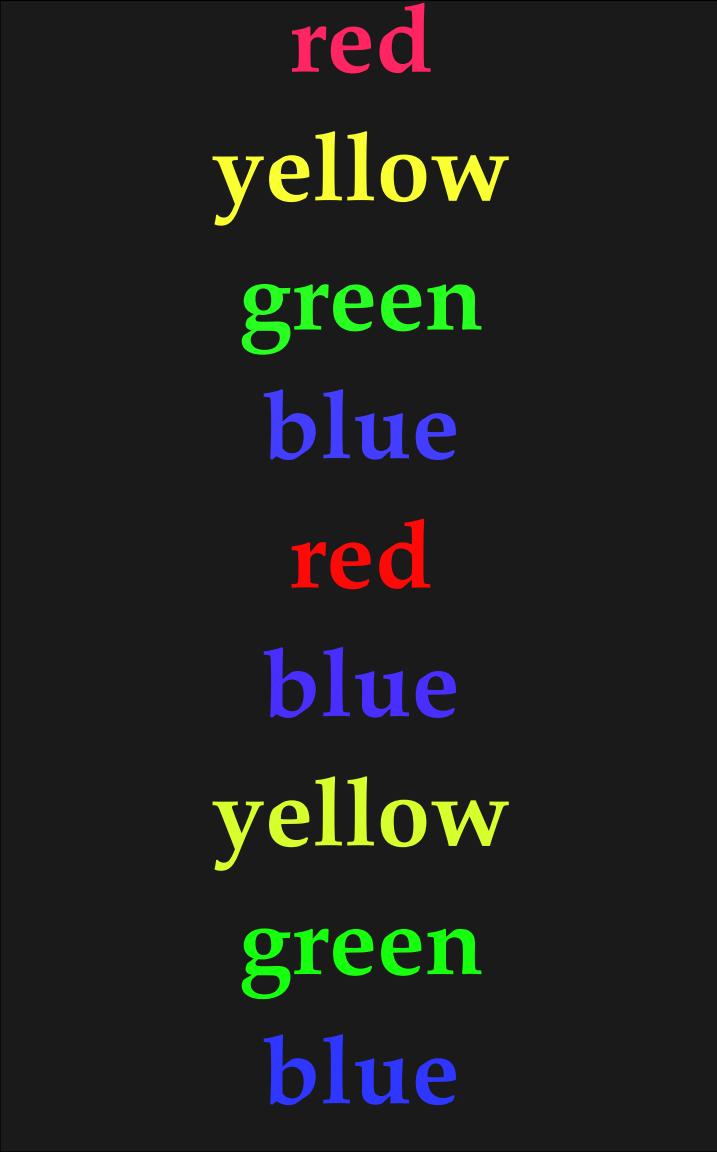
- **Incremental** means to “**add onto something**”
 - Incremental development improves your **process**
- **Iterative** means to “**re-do something**”
 - Iterative development debugs and improves your **product**
- **Adaptive** means to “**react to changing requirements**”
 - Adaptive development improves the reaction to changing customer needs



“People don’t know what they want until you show it to them.” - Steve Jobs.

Lecture Roadmap

- Software Development as Application Domain
 - Modeling the software lifecycle
- IEEE Standard 1074 for Software Lifecycles
- Software life cycle models
 - Sequential models
 - Waterfall model
 - V-model
 - Iterative models
 - Boehm's spiral model
 - Unified Process
 - Entity-oriented models
 - Scrum
- Process Control Models
 - Frequency of Change
 - Noise in communication
 - Methodology Issues
 - Defined process control model
 - Empirical process control model



red

yellow

green

blue

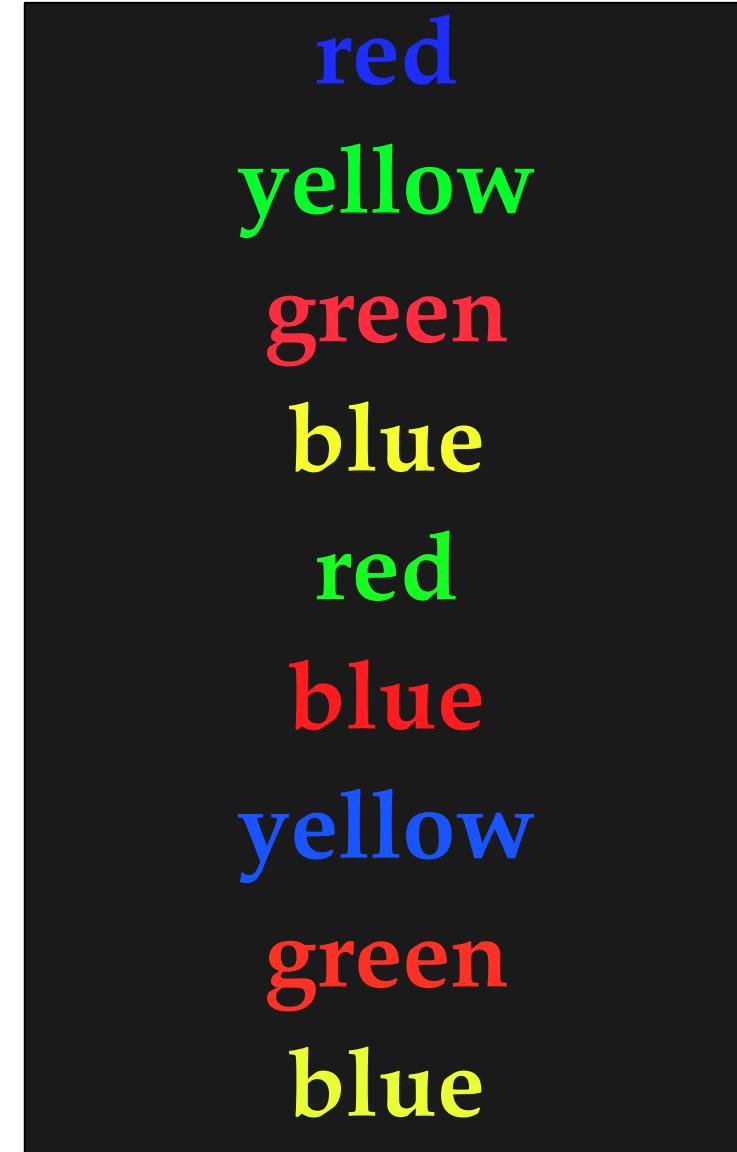
red

blue

yellow

green

blue



Source: American Psychological Association www.apa.org.

Noise in Communication

- When we look at the words in the previous slides, we see color and meaning
- **Simplification System**: At school we have learned that the meaning of a word is more important than its color
 - Saying the color of each word therefore slows us down
 - When two pieces of information are in conflict, we need to make a choice. This slows us down as well.

Noise in Communication (ctd)

- When projects become more complex, many choices have to be made
 - We experience even more unexpected events which produce conflicts between different learned simplification systems
 - Therefore we have to make even more decisions
- If you don't understand the requirements or the technology of a system, or if the requirements change, your old learned preferences may still unconsciously make decisions for you.

Lecture Roadmap

- Software Development as Application Domain
 - Modeling the software lifecycle
- IEEE Standard 1074 for Software Lifecycles
- Software life cycle models
 - Sequential models
 - Waterfall model
 - V-model
 - Iterative models
 - Boehm's spiral model
 - Unified Process
 - Entity-oriented models
 - Scrum
- Process Control Models
 - Frequency of Change
 - Noise in communication



Methodology Issues

- Defined process control model
- Empirical process control model

Methodology Issues

- Methodologies provide general principles and strategies for selecting methods and tools in a given project environment, especially when unexpected events occur
 - Key questions for which methodologies provide guidance:
 - How much involvement of the customer?
 - How much planning?
 - How much reuse?
 - How much modeling before coding?
- How much process?
- How much control and monitoring?

Controlling Software Development with a Process

How do we control software development? Two opinions:

- Through **organizational maturity** (Humphrey)
 - Defined process, Capability Maturity Model (CMM)
- Through **agility** (Schwaber):
 - Large parts of software development is empirical in nature; they cannot be modeled with a defined process
 - There is a difference between a defined and an empirical process
- How can software development better be described?
 - with a **defined process** control model?
 - with an **empirical process** control model?

Defined Process Control Model

Defined process

- Given a well-defined input, the same output are generated every time
- All activities and tasks are well-defined
- Deviations are errors that need to be corrected
- Precondition to apply this model
 - Requires that every piece of work is completely understood before the process starts
- If the preconditions are not satisfied
 - Lot of surprises, loss of control, wrong work products,...
- Conditions when to apply this model:
 - Change can be ignored, the output is predictable
- **The Waterfall Model is a defined process control model.**
 - Defined processes do not deal well with Interferences.

Empirical Process Control Model

- **Empirical process**
 - An imperfectly defined process, not all pieces of work are completely understood
 - Given a well-defined set of inputs, different outputs may be generated when the process is executed
- Deviations are seen as opportunities that need to be investigated
 - The empirical process “expects the unexpected”
- Control and risk management is exercised through frequent inspection
- **Scrum is an empirical process model.**

Definition of Scrum

- Original definition in Rugby:
 - A Scrum is a way to restart the game after an interruption. The forwards of each side come together in a tight formation and struggle to gain possession of the ball when it is tossed in among them
- Definition used in Software Engineering:
 - Scrum is a technique to manage and control software and product development when the requirements and the technology may be rapidly changing during the project.

Why Scrum ?

Linear processes are like relay races (work is performed sequentially)



Agile processes are like rugby (work is performed in parallel)



Scrum in Rugby



History of Scrum

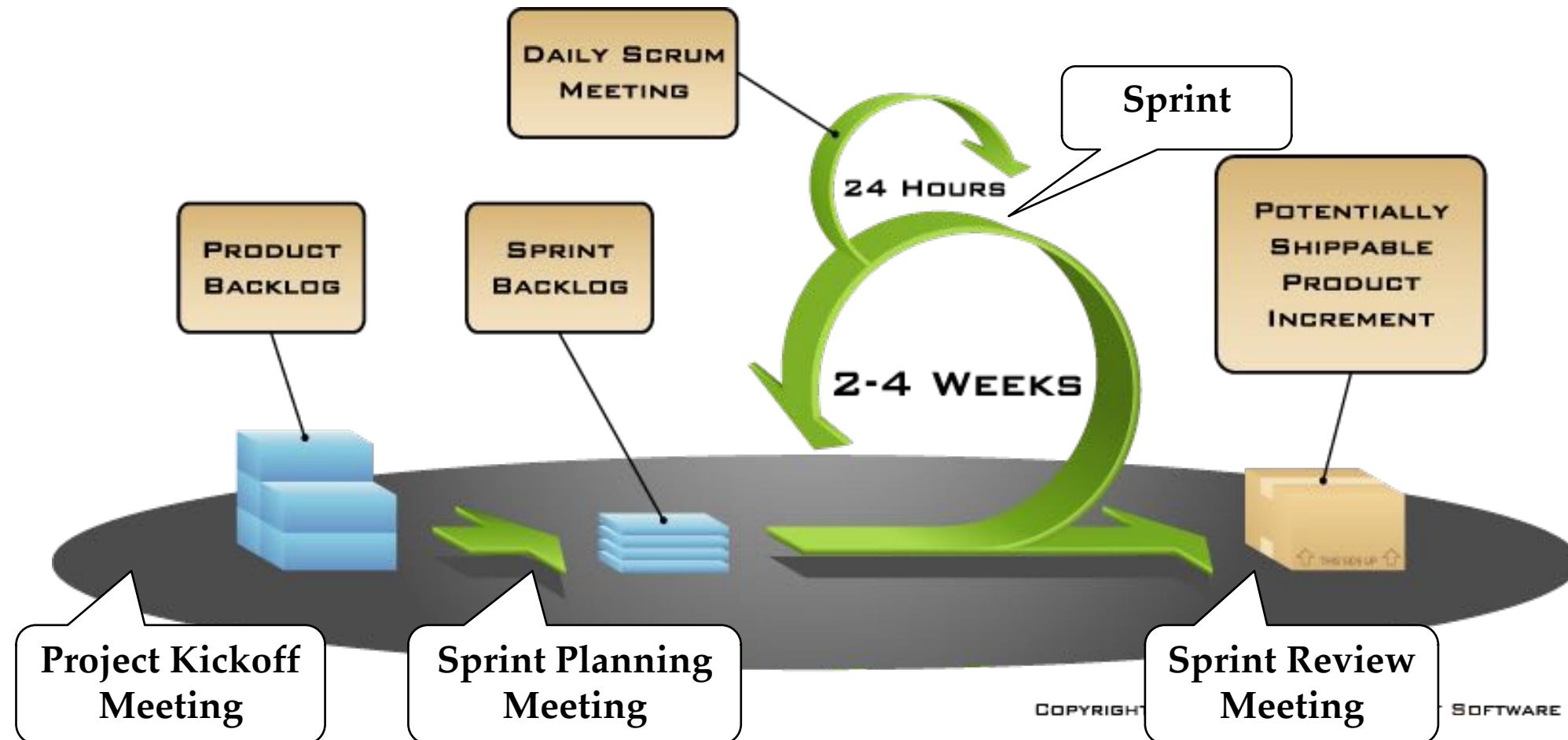
- 1995: Jeff Sutherland and Ken Schwaber analyzed common software development processes
 - “Linear and iterative processes are not suitable for unpredictable and non-repeatable situations”
- 1996: Introduction of Scrum at OOPSLA
- 2001: Publication “Agile Software Development with Scrum” by Ken Schwaber & Mike Beedle
- The Scrum founders are also members in the Agile Alliance which published the Manifesto for Agile Software Development.

Manifesto for Agile Software Development

<http://www.agilemanifesto.org/>

- Individuals and interactions are more important than processes & tools
- Working software is more important than comprehensive documentation
- Customer collaboration is more preferable than contract negotiation
- Responding to change is more preferable than following a plan.

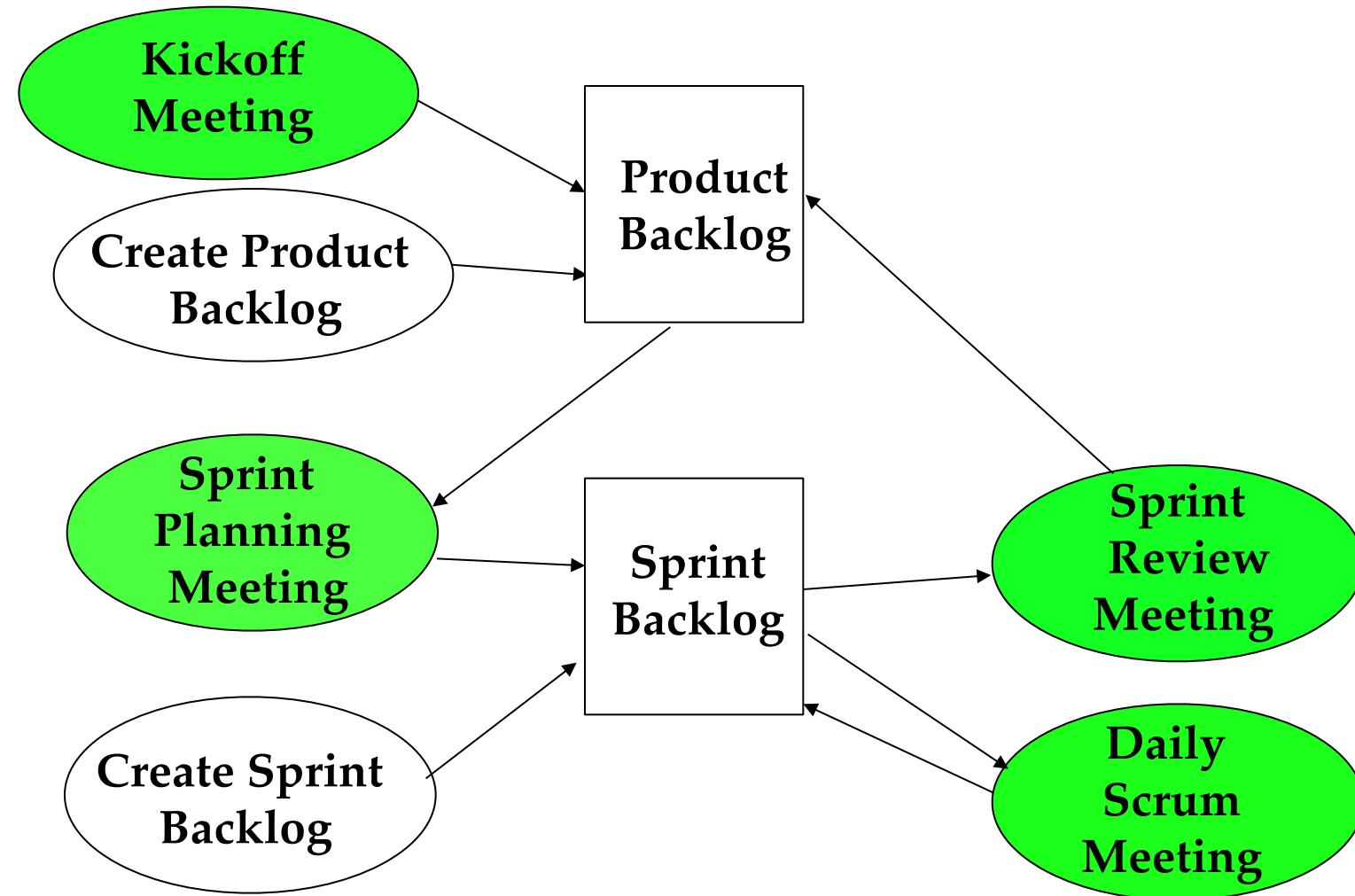
Informal Model of Scrum



Core Components in Scrum

- 3 Artifacts
 - **Product Backlog:** Vague requirements for the whole project
 - **Sprint Backlog:** Detailed requirements for one iteration ("sprint")
 - **Potentially Shippable Product Increment:** The deliverable for each Sprint
- 4 Meeting Activities
 - **Kickoff Meeting:** At the beginning of a project
 - **Sprint Planning Meeting:** List of prioritized features
 - **Daily Scrum:** Informal daily meeting, about 15 min
 - **Sprint Review Meeting:** Demonstration of features to management and customer
- 3 Roles
 - **Scrum Master:** Responsible for the process
 - **Product Owner:** Responsible for the product
 - **Developer:** Responsible for the realization of the Potentially Shippable Product Increment

Informal Scrum Workflow

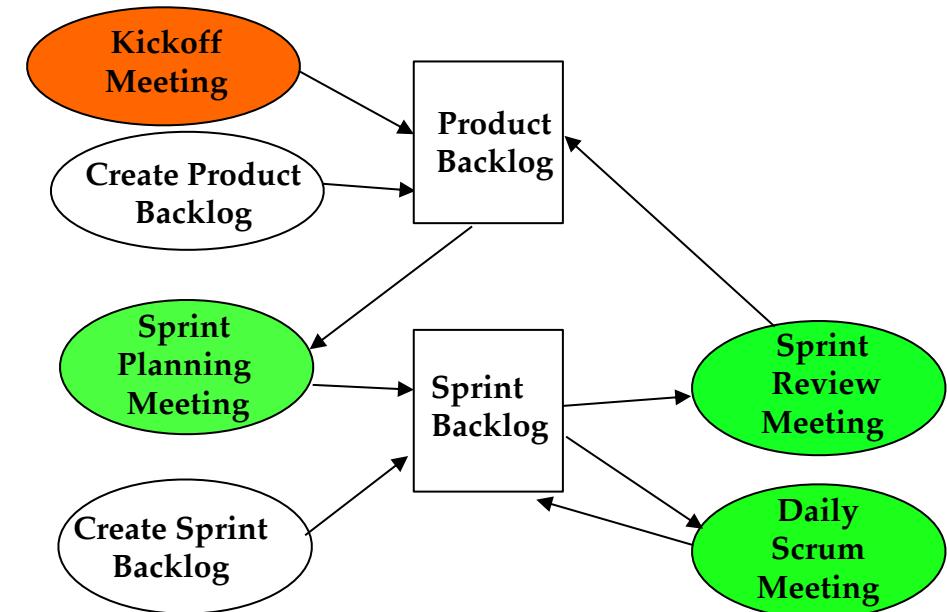


Scrum Roles

- Scrum Master
 - Responsible for enacting scrum values and practices
 - Main job is to remove impediments
 - Typically filled by a project manager or team leader
 - Should moderate and coach the team
- Product Owner
 - Knows what needs to be build and in what order
 - Responsible for the product, value and prioritization
 - Typically a product manager
- Scrum Team
 - Typically 5-6 people
 - Cross-functional (analyst, programmer, designer, tester)
 - Members should be full-time members
 - Team is self-organizing
 - Membership can change only between sprints.

Project-Kickoff Meeting

- A collaborative meeting in the beginning of the project
 - Participants: Product Owner, Scrum Master
 - Takes 8 hours and consists of 2 parts ("before lunch and after lunch")
- Goal: Create the Product Backlog



Daily Scrum Meeting

- A short (15 minutes long) meeting, which is held every day before the Team starts working
- Participants:
 - Scrum Master (which is the chairman), Scrum Team
- Every Team member should answer on 3 questions:

1. Status:

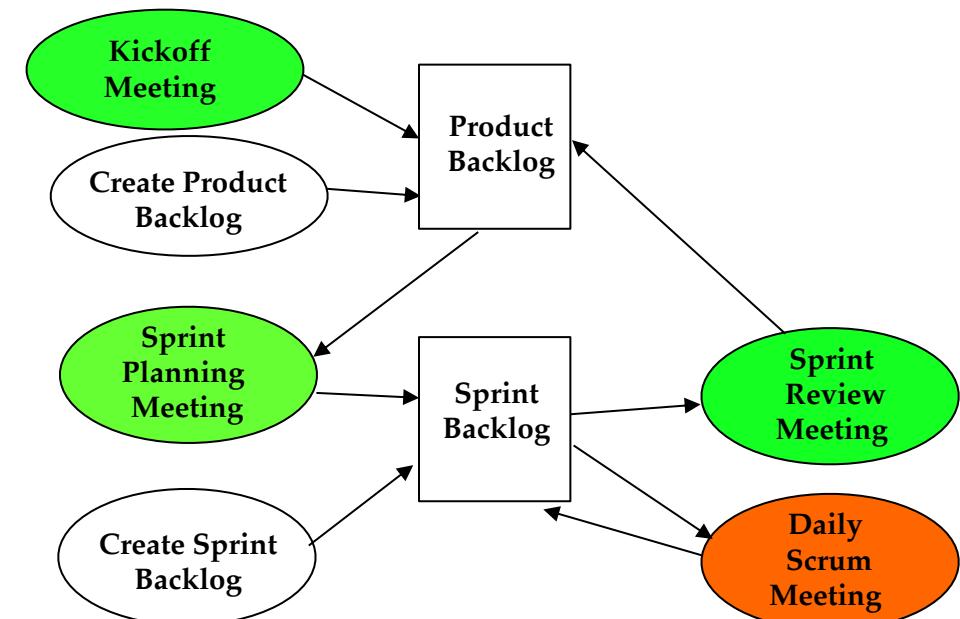
What did I do since the last Scrum meeting?

2. Issues (often called Impediments):

What is stopping me getting on with the work?

3. Action items:

What am I doing until the next Scrum meeting?



In-Class Exercise 01: Model Scrum as UML Activity Diagram

- **Task:** Model the Scrum activities and Scrum artifacts with UML
 - Draw a UML activity diagram (on paper or using your preferred tool) with your solution
 - Take a photo or export your diagram and upload a PNG, JPG or PDF to Moodle: <https://www.moodle.tum.de/mod/assign/view.php?id=762588>
- **Hint 1:** Make sure to include a start and end node
- **Hint 2:** Model the Sprint as activity which includes other activities
- **Hint 3:** Make sure to show the iterative and incremental nature of Scrum.

Summary

- **Software Lifecycle:** Set of activities and their relationships to each other to support the development of a software system
 - Software Lifecycles can be modeled as complex systems
- **Software Lifecycle Model:** An abstraction representing the development of software for the purpose of understanding, monitoring and controlling the software.
- **Different types of models:** Linear, iterative, activity-oriented and entity oriented life cycle models
- **Choice of lifecycle model:** Influenced by the frequency of change
- **Process control model:** Distinction between a defined process control model and an empirical process control model.

Morning Quiz 10

- Start Time: **8:00**
- End Time: **8:10**
- To participate in the quiz, use ArTEMiS
- Click on Open Quiz or Start Quiz
- The Lecture starts at 8:10

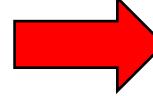
Remaining Time: **46 s**

Saved: never

● Connected

Submit

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	Start exercise
Good Morning Quiz 10		Open Quiz 

Only click on Submit when you have entered all answers!

Object-Oriented Software Engineering

Using UML, Patterns, and Java



Software Configuration Management

Bernd Bruegge

Chair for Applied Software Engineering

Technische Universität München

28 July 2018

Roadmap for Today's Lecture

- **Context and Assumptions**

- We have completed Chapter 1-10 and 15 in the OOSE text book

- **Content of this lecture**

- Today's topic: Chapter 13 Configuration Management

- **Objective:** At the end of the lecture you can

- Identify configuration items
 - Explain the difference between software configuration management, build management and release management
 - Distinguish central and distributed version control
 - Solve a merge conflict
 - Apply continuous integration
 - Explain continuous delivery

iPraktikum WS 18/19



- Focus on
 - Agile Processes
 - 7Rs (Real Clients,
Real Problems, Real Data ...)
- Info Meeting
 - June 28, 4:30pm
 - Interim Lecture Hall 1
- SS 18 Client Acceptance Test
 - July 14, 4:30pm
 - Interim Lecture Hall 1

iPraktikum Winter 2018/19

In this course you develop a mobile application in the context of a larger system architecture. Depending on the project, you work with application servers, machine learning algorithms, smart sensors, intelligent clothing, wearables like the Apple Watch or microcontrollers such as the Raspberry Pi or the Intel Curie.



You get to know the workflows, activities and tools of state-of-the-art agile software development from requirements engineering to system delivery. In particular, you learn Apple's programming language Swift and gain hands-on knowledge in the fields of system modeling, usability engineering and continuous delivery.

For this course, industry partners provide real problem statements. You get real team and project experience while working tightly together with a real client towards a real deadline.



Info Meeting

June 28, 4:30pm
Interims Lecture Hall 1



Registration

www1.in.tum.de/ios



Important Dates

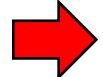
Intro Course: October 8-17
Kickoff: October 18, 4:30 pm



- ✓ Real clients
- ✓ Real problems
- ✓ Real data
- ✓ Real teamwork
- ✓ Real projects
- ✓ Real deadlines
- ✓ Real delivery

 #iostum
@ls1intum

Outline of the Lecture

- 
- Software Configuration Management
 - Change Management
 - Version Control Systems
 - In-Class Exercise 01: Solve a Merge Conflict
 - Branch Management
 - Continuous Integration
 - In-Class Exercise 02: Continuous Integration
 - Release Management

Why Software Configuration Management?



Why Software Configuration Management?

- Problems:
 - Multiple people have to work on software that is changing
 - More than one version of the software has to be supported:
 - Released systems
 - Custom configured systems (different functionality)
 - Systems under development
 - Support for different machines & operating systems

⇒ *Need for coordination*

- Software Configuration Management
 - Manages evolving software systems
 - Controls the effort involved in making changes to a system.

What is Software Configuration Management?

- **Software Configuration Management (SCM):**
 - A set of management disciplines within a software engineering process to develop a baseline
 - SCM encompasses the disciplines and techniques of initiating, evaluating and controlling change to software products during and after a software project
- Standards (approved by ANSI)
 - IEEE 828: Standard for software configuration management plans (SCMP)
 - Minimal required content for an SCMP
 - IEEE 1042: Guide to Software Configuration Management
 - Supplement to IEEE 828. Describes various approaches to configuration management.

Configuration Management Activities

- Software Configuration Management Activities:
 - Configuration item identification
 - Change management
 - Promotion management
 - Branch management
 - Release management
 - Variant management
- No fixed order:
 - These activities are usually performed in different ways (formally, informally) depending on the project type and lifecycle phase (research, development, maintenance).

Configuration Management Actors

- Configuration Manager
 - Responsible for identifying configuration items
 - Also often responsible for defining the procedures for creating promotions and releases
- Change Control Board Member
 - Responsible for approving or rejecting change requests
- Developer
 - Creates promotions triggered by change requests or the normal activities of development. The developer checks in changes and resolves conflicts
- Auditor
 - Responsible for the selection and evaluation of promotions for release and for ensuring the consistency and completeness of this release.

Configuration Management Activities

→ Configuration item identification

- Modeling the system as a *set of evolving components*

• Change management

- Management of *change requests*

• Promotion management

- Creation of *versions for other developers*

• Branch management

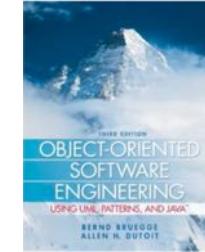
- Management of *concurrent development*

• Release management

- Creation of *versions for clients and end users*

• Variant management

- Management of *coexisting versions*



Today

Reading:
Bruegge-Dutoit
Chapter 13, p. 550ff

Terminology: Configuration Item

Configuration Item: An aggregation of hardware, software, or both, designated for configuration management and treated as a single entity in configuration management activities

- Software configuration items are not only source files but can be all types of artifacts
 - Documents, pictures, hardware, CPUs, bus speed frequencies, ...
 - Whatever needs to be put under control because it might change during the project.

Identification of Configuration Items

- Selecting the right configuration items is a skill that takes practice
 - Very similar to object modeling
 - Use techniques similar to object identification for finding configuration items!
 - Find the configuration items
 - Find relationships between configuration items.

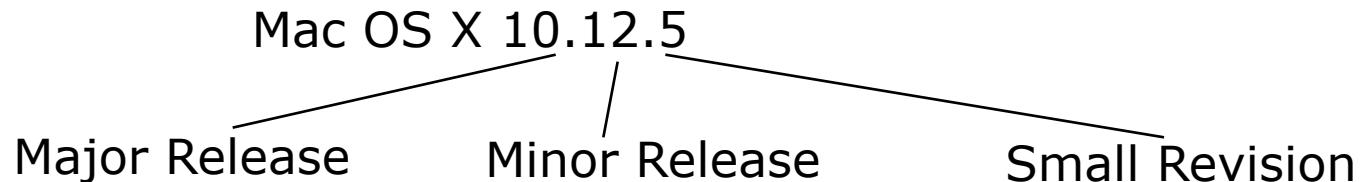
Terminology: Baseline

Baseline: A specification or product that has been formally reviewed and agreed to by responsible management

- It serves as the basis for further development
- Examples:
 - *Baseline A:* The API has been completely defined; the bodies of the methods are empty
 - *Baseline B:* All attribute setter and getter methods are implemented and tested
 - *Baseline C:* The GUI is implemented.

Naming Schemes for Baselines

- Many naming schemes for baselines exist (1.0, 6.01a, 3.14159265)
- A 3 digit scheme is quite common:



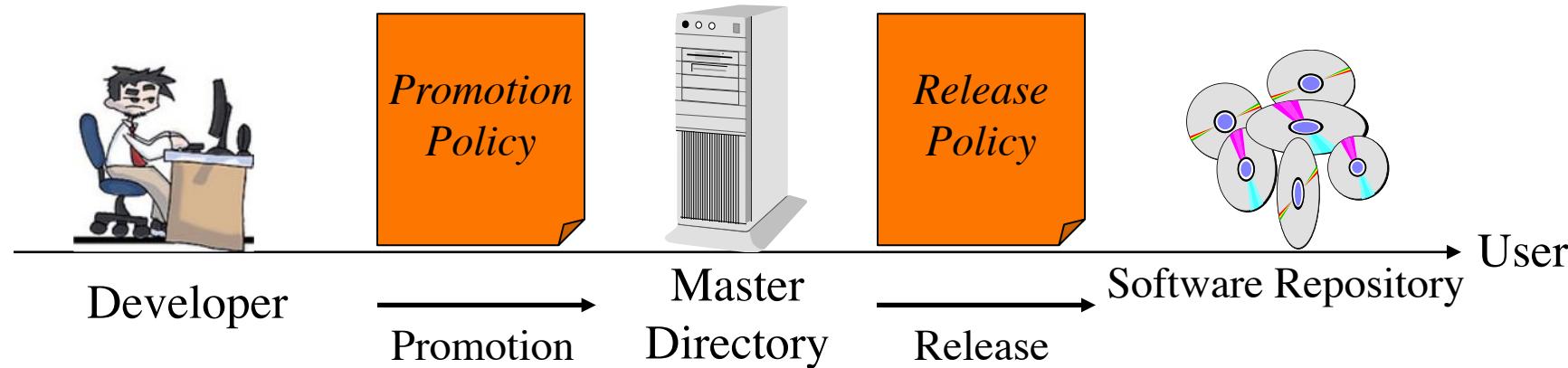
- Inconsistencies are not uncommon
- Often also arbitrary or used by marketing

Knuth adds a digit from π for every new release of \TeX
Current baseline number is 3.14159265.



Controlling Changes: Change Policies

- *Promotion*: The **internal** development state of a software is changed
- *Release*: A changed software system is made visible **outside** the development organization
- Two types of controlling change:

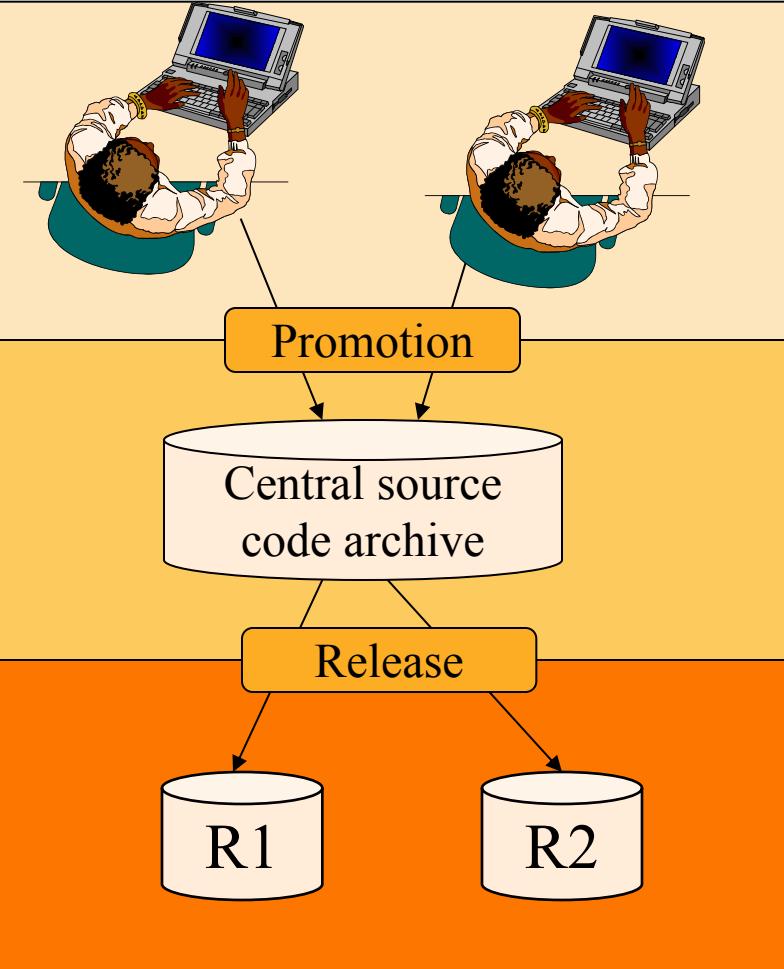


Approaches for controlling change with a change policy:

- Informal (good for research environments, promotions)
- Formal (good for externally developed Items, releases).

SCM Directories in the IEEE Std 828

- Programmer's Directory
 - (IEEE Std: "Dynamic Library")
 - Completely under control of one programmer
- Master Directory
 - (IEEE Std: "Controlled Library")
 - Central directory of all promotions
- Software Repository
 - (IEEE Std: "Static Library")
 - Externally released baselines.



Terminology: SCM Directories

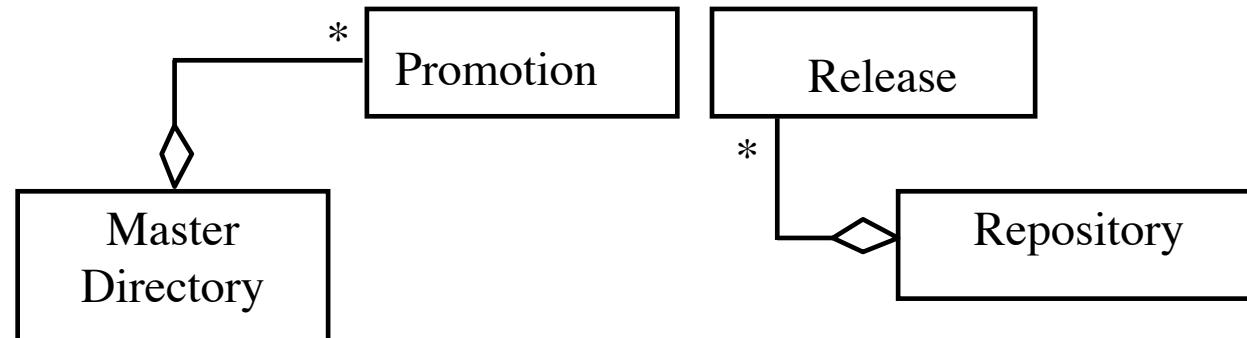
- **Programmer's Directory** (IEEE 1042: Dynamic Library)
 - Library for holding newly created or modified software entities
 - The programmer's workspace is controlled by the programmer only
- **Master Directory** (IEEE 1042: Controlled Library)
 - Manages the current baseline(s) and for controlling changes made to them
 - Changes must be authorized
- **Software Repository** (IEEE 1042: Static Library)
 - Archive for the various baselines released for general use
 - Copies of these baselines may be made available to requesting organizations.

Let's Create an Object Model for Configuration Management

„Promotions are stored in the master directory and releases are stored in the repository“

Problem: There can be many promotions and many releases

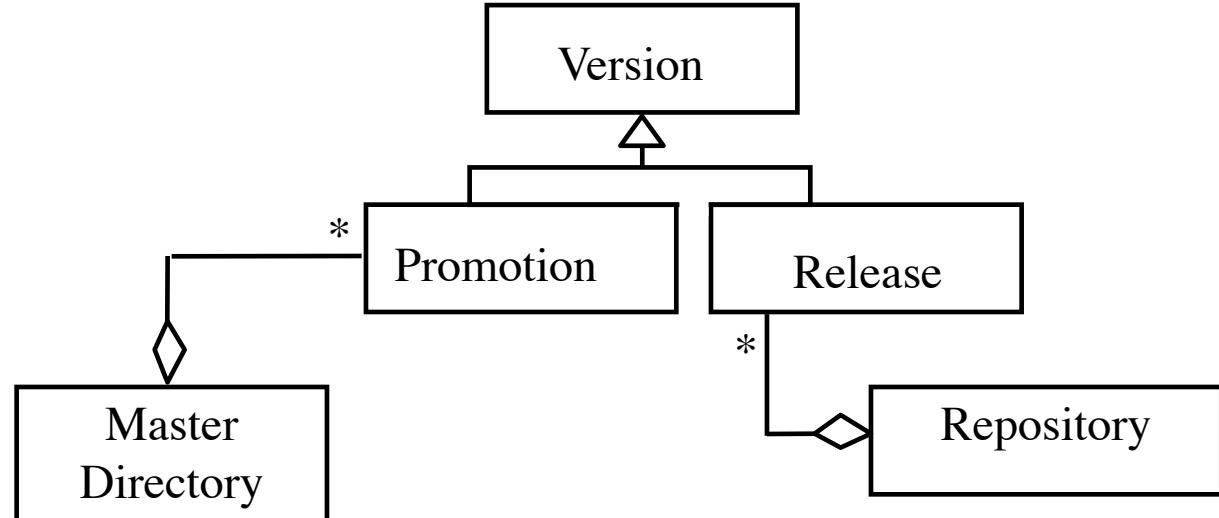
Solution: Use Multiplicity



Let's Create an Object Model for Configuration Management

Insight: Promotions and Releases are both versions

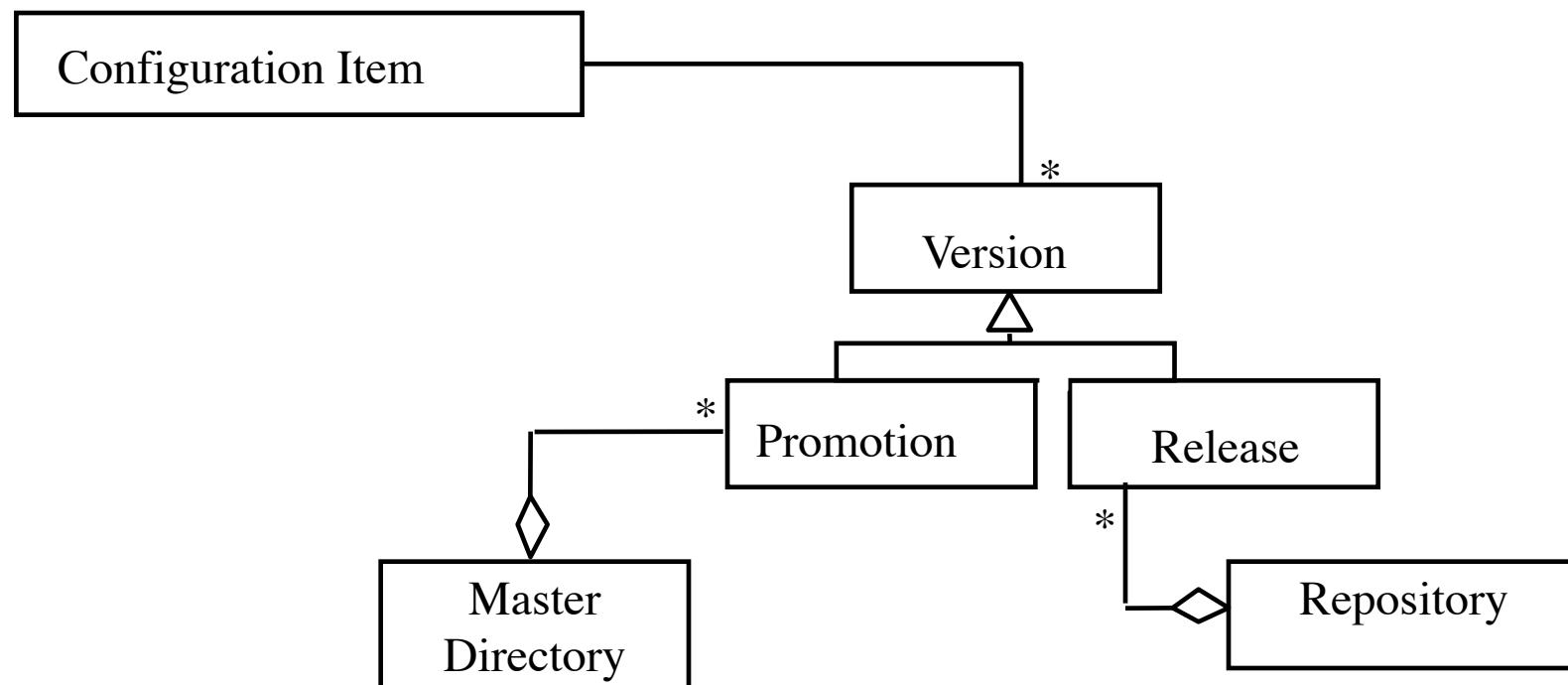
Solution: Use Inheritance



Let's Create an Object Model for Configuration Management

Problem: A configuration item can have several versions

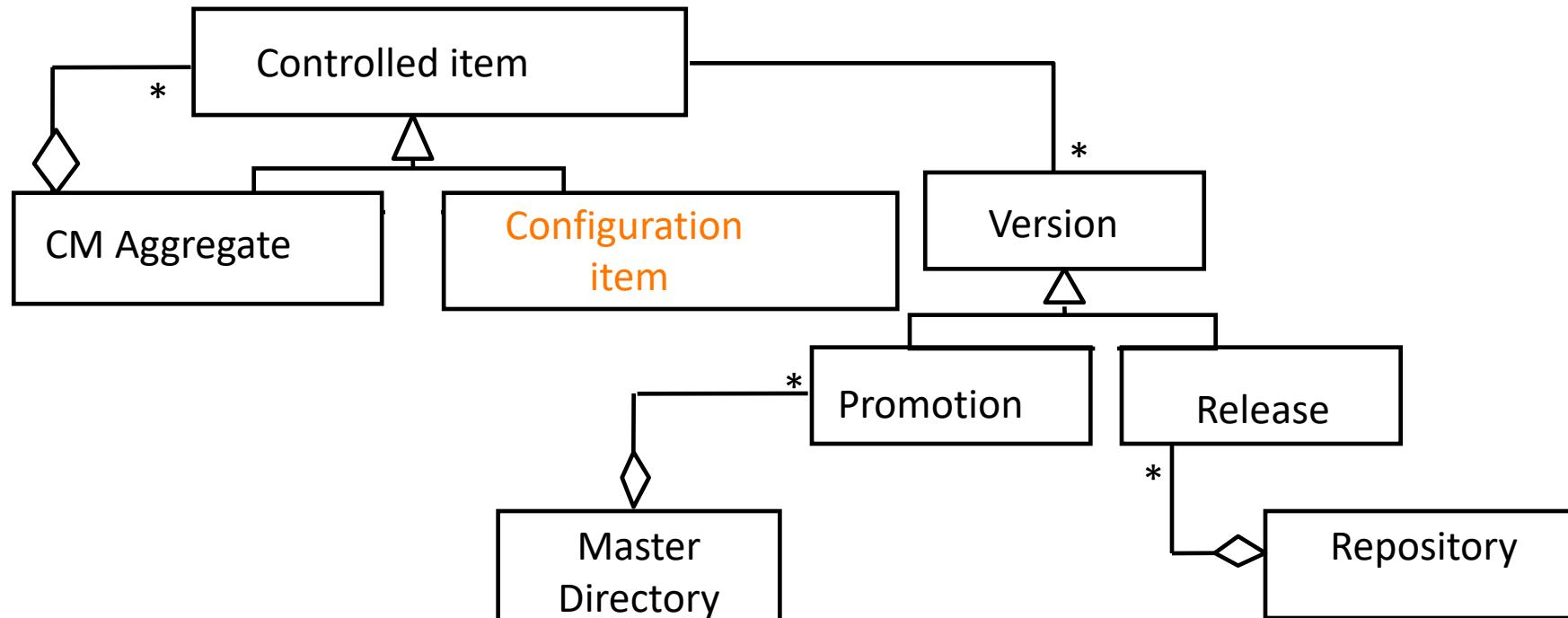
Solution: Create a 1-many association between Configuration Item and Version



Let's Create an Object Model for Configuration Management

Problem: Configuration items can themselves be grouped

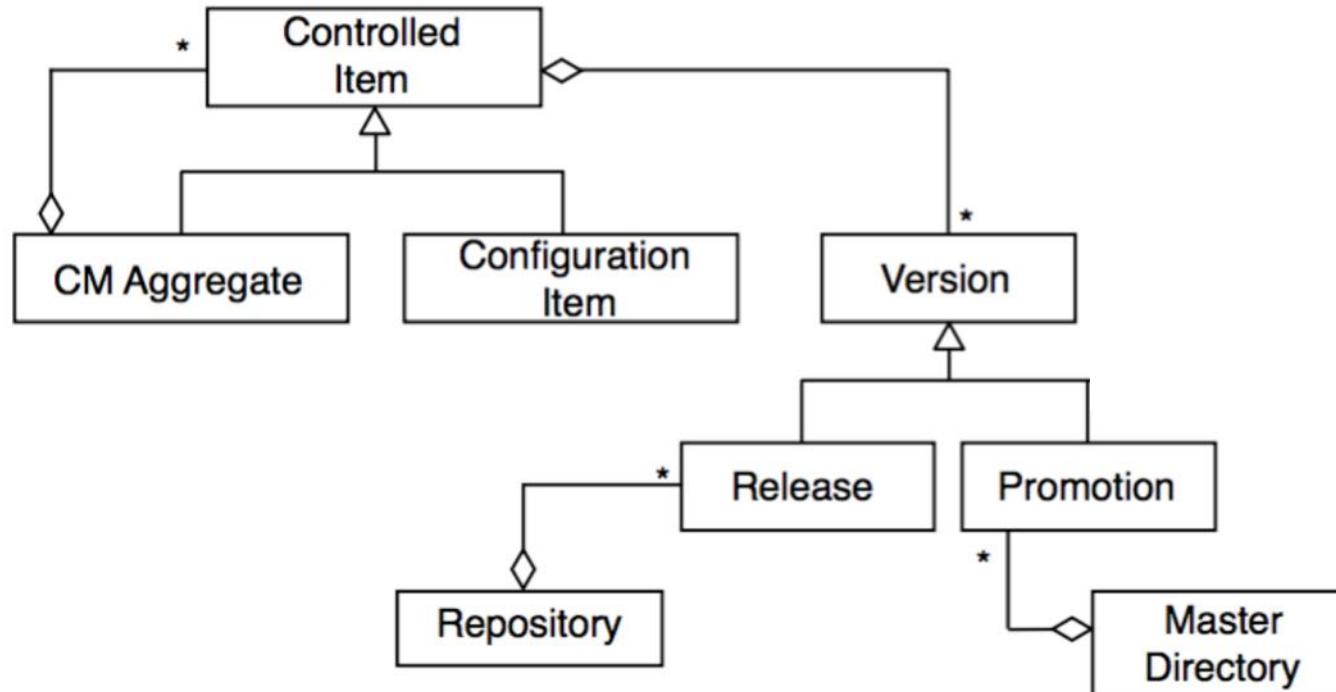
Solution: Use the composite design pattern



Let's Create an Object Model for Configuration Management

Problem: Small changes (e.g. bugfixes) lead to a Revision

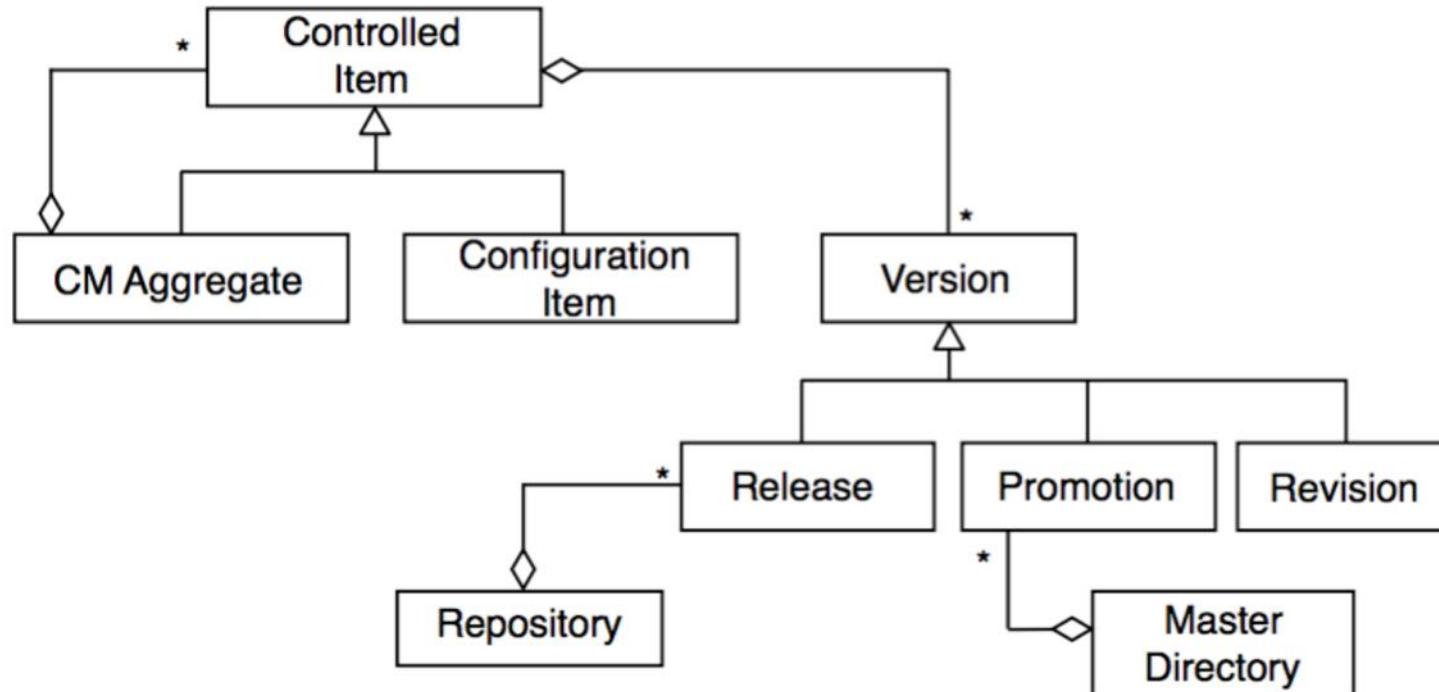
Solution: Add another subclass to Version



Let's Create an Object Model for Configuration Management

Problem: Small changes (e.g. bugfixes) lead to a Revision

Solution: Add another subclass to Version



Navigational Text to the Object Model

- **Version:** An initial release or re-release of a configuration item associated with a complete compilation or recompilation of the item. Different versions have different functionality
- **Promotion:** The distribution of a version to another developer
- **Release:** The formal distribution of an approved version
- **Revision:** Change to a version that corrects only errors in the design/code, but does not affect the documented functionality.

Outline of the Lecture

- Software Configuration Management
- Change Management
- Version Control Systems
 - In-Class Exercise 01: Solve a Merge Conflict
- Branch Management
- Continuous Integration
 - In-Class Exercise 02: Continuous Integration
- Release Management

Configuration Management Activities

- ✓ Configuration item identification

- Modeling the system as a *set of evolving components*

→ Change management

- Management of *change requests*

- Promotion management

- Creation of *versions for other developers*

- Branch management

- Management of *concurrent development*

- Release management

- Creation of *versions for clients and end users*

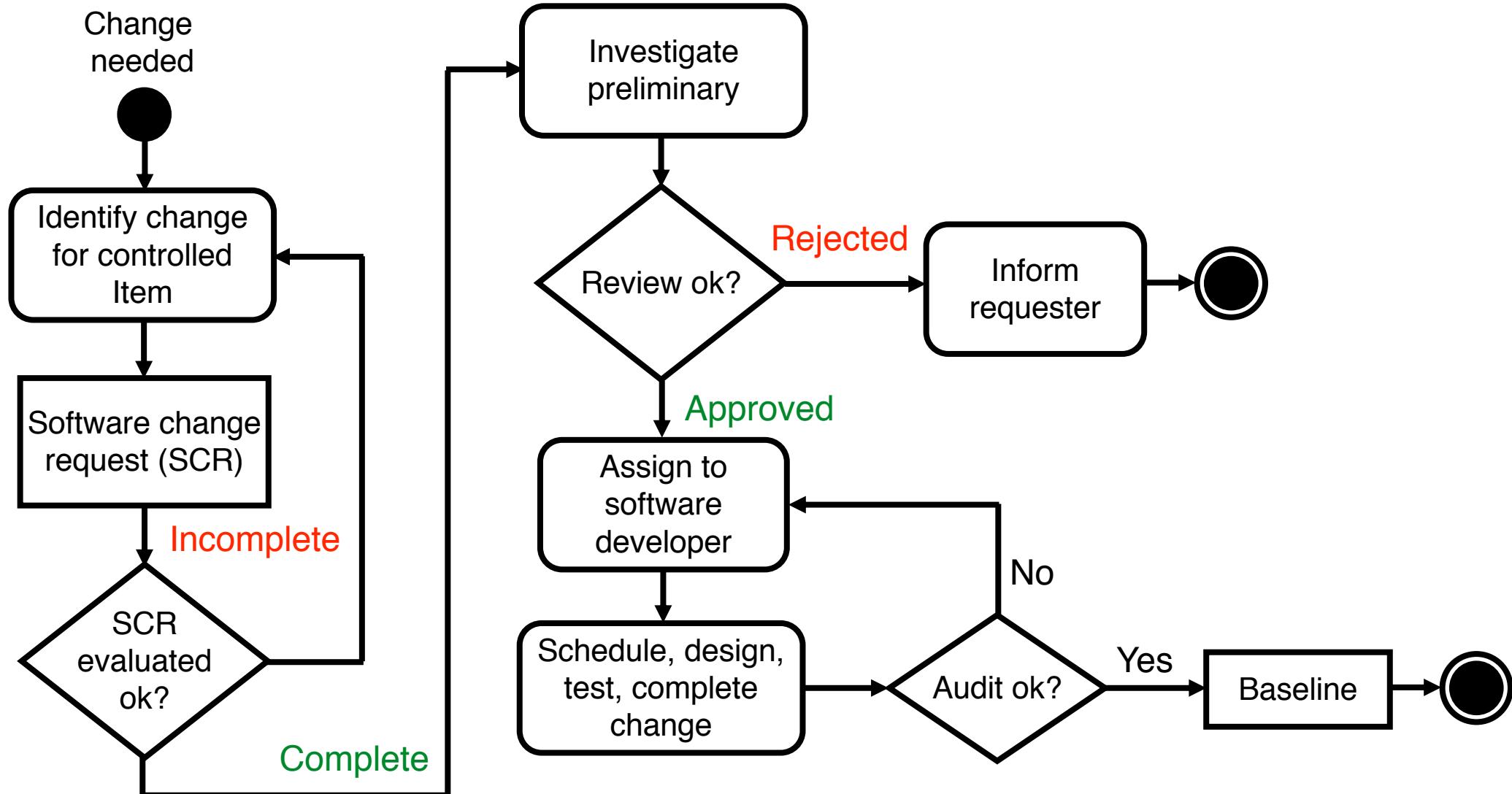
- Variant management

- Management of *coexisting versions*

Change management

- Important: A baseline can only be changed through a formal change control procedure
- **Change management** is the handling of change requests
- General change management process:
 - The change is requested
 - The change request is assessed against requirements and project constraints
 - Following the assessment, the change request is accepted or rejected
 - If it is accepted, the change is assigned to a developer and implemented
 - The implemented change is audited.

Example of a change control process



Change Policy

- The purpose of a **change policy** is to guarantee that each promotion or release conforms to commonly accepted criteria
- Examples for change policies:
 - Promotion policy:** “No developer is allowed to promote source code that has been compiled with error messages or warning messages.”
 - Release policy:** “No baseline can be released without having been beta-tested by at least 500 external people.”

Configuration Management Activities

- ✓ Configuration item identification
 - Modeling the system as a *set of evolving components*
- ✓ Change management
 - Management of *change requests*
- Promotion management
 - Creation of *versions for other developers*
- Branch management
 - Management of *concurrent development*
- Release management
 - Creation of *versions for clients and end users*
- Variant management
 - Management of *coexisting versions*

Outline of the Lecture

- Software Configuration Management
- Change Management
- **Version Control Systems**
 - In-Class Exercise 01: Solve a Merge Conflict
- Branch Management
- Continuous Integration
 - In-Class Exercise 02: Continuous Integration
- Release Management

Version Control Systems (VCS)

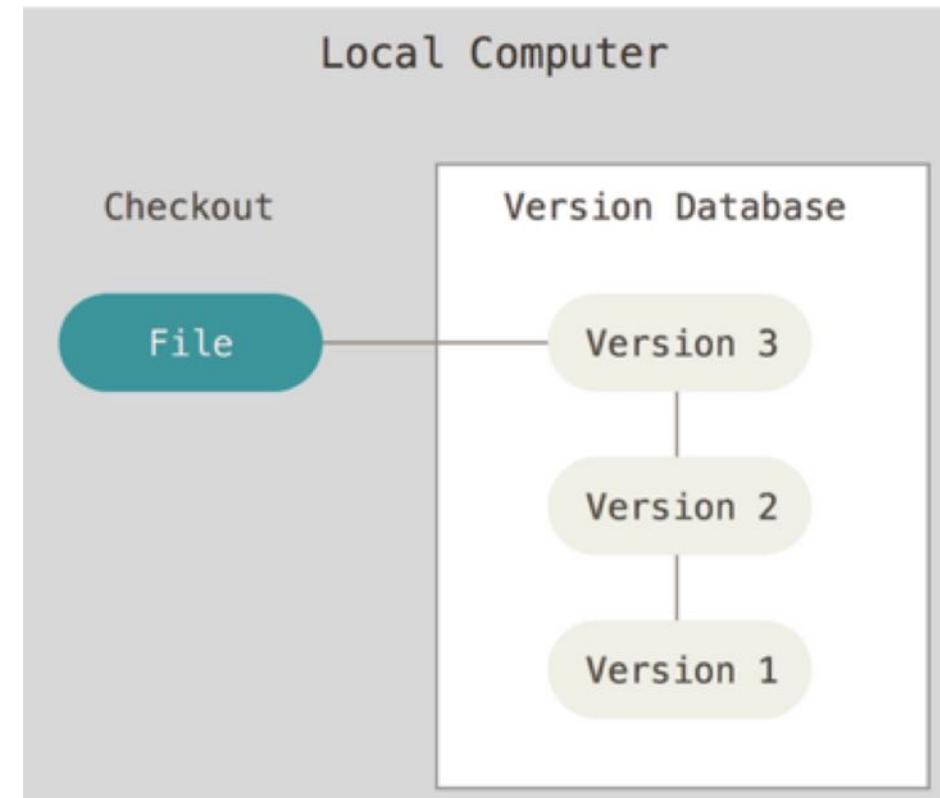
- Version Control Systems allow many software developers to collaborative work on the configuration items in a given project
- VCSs store different versions of configuration items (e.g. source code and configuration data) in a commit history and allow to restore previous versions
- The commit history allows developers to see how the configuration items changed over time and to see who changed a certain item
- Revisions are stored in a repository and developers can check out a revision into a working copy
- Distributed VCSs - also known as distributed revision control or decentralized version control systems -, provide more flexibility and features.

Architectural Choices for Version Control Systems

- Monolithic Architecture
- Repository Architecture
- Peer-to-Peer Architecture

Monolithic Architecture for Version Control

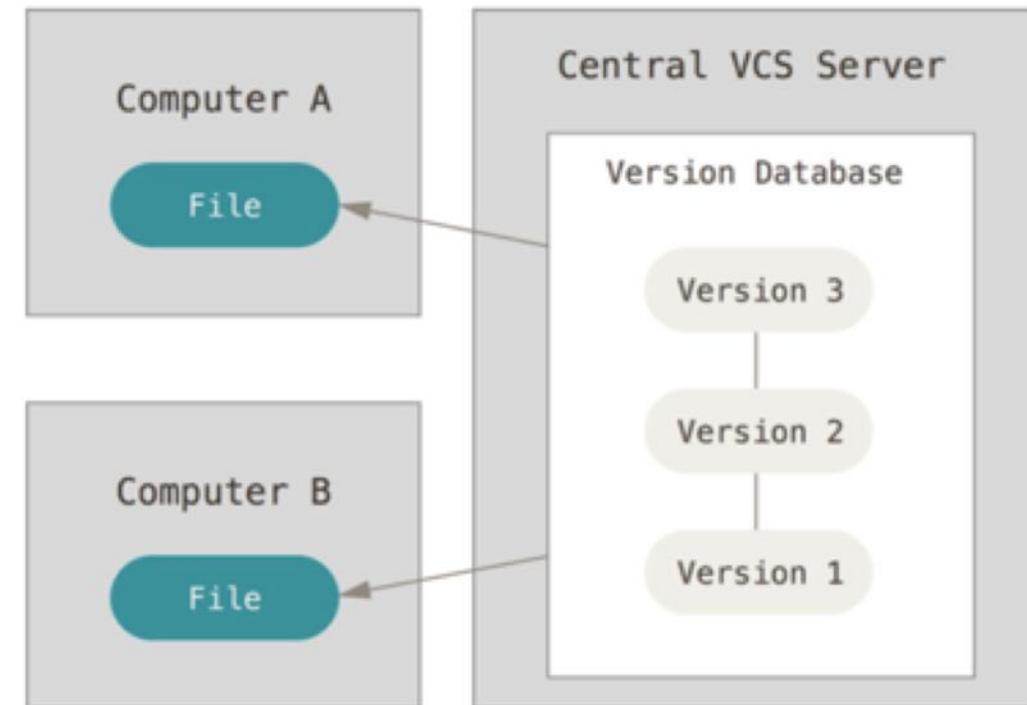
- Programmer has a simple local database that keeps all the changes to files under revision control
- Example of a Monolithic Version Control Architecture:
 - RCS (Revision Control System)
 - Historically the first version control system
 - Walter F. Tichy: RCS—A System for Version Control. In: Software—Practice and Experience. Volume 15, Number 7, 1985.
- Still distributed with many computers today



Source: Chacon, Scott. *Pro git*. Apress, 2009

Repository Architecture for Version Control

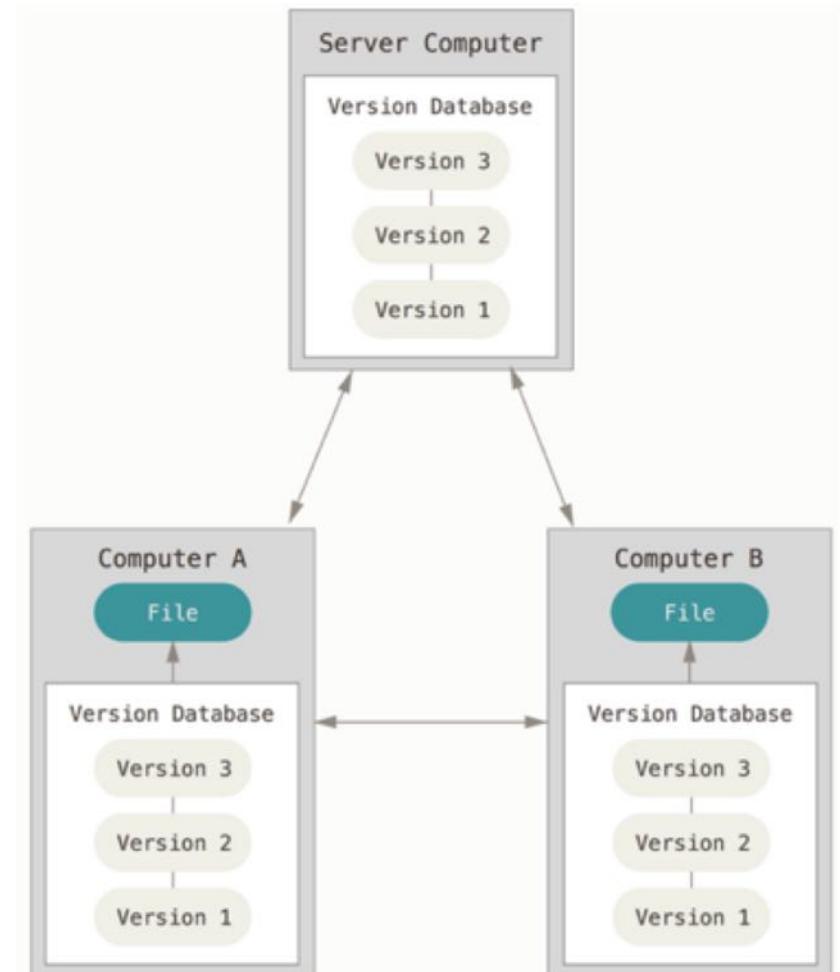
- A single server contains all the versioned files
- Programmers check out files from to the server to their computer,
- They change them and check them back into the server
- Administrators have fine-grained control over who can do what
- **Problem:** Single point of failure in the central Version Control Server (VCS)
 - Possibility of loosing all the versions and their history if the server crashes



Source: Chacon, Scott. *Pro git*. Apress, 2009

Peer-to-Peer Architecture for Distributed Version Control

- Addresses the single point of failure problem
- Each programmer's directory (Computer A, Computer B,...) fully mirrors the master directory (Server Computer)
- Programmers can work offline on their own versions (commits and branches)
- If the Server Computer fails and a programmer has a full copy of the master directory, it can be copied back to the Server Computer
- Example of a Peer-to-Peer Architecture: Git
 - Created by Linus Torvalds for development of the Linux kernel (2005)"



Source: Chacon, Scott. *Pro git*. Apress, 2009

Pros and Cons of Distributed Version Control Systems

- **Advantages**
 - Ability to work offline (local commits)
 - Ability to work incrementally (small commits)
 - Ability to context switch efficiently (lightweight branching)
 - Ability to do exploratory coding efficiently (lightweight branching)
- **Disadvantages**
 - High learning curve
 - Scaling issues

Git

- Open Source Project (<http://git-scm.com>)
- Supports light-weight local branching
- Commands: add, clone, commit, push, fetch, merge, pull
 - **Add:** Select changes for the commit
 - **Clone:** creates a copy of the master repository
 - **Commit:** Commit the change to the local repository
 - **Push:** Push local changes to the master repository
 - **Fetch:** Imports changes from the master repository into the local repository
 - **Merge:** Merge changes into the programmer's directory (working copy)
 - **Pull:** Fetch followed by merge
- Additional information: <http://git-scm.com/documentation>

Most Important Git Commands

git add:

Add changed files to the staging area

git commit:

Commit selected changed files of the staging area to your local repository

git push:

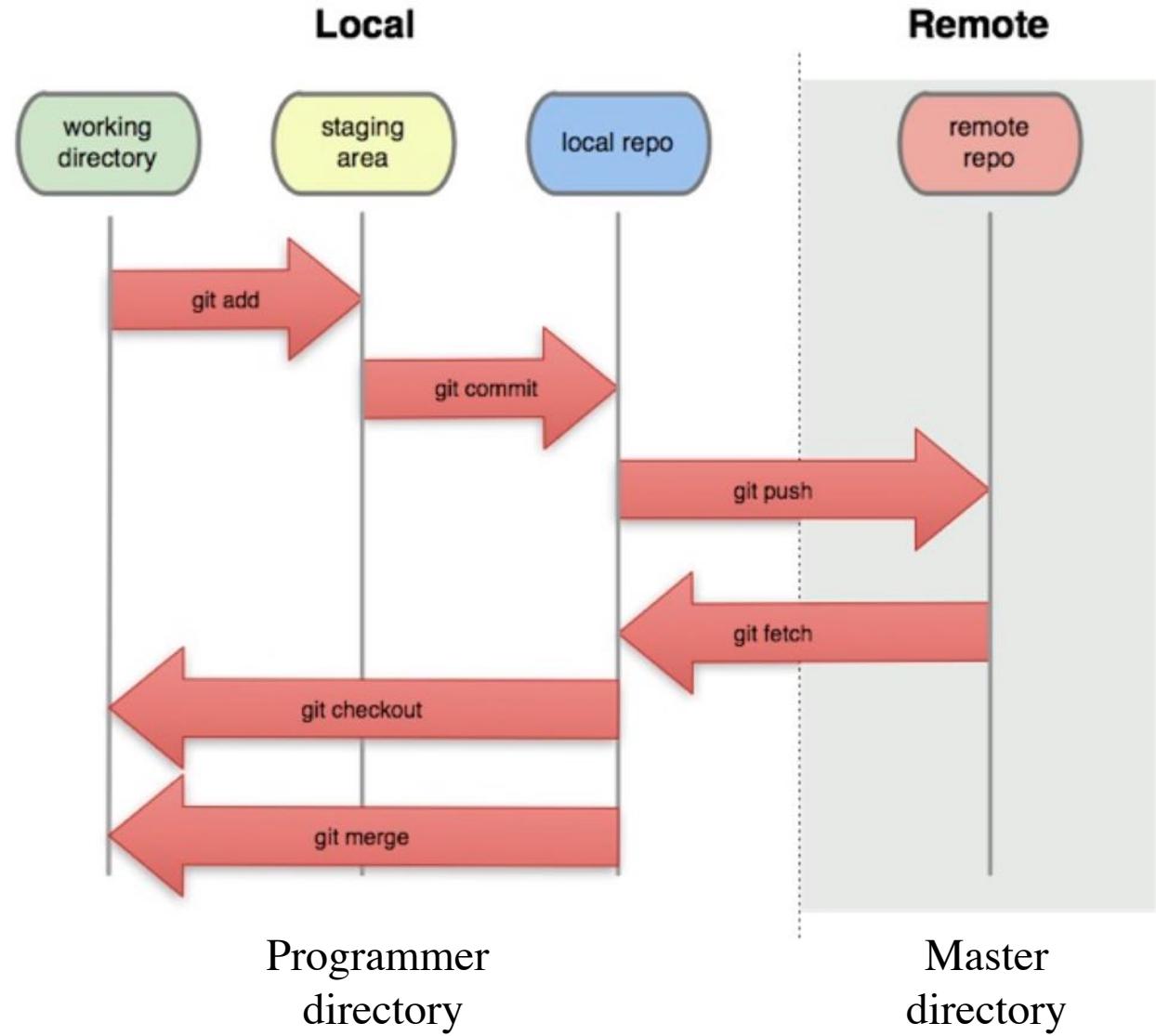
Upload local commits to a remote repository

git pull (fetch & merge):

Download and merge remote commits into your working copy

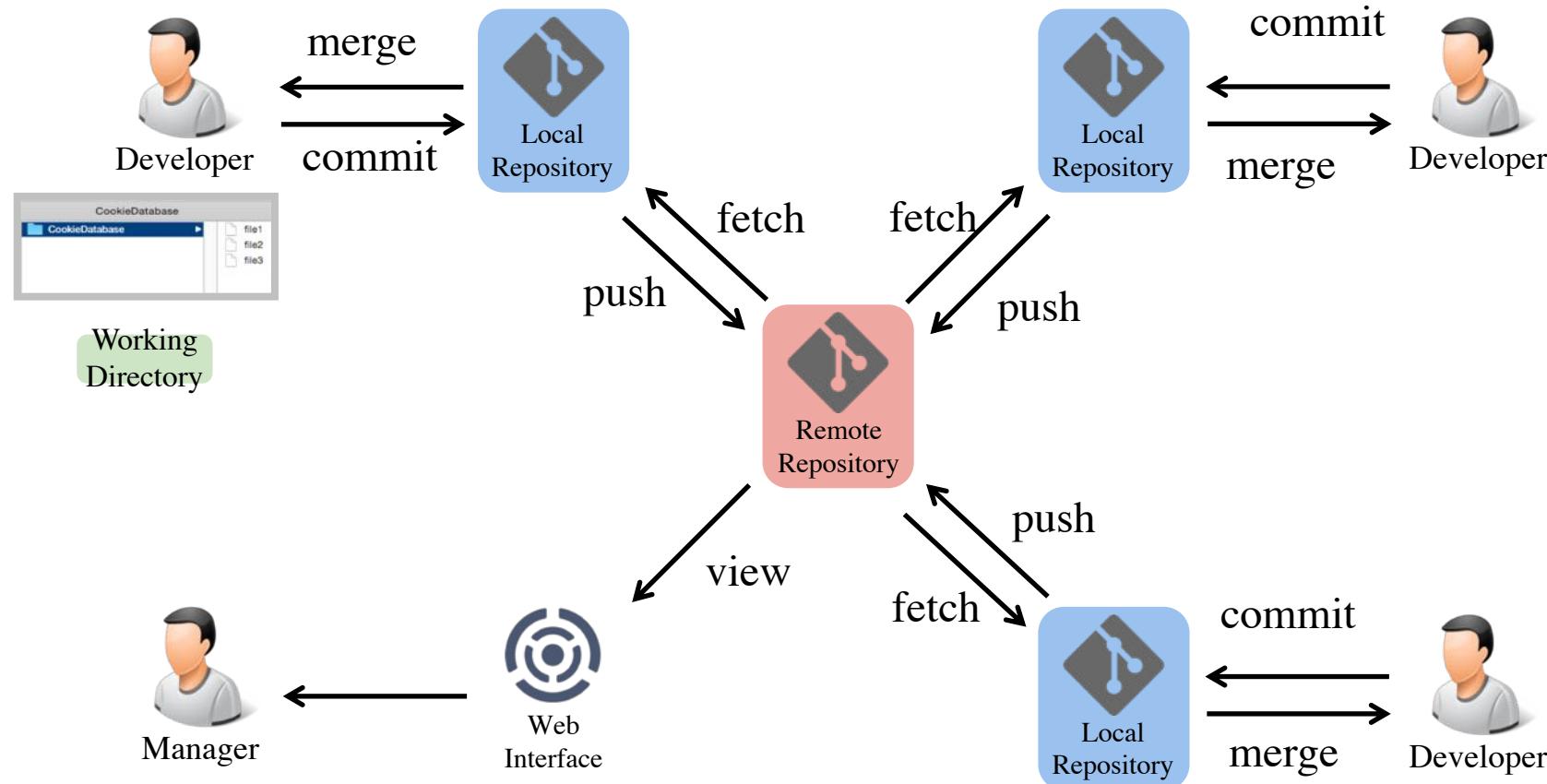
git clone (fetch & initial checkout):

Clone a complete repository into a new working directory



Please note: pull = fetch + merge

Distributed Version Control



This is an informal model. How would you model this in UML?

Committing Changes: Commit Command



Developer

I want to **commit all my changes** in the
CookieDatabase to my Local Repository



Working
Directory

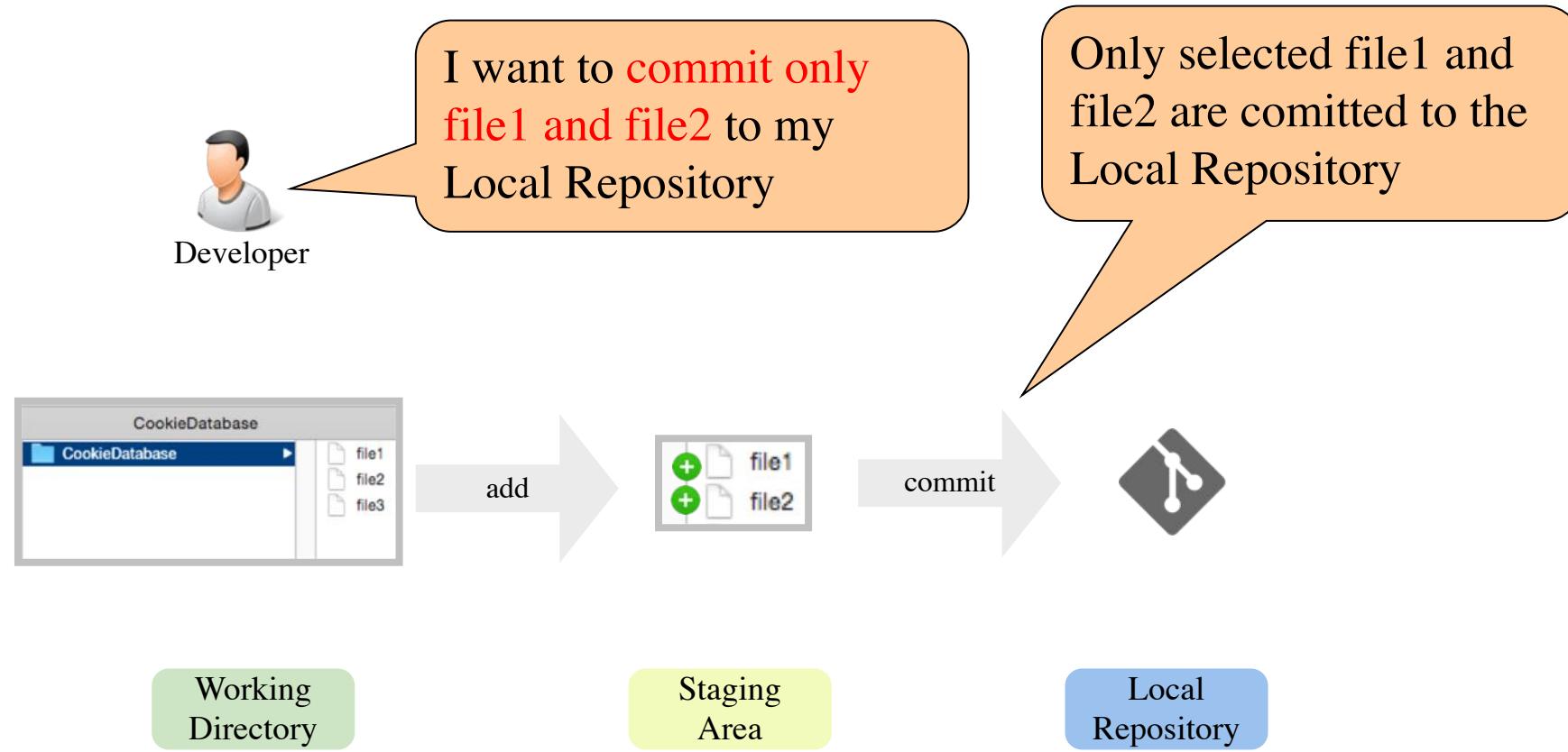
commit



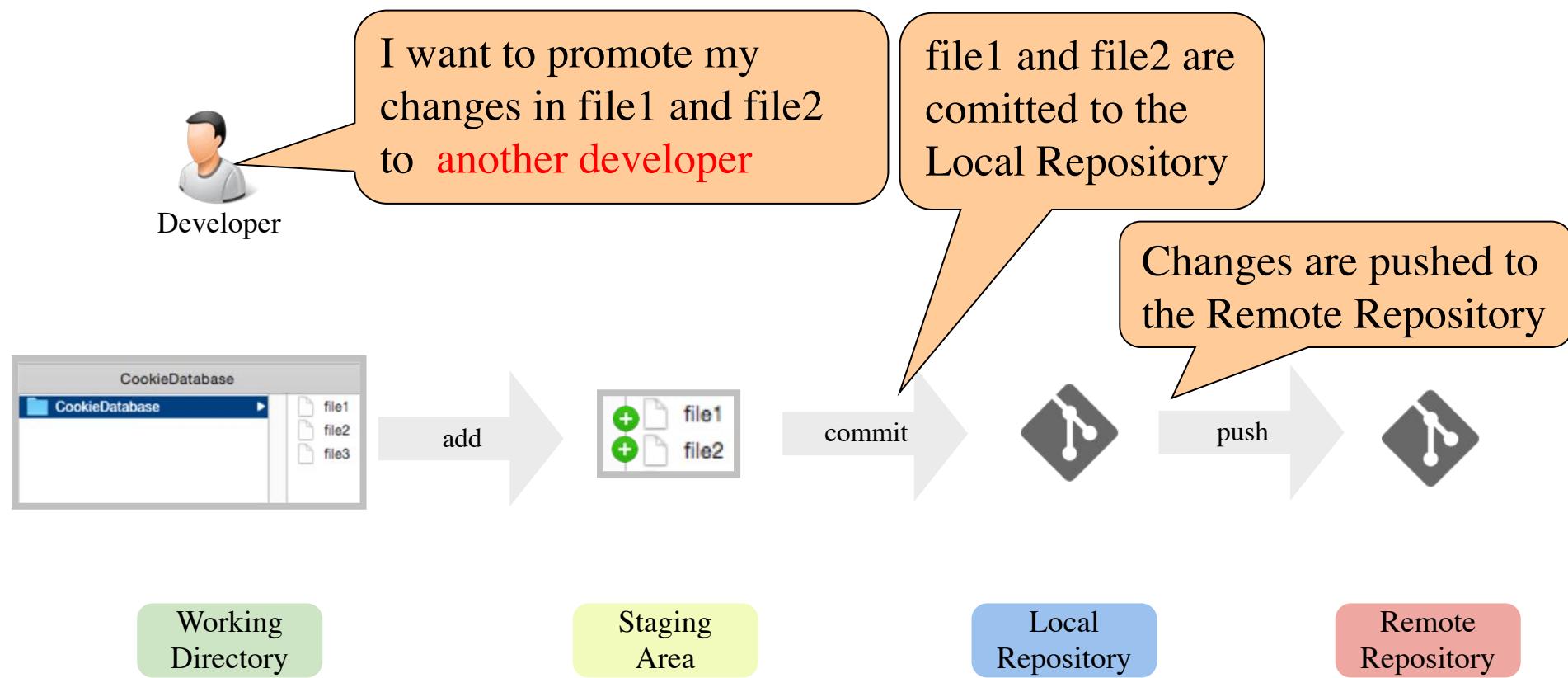
Local
Repository

The git commit command
contains **all** the changes
between two versions

Committing Selected Changes: Add Command



Promoting selected Changes to other developers: Add & Commit & Push Command



This is an informal model. How would you model this in UML?

In-Class Exercise 01: Merge Conflict (ArTEMiS)



Tasks:

1. Start the exercise **Lecture 10 In-Class Exercise 01** on ArTEMiS
2. Clone your exercise repository
3. Commit your solution
4. Push your solution to the remote server
- 5. Change:** Update your solution
6. Resolve the merge conflict

Task 1: Start the exercise on ArTEMiS

Introduction to Software Engineering (Summer 2018) ⓘ

Exercise	Due date	Results	Actions
Show 12 overdue exercises			
Lecture 10 In-Class Exercise 01 Merge Conflict	in a day	You have not started this exercise yet.	Start exercise

Click on **Start exercise**

Task 2: Clone your exercise repository

Introduction to Software Engineering (Summer 2018) ⓘ

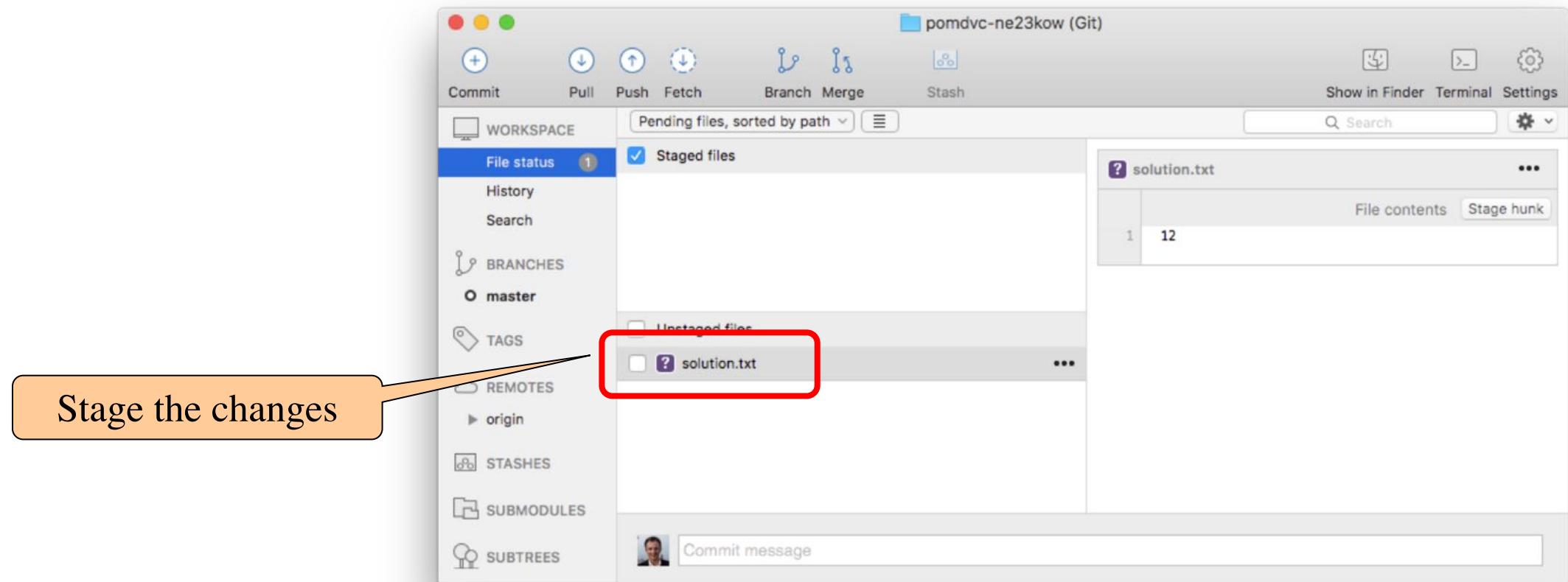
Exercise	Due date	Results	Actions
Show 12 overdue exercises		Clone your personal repository for this exercise: https://ne23kow@repobruegge.in.tum.de/scm/eist2018l10e01/eist2018-l10-e01-exercis	Clone repository
Lecture 10 In-Class Exercise 01	in a day	Merge Conflict	Clone in SourceTree Atlassian SourceTree is the free Git client for Windows or Mac.

Click on Clone in SourceTree

Click on Clone repository

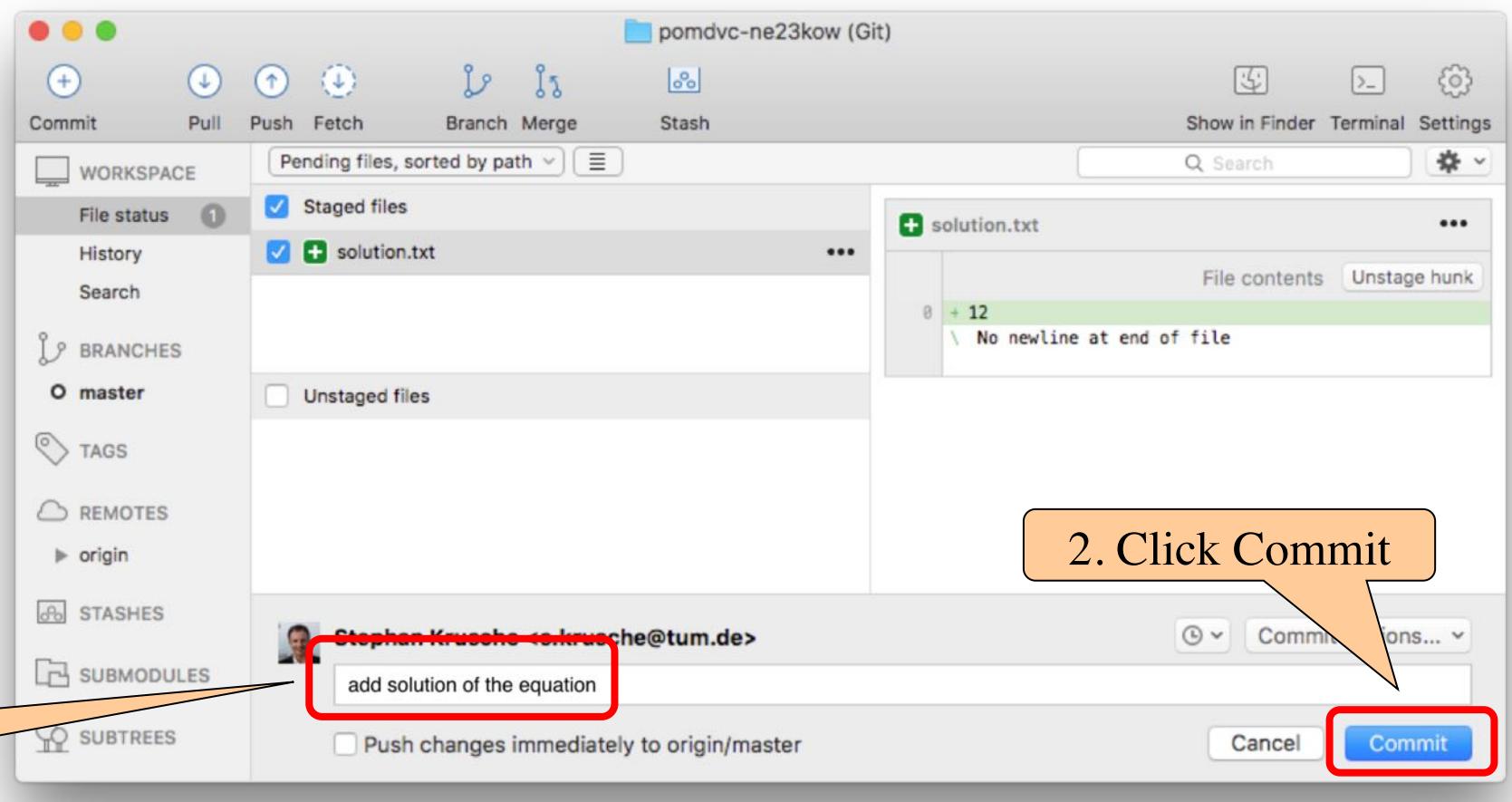
Task 3: Commit your solution

- Create a new file **solution.txt** in your local working copy
- Write the result of the following equation into this file: **6 + 6 = ?**



Task 3: Commit your solution

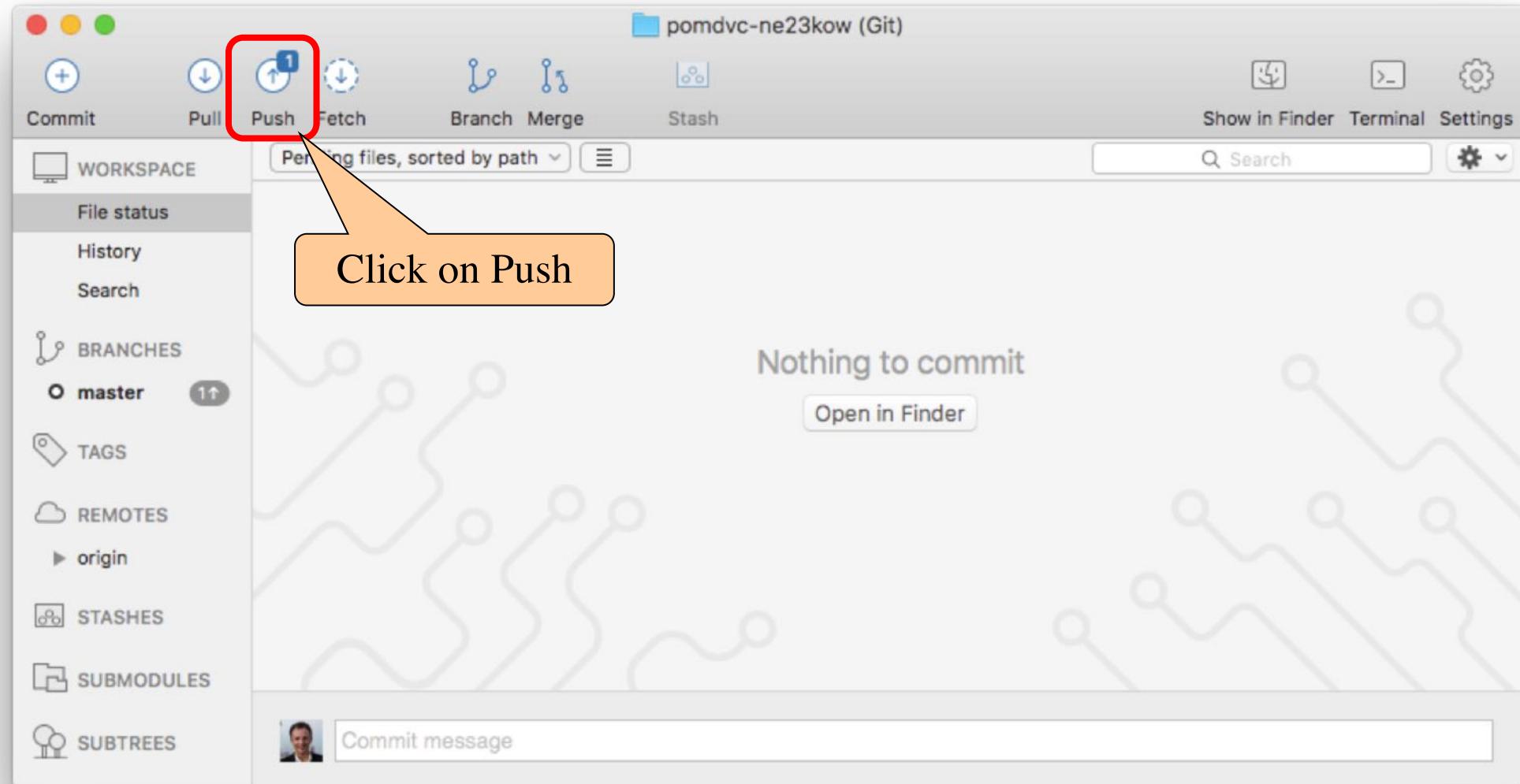
- Insert a **meaningful commit message**, e.g. “add solution of the equation” and click on **Commit**



1. Insert a meaningful commit message

2. Click Commit

Task 4: Push your solution to the remote server

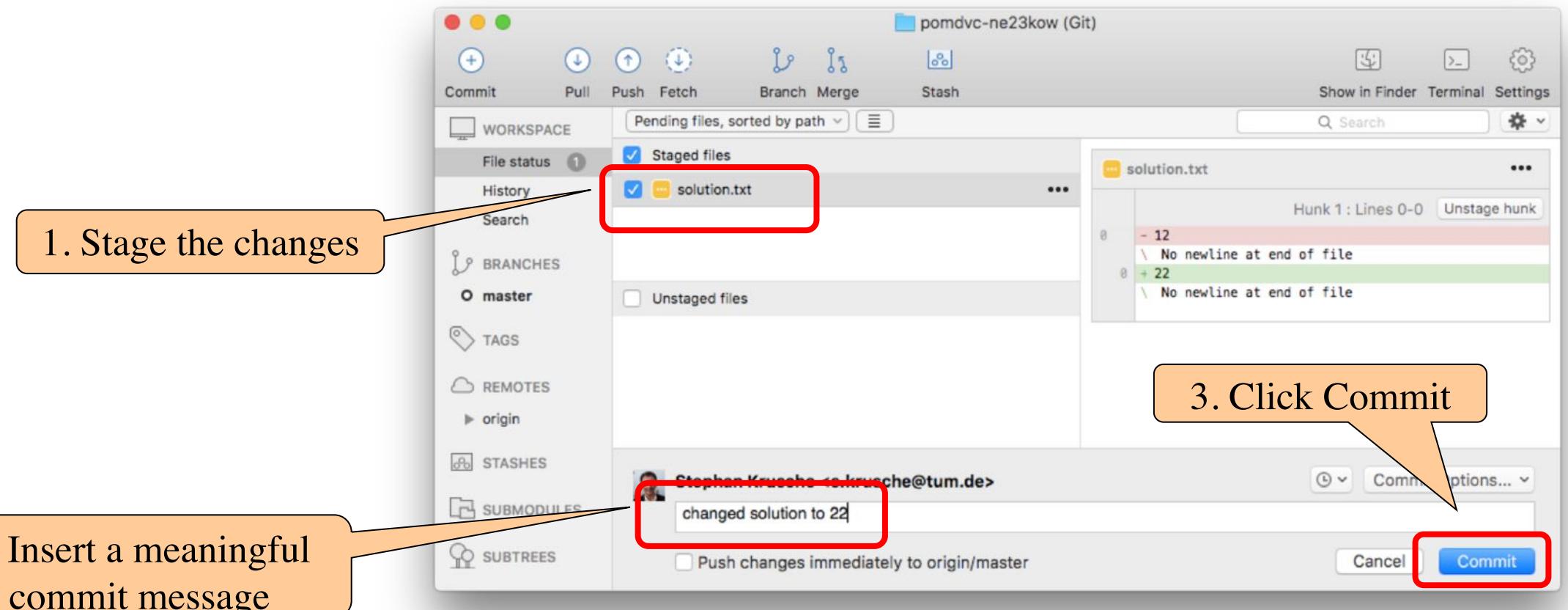


Merge Conflict

- We now simulate a merge conflict so that you learn how to properly handle it
- Merge conflicts happen if two persons work on the same lines in the same files at the same time
- **However:** only one person at the same time can push commits to the remote server
- **Important:** from now on, do not pull changes except we ask you :-)

Task 5: Change: Update your solution

- **Change:** the equation for the solution changed to **$11 + 11 = ?$**
- Update the number in the **solution.txt** file accordingly, stage your changes and commit your changes to the local repository



Task 5: Change: Update your solution

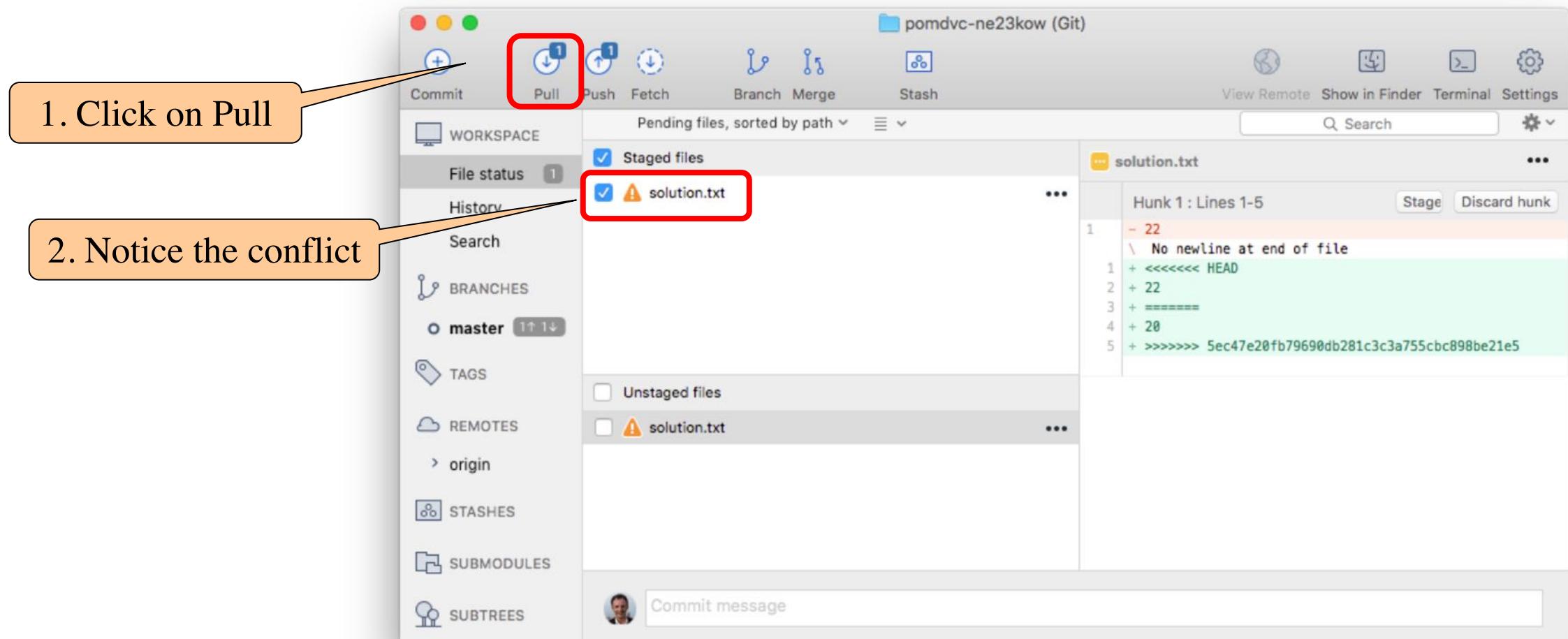
- Try to push the changes



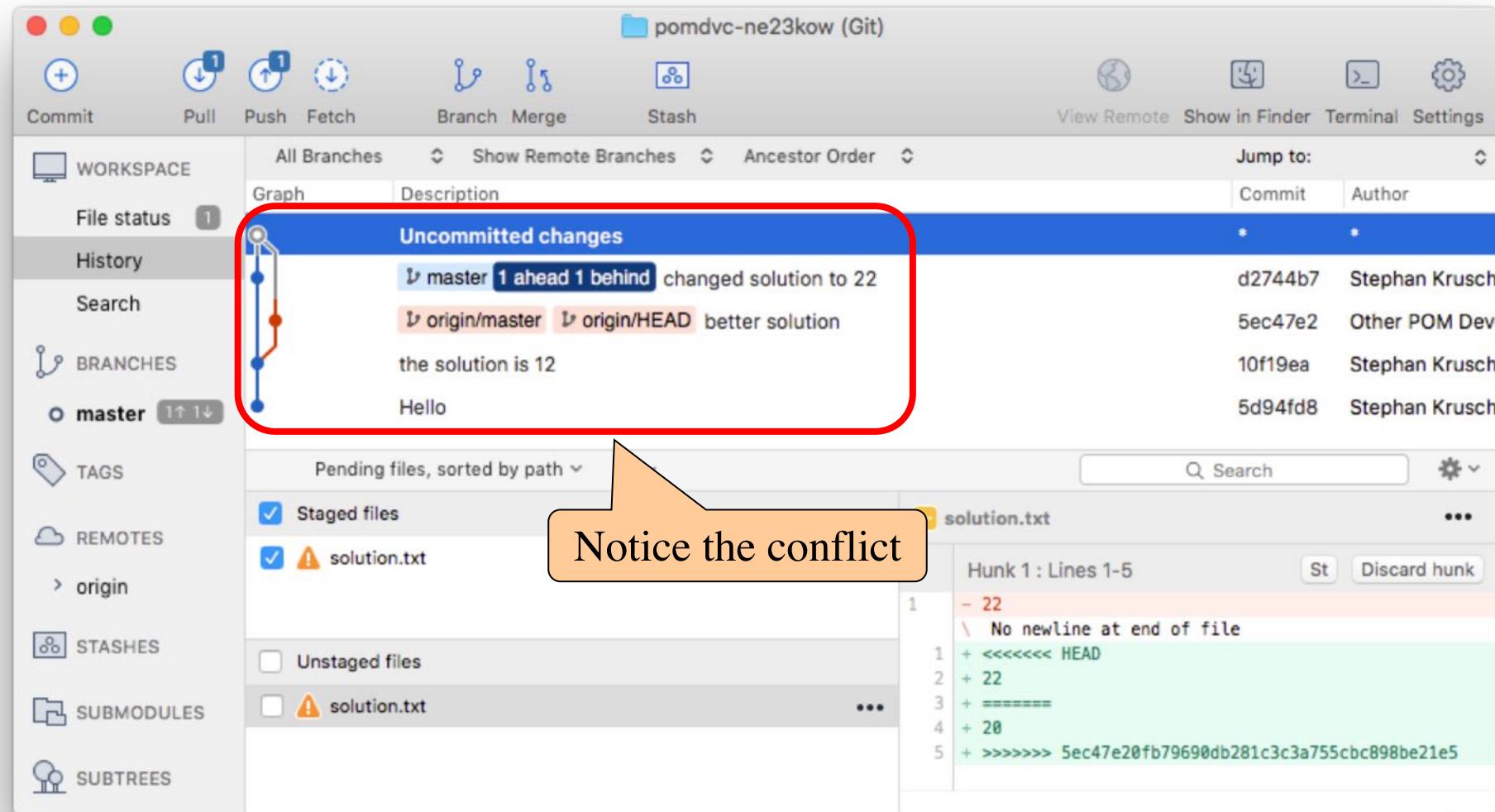
Explanation: another developer pushed his changes before and you first have to pull

Task 5: Change: Update your solution

- Pull the remote changes



Task 6: Resolve the Merge Conflict

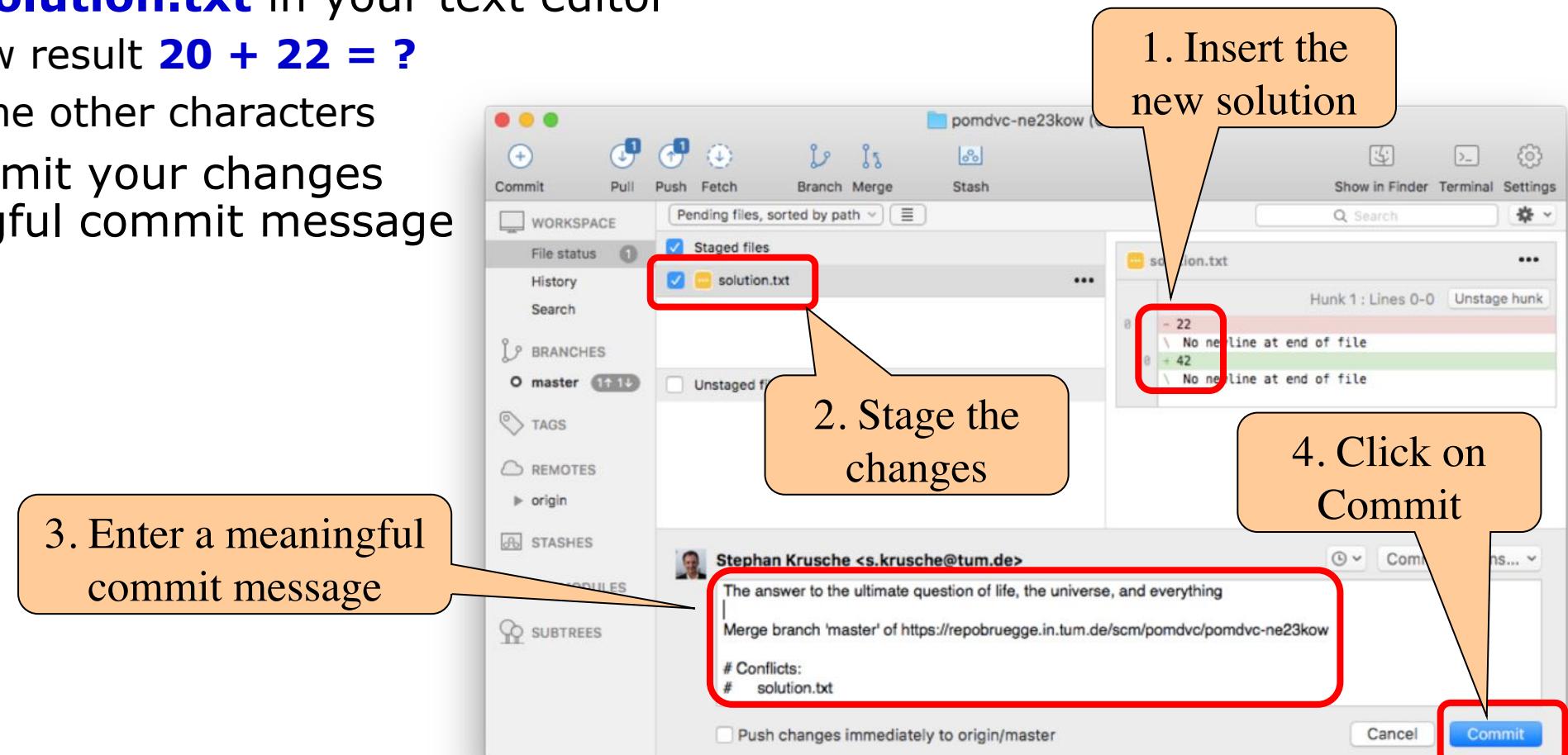


Task 6: Resolve the Merge Conflict

- Three possibilities to resolve merge conflicts:
 - 1) Take mine (i.e. my changes and ignore the other changes)
 - 2) Take theirs (i.e. the other changes and ignore my changes)
 - 3) Merge the overlapping changes manually and decide per case possibly taking both changes

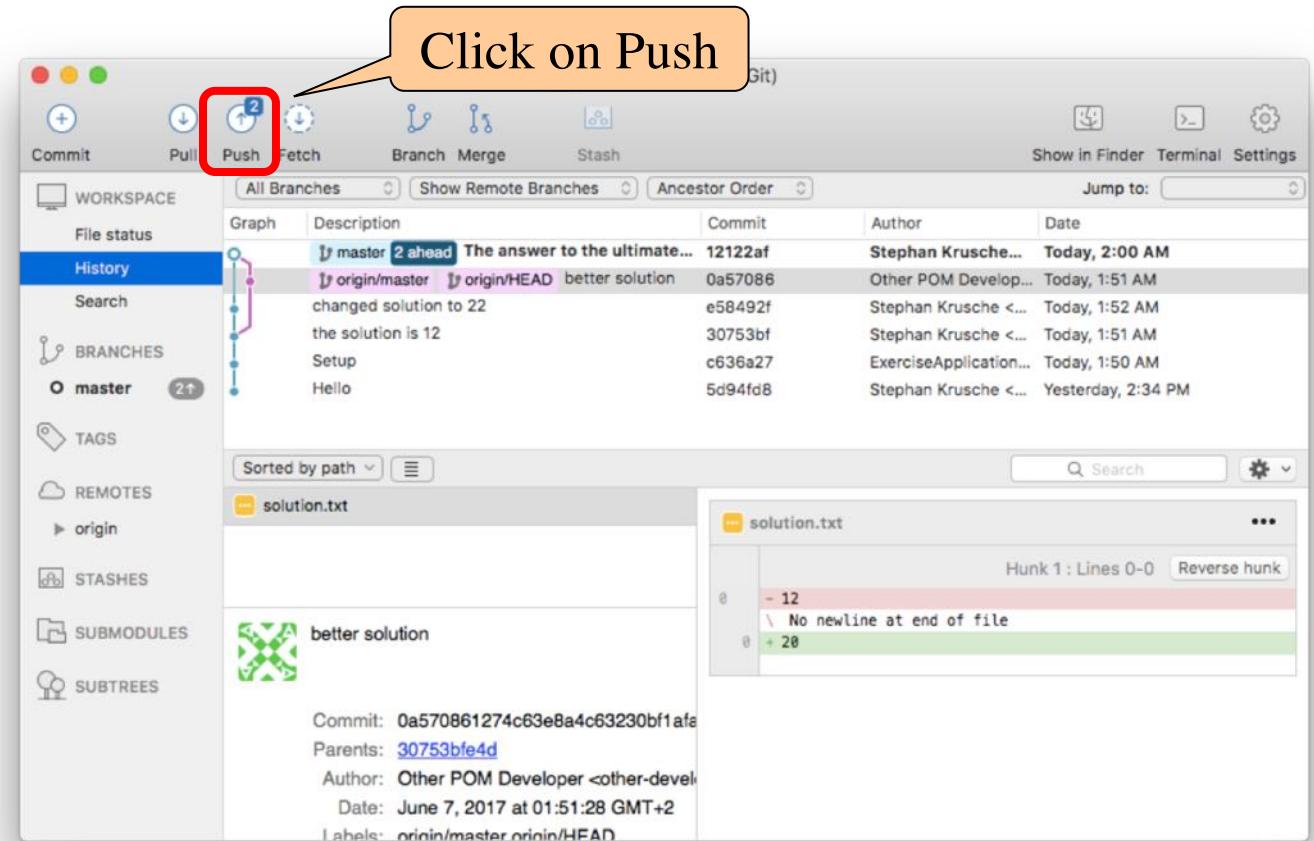
Task 6: Resolve the Merge Conflict

- In this case, we want to sum the 2 numbers of you and the other developer to find the answer to the ultimate question of life, the universe, and everything ;-)
- Open the file **solution.txt** in your text editor
 - Enter the new result **20 + 22 = ?**
 - Remove all the other characters
- Stage and commit your changes with a meaningful commit message



Task 6: Resolve the Merge Conflict

- Notice that we now have 2 commits in the local working copy that are not yet pushed
- Push these 2 commits
- Now it works :-)



Task 6: Resolve the Merge Conflict

- Open ArTEMiS again and notice the green result
- You should see **2 passed, Score: 100%**

Lecture 10 In-Class
Exercise 01 Merge
Conflict

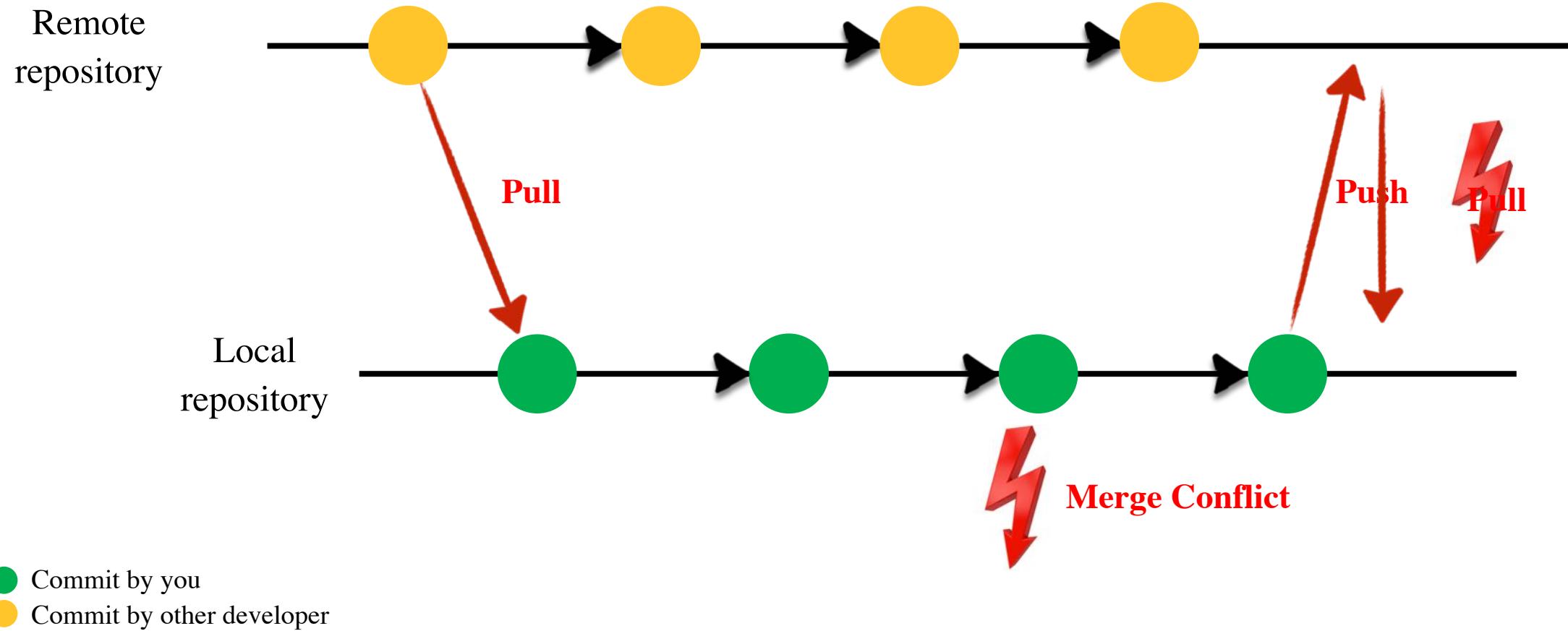
in a day

 2 passed, **Score: 100%**, Submission: a few seconds ago

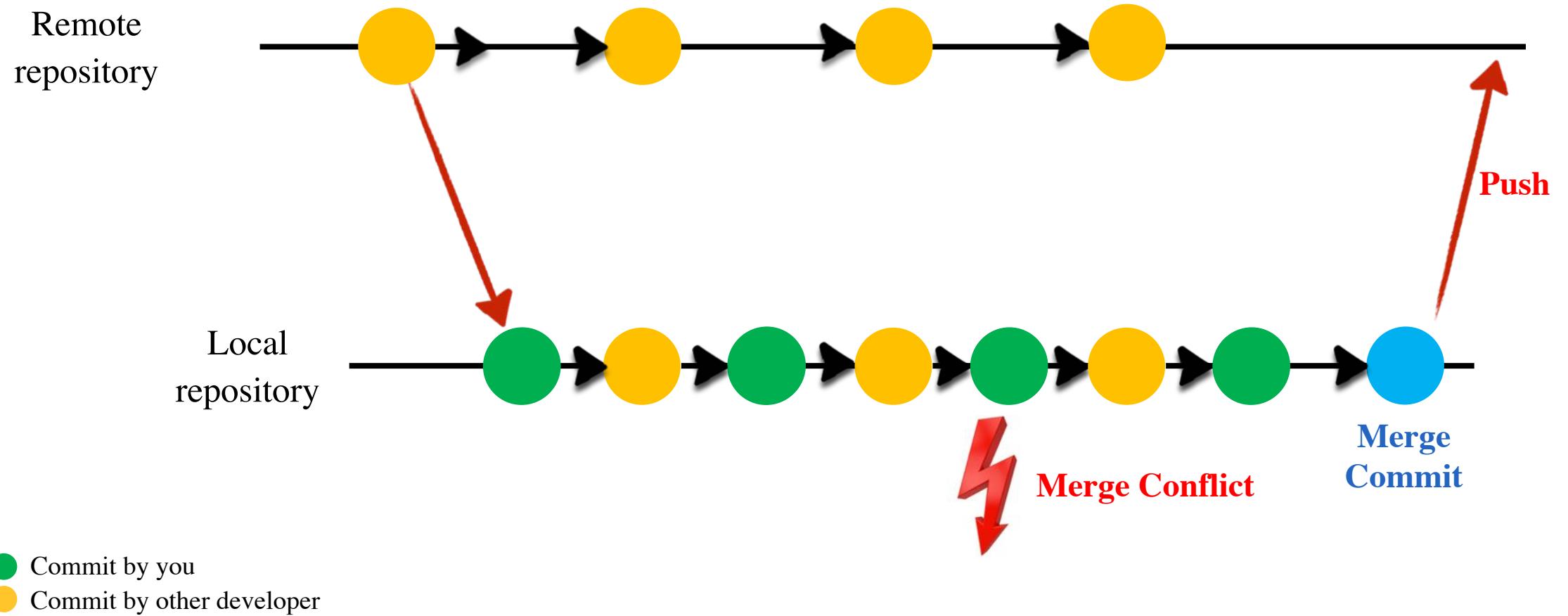


Clone repository

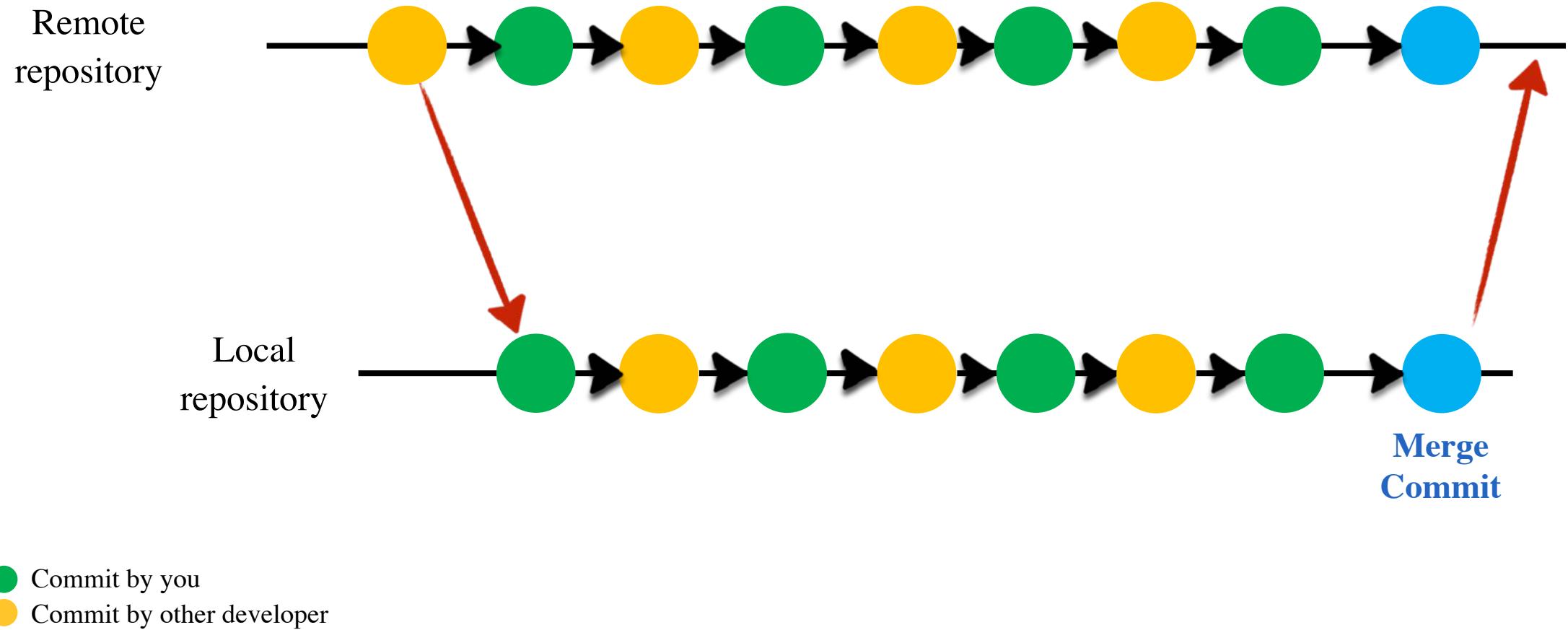
Example merge conflict handling (1)



Example merge conflict handling (2)



Example merge conflict handling (end result)



Distributed Version Control: Best Practices

- Commit related changes
- Commit and push often
- Do not commit half done work
- Test before you commit
- Write meaningful and understandable commit messages
- Do not use version control as a backup system
- Keep your working copy of the repository up to date (regularly pull and push)
- Do not change published (promoted) history

Configuration Management Activities

- ✓ Configuration item identification
 - Modeling the system as a *set of evolving components*
- ✓ Change management
 - Management of *change requests*
- ✓ Promotion management
 - Creation of *versions for other developers*
- Branch management
 - Management of *concurrent development*
- Release management
 - Creation of *versions for clients and end users*
- Variant management
 - Management of *coexisting versions*

15 Minute Break



Outline of the Lecture

- Software Configuration Management
- Change Management
- Version Control Systems
 - In-Class Exercise 01: Solve a Merge Conflict

→ Branch Management

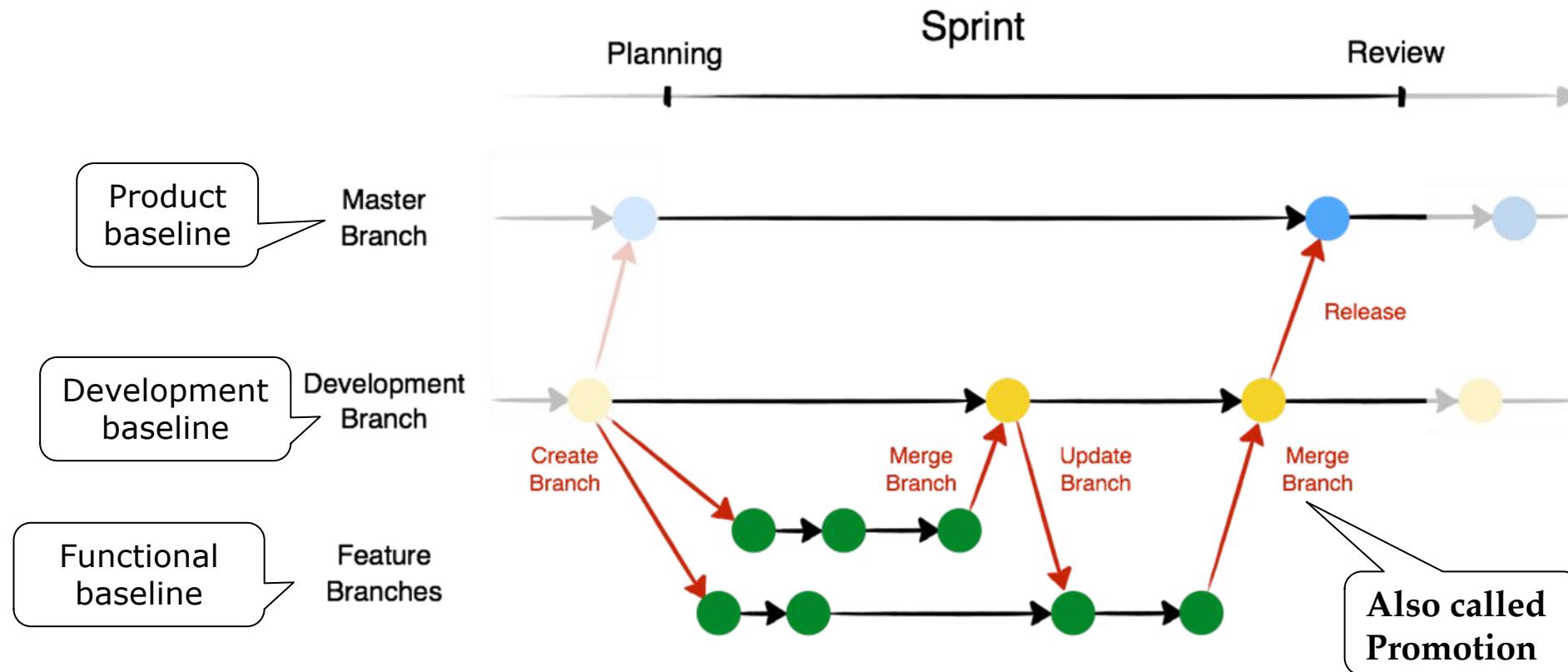
- Continuous Integration
 - In-Class Exercise 02: Continuous Integration
- Release Management

Types of Baselines

- As systems are developed, a series of baselines is developed, usually after a review
 - Product baseline
 - Developmental baseline
 - Functional baseline
- Types of reviews: Analysis review, design review, code review, system test, client acceptance test.

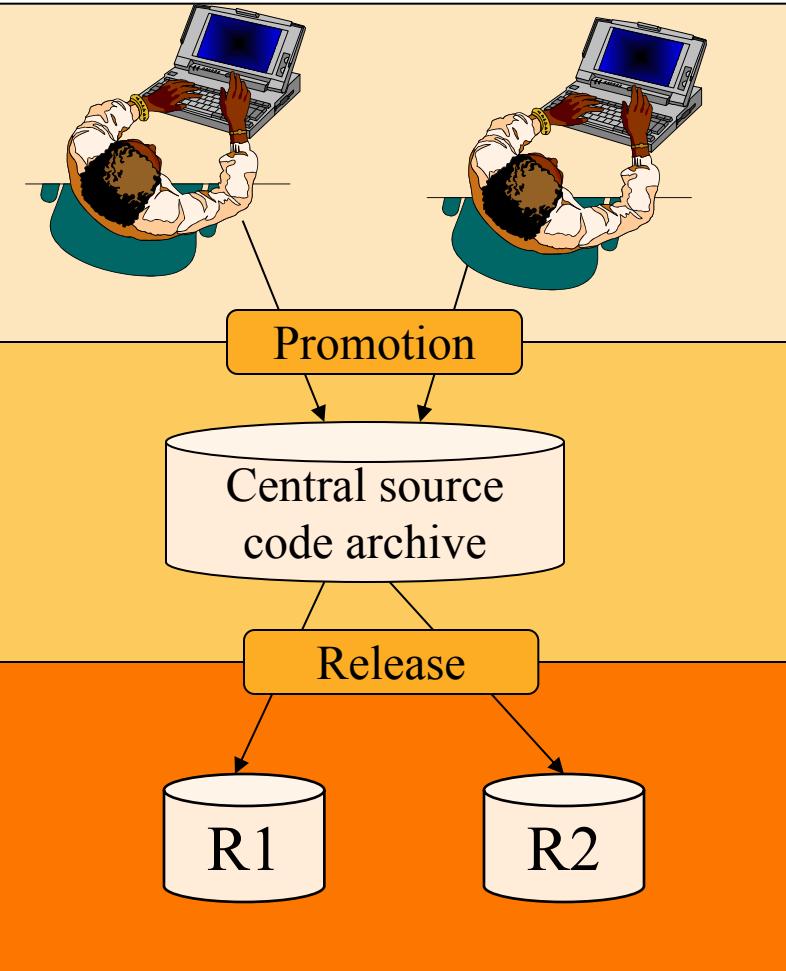
Example of the Git Branch Management Model (simplified)

- **Master Branch:** External Release (e.g. Product Increment)
- **Development Branch:** Internal Release
- **Feature Branches:** Incremental development and explorations



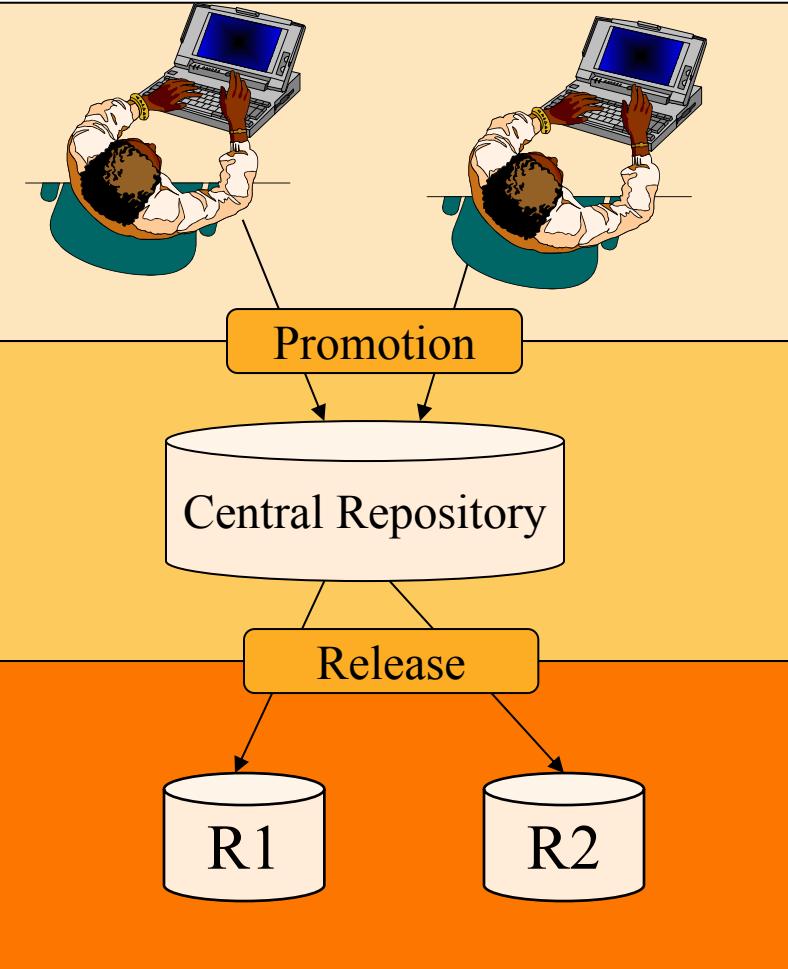
Review: SCM Directories in the IEEE Std 828

- Programmer's Directory
 - (IEEE Std: "Dynamic Library")
 - Completely under control of one programmer
- Master Directory
 - (IEEE Std: "Controlled Library")
 - Central directory of all promotions
- Software Repository
 - (IEEE Std: "Static Library")
 - Externally released baselines.



Git Terminology mapped to SCM Directories the IEEE Std 828

- Feature Branch
 - Completely under control of one programmer
- Development Branch
 - Central directory of all promotions
 - Candidates for releases
- Master Branch
 - External releases



Best Practices for Branch Management

- You can use branches to control concurrent development and explorations, but you should agree on a branching model
 - Example: Map functional requirements to feature branches
 - Minimize feature branch lifetime (not more than “some” days)
 - Split complex requirements into smaller ones
 - Use functional decomposition
 - Use include and extend use cases
 - Pull regularly from the development branch into the feature branch
- Otherwise your branches diverge heavily and you will have serious merge conflicts.

Configuration Management Activities

- ✓ Configuration item identification
 - Modeling the system as a *set of evolving components*
- ✓ Promotion management
 - Creation of *versions for other developers*
- ✓ Change management
 - Management of *change requests*
- ✓ Branch management
 - Management of *concurrent development*
- Release management
 - Creation of *versions for clients and end users*
- Variant management
 - Management of *coexisting versions*

Outline of the Lecture

- Software Configuration Management
- Change Management
- Version Control Systems
 - In-Class Exercise 01: Solve a Merge Conflict
- Branch Management
- Continuous Integration
 - In-Class Exercise 02: Continuous Integration
- Release Management

Reasons for Continuous Integration

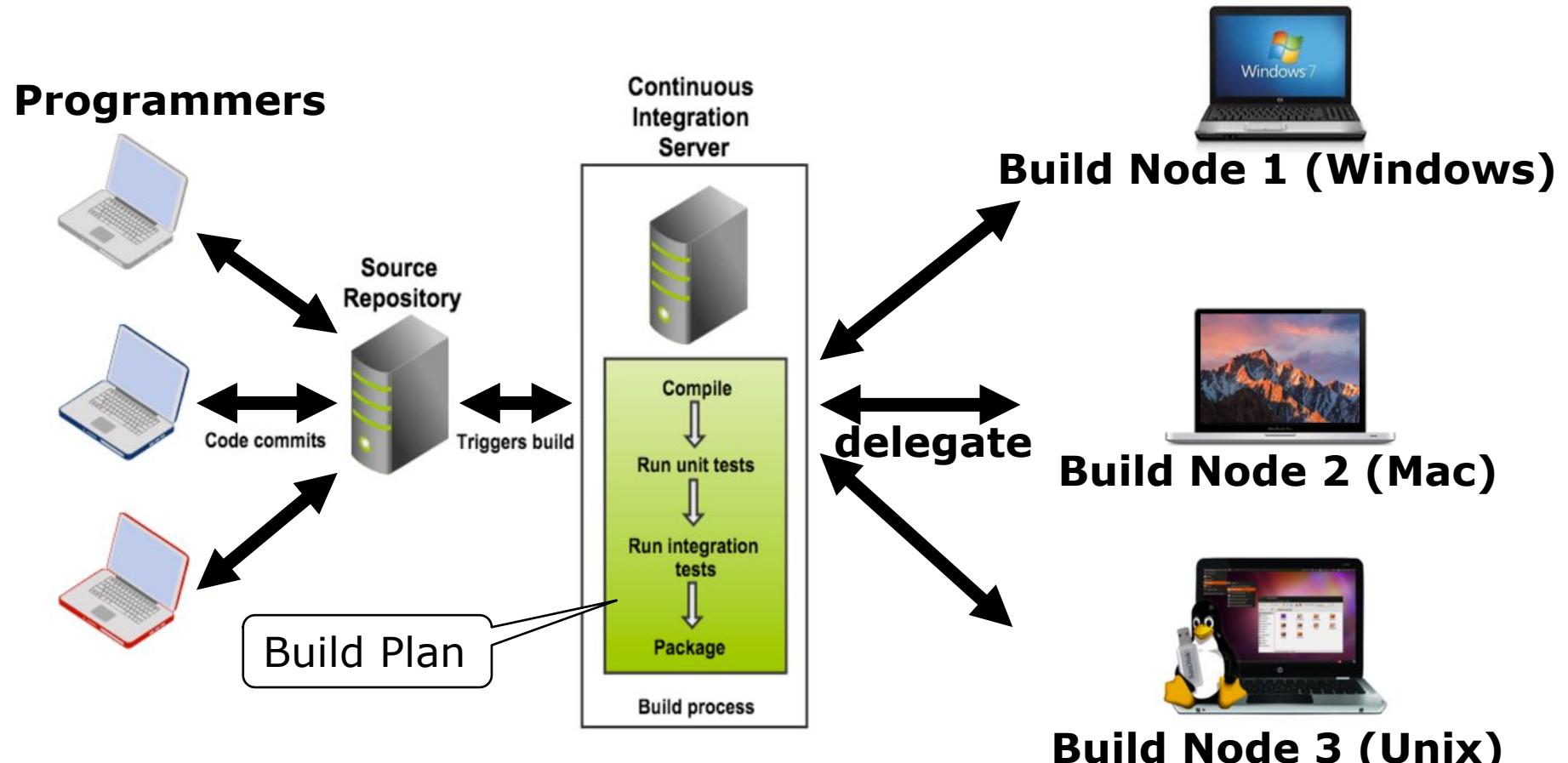
- **Risk #1:** The later integration occurs in a project, the bigger is the risk that unexpected failures occur
- **Risk #2:** The higher the complexity of the software system, the more difficult is the integration of its components
- Continuous integration addresses these risks by building as early and frequently as possible
- Additional advantages:
 - There is always an executable version of the system
 - Team members have a good overview of the project status

Definition: Continuous Integration

- **Continuous Integration:** A software development technique where members of a team integrate their work frequently.
- Usually each person integrates at least daily, leading to multiple integrations per day.
- Each integration is verified by an automated build which includes the execution of tests – regression tests – to detect integration errors as quickly as possible.

Source: <http://martinfowler.com/articles/continuousIntegration.html>

Continuous Integration Overview



Challenge: Can you formulate the build plan in UML?

Advantages of Continuous Integration

- + There is always an executable version of the system
- + Developers and managers have a good overview of the project status
- + Automatic regression testing

Regression Testing

- **Goal:** verify that software previously developed and tested still performs correctly even after it was changed or interfaced with other software
 - + **Benefit:** finds errors in the existing source code immediately after a change is introduced
 - **Drawback:** can be very costly to execute a large test suite after each change
- Techniques:
 - Retest all
 - Regression test selection (e.g. using dependency analysis)
 - Test case prioritization
- When to execute the selected test cases?
 - After each change, Nightly, Weekly



Apache Maven



- Open source build and dependency management tool
- Good for build automation for Java projects
- Uses conventions for build structure / procedure
- Stores libraries and plugins in a central repository → easy reuse of existing components such as unit tests, logging frameworks
- **POM = Project Object Model**
 - Main artifact and configuration file: **pom.xml**
- Support for multiple build lifecycle phases, e.g. compile, test, package, install, deploy
 - Very powerful build tool
- <https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>

Example pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>de.tum.in.www1.UniversityApp</groupId>
  <artifactId>UniversityApp</artifactId>
  <packaging>jar</packaging> <dependencies><dependency>
  <version>1.0</version>
  <name>UniversityApp</name>
    <dependencies><dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency></dependencies>
```

`<build>`

```
  <sourceDirectory>${project.basedir}/src</sourceDirectory>
  <plugins><plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.3</version>
  </plugin>
  <plugin>
    <artifactId>maven-jar-plugin</artifactId>
    <version>3.0.2</version>
    <configuration><archive><manifest>
      <mainClass>de.tum.in.www1.UniversityApp</mainClass>
    </manifest></archive></configuration>
  </plugin></plugins></build>
</project>
```

Packaging format

Dependency to
external framework

Typical commands

- mvn clean
- mvn compile
- mvn test
- mvn package
- mvn deploy

Build and package
instructions

In-Class Exercise 02: Continuous Integration



1. Start the exercise **Lecture 10 In-Class Exercise 02** on ArTEMiS
2. Clone your exercise repository
3. Import the project into Eclipse
4. Run the application
5. Configure Maven to run tests
6. Run the test
7. Fix the failing test
8. Configure Maven to create an executable
9. Commit and push your changes
10. Download the Java executable
11. Add a new dependency in Maven
12. Commit and push your changes

Task 1: Start the exercise on ArTEMiS

Introduction to Software Engineering (Summer 2018) ⓘ

Exercise	Due date	Results	Actions
Show 12 overdue exercises			
Lecture 10 In-Class Exercise 02 Continuous Integration	in a day	You have not started this exercise yet.	Start exercise

Click on **Start exercise**

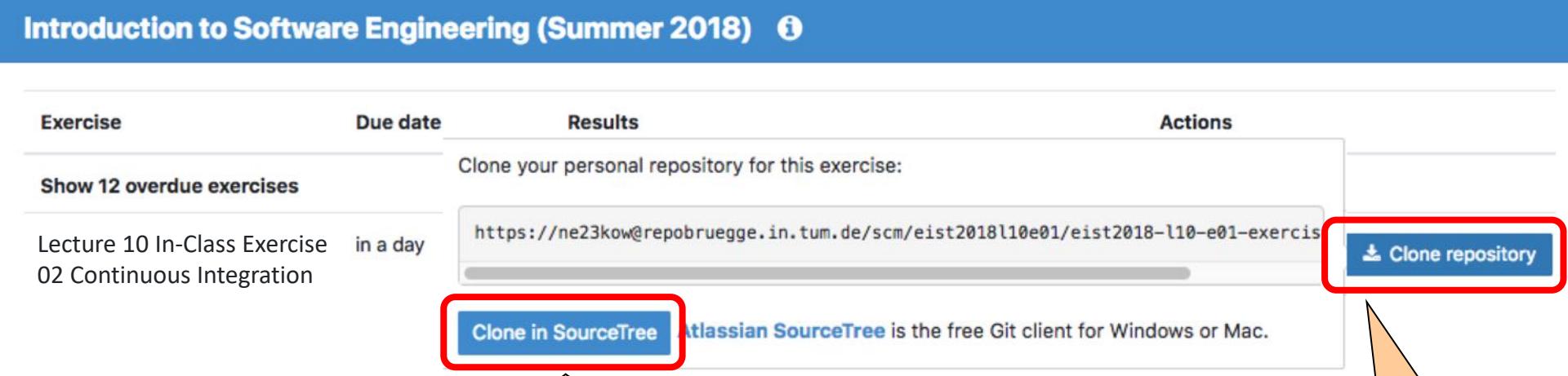
Task 2: Clone your exercise repository

Introduction to Software Engineering (Summer 2018) ⓘ

Exercise	Due date	Results	Actions
Show 12 overdue exercises		Clone your personal repository for this exercise: https://ne23kow@repobruegge.in.tum.de/scm/eist2018l10e01/eist2018-l10-e01-exercis	Clone repository
Lecture 10 In-Class Exercise 02 Continuous Integration	in a day	Clone in SourceTree Atlassian SourceTree is the free Git client for Windows or Mac.	

Click on Clone in SourceTree

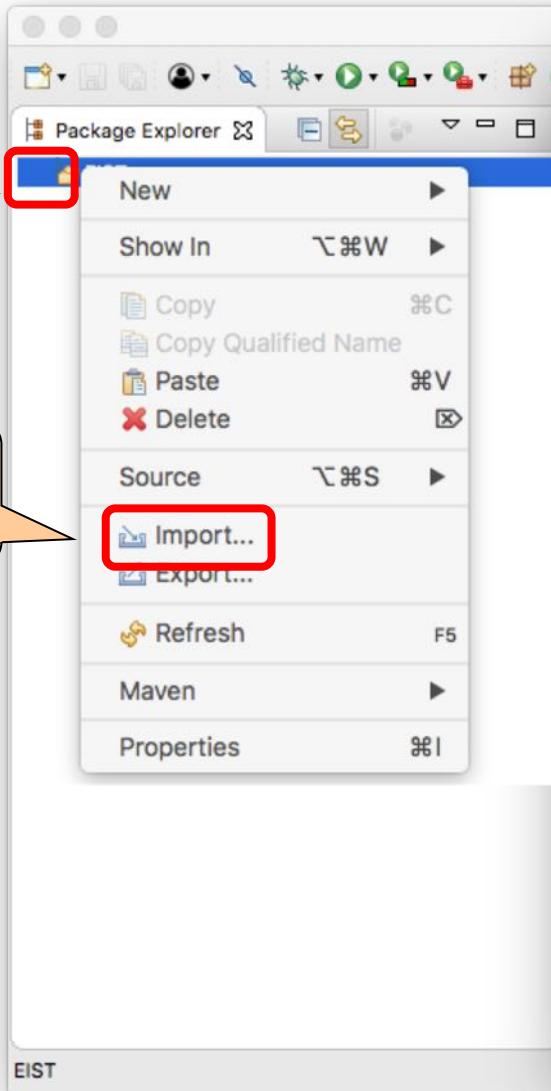
Click on Clone repository



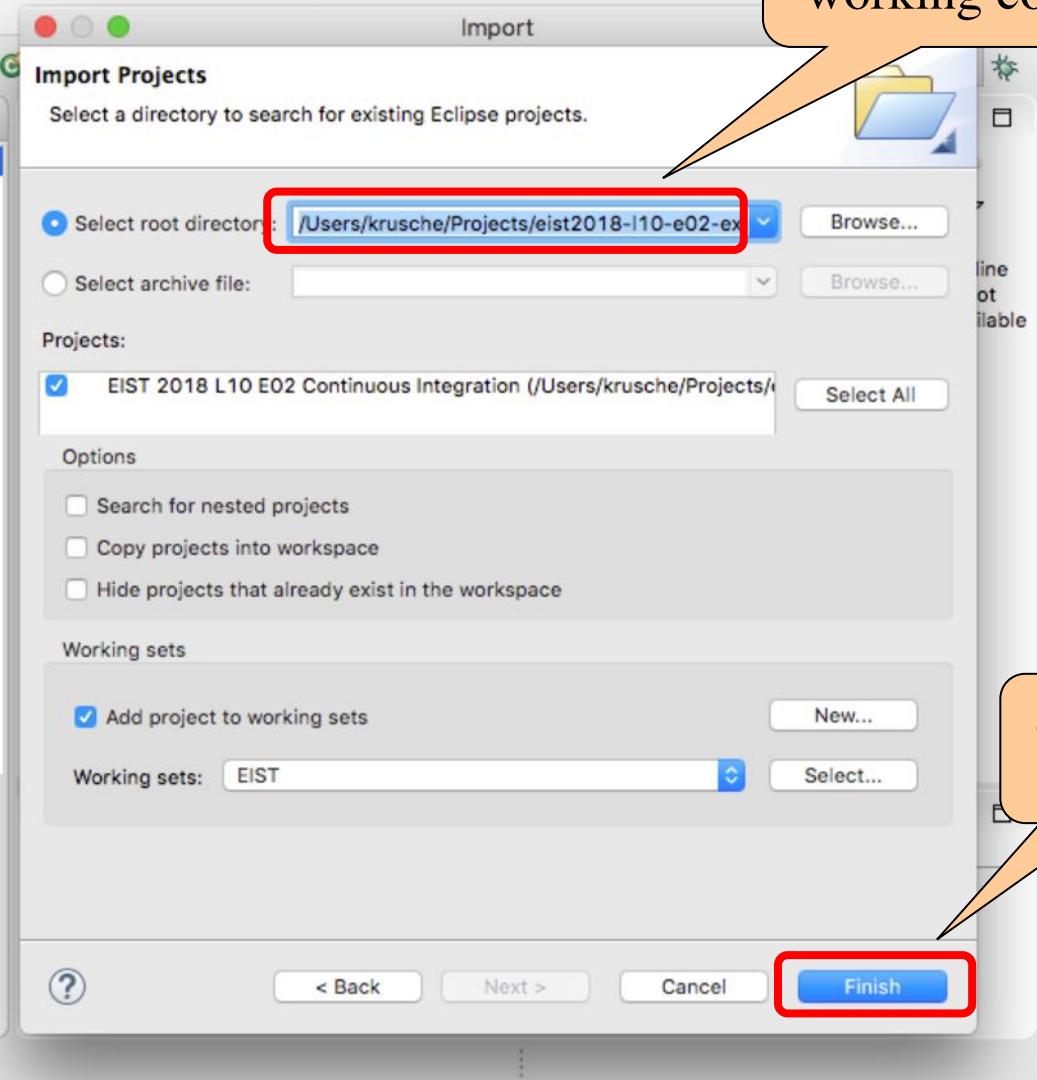
The screenshot shows a table from a software engineering course website. The first row has a blue header bar with the title. The second row contains a single exercise entry. The third row is empty. The 'Actions' column for the first row contains two buttons: 'Clone repository' (highlighted with a red box) and 'Clone in SourceTree' (highlighted with a red box). Below the table, two orange callout boxes point to these buttons with the text 'Click on Clone in SourceTree' and 'Click on Clone repository' respectively.

Task 3: Import the project into Eclipse

1. Right click



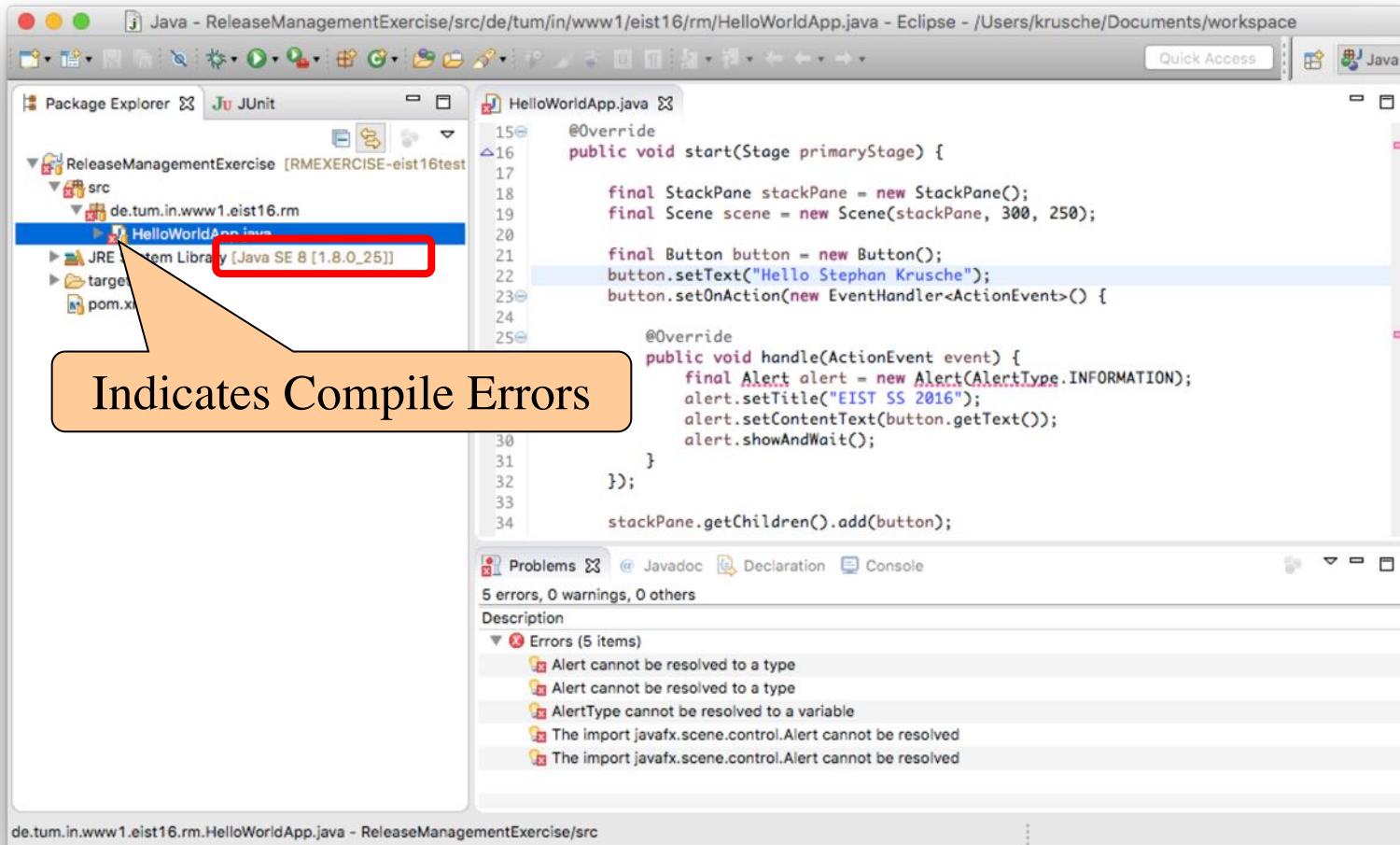
2. Click
Import...



3. Choose your
working copy

4. Click
Finish

In case you have compiler errors



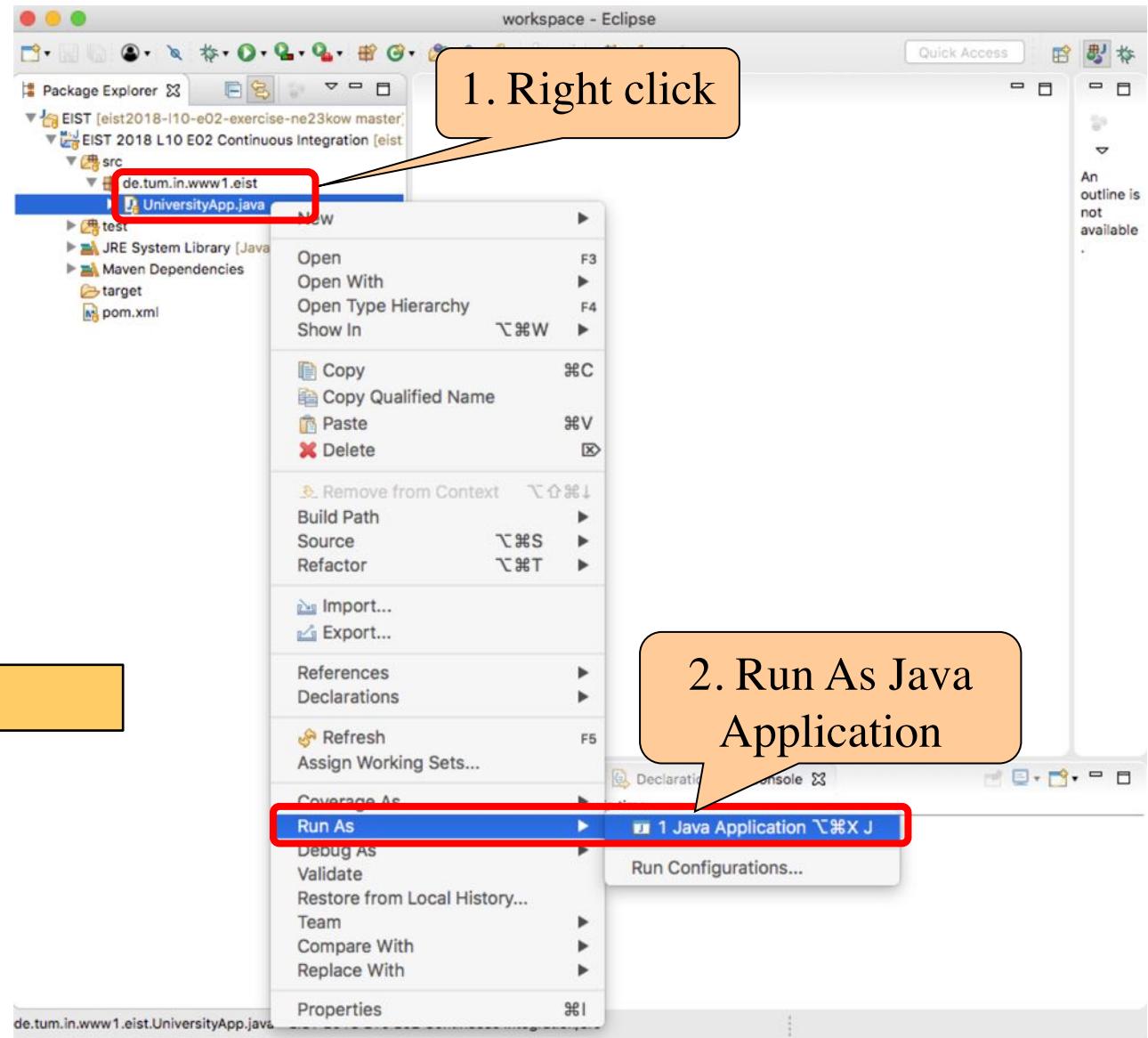
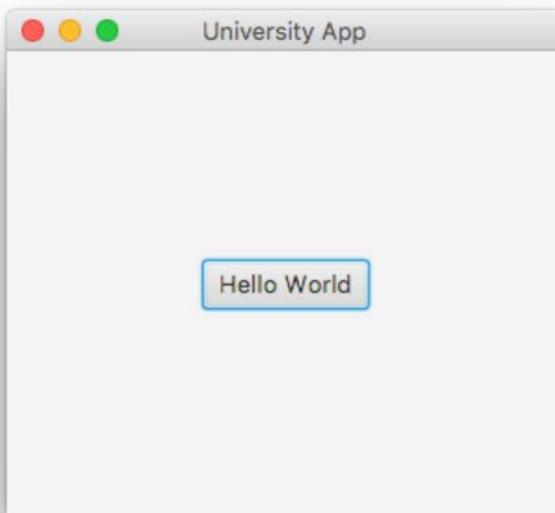
Other issues:

- Right click the project
→ Maven → Update project
- Clean the Maven Cache on your computer

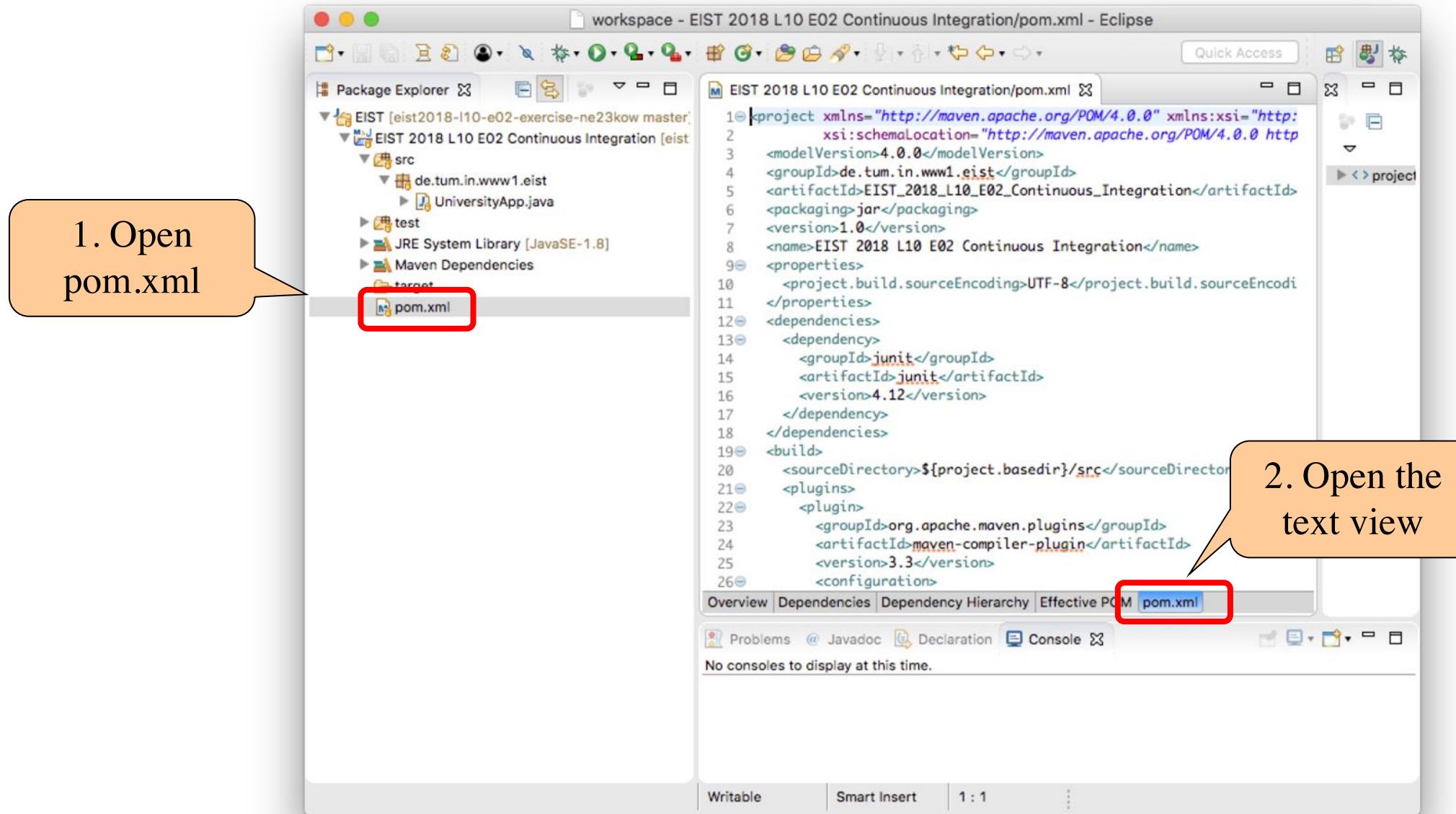
➤ Make sure to install the newest Java 8 JDK version (>= Java 8u172):
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Task 4: Run the Java Application in Eclipse

- Navigate into the project and select UniversityApp
- Right click → Run As → Java Application

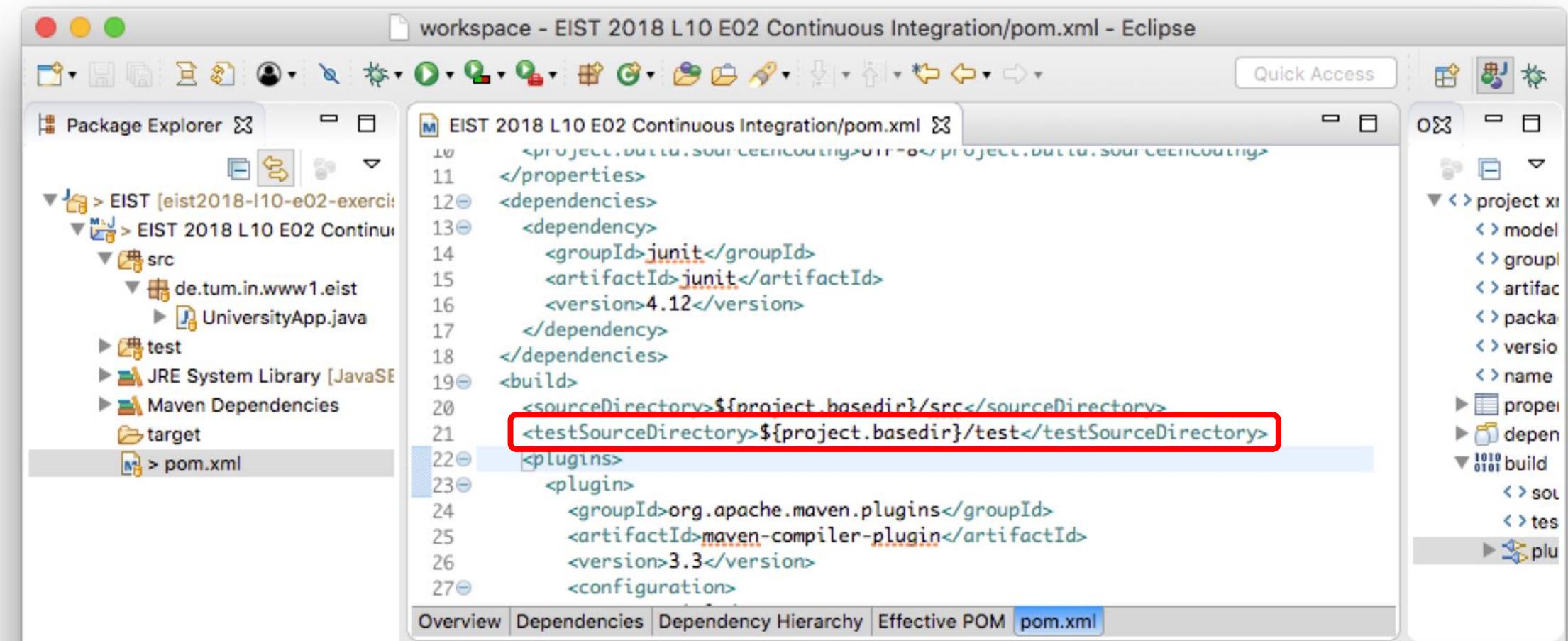


Task 5: Configure Maven to run tests



Task 5: Configure Maven to run tests

Add <testSourceDirectory>\${project.basedir}/test</testSourceDirectory>

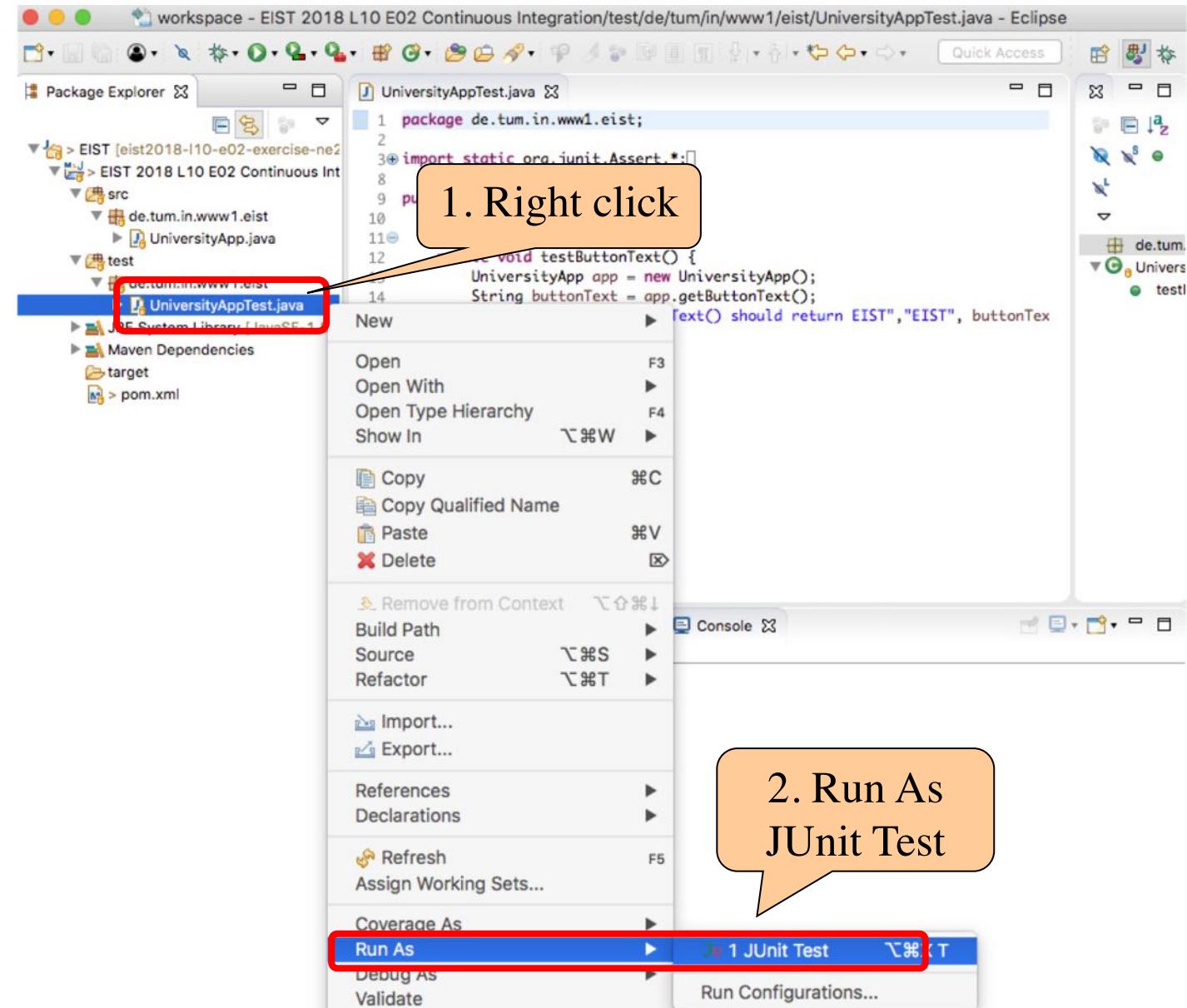


```
workspace - EIST 2018 L10 E02 Continuous Integration/pom.xml - Eclipse
Package Explorer EIST 2018 L10 E02 Continuous Integration/pom.xml
src
  de.tum.in.www1.eist
    UniversityApp.java
test
JRE System Library [JavaSE-1.8]
Maven Dependencies
target
pom.xml

<sourceDirectory>${project.basedir}/src</sourceDirectory>
<testSourceDirectory>${project.basedir}/test</testSourceDirectory>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.3</version>
    <configuration>
```

Task 6: Run the test

- Right click the test case **UniversityAppTest**
- Run it as JUnit Test (using Eclipse)



Test Result in Eclipse

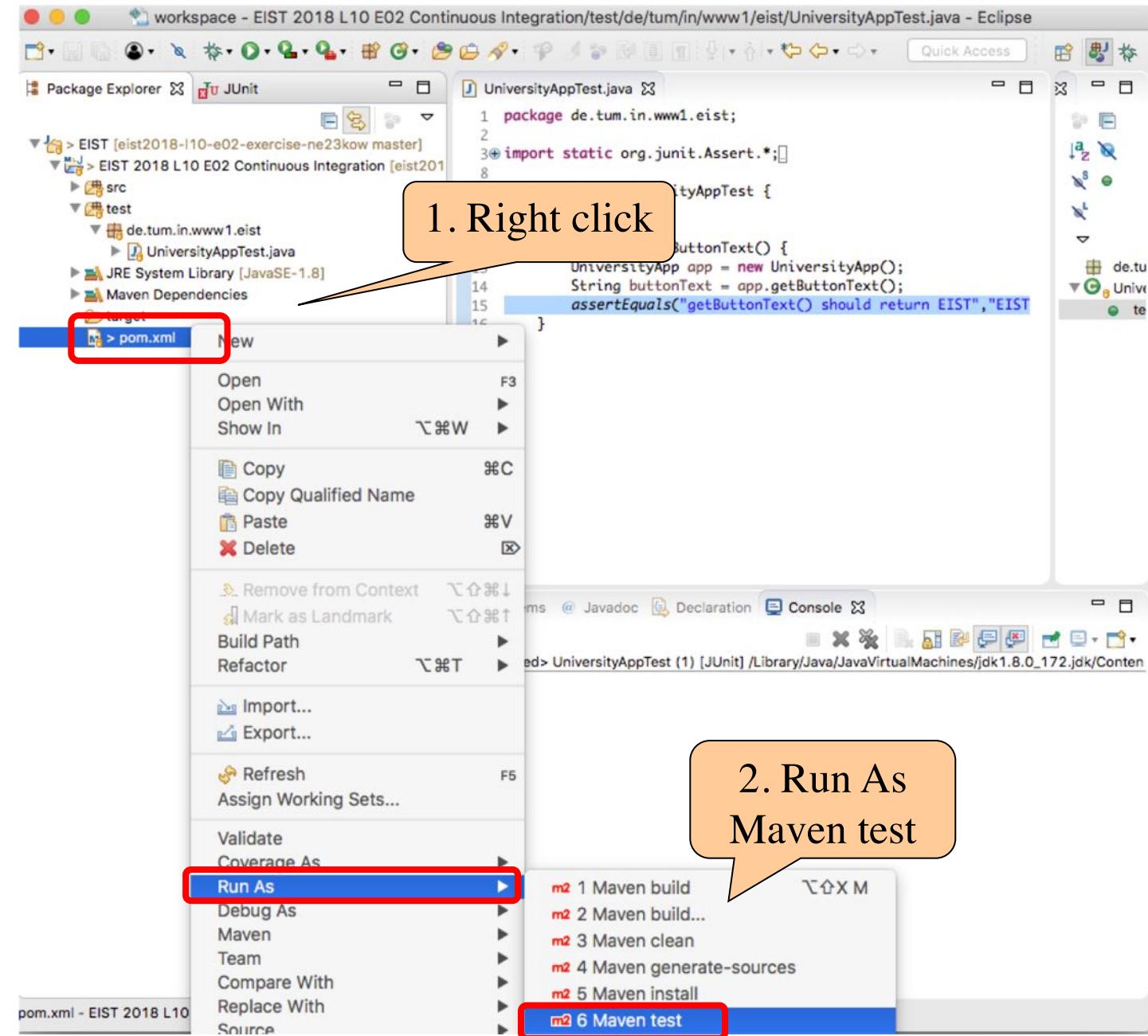
The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** workspace - EIST 2018 L10 E02 Continuous Integration/test/de/tum/in/www1/eist/UniversityAppTest.java - Eclipse
- Toolbar:** Standard Eclipse toolbar with various icons for file operations, search, and navigation.
- Package Explorer:** Shows the project structure with a tree view of packages and files.
- JUnit View:** Displays the test results:
 - Finished after 0,051 seconds
 - Runs: 1/1 Errors: 0 Failures: 1
 - Test: de.tum.in.www1.eist.UniversityAppTest [Runner: JUnit 4] - testButtonText (0,000 s)
- Code Editor:** Shows the Java code for `UniversityAppTest.java`. The failing test method is highlighted:

```
1 package de.tum.in.www1.eist;
2
3 import static org.junit.Assert.*;
4
5 public class UniversityAppTest {
6
7     @Test
8     public void testButtonText() {
9         UniversityApp app = new UniversityApp();
10        String buttonText = app.getButtonText();
11        assertEquals("getButtonText() should return EIST", "EIST");
12    }
13 }
```
- Failure Trace:** Shows the error message: `org.junit.ComparisonFailure: getButtonText() should return EIST` at `at de.tum.in.www1.eist.UniversityAppTest.testButtonText(UniversityAppTest.java:11)`.
- Console:** Shows the output: <terminated> UniversityAppTest (1) [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_172.jdk/Content
- Status Bar:** Shows Writable, Smart Insert, and line number 16 : 1.

Task 6: Run the test

- Right click pom.xml
- Run **UniversityAppTest** as JUnit Test (using Maven)



Test Result using Maven

```
-----  
T E S T S  
-----
```

```
Running de.tum.in.www1.eist.UniversityAppTest
```

```
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.15 sec <<< FAILURE!
```

```
testButtonText(de.tum.in.www1.eist.UniversityAppTest) Time elapsed: 0.03 sec <<< FAILURE!
```

```
org.junit.ComparisonFailure: getButtonText() should return EIST expected:<[EIST]> but was:<[Hello World]>
```

```
    at org.junit.Assert.assertEquals(Assert.java:115)
```

```
    at de.tum.in.www1.eist.UniversityAppTest.testButtonText(UniversityAppTest.java:15)
```

```
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

```
Results :
```

```
Failed tests: testButtonText(de.tum.in.www1.eist.UniversityAppTest): getButtonText() should return EIST expected:<[EIST]> but was:<[Hello World]>
```

```
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
```

```
[INFO] -----
```

```
[INFO] BUILD FAILURE
```

```
[INFO] -----
```

```
[INFO] Total time: 4.143 s
```

```
[INFO] Finished at: 2018-06-28T00:14:36+02:00
```

```
[INFO] Final Memory: 17M/207M
```

```
[INFO] -----
```

→ This is a typical test result on a Continuous Integration Server

Task 6: Fix the failing test

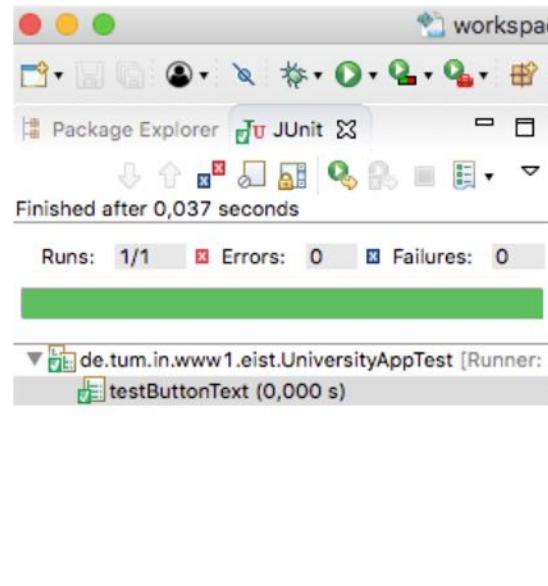
1. Open the UniversityApp class

2. Change the returned String in Line 44

```
35     stackPane.getChildren().add(button);
36
37     primaryStage.setTitle(title);
38     primaryStage.setScene(scene);
39     primaryStage.show();
40 }
41
42
43 public String getButtonText() {
44     return "EIST";
45 }
46
47 public static void main(String[] args) {
48     launch(args);
49 }
```

Task 6: Fix the failing test

→ Run the test again using Eclipse and Maven



```
workspace - EIST 2018 L10 E02 Continuous Integration
Package Explorer JUnit
Finished after 0,037 seconds
Runs: 1/1 Errors: 0 Failures: 0
de.tum.in.www1.eist.UniversityAppTest [Runner: ]
  testButtonText (0,000 s)

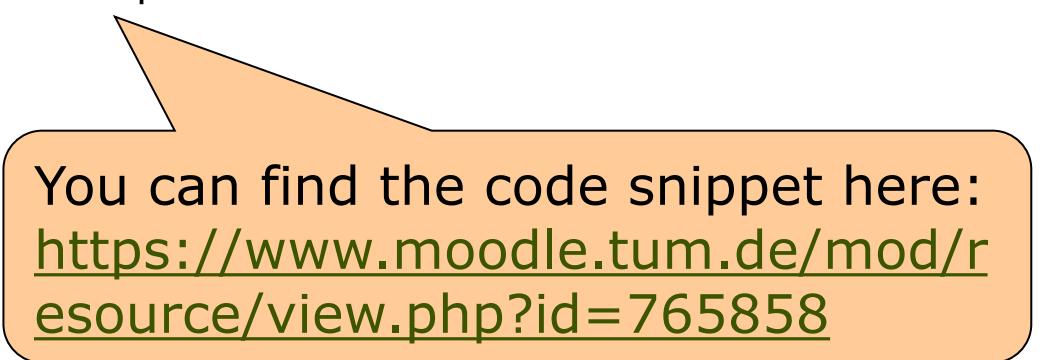
UniversityAppTest.java
1 package de.tum.in.www1.eist;
2
3 import static org.junit.Assert.*;
4
5 public class UniversityAppTest {
6
7     @Test
8     public void testButtonText() {
9         UniversityApp app = new Uni
10        String buttonText = app.getI
11        assertEquals("getButtonText"
12    }
13}
```

```
-----  
TESTS  
-----  
Running de.tum.in.www1.eist.UniversityAppTest  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.149 sec  
  
Results :  
  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0  
  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 2.564 s  
[INFO] Finished at: 2018-06-28T00:20:42+02:00  
[INFO] Final Memory: 11M/220M  
[INFO] -----
```

Task 8: Configure Maven to create an executable

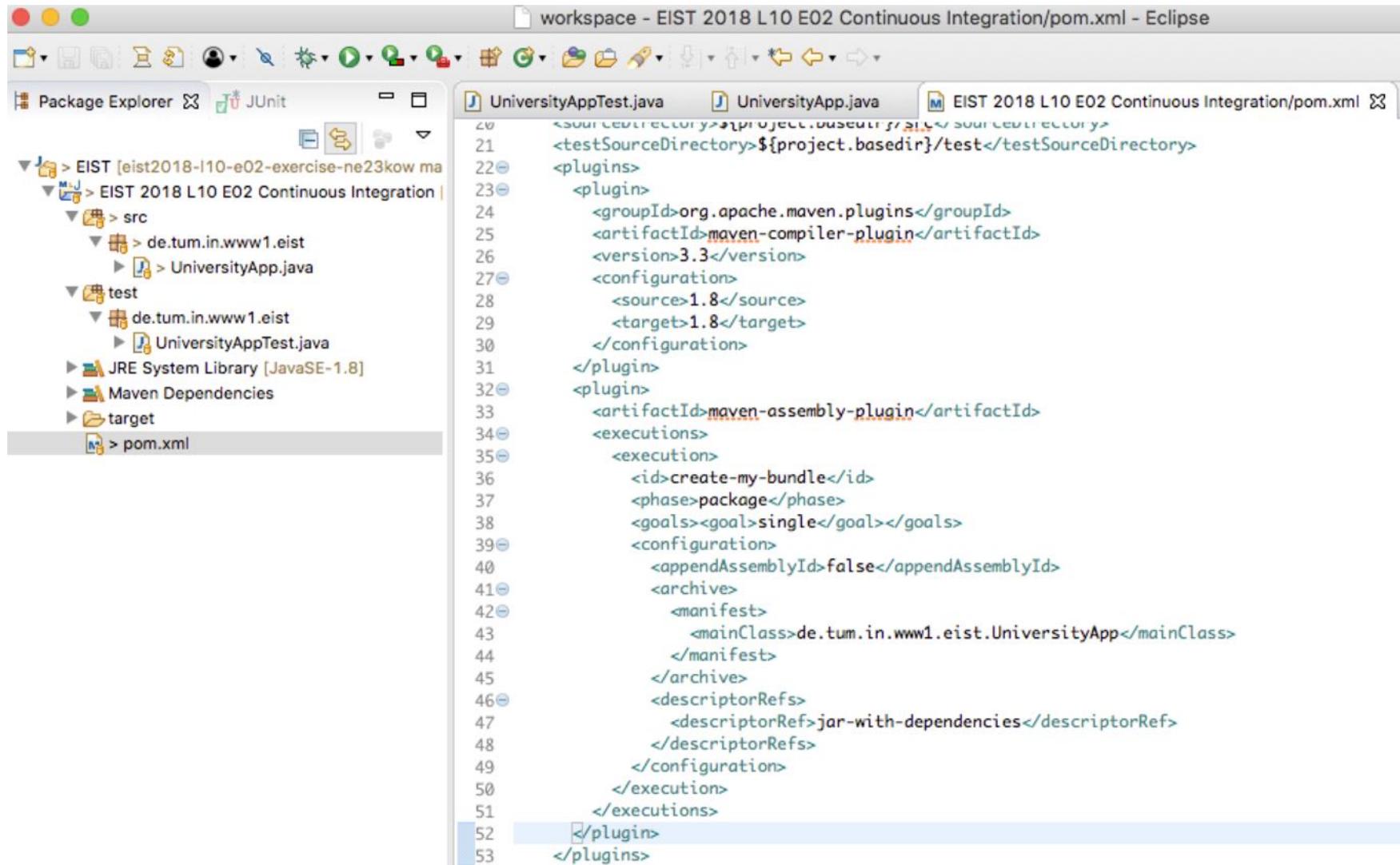
Edit the pom.xml again and add the following into the <plugins> section (within the <build> section):

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <id>create-my-bundle</id>
      <phase>package</phase>
      <goals><goal>single</goal></goals>
      <configuration>
        <appendAssemblyId>false</appendAssemblyId>
        <archive>
          <manifest>
            <mainClass>de.tum.in.www1.eist.UniversityApp</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
</plugin>
```



You can find the code snippet here:
<https://www.moodle.tum.de/mod/resource/view.php?id=765858>

Resulting pom.xml



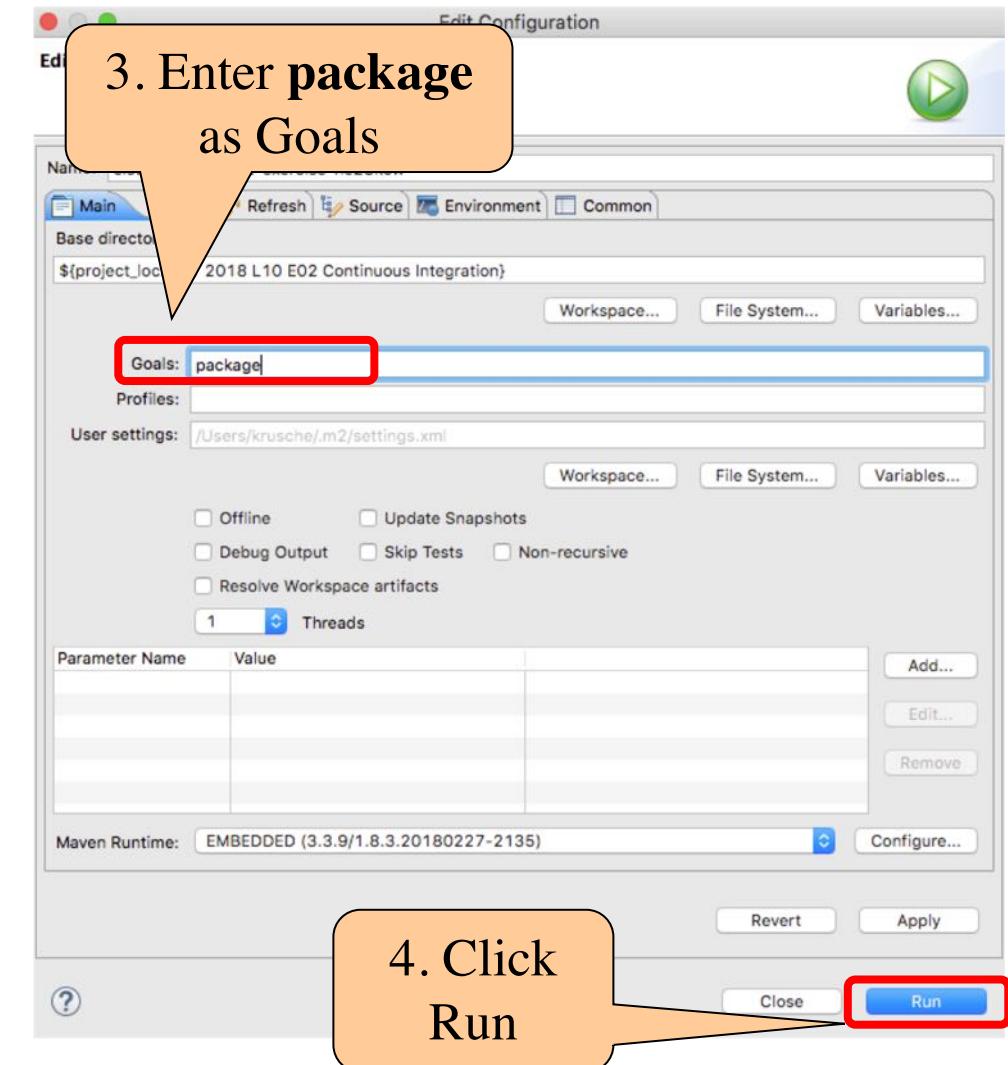
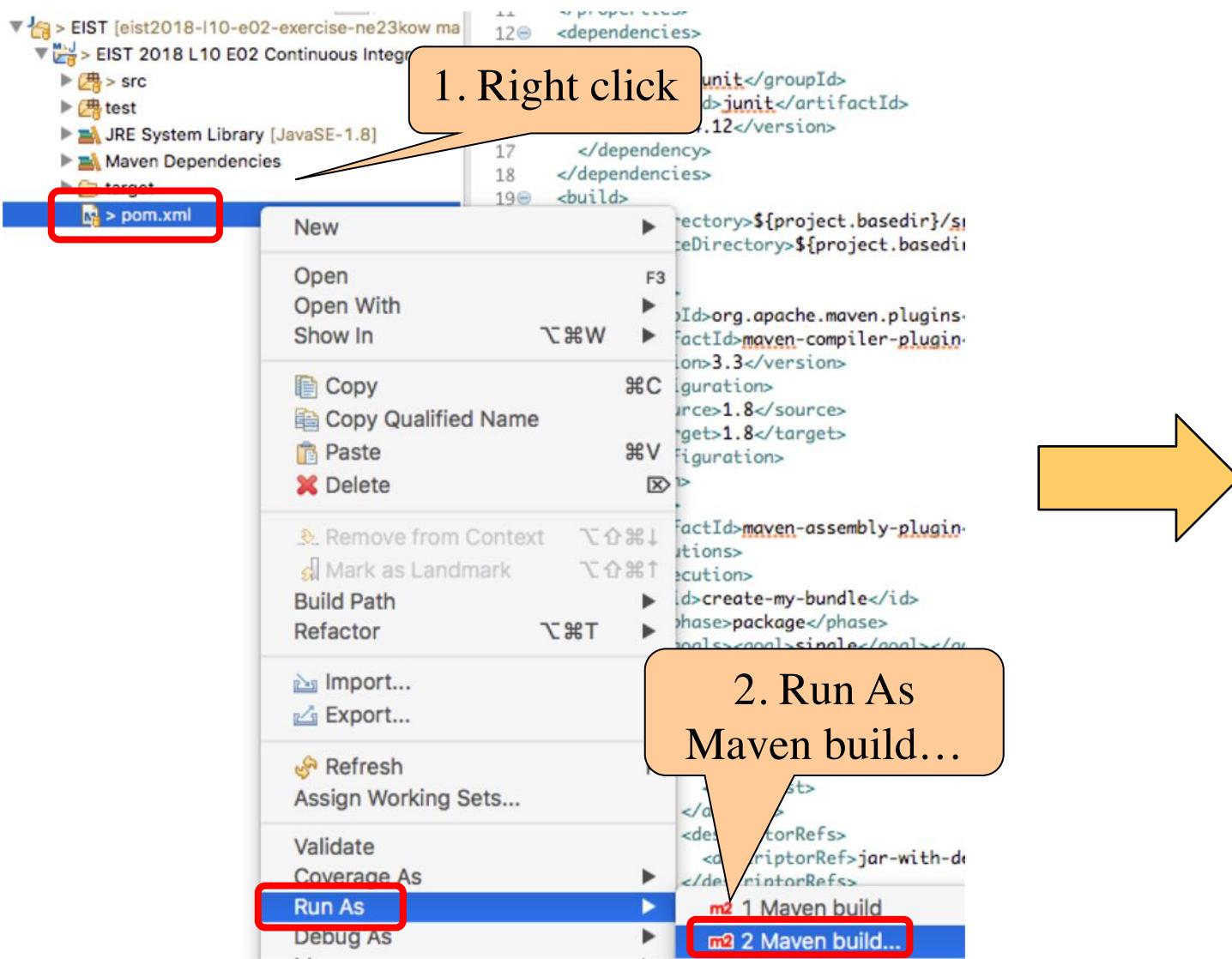
The screenshot shows the Eclipse IDE interface with the title bar "workspace - EIST 2018 L10 E02 Continuous Integration/pom.xml - Eclipse". The left side features the "Package Explorer" view, which displays the project structure:

- Project: EIST [eist2018-l10-e02-exercise-ne23kow ma]
- Source Folder: src
 - de.tum.in.www1.eist
 - UniversityApp.java
- Test Folder: test
 - de.tum.in.www1.eist
 - UniversityAppTest.java
- JRE System Library [JavaSE-1.8]
- Maven Dependencies
- target
- pom.xml (selected)

The right side shows the content of the "pom.xml" file:

```
<sOURCEDirectory>${project.basedir}/src</sOURCEDirectory>
<tESTSourceDirectory>${project.basedir}/test</tESTSourceDirectory>
<pLUGINS>
    <pLUGIN>
        <gROUPId>org.apache.maven.plugins</gROUPId>
        <aRTIFID>maven-compiler-plugin</aRTIFID>
        <vERSION>3.3</vERSION>
        <cONFIGURATION>
            <sOURCE>1.8</sOURCE>
            <tARGET>1.8</tARGET>
        </cONFIGURATION>
    </pLUGIN>
    <pLUGIN>
        <aRTIFID>maven-assembly-plugin</aRTIFID>
        <eXECUTIONS>
            <eXECUTION>
                <iD>create-my-bundle</iD>
                <pHASE>package</pHASE>
                <gOALS><gOAL>single</gOAL></gOALS>
                <cONFIGURATION>
                    <aPPENDASSEMBLYID>false</aPPENDASSEMBLYID>
                    <aRCHIVE>
                        <mANIFEST>
                            <mAINCLASS>de.tum.in.www1.eist.UniversityApp</mAINCLASS>
                        </mANIFEST>
                    </aRCHIVE>
                    <dESCRIPTORREFS>
                        <dESCRIPTORREF>jar-with-dependencies</dESCRIPTORREF>
                    </dESCRIPTORREFS>
                </cONFIGURATION>
            </eXECUTION>
        </eXECUTIONS>
    </pLUGIN>
</pLUGINS>
```

Task 8: Configure Maven to create an executable



Result of Maven Package

```
Running de.tum.in.www1.eist.UniversityAppTest  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.181 sec
```

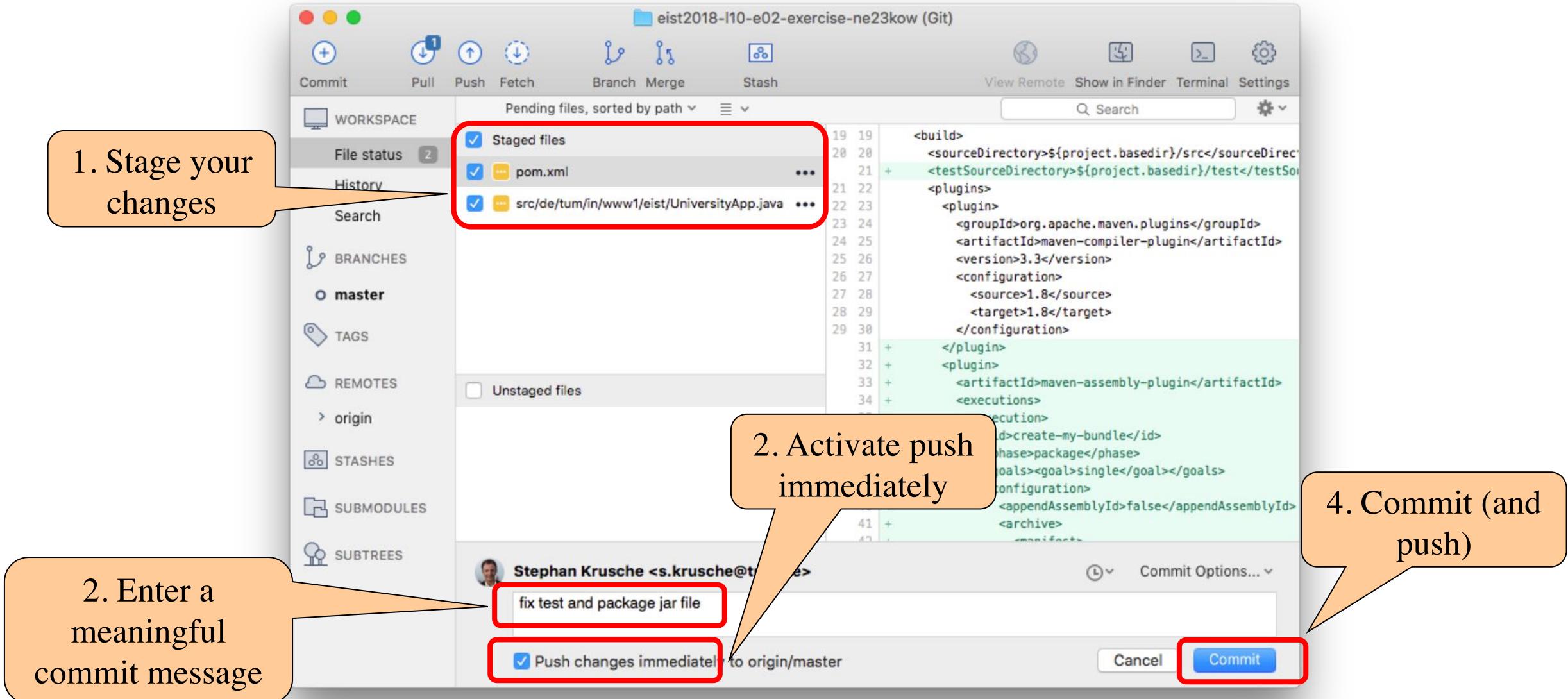
Results :

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

```
[INFO]  
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ EIST_2018_L10_E02_Continuous_Integration -  
[INFO] Building jar: /Users/krusche/Projects/eist2018-l10-e02-exercise-ne23kow/target/EIST_201  
[INFO]  
[INFO] --- maven-assembly-plugin:2.2-beta-5:single (create-my-bundle) @ EIST_2018_L10_E02_Conti  
[INFO] META-INF/ already added, skipping  
[INFO] META-INF/MANIFEST.MF already added, skipping  
[INFO] org/ already added, skipping  
[INFO] Building jar: /Users/krusche/Projects/eist2018-l10-e02-exercise-ne23kow/target/EIST_2018_L10_E02_Continuous_Integration-1.0.jar  
[INFO] META-INF/ already added, skipping  
[INFO] META-INF/MANIFEST.MF already added, skipping  
[INFO] org/ already added, skipping  
[WARNING] Configuration options: 'appendAssemblyId' is set to false, and 'classifier' is missing.  
Instead of attaching the assembly file: /Users/krusche/Projects/eist2018-l10-e02-exercise-ne23kow/target/EIST_2018_L10_E02_Continuous_Int  
NOTE: If multiple descriptors or descriptor-formats are provided for this project, the value of this file will be non-deterministic!  
[WARNING] Replacing pre-existing project main-artifact file: /Users/krusche/Projects/eist2018-l10-e02-exercise-ne23kow/target/EIST_2018_L  
with assembly file: /Users/krusche/Projects/eist2018-l10-e02-exercise-ne23kow/target/EIST_2018_L10_E02_Continuous_Integration-1.0.jar  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 4.561 s  
[INFO] Finished at: 2018-06-28T00:27:03+02:00  
[INFO] Final Memory: 14M/202M  
[INFO] -----
```

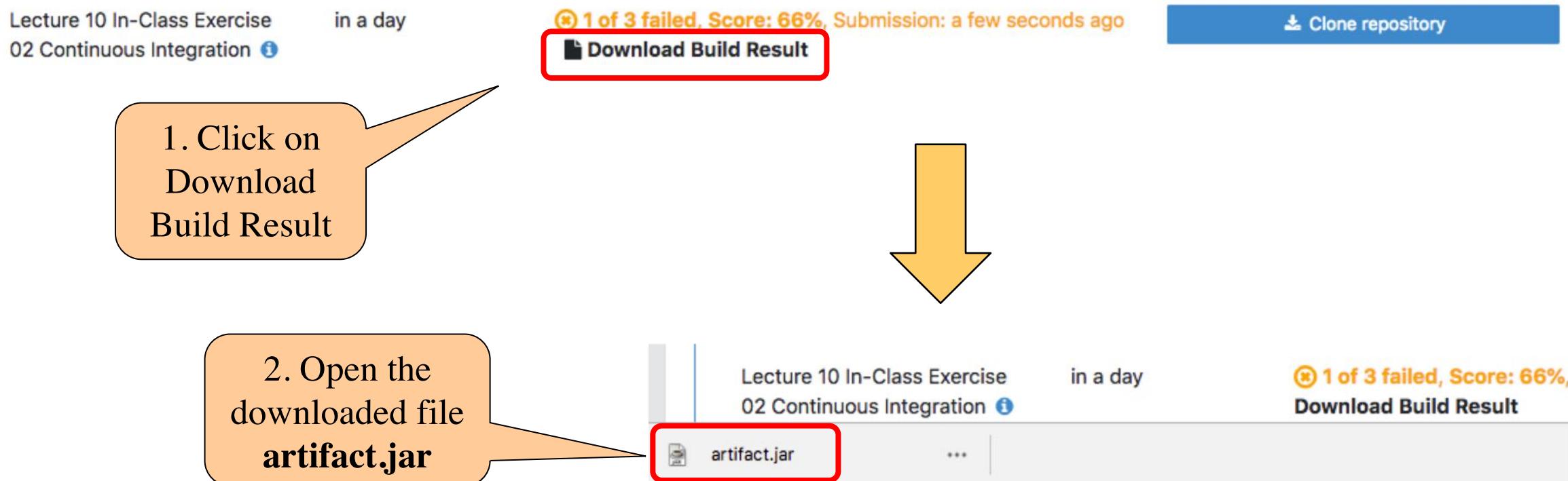
Find the jar file in the target folder: Open it to start the application

Task 9: Commit and push your changes



Task 10: Download the Java executable

- Open ArTEMiS and download the jar file from the Continuous Integration Server



Task 11: Add a new dependency in Maven

- There is one more test case failing on ArTEMiS

Feedback

X

Error in method `testIfLog4jIsConfiguredCorrectly`:

`java.lang.AssertionError: Log4j dependency not found in pom.xml. Make sure to add the correct dependency in pom.xml`

Close

→ This is our last task!

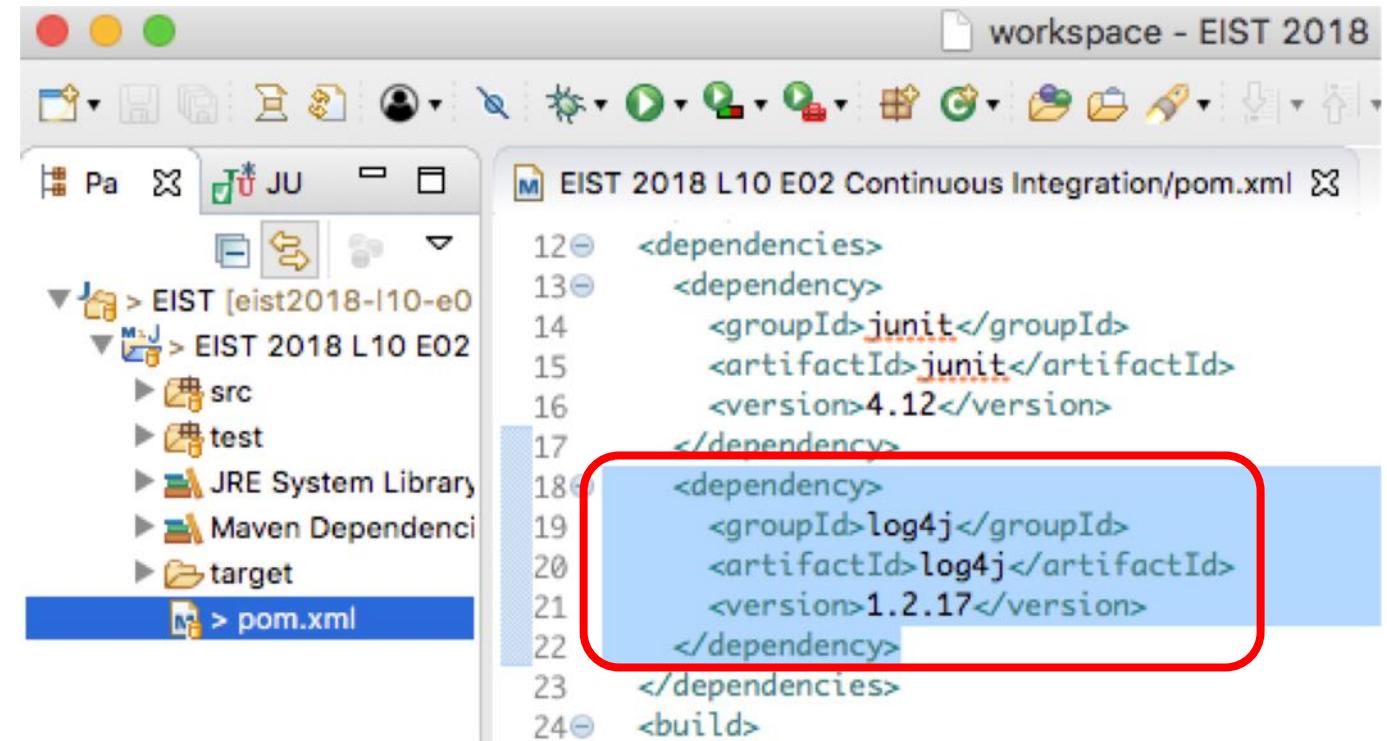
Task 11: Add a new dependency in Maven

- Maven also manages dependencies and lets you easily add new libraries and frameworks to your Java project
- We want to add Logging to the UniversityApp.
- To save time, we want to reuse an external component, the logging framework **Log4j**.
- Follow [these instructions](#) to add Log4j as external Maven dependency in the pom.xml file (starting at step 3 in the tutorial).
- Do some reasonable logging within the UniversityApp class, e.g. give out a log message using Log4j when the app starts.

Task 11: Add a new dependency in Maven

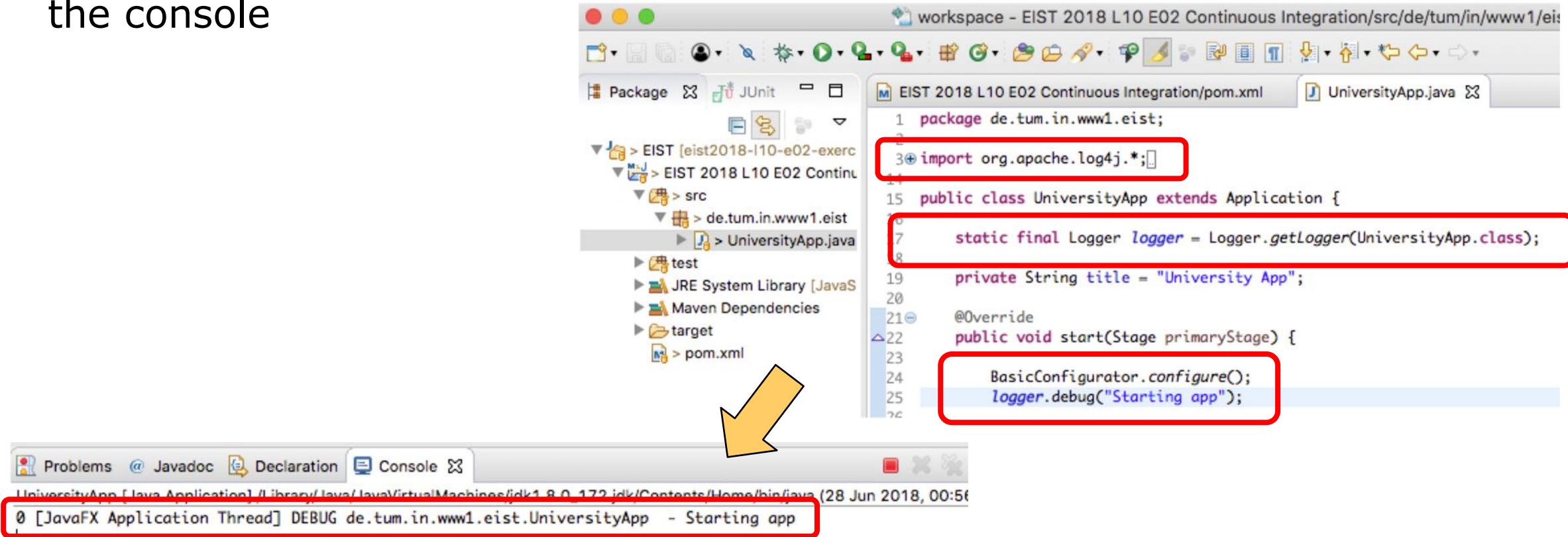
- Open the pom.xml and add the new dependency

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```



Task 11: Add a new dependency in Maven

- Open **UniversityApp.java**
- **import org.apache.log4j.***
- Initialize a logger with a basic configuration and log the start of the app
- Start the application again using Eclipse and review if the log appears in the console



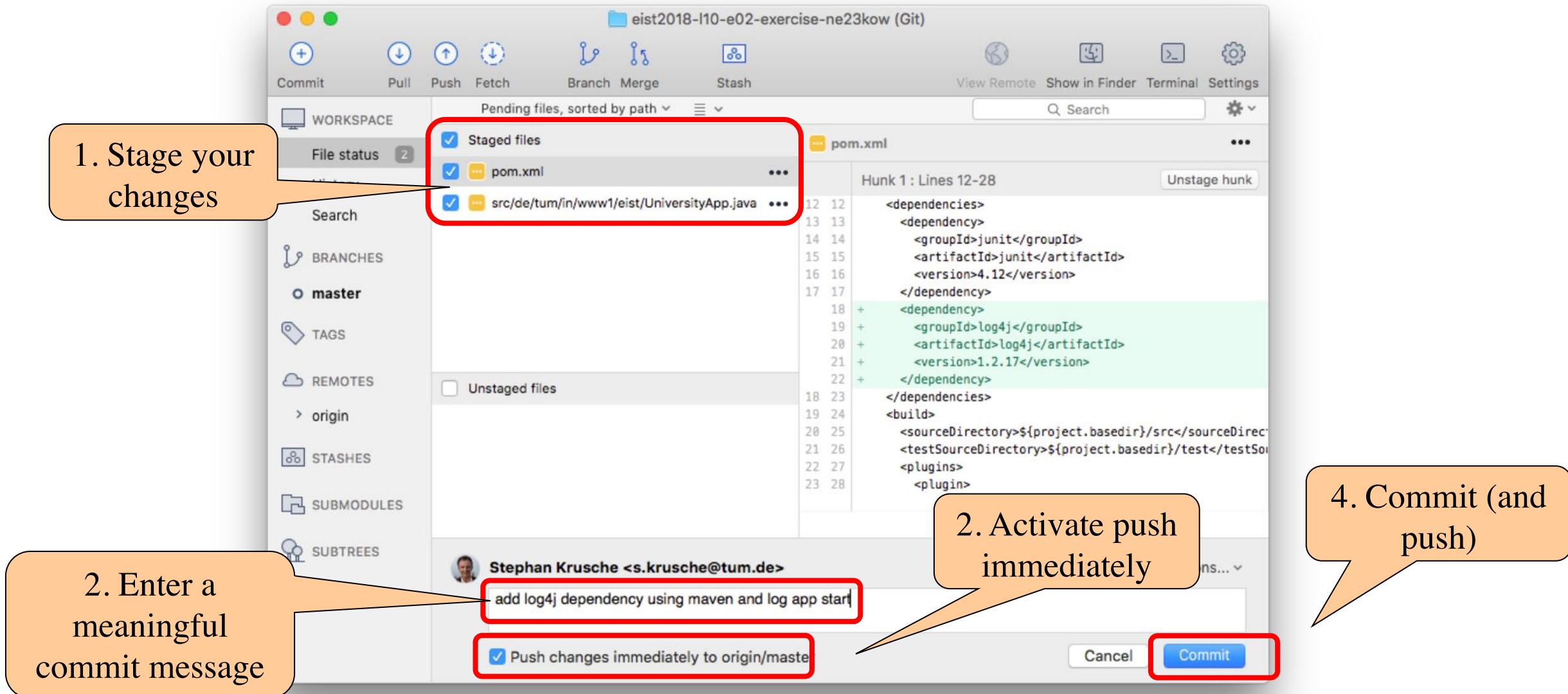
The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer shows a project structure with a file named `UniversityApp.java` selected. In the center, the code editor displays the following Java code:

```
1 package de.tum.in.www1.eist;
2
3+import org.apache.log4j.*;
4
5 public class UniversityApp extends Application {
6
7     static final Logger logger = Logger.getLogger(UniversityApp.class);
8
9     private String title = "University App";
10
11    @Override
12    public void start(Stage primaryStage) {
13
14        BasicConfigurator.configure();
15        logger.debug("Starting app");
16    }
17}
```

Three specific sections of the code are highlighted with red boxes: the `import` statement at line 3, the logger declaration at line 7, and the logger call at line 15. A large yellow arrow points from the bottom of the code editor towards the bottom of the screen, where the Console tab is located. The Console tab shows the output of the application's debug logs:

```
0 [JavaFX Application Thread] DEBUG de.tum.in.www1.eist.UniversityApp - Starting app
```

Task 12: Commit and push your changes



Final Result on ArTEMiS

Lecture 10 In-Class
Exercise 02
Continuous
Integration

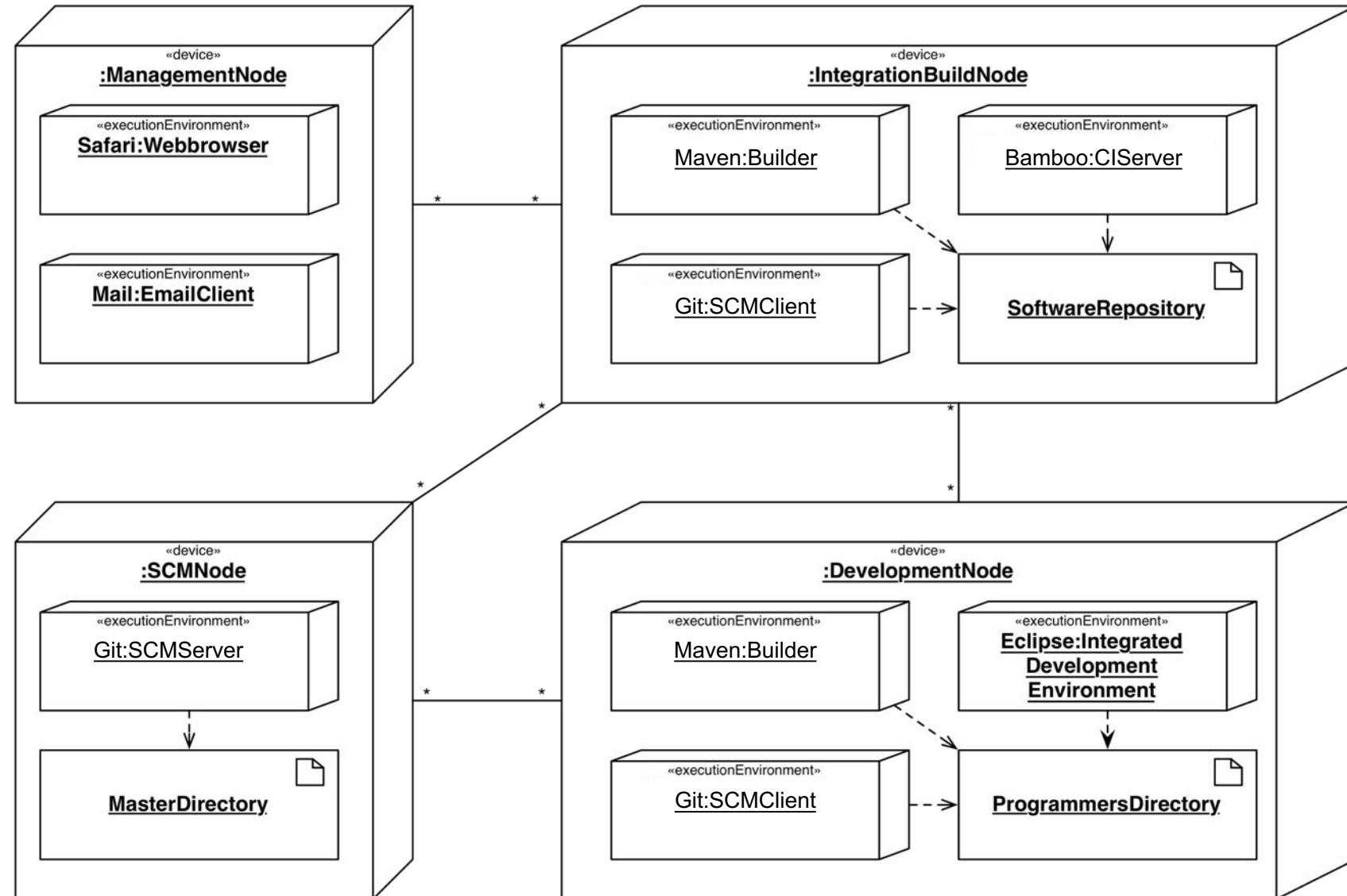
in a day

 3 passed, Score: 100%, Submission: a few seconds ago  [Download Build Result](#)



[Clone repository](#)

Deployment Diagram of a Continuous Integration System



Examples of Continuous Integration Systems

- Atlassian Bamboo
- Jenkins
- Cockpit
- Anthill
- Continuum
- Hudson
- CruiseControl and CruiseControl.NET

Outline of the Lecture

- Software Configuration Management
 - Change Management
 - Version Control Systems
 - In-Class Exercise 01: Solve a Merge Conflict
 - Branch Management
 - Continuous Integration
 - In-Class Exercise 02: Continuous Integration
- Release Management

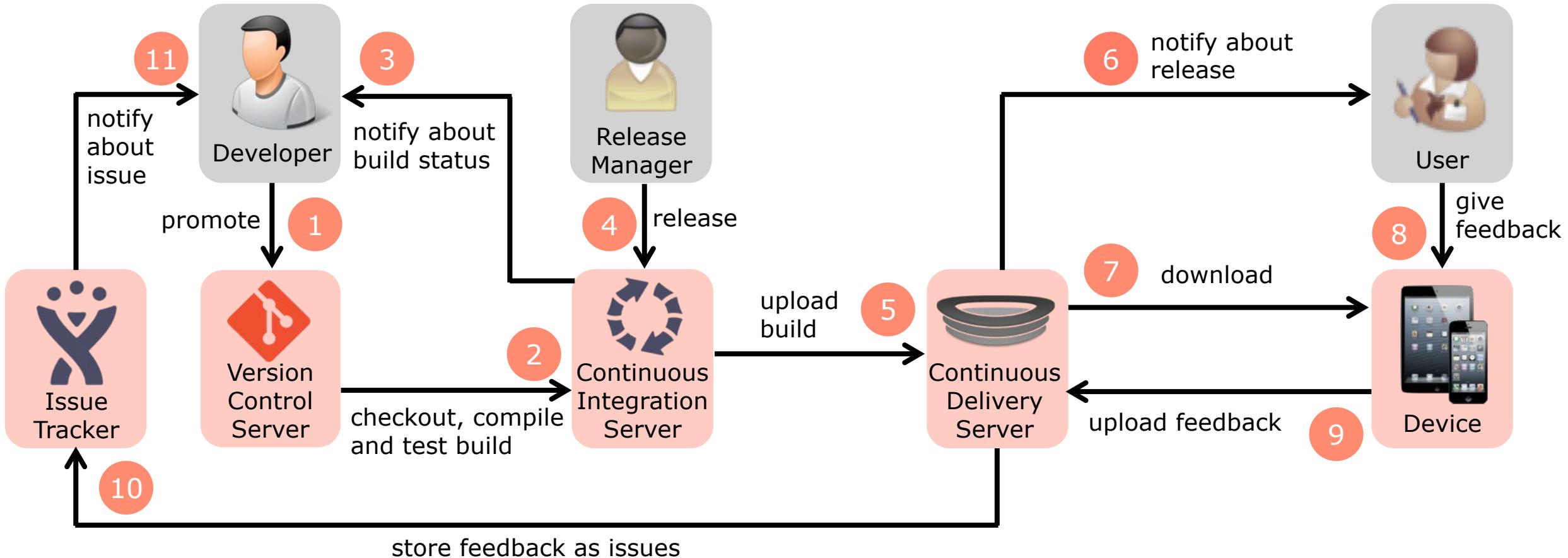
Requirements for Release Management

- Large and distributed software projects need to provide a development infrastructure with an integrated build management that supports:
 - Regular builds from the master directory
 - Automated execution of tests
 - E-mail notification
 - Determination of code metrics
 - Automated publishing of the applications and test results (e.g. to a website)

Requirements for Release Management (2)

- The transition from source code to the executable application consists of these activities:
 1. Setting required paths and libraries
 2. Compiling source code
 3. Copying source files (e.g. images, sounds, start scripts)
 4. Setting file permissions (e.g. to executable)
 5. Packaging the application (e.g. zip, tar, dmg)
- Performing these activities manually: Chance of introducing failures is high
- Tools to automate these activities with a build plan: Unix's Make, Ant, Maven.

Release Management Model



This is an informal model. How would you model this in UML?

Configuration Management Activities

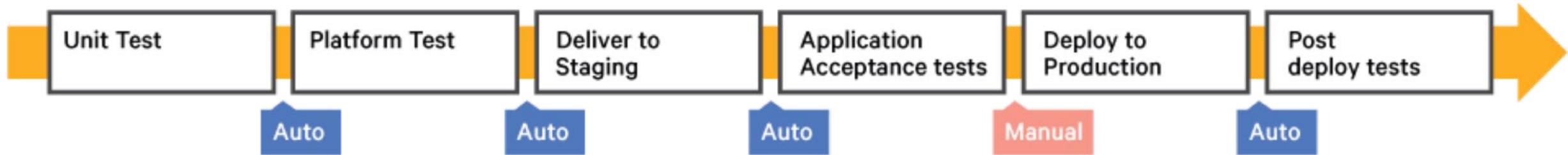
- ✓ Configuration item identification
 - Modeling the system as a *set of evolving components*
- ✓ Promotion management
 - Creation of *versions for other developers*
- ✓ Change management
 - Management of *change requests*
- ✓ Branch management
 - Management of *concurrent development*
- ✓ Release management
 - Creation of *versions for clients and end users*
- Variant management
 - Management of *coexisting versions*

Terminology

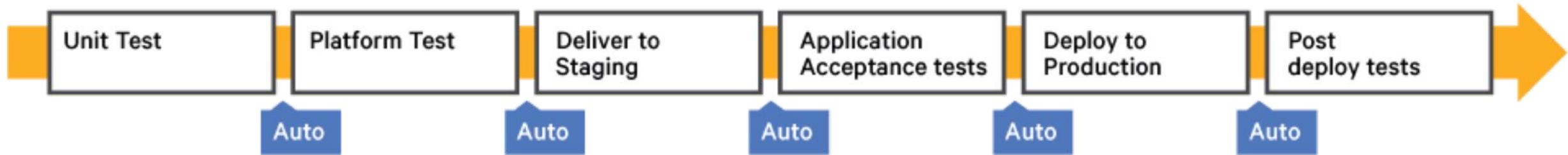
- **Continuous integration:** technique where members of a team integrate their work frequently. Usually each person integrates at least daily, leading to multiple integrations per day.
- **Continuous delivery:** approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time.
- **Continuous deployment:** every change that passes automated tests is deployed **automatically**.
- **Continuous software engineering:** organizational capability to develop, release and learn from software in short cycles.

Continuous Delivery vs. Continuous Deployment

Continuous Delivery



Continuous Deployment



Source: <https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff>

Summary

- Software Configuration Management:
 - A set of management disciplines within a software engineering process to develop a *baseline*
- Promotions are internal versions and Releases are external version
- Change Management allows control of change requests
- Branch Management allows exploration of multiple feature requests
- Git allows lightweight branching and merging
- Continuous Integration has emerged as a central activity in modern software projects.

References

- IEEE Standard 828 for Configuration Management in Systems and Software Engineering
- IEEE Standard 1042 Guide to Software Configuration Management
- Martin Fowler: Continuous Integration
 - <http://martinfowler.com/articles/continuousIntegration.html>

Morning Quiz 11

- Start Time: **8:05**
- End Time: **8:15**
- To participate in the quiz, use ArTEMiS
- Click on Open Quiz or Start Quiz
- The Lecture starts at 8:15

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	Start exercise
Good Morning Quiz 11		Open Quiz

Remaining Time: **46 s**
Saved: never
● Connected

Submit

Only click on Submit when you have entered all answers!

A photograph of a person climbing a steep, rocky mountain slope. The climber is wearing a red jacket and blue pants, and is using ice axes and crampons. The background shows a vast, snow-covered mountain range under a clear sky.

Testing

Bernd Bruegge

Chair for Applied Software Engineering
Technische Universität München

5 July 2018

Roadmap for today

- **Context and Assumptions**

We have completed Chapter 1-10, 13 and 15

- **Content of this lecture**

- Chapter 11 Testing

- **Objective:** At the end of the lecture you understand

- The difference between fault, failure and error
- The difference between model-based and object-oriented testing
- The difference between white-box and black box testing
- The testing activities unit testing, integration testing, system testing
- The difference between horizontal and vertical integration
- The JUnit testing framework

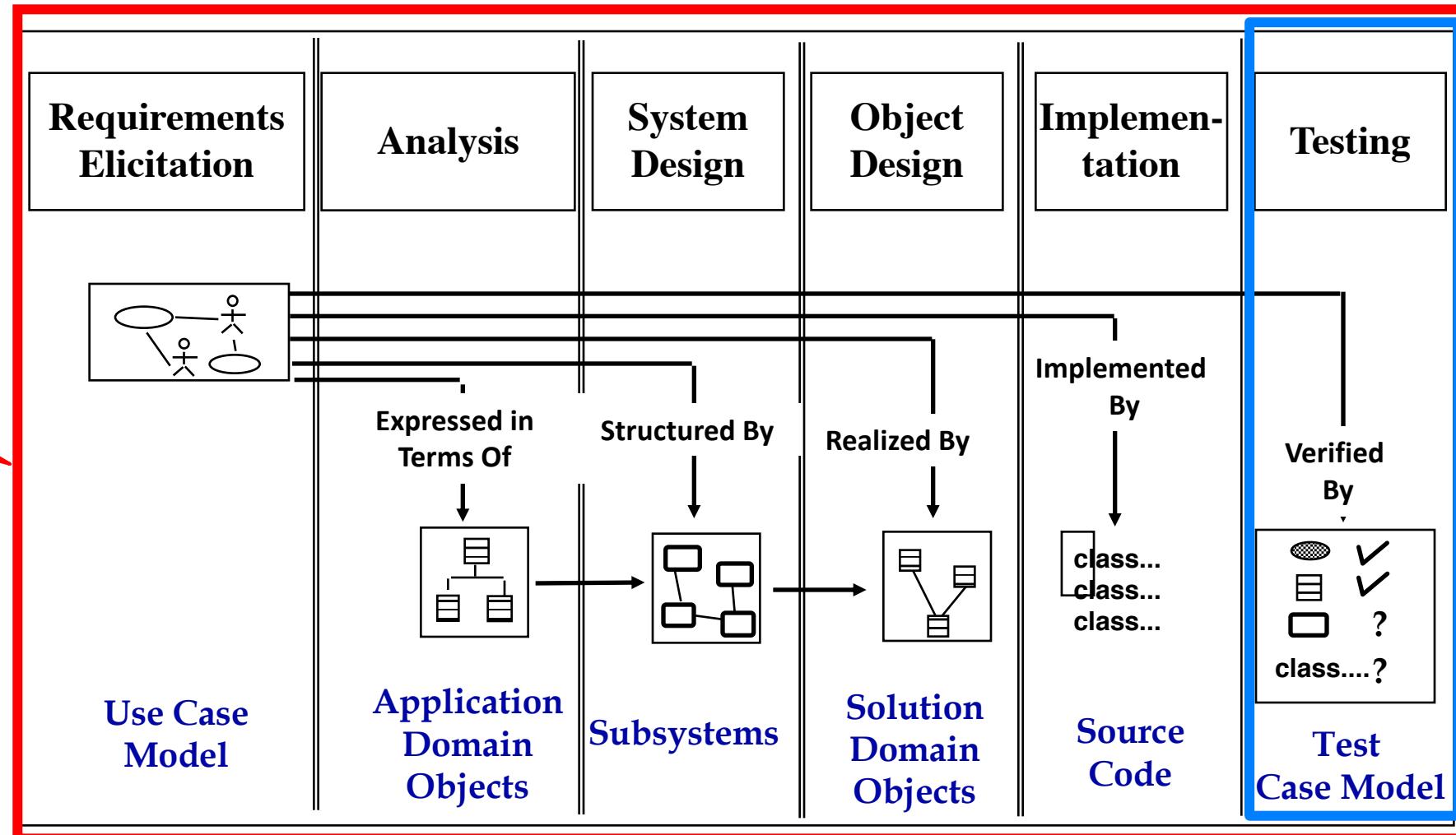
Software Lifecycle Activities and their Models in EIST

✓ Chapter 4

✓ Chapter 5

✓ Chapter 6 to 9

✓ Chapter 10 ➤ Chapter 11



Outline of the Lectures on Testing

→ Terminology

- Failure, Error, Fault
- Test Model
- Model-based testing
- Object-Oriented testing
- Testing activities
 - Static Analysis
 - Black box and White Box testing
 - Unit testing
 - Integration testing
 - System Testing
 - Function testing
 - Performance testing
 - Acceptance Testing.

Famous Problems

- F-16 : crossing equator using autopilot
 - Result: plane flipped over
 - Reason?
 - Reuse of autopilot software from a rocket



- Urban legend?
- Some people say, the fault was actually found during simulation
 - Source: <http://catless.ncl.ac.uk/Risks/3.44.html#subj1>

Many Other Problems

- NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
 - Reason: Unit conversion problem
- The Therac-25 accidents (1985-1987)
 - A medical linear accelerator that creates energy beams to destroy tumors
 - At least five patients died
 - Reason: Cryptic error messages caused confusion between high energy x-ray beams (25000 rads, 25million eV) and low energy beams
- Risk Forum by Peter G. Neumann
 - Risks in computers and related systems
 - <http://catless.ncl.ac.uk/Risks/>
 - Today's bug list:
<http://catless.ncl.ac.uk/Risks/30/38>

THE RISKS DIGEST

Forum on Risks to the Public in Computers and Related Systems

ACM Committee on Computers and Public Policy, Peter G. Neumann, moderator

Volume 30 Issue 38

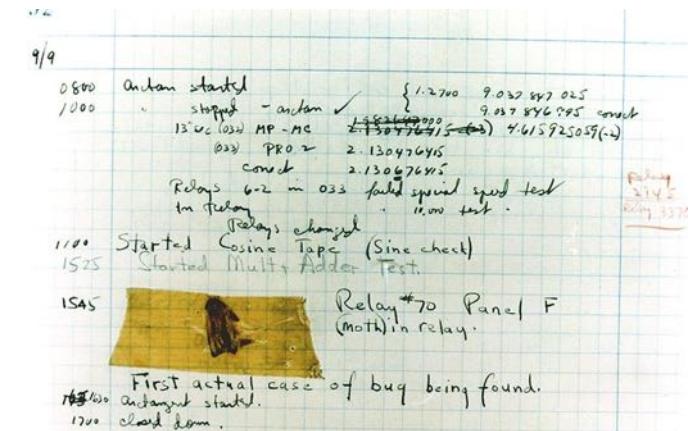
Monday 17 July 2017

Contents

- [A Solar Eclipse Could Wipe Out 9,000 Megawatts of Power Supplies](#)
[Bloomberg](#)
- [Massachusetts tax system blocks payments, sends refunds in error](#)
[MassLive](#)
- [The AlphaBay Takedown Sends Dark Web Markets Reeling](#)
[WiReD](#)
- [Cloud Leak: How A Verizon Partner Exposed Millions of Customer Accounts](#)
[UpGuard](#)
- [How Fake News Goes Viral—Here's the Math](#)
[Scientific American](#)
- [While Some Cry 'Fake,' Spotify Sees No Need to Apologize](#)
[The New York Times](#)
- [Nearly 90,000 Sex Bots Invaded Twitter in 'One of the Largest Malicious Campaigns Ever Recorded on a Social Network'](#)
[Gizmodo](#)
- [Elon Musk says preventing a 'fleet-wide hack' is Tesla's top security priority](#)
[Electrek](#)
- [Weekend Video Extra: A Prescient Warning re: AI and Robotics, from 1956!](#)
[Lauren Weinstein](#)
- [Your pacemaker is spying on you](#)
[Mark Thorson](#)
- [Leaping Kangaroos](#)
[Anthony Thorn](#)

Terminology

- **Failure:** Any deviation of the observed behavior from the specified behavior
- **Erroneous state (error):** The system is in a state such that further processing by the system can lead to a failure
- **Fault:** The mechanical or algorithmic cause of an error (informally often called "bug"¹)
- **Validation:** Activity of checking for deviations between the **observed behavior** of a system and its **specification**.



¹The first bug was found by Grace Hopper

- It was actually a moth. The remains of the moth can be found at the Smithsonian Institution's National Museum of American History

https://en.wikipedia.org/wiki/Grace_Hopper

What is this?

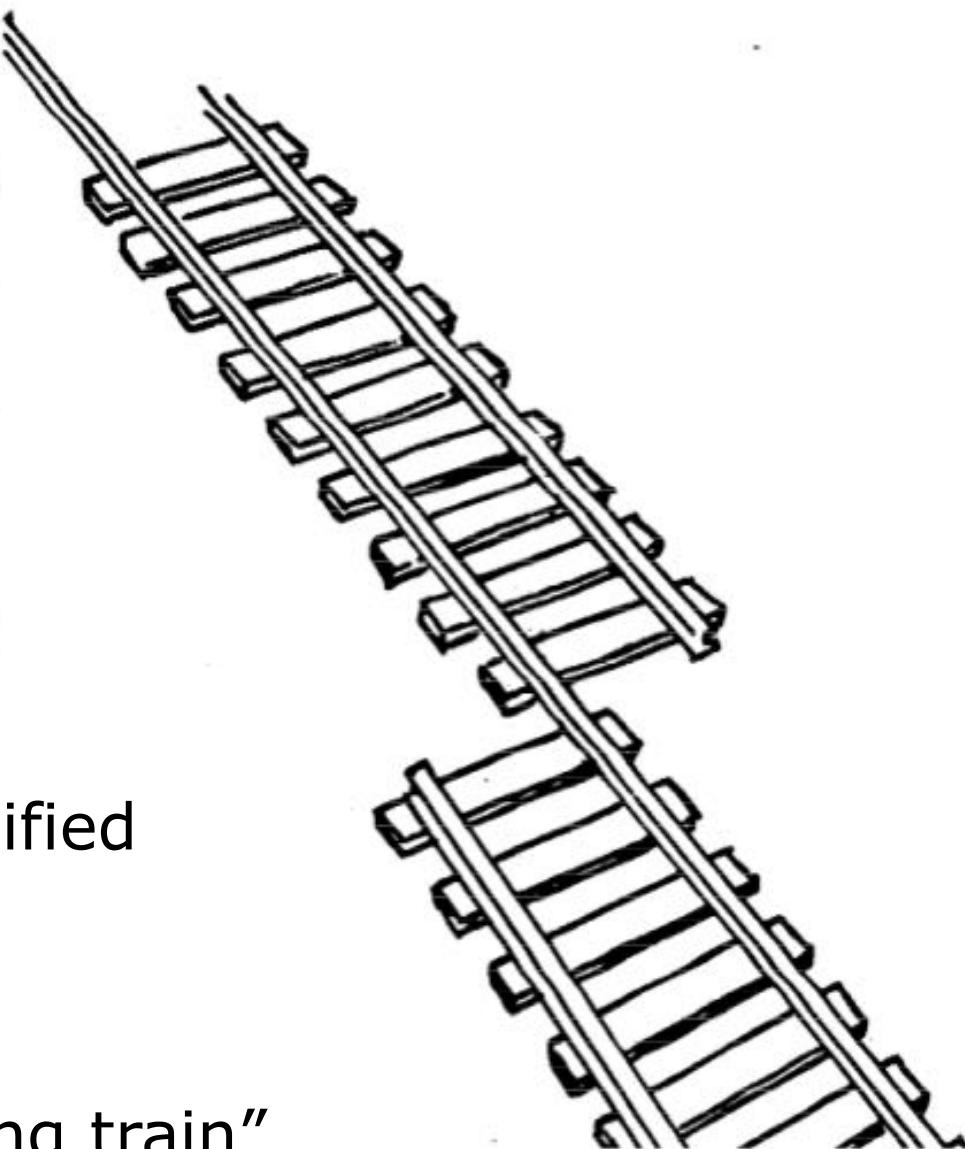
A failure?

An error?

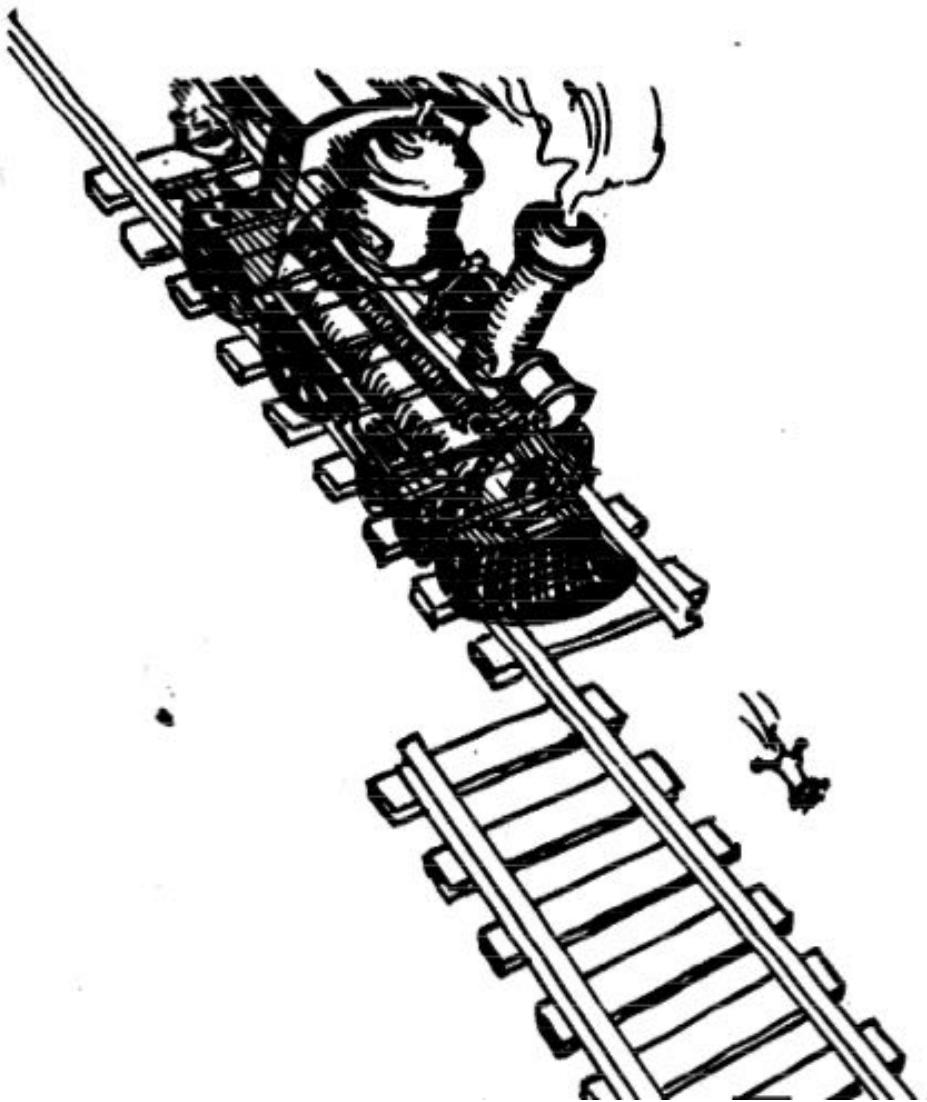
A fault?

We need to describe the specified behavior first!

Requirements Specification:
“A track shall support a moving train”



Erroneous State (“Error”)



Fault

Algorithmic fault: Compass shows wrong values

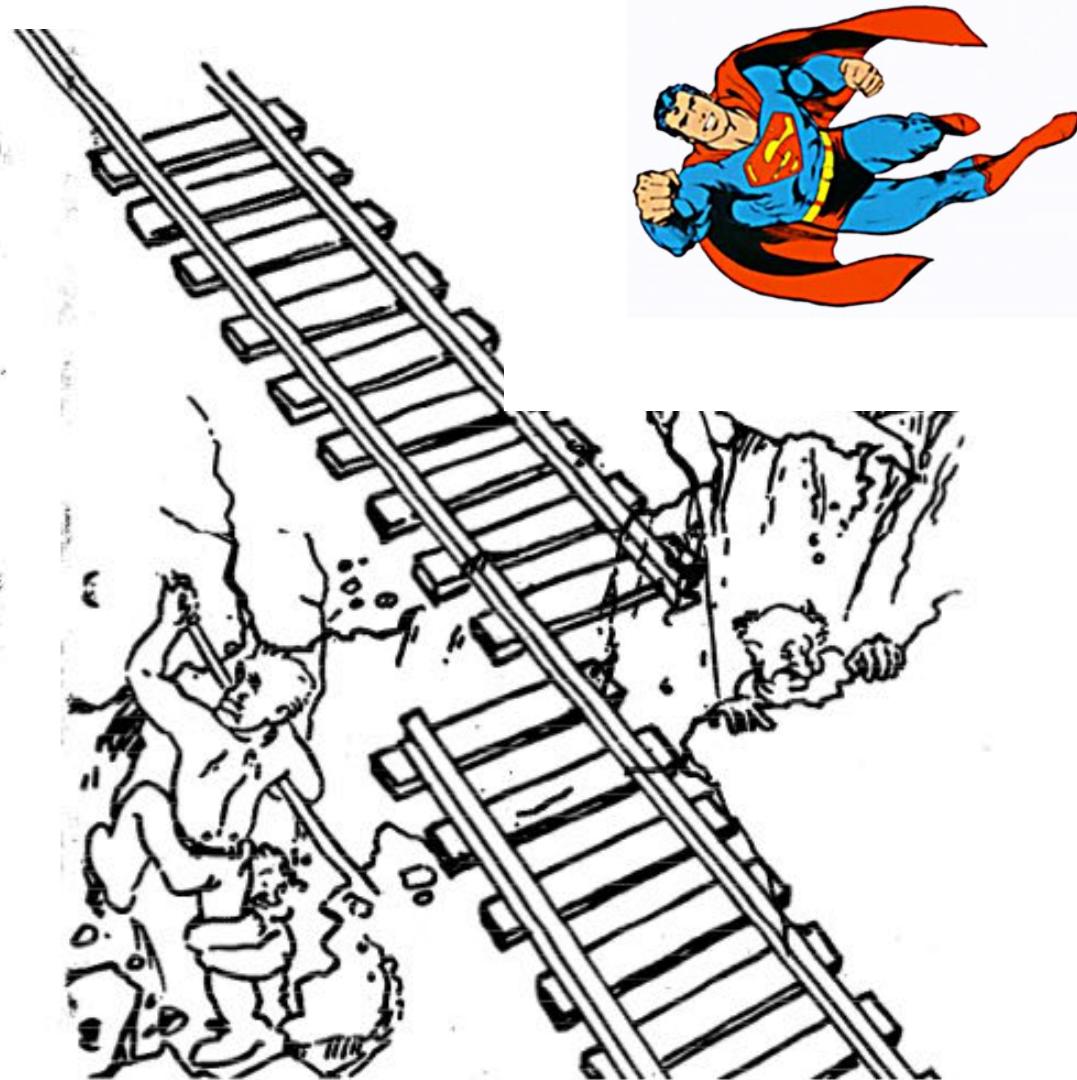
Or *usage fault*:

Wrong usage of compass

Or *communication fault*: The two teams had problems talking to each other.



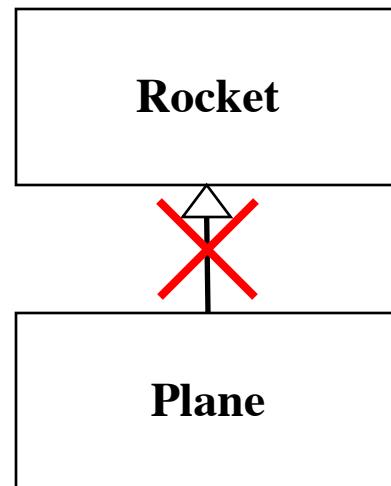
Mechanical Fault



F-16 Bug



- Where is the failure?
- Where is the error?
- Where is the fault?
 - Bad use of implementation inheritance
 - A Plane is **not** a rocket.



Examples of Faults and Errors

- **Faults in the Interface specification**

- Mismatch between what the client app needs and what the server offers
- Mismatch between requirements and implementation

- **Algorithmic Faults**

- Missing initialization
- Incorrect branching condition
- Missing test for null

- **Mechanical Faults (very hard to find)**

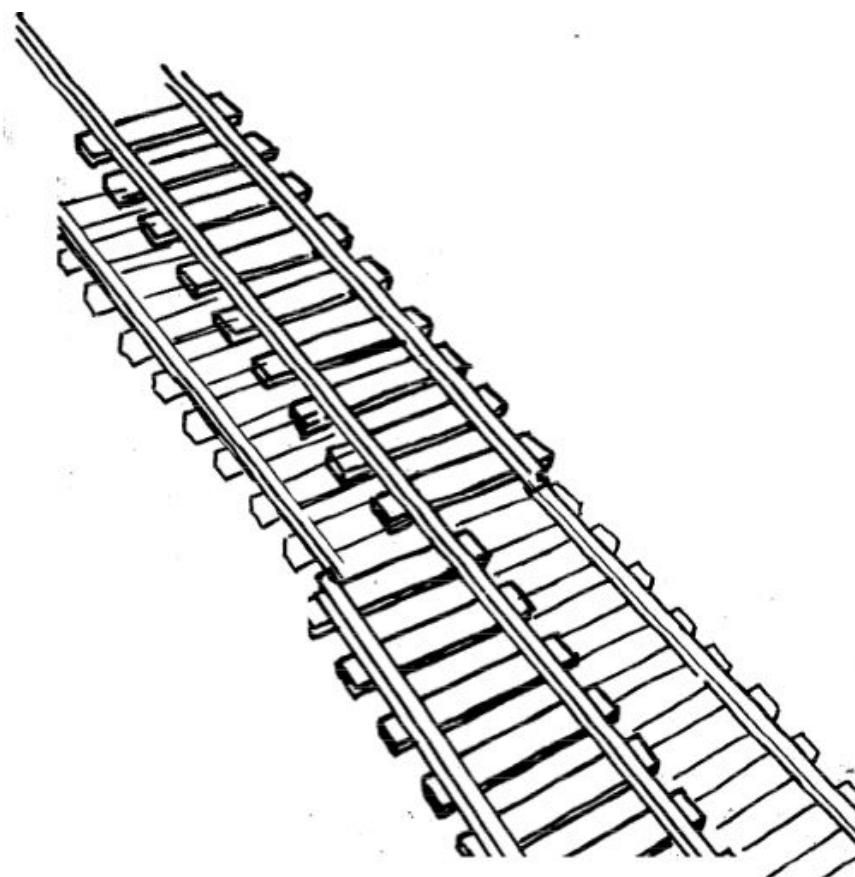
- Operating temperature outside of equipment specification

- **Errors**

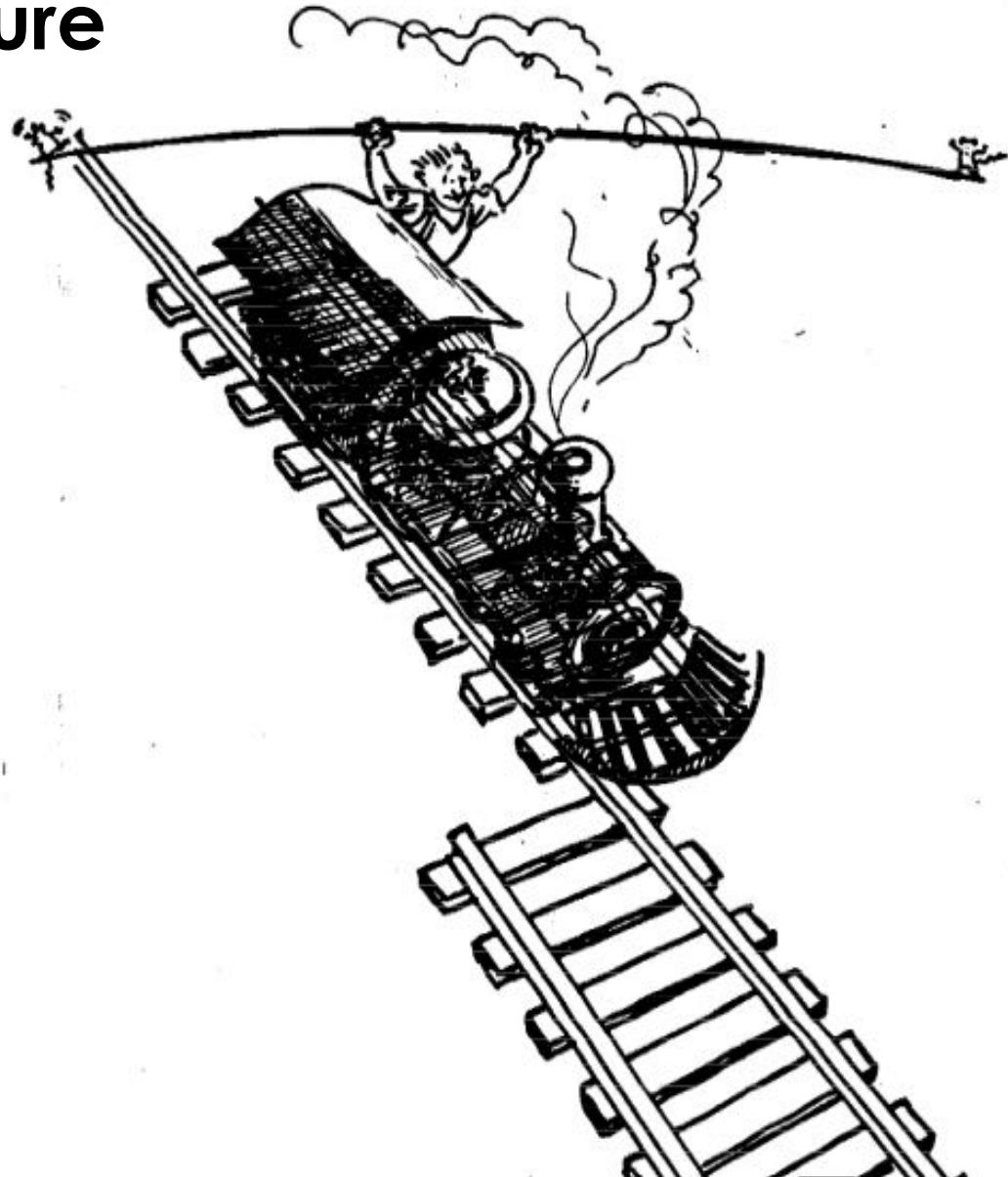
- Wrong user input
- Null reference errors
- Concurrency errors
- Exceptions.

How do we deal with Errors, Failures and Faults?

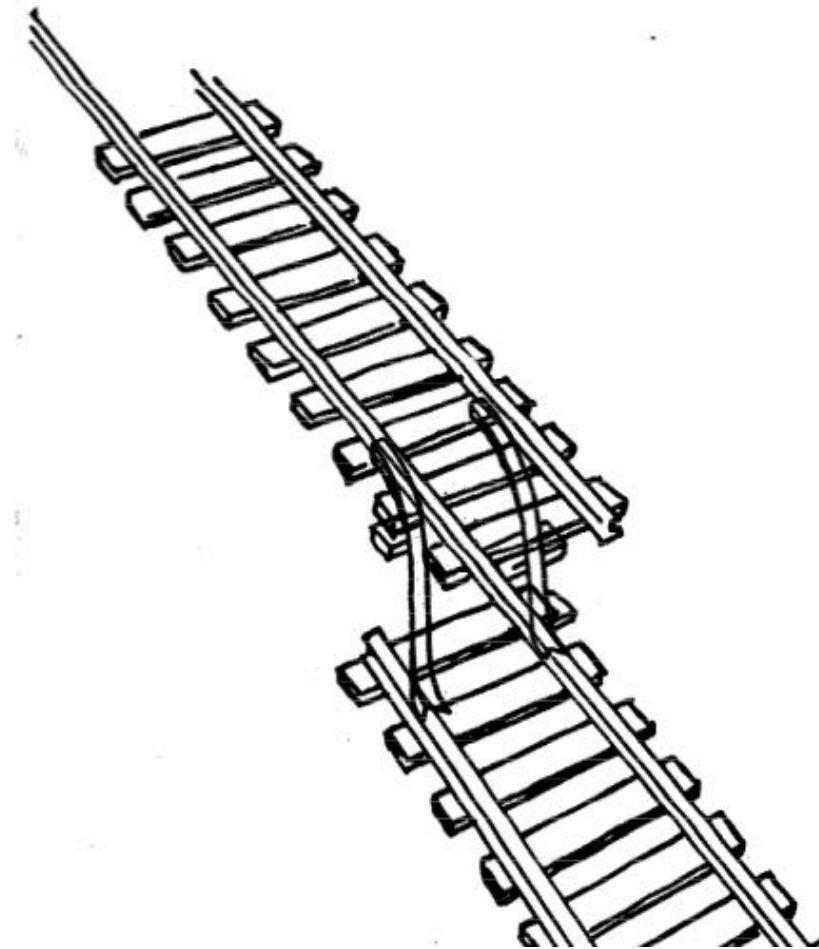
Modular Redundancy



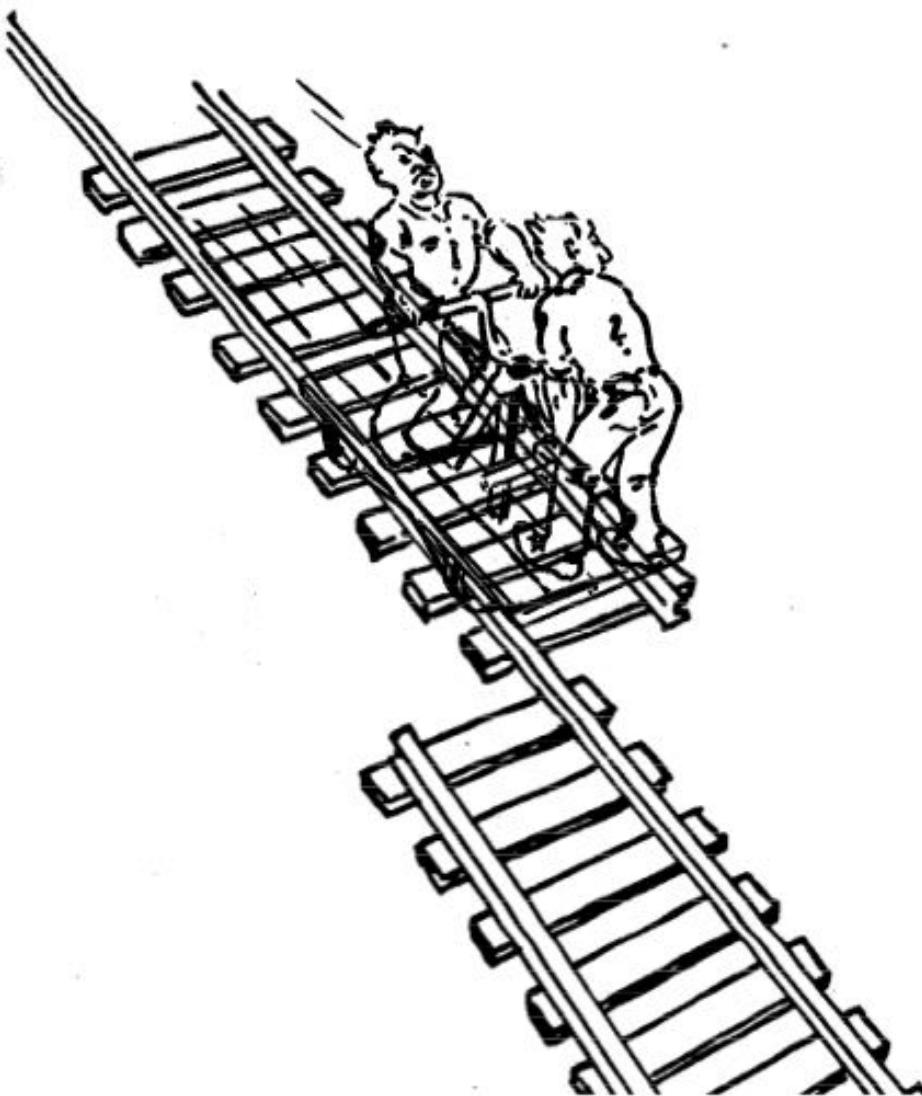
Declaring the Bug as a Feature



Patching



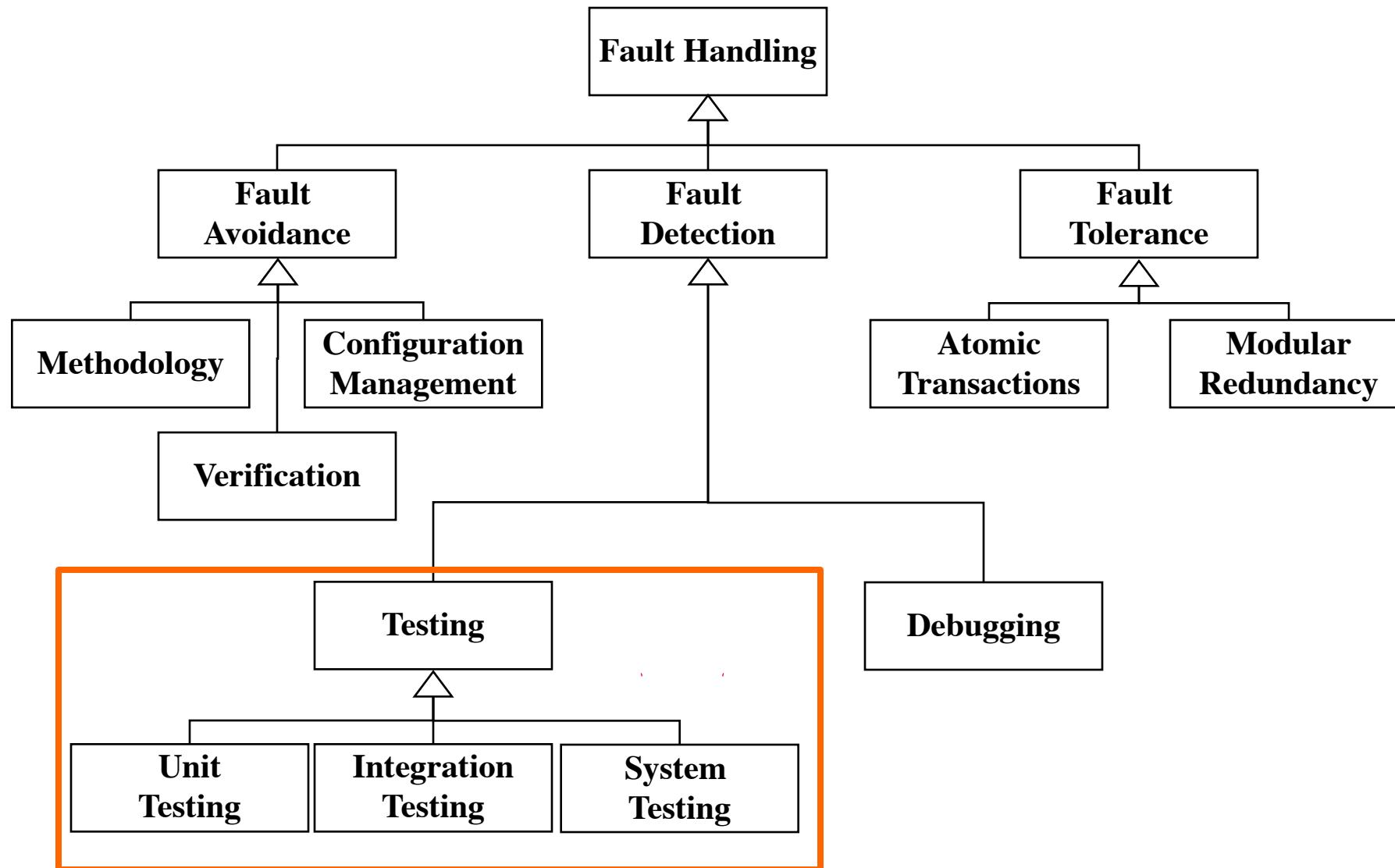
Testing



Another View on How to Deal with Faults

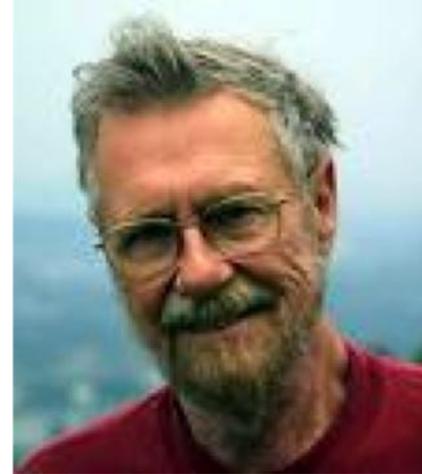
- **Fault avoidance**
 - Use methodology to reduce complexity
 - Use configuration management to prevent inconsistency
 - Apply verification to prevent algorithmic faults
 - Use reviews to identify faults already visible in the design
- **Fault detection**
 - Testing: Activity to provoke failures in a planned way
 - Debugging: Find and remove the cause (fault) of an observed failure
 - Monitoring: Deliver information about state and behavior => Used during debugging
- **Fault tolerance**
 - Exception handling
 - Modular redundancy.

Taxonomy for Fault Handling Techniques



Observations

- It is impossible to completely test any nontrivial module or system
 - Practical limitations: Complete testing is prohibitive in time and cost
 - Theoretical limitations: e.g. Halting problem
- “Testing can only show the presence of bugs, not their absence” (Dijkstra)
- Testing is not for free
=> Define your goals and priorities.



Edsger W. Dijkstra (1930-2002)

- First Algol 60 Compiler
- 1968:
 - T.H.E.
 - Go To considered Harmful, CACM
- Since 1970 Focus on Verification and Foundations of Computer Science
- 1972 A. M. Turing Award

Testing takes creativity

- To develop an effective test, one must have:
 - Detailed understanding of the system
 - Application and solution domain knowledge
 - Knowledge of the testing techniques
 - Skill to apply these techniques
- Testing is done best by independent testers
 - We often develop a certain mental attitude that the program should ~~in~~ in a certain way when in fact it does not
 - Program ~~behave~~s often stick to the data set that makes the program work
 - A program often does not work when tried by somebody else.

Test Model

- The **Test Model** consolidates all test related decisions and components into one package (sometimes also test package or test requirements)
- The test model contains a test driver, input data, the oracle, the test harness and of course the test cases
 - The **test driver** is the program that executes the test cases
 - The **test case** (often just called **test**) a function usually derived from a use case
 - The **input data** consist of the data needed for the test cases
 - The **oracle** predicts the expected output data
 - The **test harness**
 - A framework or software components that allows to run the tests under varying conditions and monitor the behavior and outputs of the system under test (SUT)
 - Test harnesses are necessary for automated testing.

Automated Testing

- There are two ways to generate the test model
 - **Manually:** The developers set up the test data, run the test and examine the results themselves. Success and/or failure of the test is determined through observation by the developers
 - **Automatically:** *Automated generation* of test data and test cases. Running the test is also done automatically, and finally the comparison of the result with the oracle is also done automatically
- **Definition Automated Testing**
 - All the test cases are *automatically executed* with a test harness
- Advantage of automated testing:
 - Less boring for the developer
 - Better test thoroughness
 - Reduces the cost of test execution
 - Indispensable for regression testing.

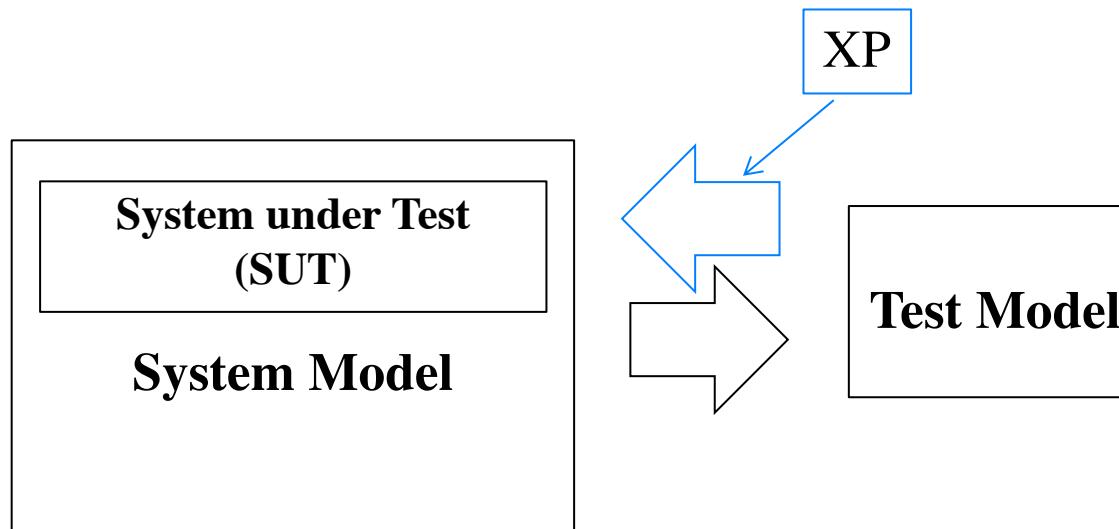
Model-Based Testing

Definition: Model Based Testing

- The system model is used for the generation of the test model
- Extreme Programming (XP) variant:
 - The test model is used for the generation of the system model

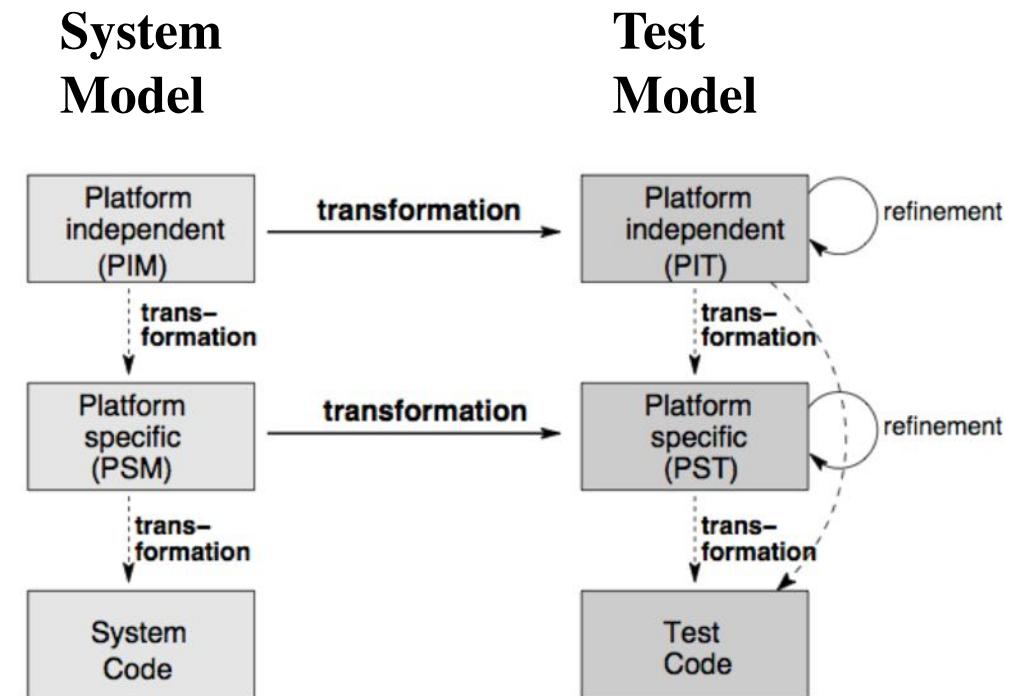
Definition: System under test (SUT)

- The part of the system model which is being tested



Model-Based Testing is part of Model-Driven Software Development

- System models are transformed into test models
 - When the system model is defined at the platform independent level (PIM), the platform-independent test model (PIT) can be derived
 - When platform specific model (PSM) level is defined, the platform-specific test model (PST) can be derived
 - The PST can also be derived by transforming the PIT model
 - Executable Test Code is then derived from the PST and PIT models
- After each transformation, the test model may change.
Examples:
 - If PIT and PST models must cover unexpected system behavior, special exception handling code must be added to the test code
 - Model-based testing enables the early introduction of testing into the system development process.



Model-driven Software Development

Some acronyms for model-driven software development

- **MDE (Model-Driven Engineering)**
 - A development methodology which focuses on creating and exploiting domain models (application domain models, solution domain models). Usually used when the development consists of hardware and software components

MDD (Model-Driven Development)

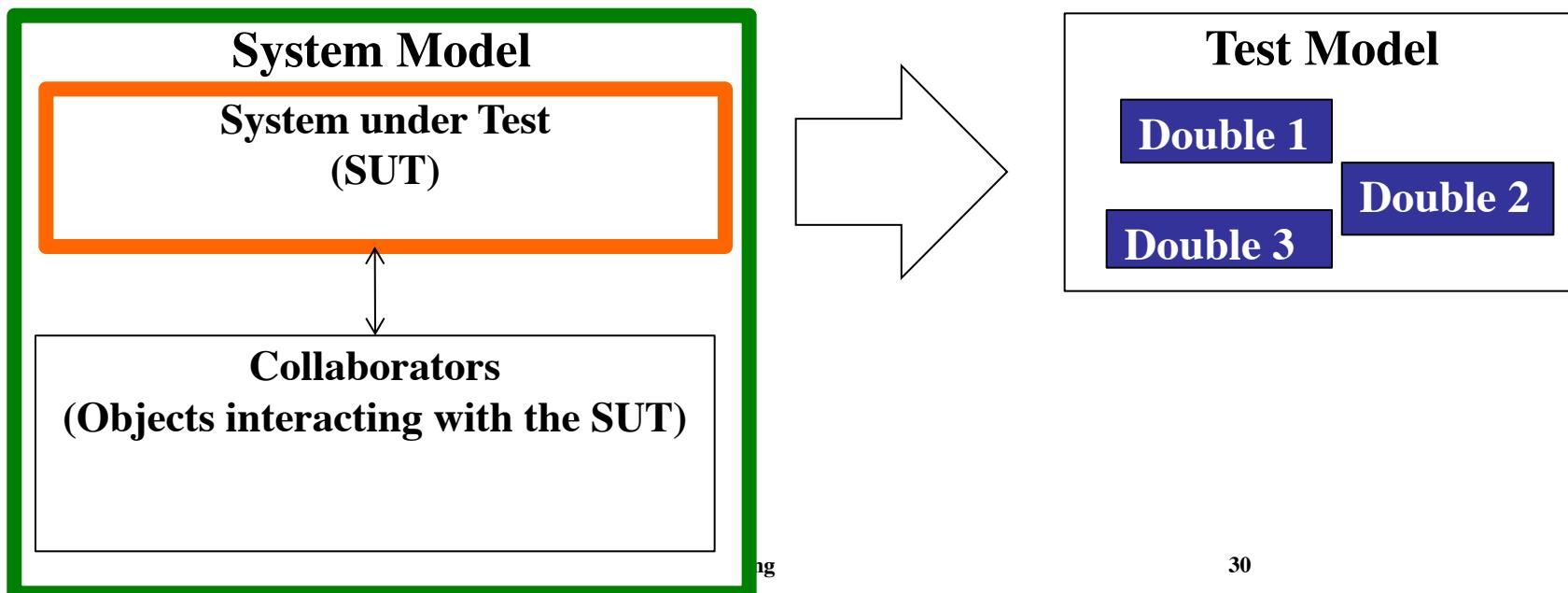
- It is impossible to keep a model consistent automatically after a manual change of the generated code. Manual changes to generated code should be avoided. MDD forbids round-trip engineering
- **MDA (Model-Driven Architecture)**
 - The system functionality is defined using a platform-independent model (PIM) and an appropriate domain-specific language (DSL). The MDA acronym is used mainly by the Object Management Group (OMG).

Outline of the Lectures on Testing

- ✓ Terminology
 - ✓ Failure, Error, Fault
- ✓ Test Model
- ✓ Model-based testing
- ➡ Object-Oriented testing
 - Testing activities
 - Static Analysis
 - Black box and White Box testing
 - Unit testing
 - Integration testing
 - System Testing
 - Function testing
 - Performance testing
 - Acceptance Testing.

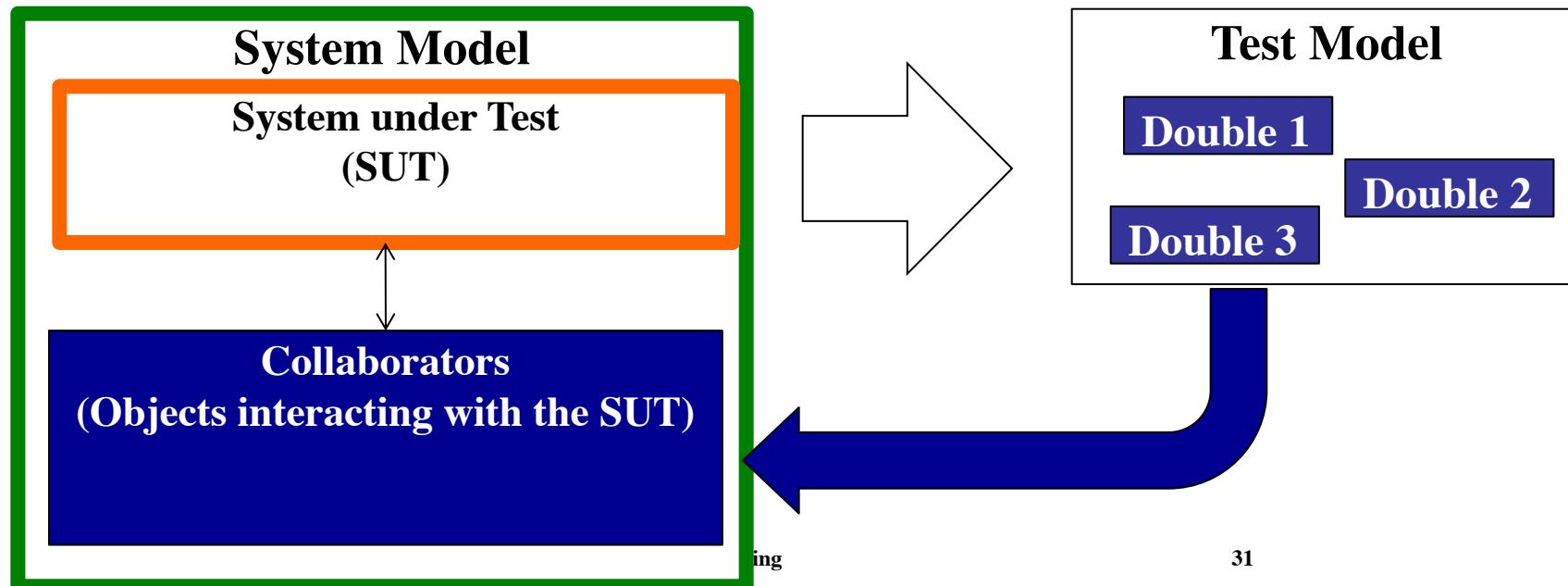
Object-Oriented Test Modeling

- We start with the system model
- The system contains the SUT
- The SUT does not exist in isolation, it collaborates with other objects in the system model
- The test model is derived from the SUT
- To be able to interact with collaborators, we add *objects to the test model*
- These objects are called **test doubles**



Object-Oriented Test Modeling

- We start with the system model
- The system contains the SUT (the unit we want to test)
- The SUT does not exist in isolation, it collaborates with other objects in the system model
- The test model is derived from the SUT
- To be able to interact with collaborators, we add *objects to the test model*
- These objects are called **test doubles**
- These doubles are substitutes for the Collaborators during testing



Taxonomy of Test Doubles

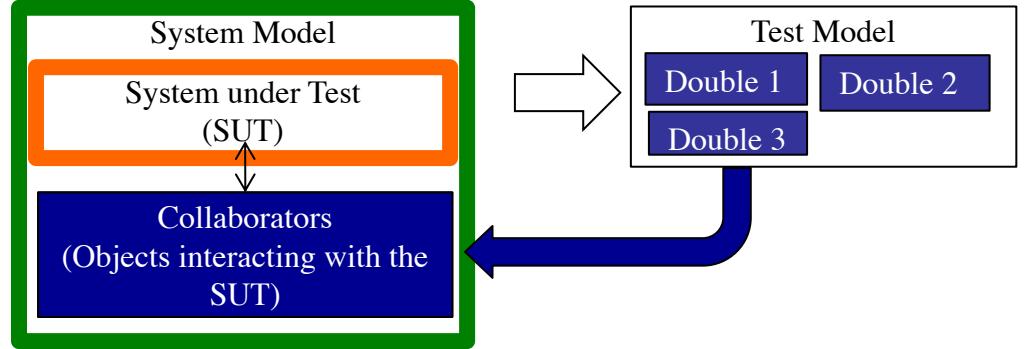
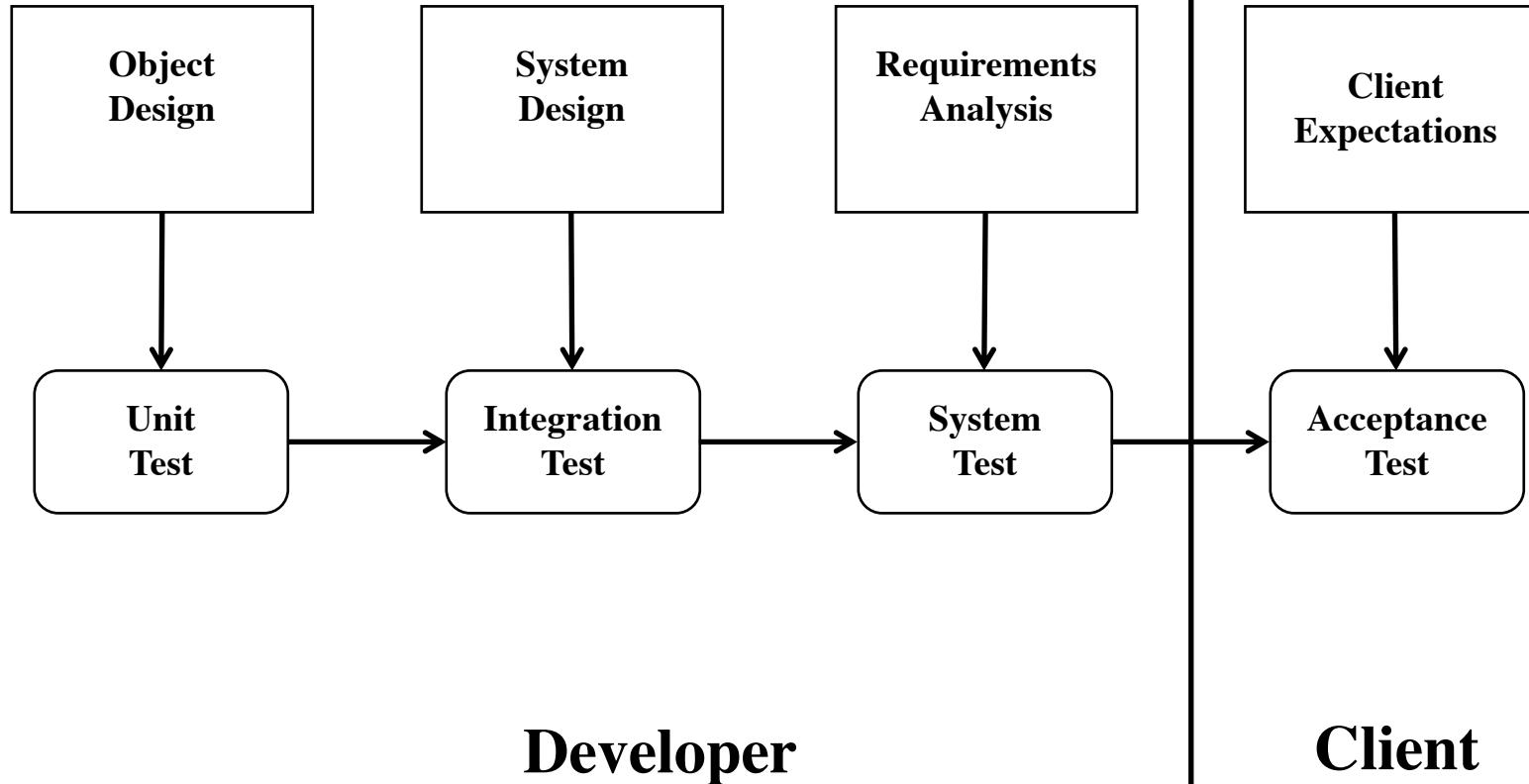


There are 4 types of test doubles:

- **Dummy object:** Often used to fill parameter lists, passed around but never actually used
 - **Fake object:** A working implementation that contains a “shortcut” which makes it not suitable for production code
 - Example: A database stored in memory instead on a disk
 - **Stub:** Provides canned answers to calls made during the test. Provides always the same answer
 - Example: Random number generator that always return 3.14
- **Mock object:** Mock objects are able to mimic the behavior of the real object. They know how to deal with a specific sequence of calls they are expected to receive

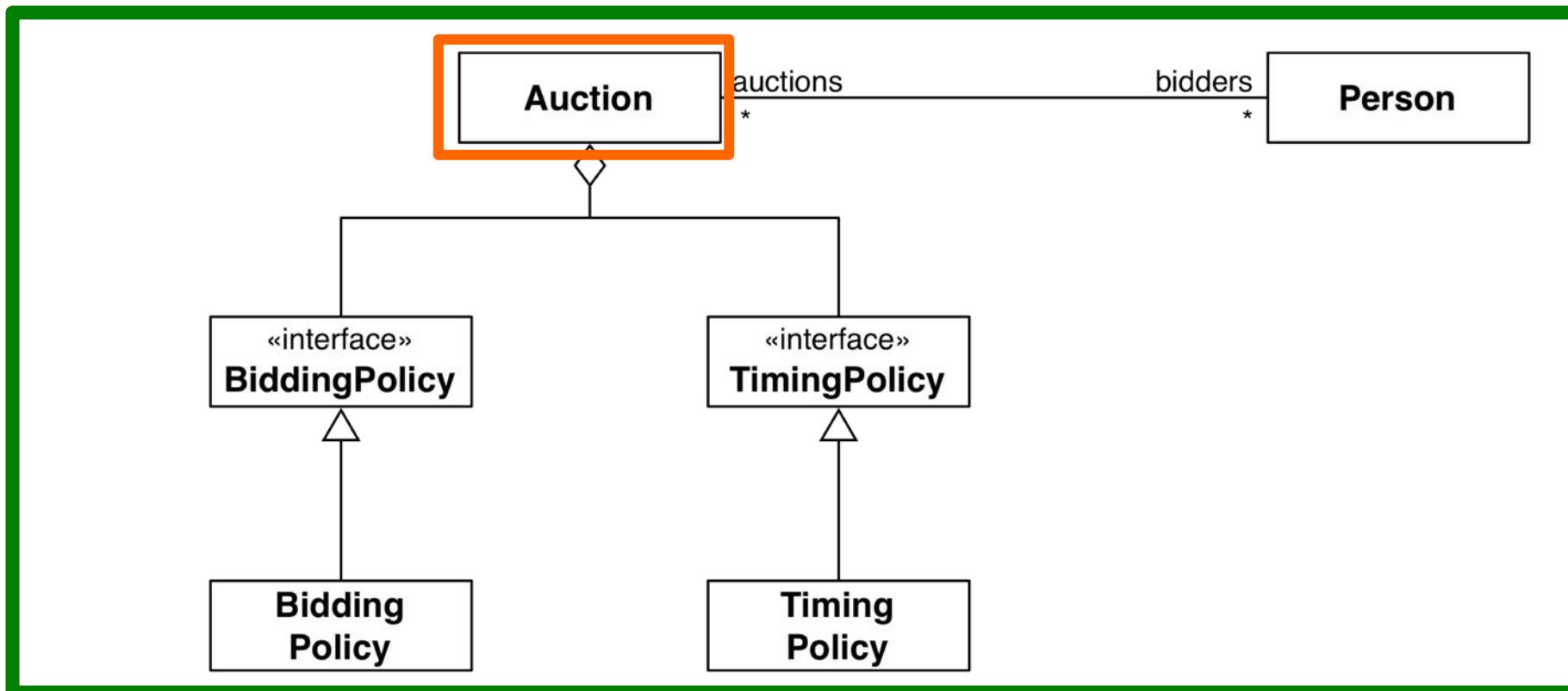
**Good design is crucial when using mock objects:
The real object (subsystem) must be specified with an interface (façade) and a class for the implementation.**

SUT Examples



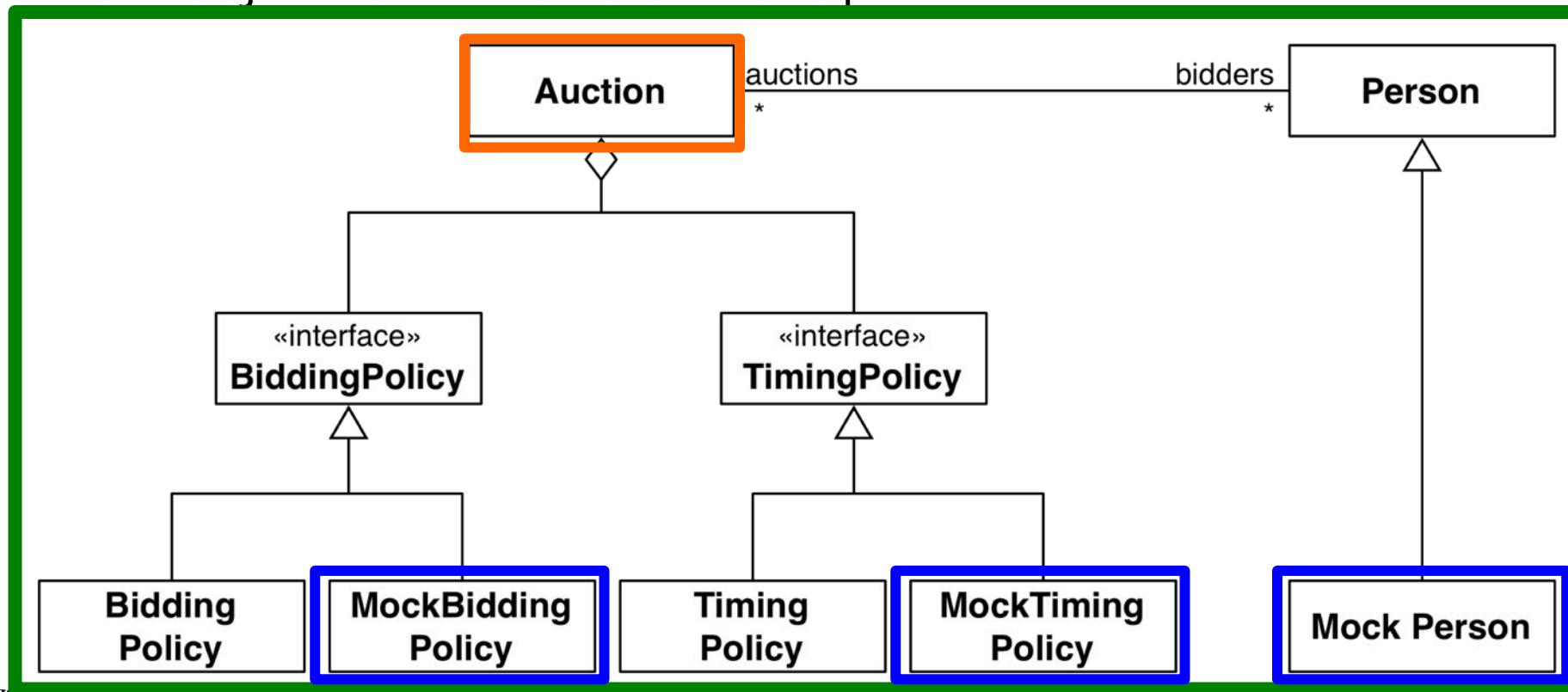
Motivation for Mock Objects

- Let us assume we have a **system model** for an auction system with 2 types of policies. We want to unit test Auction, which is our **SUT**

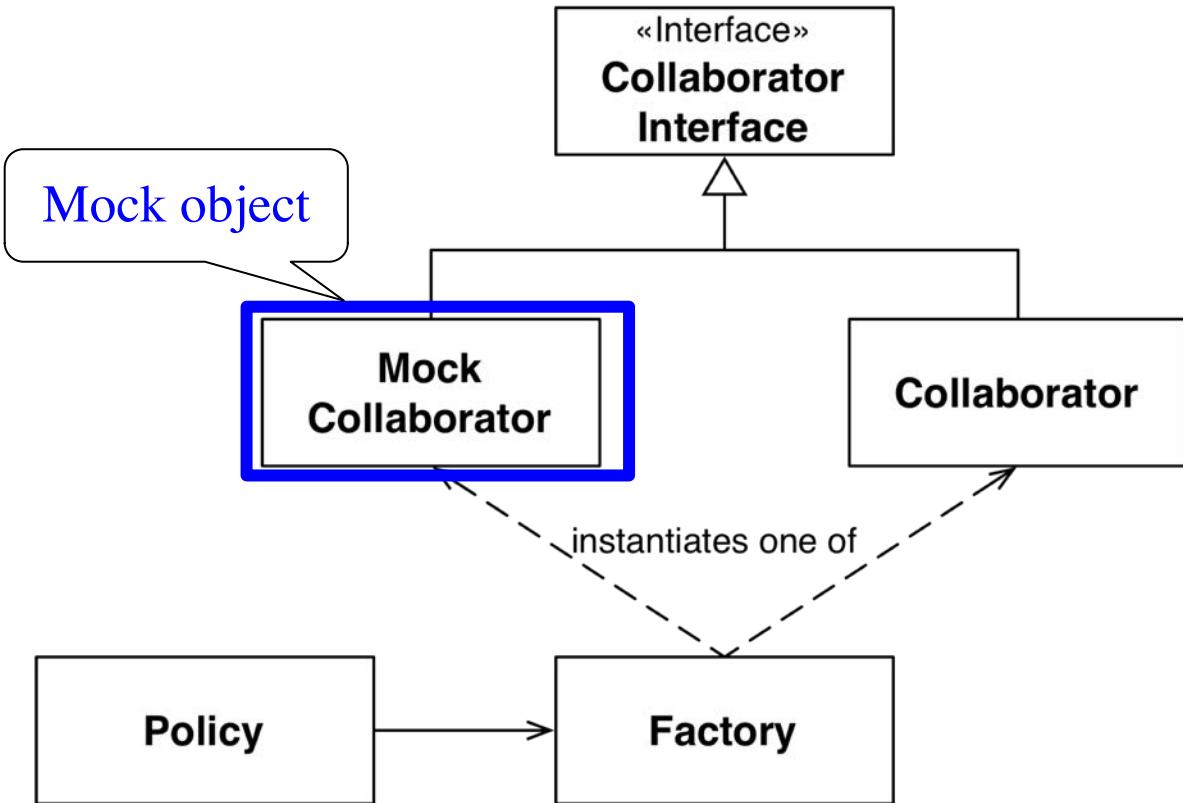


Motivation for Mock Objects

- Let us assume we have a **system model** for an auction system with 2 types of policies. We want to unit test Auction, which is our **SUT**
- The mock object test pattern is based on the idea to replace the interaction with the collaborators in the system model, that is Person, the Bidding Policy and the TimingPolicy by **mock objects**
- These mock objects are created at startup-time.



Mock-Object Pattern



- In the mock object pattern a **mock object** replaces the *behavior* of a real object called the collaborator and returns hard-coded values
- A mock object can be created at startup-time with the factory pattern (Not covered in the lecture, look it up in Gamma's book)
- Mock objects can be used for testing *state of individual objects* as well as the *interaction between objects*, that is, to validate that the interactions of the SUT with its collaborators behave as expected
- The use of Mock objects is based on the **Record-Play Metaphor**.

Record-Replay Metaphor

Assume you want to perform a musical, which requires an orchestra and a choir. Most of the time the orchestra will not be available (too expensive), when the choir practices. But the choir needs to be accompanied by the music played by the orchestra when rehearsing the musical:



During Rehearsal the Choir must sing in sync with the music on the Mock (Tape).

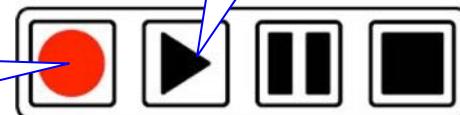


During Rehearsal the music on the Mock is played

The Tape is the Mock

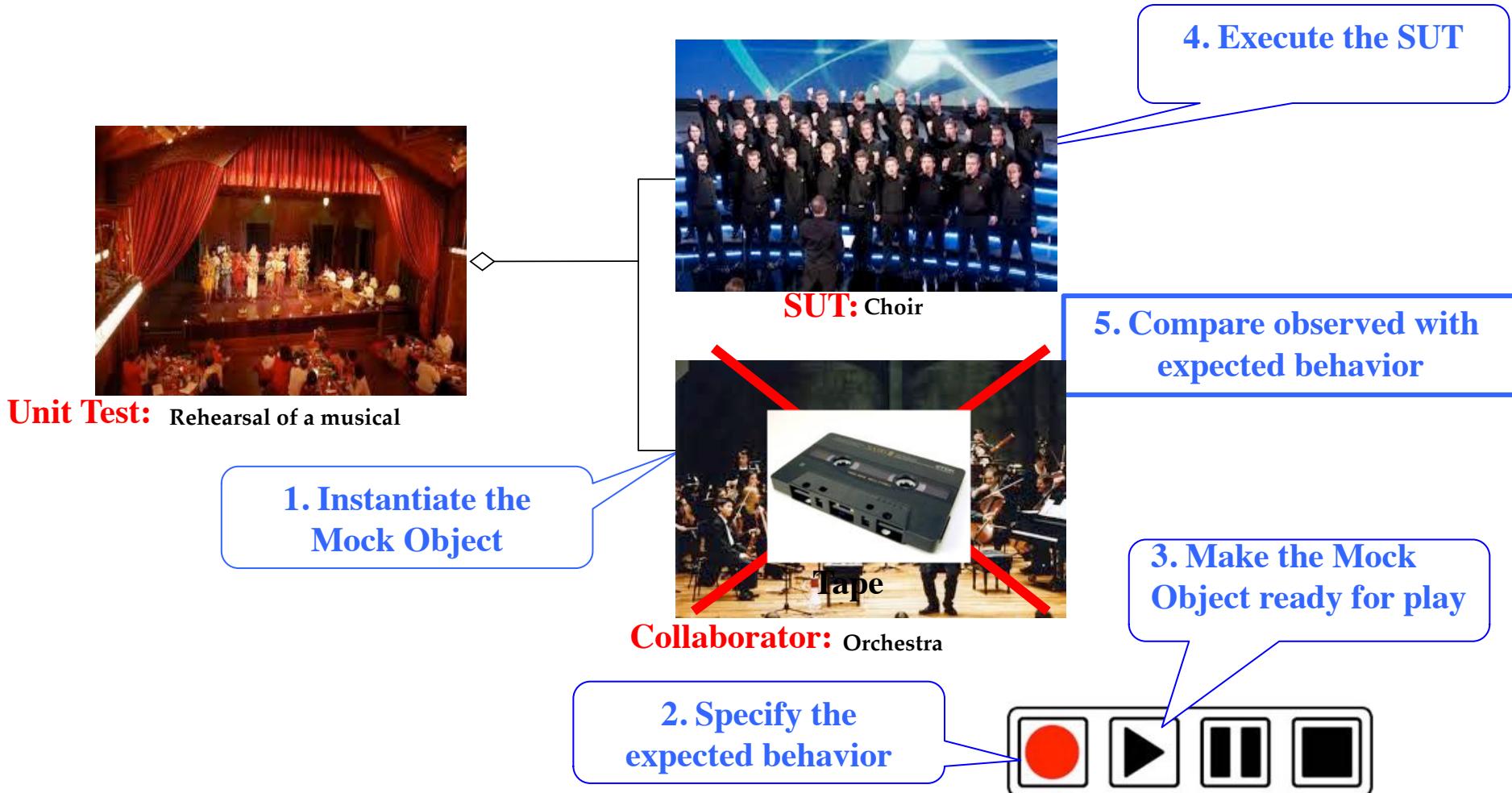


The Orchestra records the music onto the Mock



Record-Replay Metaphor for Mock Objects

- Mock objects are proxy collaborators in tests where the real collaborators are not available



Outline of the Lectures on Testing

- ✓ Terminology
 - ✓ Failure, Error, Fault
- ✓ Test Model
- ✓ Model-based testing
- ✓ Model-driven testing
- Testing activities
 - Static Analysis
 - Black box and White Box testing
 - Unit testing
 - Integration testing
 - System Testing
 - Function testing
 - Performance testing
 - Acceptance Testing.

Dynamic Analysis and Static Analysis

- **Dynamic Analysis**

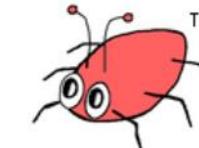
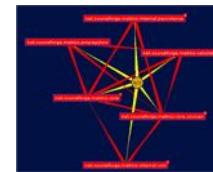
- Black-box testing: Tests the input/output behavior
- White-box testing: Tests the implementation of the subsystem or class.

- **Static Analysis**

- Hand execution by reading the source code
- Walk-Through by informal presentation to others
- Code Inspection by formal presentation to others
- Automated tools that check for
 - syntactic and semantic errors
 - departure from coding standards

Static Analysis Tools in Eclipse

- Compiler Warnings and Errors
 - Possibly uninitialized variable
 - Undocumented empty block
 - Assignment with no effect
 - Missing semicolon, ...
- Checkstyle
 - Checks for Java code guideline violations
 - <http://checkstyle.sourceforge.net>
 - Good example: <http://maven.apache.org/plugins/maven-checkstyle-plugin/checkstyle.html>
- Metrics
 - Checks for structural anomalies
 - <http://metrics.sourceforge.net>
- SpotBugs (formerly called FindBugs)
 - Uses static analysis to look for bugs in Java code
 - <https://spotbugs.github.io/>



Observation about Static Analysis

- Static analysis typically finds mistakes but some mistakes do not matter
 - Important to find the intersection of stupid and important mistakes
 - Not a magic bullet but if used effectively, static analysis is cheaper than other techniques for catching the same bugs
- Static analysis catches at best 5-10% of software quality problems
- Source: William Pugh, Mistakes that Matter, JavaOne Conference
 - <http://www.cs.umd.edu/~pugh/MistakesThatMatter.pdf>

In-Class Exercise: FindBugs

Tasks:

1. Start the exercise **Lecture 11 In-Class Exercise 01** on ArTEMiS
2. Clone your exercise repository
3. Commit your solution
4. Push your solution to the remote server

Task 1: Start the exercise on ArTEMiS

Introduction to Software Engineering (Summer 2018) ⓘ

Exercise	Due date	Results	Actions
Show 12 overdue exercises			
Lecture 10 In-Class	in a day	You have not started this exercise yet.	Start exercise
Exercise 02 Continuous Integration			

Click on Start exercise



Task 2: Clone your exercise repository

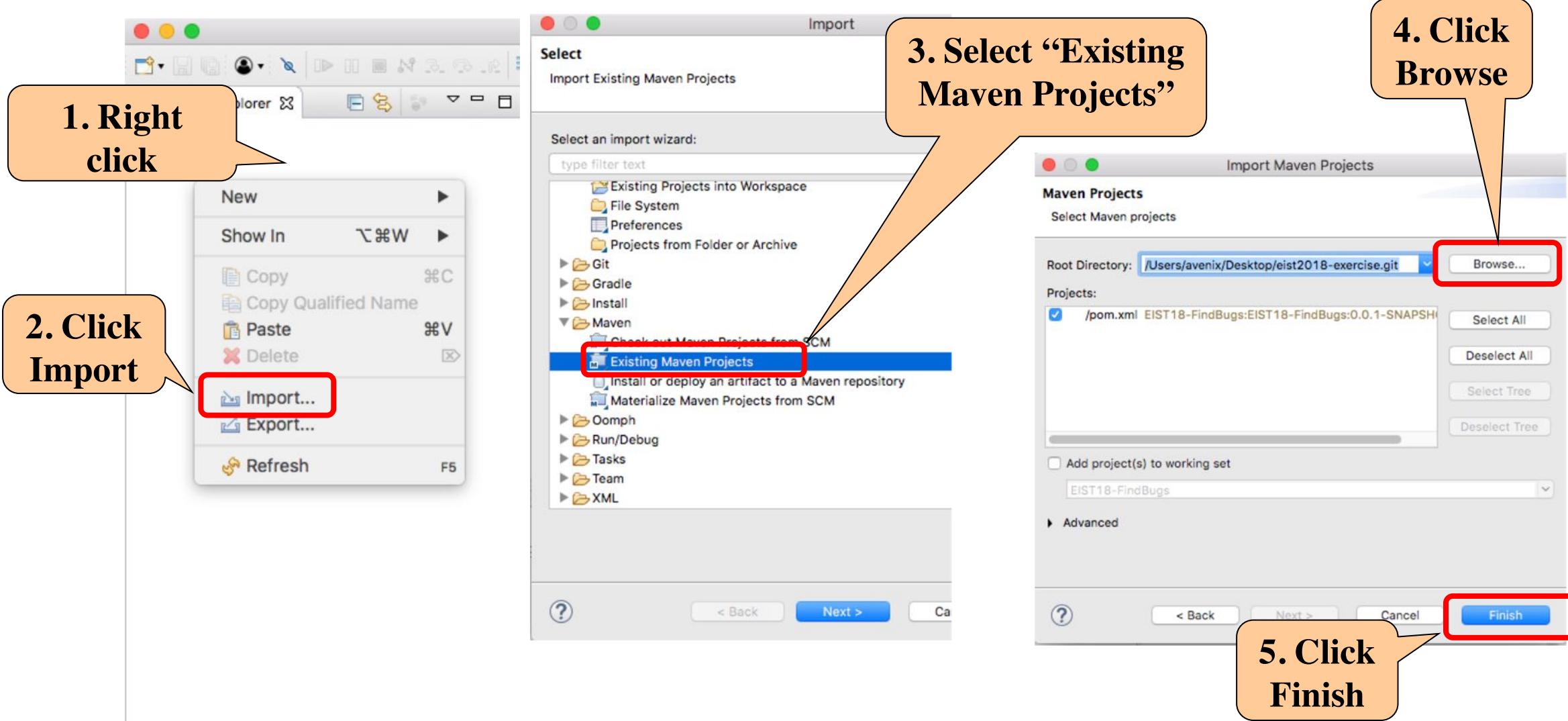
The screenshot shows a table of exercises. The first row has a blue header with the title "Introduction to Software Engineering (Summer 2018)". The columns are labeled "Exercise", "Due date", "Results", and "Actions".

Exercise	Due date	Results	Actions
Show 12 overdue exercises		Clone your personal repository for this exercise: https://ne23kow@repobruegge.in.tum.de/scm/eist2018l10e01/eist2018-l10-e01-exercis	Clone repository
Lecture 10 In-Class Exercise 02 Continuous Integration	in a day	Clone in SourceTree Atlassian SourceTree is the free Git client for Windows or Mac.	

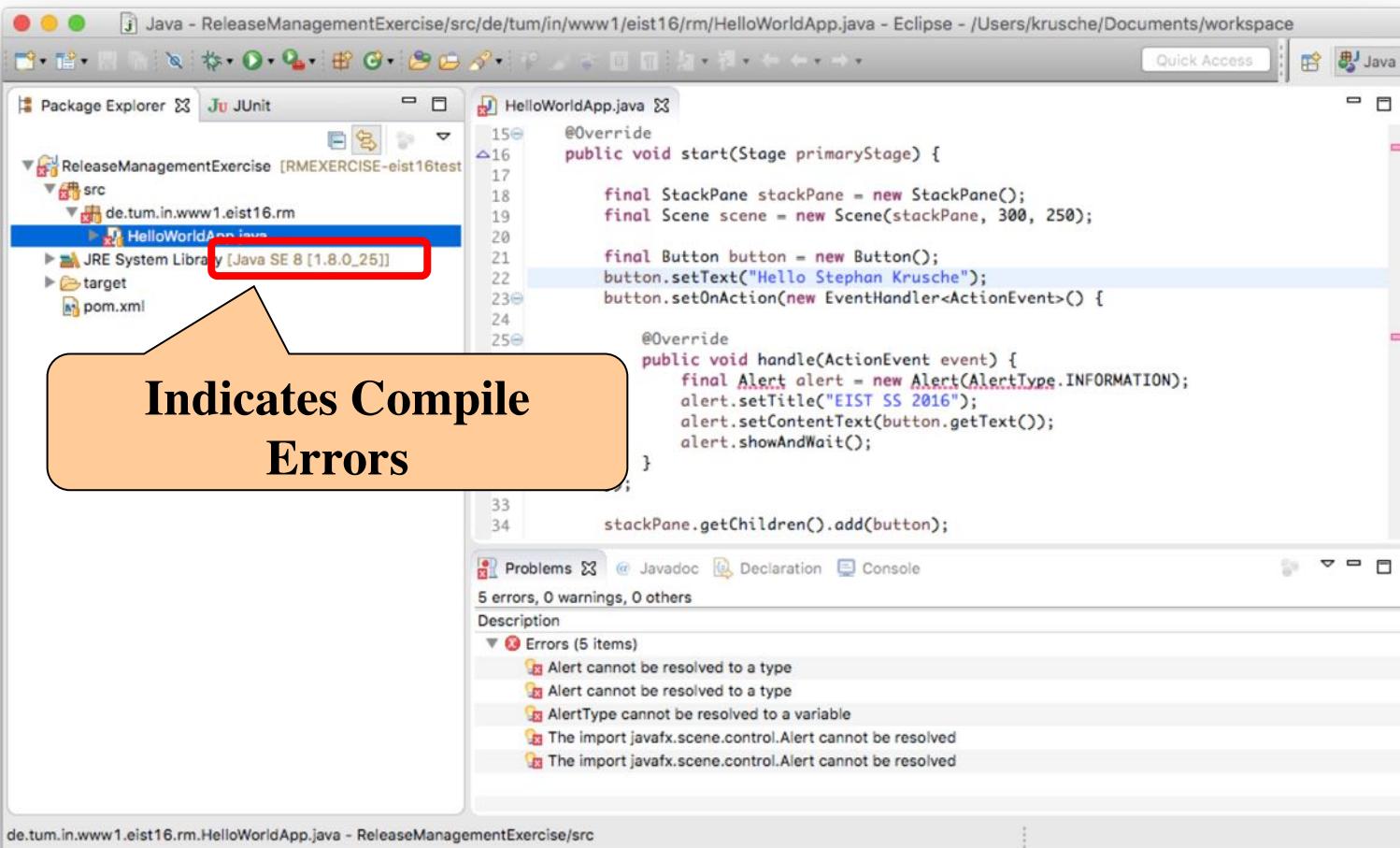
Two specific buttons are highlighted with red boxes and orange callout arrows:

- A blue button labeled "Clone in SourceTree" is highlighted with a red box and an orange arrow pointing to it from below, with the text "Click on Clone in SourceTree".
- A blue button labeled "Clone repository" is highlighted with a red box and an orange arrow pointing to it from the right, with the text "Click on Clone repository".

Task 3: Import the Maven project into Eclipse



In case you have compilation errors



Other issues:

- Right click the project → Maven → Update project
- Clean the Maven Cache on your computer

➤ Make sure to install the newest Java 8 JDK version (>= Java 8u172):
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

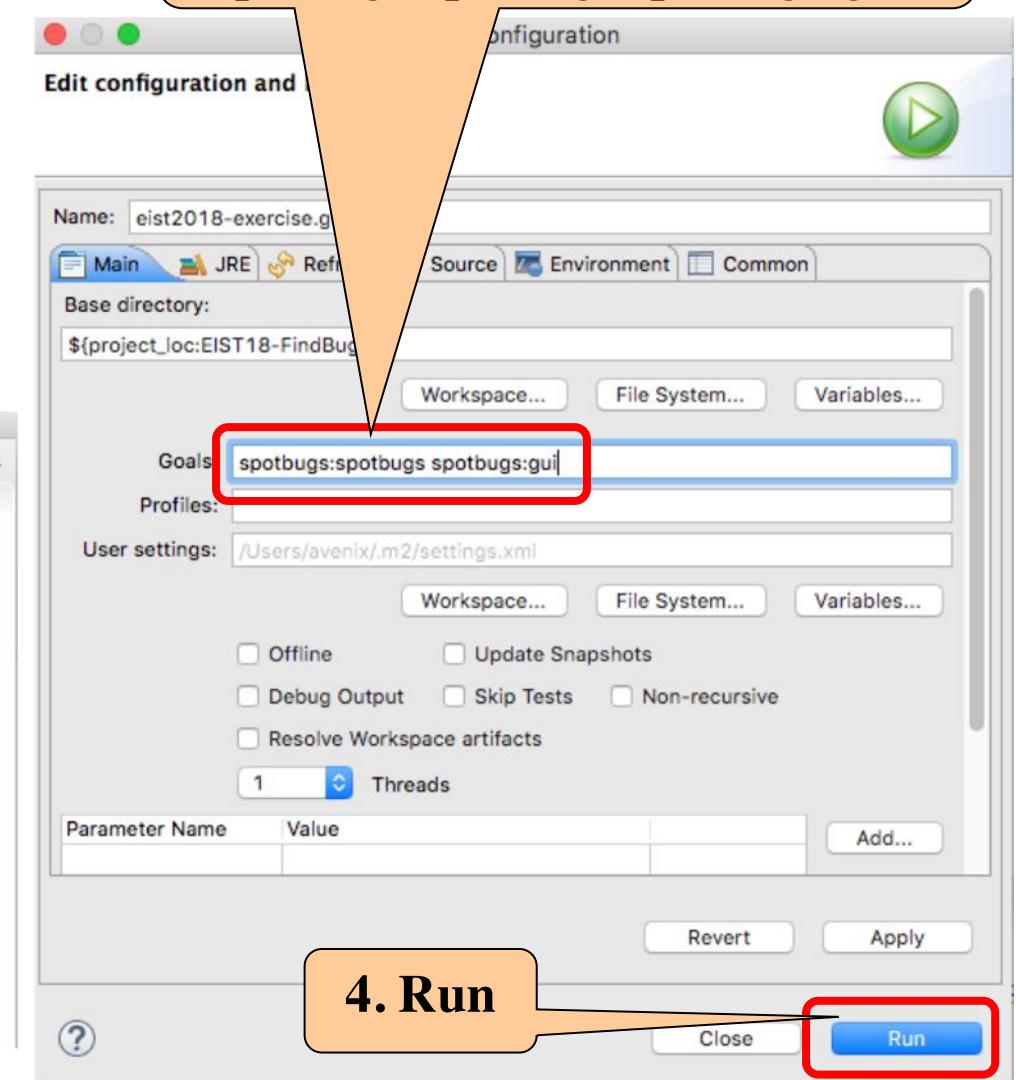
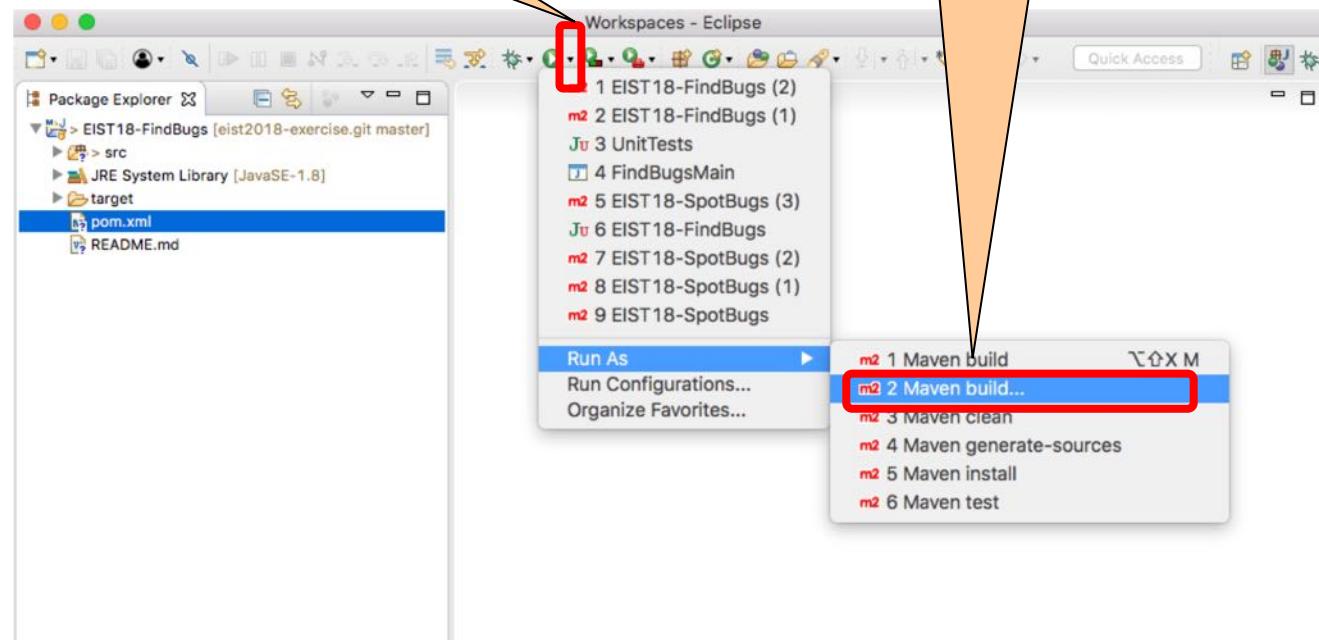
Task 4: Run Spotbugs (Maven Plugin)

3. Goals are:

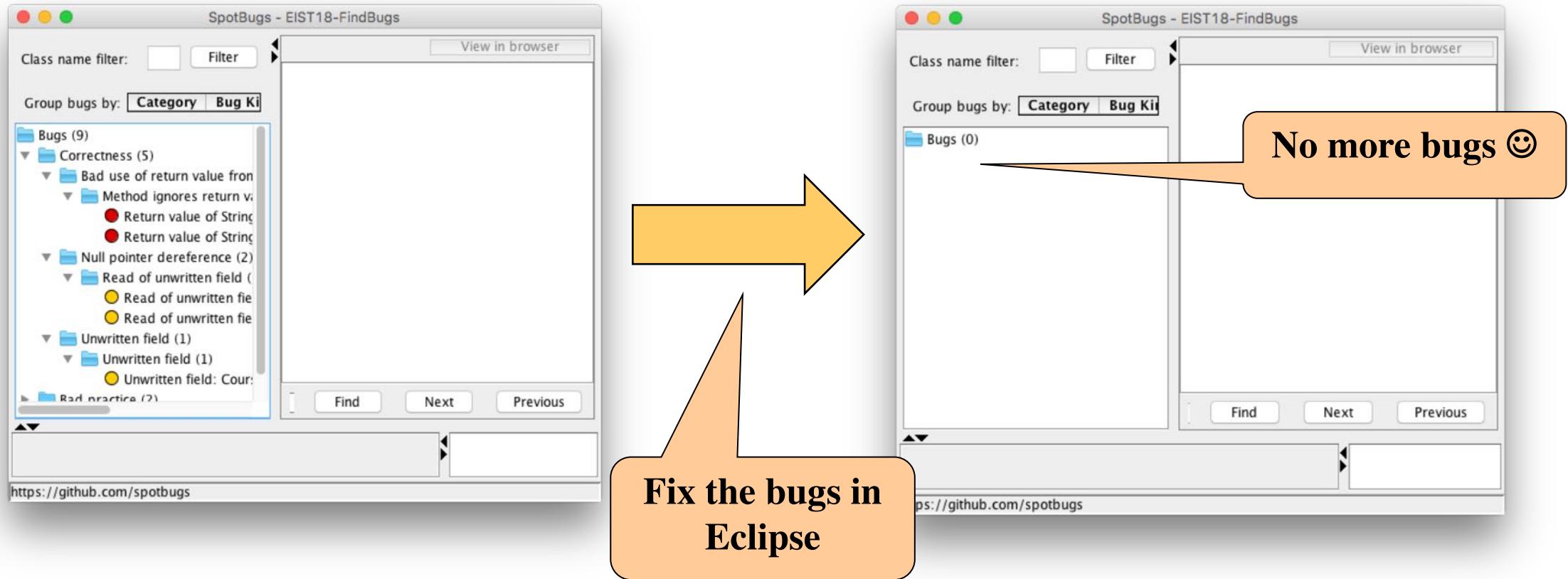
spotbugs:spotbugs spotbugs:gui

1. Arrow next to
Run Button

2. Run As
Maven
build...

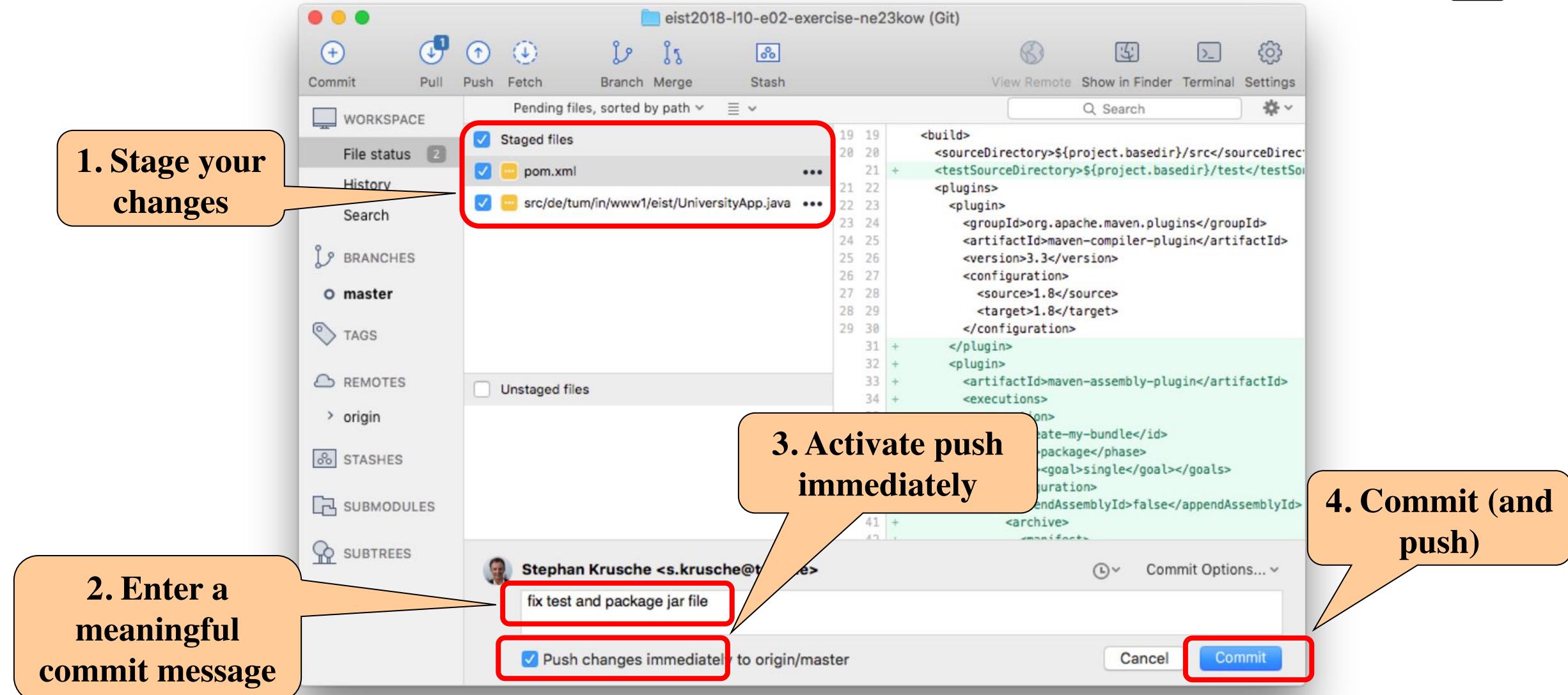


Task 5: Fix the bugs



Task 6: Commit and push your changes

Duration
10 min



Solution: Bug in the Student class

SpotBugs - EIST18-FindBugs

Student.java in de.tum.jk.findbugs [View in browser](#)

```
7  public Student(String firstName, String lastName) {
8      setFirstName(firstName);
9      setLastName(lastName);
10 }
11
12 public String getFirstName() {
13     return firstName;
14 }
15
16 public void setFirstName(String firstName) {
17     firstName.replaceAll("[^A-Za-z]", "");
18     this.firstName = firstName;
19 }
20
21 public String getLastName() {
22     return lastName;
23 }
24
25 public void setLastName(String lastName) {
26     lastName.replaceAll("[^A-Za-z]", "");
```

Find Next Previous

Method ignores return value

The return value of this method should be checked. One common cause of this warning is to invoke a method on an immutable object, thinking that it updates the object. For example, in the following code fragment,

```
String dateString = getHeaderField(name);
dateString.trim();
```

the programmer seems to be thinking that the trim() method will update the String referenced by dateString. But since Strings are immutable, the trim() function returns a new String value, which is being ignored here. The code should be corrected to:

```
String dateString = getHeaderField(name);
dateString = dateString.trim();
```

```
public void setFirstName(String firstName) {
    firstName = firstName.replaceAll("[^A-Za-z]", "");
    this.firstName = firstName;
}
```

Solution: Bug in the Course class

SpotBugs - EIST18-FindBugs

Course.java in de.tum.jk.findbugs View in browser

```
1 package de.tum.jk.findbugs;
2 import java.util.List;
3
4 public class Course {
5     private String name;
6     public List<Student> participants;
7
8     public Course(String name) {
9         this.name = name;
10    }
11
12    public String getName() {
13        return name;
14    }
15
16    public void setName(String name) {
17        this.name = name;
18    }
19
20    public int getNumberOfParticipants() {
21        return this.participants.size();
22    }
23}
```

Find Next Previous

Read of unwritten public or protected field
The program is dereferencing a public or protected field that does not seem to ever have a non-null value written to it. Unless the field is initialized via some mechanism not seen by the analysis, dereferencing this value will generate a null pointer exception.

Bug kind and pattern: NP - NP_UNWRITTEN_PUBLIC_OR_PROTECTED_FIELD

```
public Course(String name) {
    this.name = name;
    participants = new ArrayList<Student>();
}
```

Solution: Bug in the TUMonline class

SpotBugs - EIST18-FindBugs

TUMOnline.java in de.tum.jk.findbugs

View in browser

```
4 public class TUMOnline {
5     private ArrayList<Course> registeredCourses;
6     private ArrayList<Student> enrolledStudents;
7
8     public TUMOnline() {
9         registeredCourses = new ArrayList<Course>();
10        enrolledStudents = new ArrayList<Student>();
11    }
12
13    public void enrolStudent(Student student) {
14        enrolledStudents.add(student);
15    }
16
17    public void registerCourse(Course course) {
18        this.registeredCourses.add(course);
19    }
20
21    public Student findStudentByName(String firstName, String lastName) {
22        for (Student student : this.enrolledStudents) {
23            if (student.getFirstName() == firstName && student.getLastName() != lastName)
24                return student;
25        }
26    }
27    return null;
}
```

Find Next Previous

Comparison of String parameter using == or !=

This code compares a java.lang.String parameter for reference equality using the == or != operators. Requiring callers to pass only String constants or interned strings to a method is unnecessarily fragile, and rarely leads to measurable performance gains. Consider using the equals(Object) method instead.

Bug kind and pattern: ES – ES_COMPARING_PARAMETER_STRING_WITH_EQ

```
public Student findStudentByName(String firstName, String lastName) {
    for (Student student : this.enrolledStudents) {

        if (student.getFirstName().equals(firstName) && student.getLastName().equals(lastName))
            return student;
    }
    return null;
}
```

Outline of the Lectures on Testing

- Terminology
 - Failure, Error, Fault
- Test Model
- Model-based testing
- Object-Oriented testing
- Testing activities
 - Static Analysis
- ➡ Black box and White Box testing
 - Unit testing
 - Integration testing
 - System Testing
 - Function testing
 - Performance testing
 - Acceptance Testing.

Black-box Testing

- Focus: I/O behavior. If for any given input, we can predict the output, then the unit passes the test.
 - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by **equivalence partitioning**:
 - Divide inputs into equivalence classes
 - Allowed input, illegal input
 - For an enumerated type or array: Below the range, in the range, above the range
 - Choose test cases for each equivalence class
 - Example: If an object is supposed to accept a negative number, testing one negative number is enough.

White-box Testing

- Focus: Thoroughness (Coverage). Every statement in the component is executed at least once
- Four types of white-box testing
 - Statement Testing
 - Loop Testing
 - Path Testing
 - Branch Testing.

White-box Testing (Continued)

- Statement Testing (Algebraic Testing)
 - Tests each statement (Choice of operators in polynomials, etc)
- Loop Testing
 - Loop to be executed exactly once
 - Loop to be executed more than once
 - Cause the execution of the loop to be skipped completely
- Path testing:
 - Makes sure all paths in the program are executed
- Branch Testing (Conditional Testing)
 - Ensure that each outcome in a condition is tested at least once
 - Example of 2 statements:

```
i = FALSE;  
if ( i = TRUE) printf("Yes") else printf("No");
```

- How many test cases do we need to test these statements?

Example of Branch Testing

```
if ( i = TRUE) printf("Yes");else printf("No");
```

- We need two test cases with the following input data
 - 1) i = TRUE
 - 2) i = FALSE
- What is the expected output for the two cases?
 - In both cases: "Yes"
 - This a typical beginner's mistake in languages, where the assignment operator also returns the value assigned (C, Java)
- So tests can be faulty as well ☹
- Some of these faults can be identified with static analysis.

Comparison of White & Black-box Testing

- White-box Testing
 - Potentially infinite number of paths have to be tested
 - White-box testing often tests what is done, instead of what should be done
 - Cannot detect missing use cases
- Black-box Testing
 - Potential combinatorical explosion of test cases (valid & invalid data)
 - Does not discover extraneous use cases ("features")
- Both types of testing are needed
 - White-box testing and black box testing are the ends of a testing continuum
 - Any choice of test case lies in between these ends and depends on the following:
 - Number of possible logical paths
 - Nature of input data
 - Amount of computation
 - Complexity of algorithms and data structures.

Outline of the Lectures on Testing

- ✓ Terminology
 - ✓ Failure, Error, Fault
- ✓ Test Model
- ✓ Model-based testing
- ✓ Object-Oriented testing
- ✓ Testing activities
 - Static Analysis
 - ➡ Unit testing
 - Integration testing
 - System Testing
 - Function testing
 - Performance testing
 - Acceptance Testing.

Unit Testing

- Unit testing is a testing method where individual units in a program are tested.
- In procedural programming, a unit is usually a function or procedure
- In object-oriented programming, a unit is usually the class: This can be an attribute, an individual method or the interface of the class
- Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process.
- Unit tests form the basis of integration testing.

Guidance for Unit Test Case Selection

- Use *analysis knowledge* about functional requirements (black-box testing):
 - Scenarios Use cases
 - Expected input data
 - Invalid input data
- Use *design knowledge* about system structure, algorithms, data structures (white-box testing):
 - Control structures
 - Test branches, loops, ...
 - Classes and Data structures
 - Test methods, attributes, records fields, arrays, ...
- Use *implementation knowledge* about algorithms and datastructures:
 - Force a division by zero
 - If the upper bound of an array is 10, then use 11 as index.

JUnit Overview

- A Java framework for writing and running unit tests
- JUnit is Open Source
 - www.junit.org
 - <https://github.com/junit-team/junit>
- Designed initially by Kent Beck and Erich Gamma (JUnit 3)
 - Designed with “Test First” in mind
 - In “Test First” the tests are written before coding the SUT
 - Observe those test cases that creating failures
 - Write new SUT code or fix existing code to make the tests pass
 - JUnit 3 used inheritance for specifying test cases
 - The programmer had to inherit from `junit.framework.TestCase`
- The newest version JUnit 4 uses Java **annotations** and Java **assertions** instead of inheritance.

15 Minute Break



JUnit example: Testing Money class

Problem Statement: Storing, adding, and subtracting **money** in a computer currently is not possible. We can do these operations only on **integers**. As a result we might accidentally add 5 Euros (€) and 7 US Dollars (\$), which is of course invalid

Solution: Create a Money class that provides the currency abstraction (encapsulating amount and type of currency)

Functional Requirements:

1. Store an amount and currency
2. Add money with the same currency
3. Return **Null** if the addition is invalid (e.g. 6 Euro + 5 US Dollars is invalid).

Money class

```
class Money {  
    private int amount;  
    private Currency currency;  
    public Money(int amount, Currency currency) {  
        this.amount = amount;  
        this.currency = currency;  
    }  
    public int amount() {  
        return amount;  
    }  
    public Currency currency() {  
        return currency;  
    }  
    ...  
}
```

Money class, ctd.

```
class Money {  
    private int amount;  
    private Currency currency;  
    ...  
    public boolean equals(Object object) {  
        if (object instanceof Money) {  
            Money money = (Money)object;  
            return money.currency().equals(currency) &&  
                money.amount() == amount;  
        }  
        return false;  
    }  
  
    public Money add(Money money) {  
        return new Money(amount() + money.amount(), currency());  
    }  
}
```

add() is the SUT method
from the System Model

A Unit Test for the add() method

The unit test for the class Money should test the add method. Below you can see an example test for the addition of Money. Currently this test is very limited.

```
import org.junit.Test;  
  
public class MoneyTest {  
    @Test public void simpleAdd() {  
        Money m12CHF= new Money(12, "CHF");  
        Money m14CHF= new Money(14, "CHF");  
        Money expected= new Money(26, "CHF");  
        Money observed= m12CHF.add(m14CHF);  
    }  
}
```

@Test annotation:
simpleAdd() is the name of the test case in the test driver

MoneyTest is a test driver in the Test Model

This is still an incomplete unit test as we do not yet compare expected and observed state.

Here we are calling the SUT Method add() from the System Model

Annotations in JUnit 4.x

→ **@Test** public void foo()

Annotation @Test identifies that foo() is a test method

@Test(expected=IllegalArgumentException.class)

Tests if the test method throws the named exception

@Test(timeout=100)

The test fails if it takes longer than 100 milliseconds.

@Before public void bar()

Perform bar() before executing any test method

@After public void foobar()

Any test method must finish with call to foobar()

@BeforeClass public void foofoo()

Perform foofoo() before the start of all tests. Used to perform time intensive activities, e.g. to connect to a database

@AfterClass public void blabla()

Perform blabla() after all tests have finished. Used to perform clean-up activities, e.g. to disconnect to a database

@Ignore(string S)

Ignore the test method prefixed by @Ignore, print out the string S instead. Useful if the code has been changed but the test has not yet been adapted

Annotations in JUnit 4

- @Test public void foo()
 - Annotation @Test identifies that foo() is a test method
- @Before public void bar()
 - Perform bar() before executing any test method
- @After public void foobar()
 - Any test method must finish with call to foobar()
- @BeforeClass public void foofoo()
 - Perform foofoo() before the start of all tests.,
- @AfterClass public void blabla()
 - Perform blabla() after all tests have finished.
- @Ignore(string S)
 - Ignore the test method prefixed by @Ignore, print out the string S instead
- @Test(expected=IllegalArgumentException.class)
 - Tests if the test method throws the named exception
- @Test(timeout=100)
 - Fails if the test method takes longer than 100 milliseconds

@Before and @After: Ensuring Pre- and Post Conditions

Any Method can be annotated with @Before and @After:

```
public class CalculatorTest {  
    @Test public void addTest()  
    @Test public void subTest()  
    @Before public void setupTestData(){}/>executed before every addTest/subTest  
    @After public void teardownTestData(){}/>executed after every addTest/subTest  
}
```

Any Class containing a set of tests can be annotated with @BeforeClass and @AfterClass

This is useful for expensive setups that do not need to be run for every test, such as setting up a database connection.

```
public class CalculatorTest {  
    @BeforeClass // executed at instantiation of class  
        public static void setupDatabase Connection() { ... }  
    @AfterClass // executed after removing instance of class  
        public static void teardownDatabase Connection() { ... }  
}
```

Annotations in JUnit 4

- `@Test public void foo()`
 - Annotation `@Test` identifies that `foo()` is a test method
- `@Before public void bar()`
 - Perform `bar()` before executing any test method
- `@After public void foobar()`
 - Any test method must finish with call to `foobar()`
- `@BeforeClass public void foofoo()`
 - Perform `foofoo()` before the start of all tests.,
- `@AfterClass public void blabla()`
 - Perform `blabla()` after all tests have finished.

→ `@Ignore(string S)`

- Ignore the test method prefixed by `@Ignore`, print out the string `S` instead
- `@Test(expected=IllegalArgumentException.class)`
 - Tests if the test method throws the named exception

→ `@Test(timeout=100)`

- Fails if the test method takes longer than 100 milliseconds

@Ignore: Omitting Tests

- There are situations where certain tests should not be executed by the test harness
- Example:
 - The current release of a third-party library used in the SUT has a bug in a routine Foo. We cannot test Bar until Foo is fixed.

```
public class CalculatorTest {  
    @Ignore("Don't run the Bar test until bug in Foo  
    is fixed")  
    @Test public void Bar() {  
        ...  
    }  
}
```

@Test(timeout): Making Sure Tests are short

- Unit tests should be short
- But some tests take their time, particularly if network connectivity is involved
 - In these cases it is recommended to set an upper bound for the test

```
@Test(timeout=5000)  
public void testNetworkOperation() {
```

```
    ...
```

```
}
```

```
.
```

Assertions in JUnit 4

- • **assertTrue(Predicate);**
 - Checks if Predicate evaluates to true; otherwise throws an Exception
- **assertFalse(Predicate);**
 - Checks if Predicate evaluates to false; otherwise throws an Exception
- **fail(String)**
 - Let the method fail, useful to check that a certain part of the code is not reached.
- **assertEquals([message], expected, actual)**
 - Checks if the values **expected** and **actual** are the same; otherwise throws an Exception including the **message**
- **assertEquals([message], expected, actual, tolerance)**
 - Used for float and double; tolerance specifies the number of decimals which must be the same
- **assertNull([message], object)**
 - Checks if the object is null; otherwise throws an Exception including the **message**
- **assertNotNull([message], object)**
 - Check if the object is **not** null; otherwise throws an Exception including the **message**
- **assertSame([message], expected, actual)**
 - Check if both variables **expected** and **actual** refer to the same object; otherwise throws an Exception including the **message**
- **assertNotSame([message], expected, actual)**
 - Check that both variables **expected** and **actual** do **not** refer to the same object; otherwise throws an Exception including the **message**

Please note: Parameters in square brackets are optional, e.g. [message]

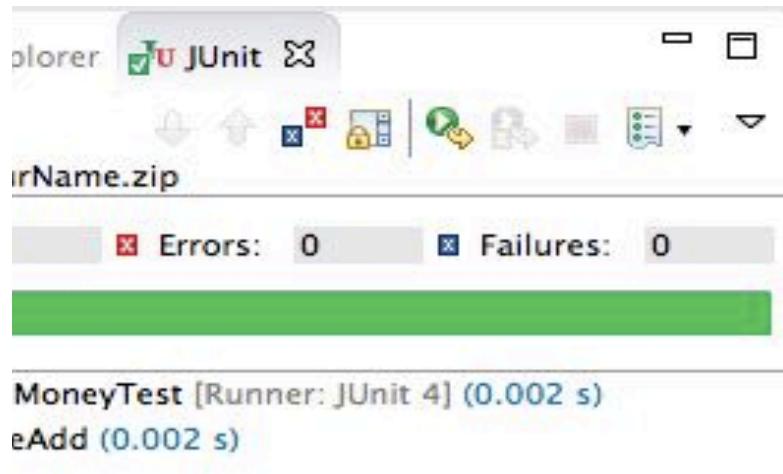
Using assertTrue in a Unit Test

The unit test for the class Money should test all public and protected methods in the class, except getters and setters. Below you can see an example test for the add() method of Money.

```
import org.junit.Test;  
import static org.junit.Assert.*;  
  
public class MoneyTest {  
    @Test public void simpleAdd() {  
        Money m12CHF = new Money(12, "CHF");  
        Money m14CHF = new Money(14, "CHF");  
        Money expected = new Money(26, "CHF");  
        Money observed = m12CHF.add(m14CHF);  
        assertTrue(expected.equals(observed));  
    }  
}
```

The test passes, if the parameter of type Boolean evaluates to True, otherwise the test throws an exception of typeAssertionError

Demo in Eclipse



```
1 package tum.pse;
2 import static org.junit.Assert.*;
3 
4 public class MoneyTest {
5     @Test public void eAdd() {
6         Money m12CHF= new Money(12, "CHF");
7         Money m14CHF= new Money(14, "CHF");
8         Money expected= new Money(26, "CHF");
9         Money observed= m12CHF.add(m14CHF);
10        assertEquals(expected, observed);
11    }
12 }
```

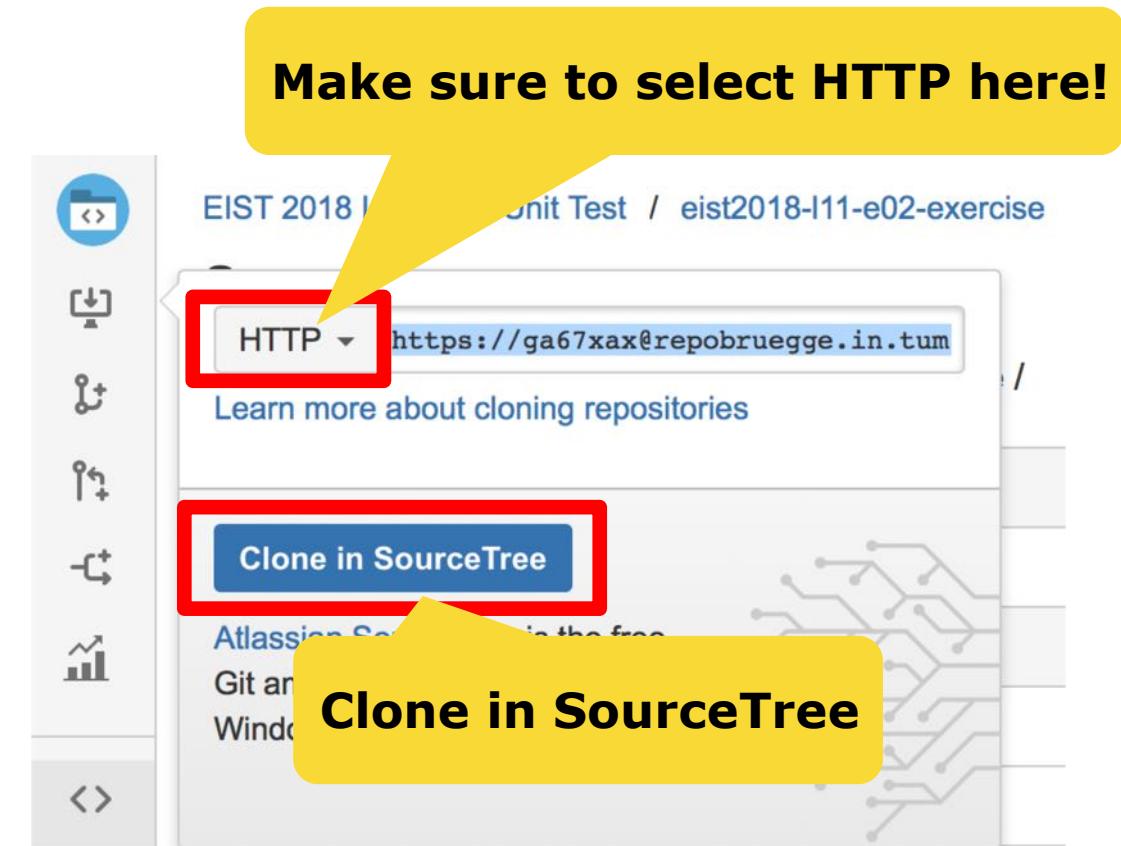
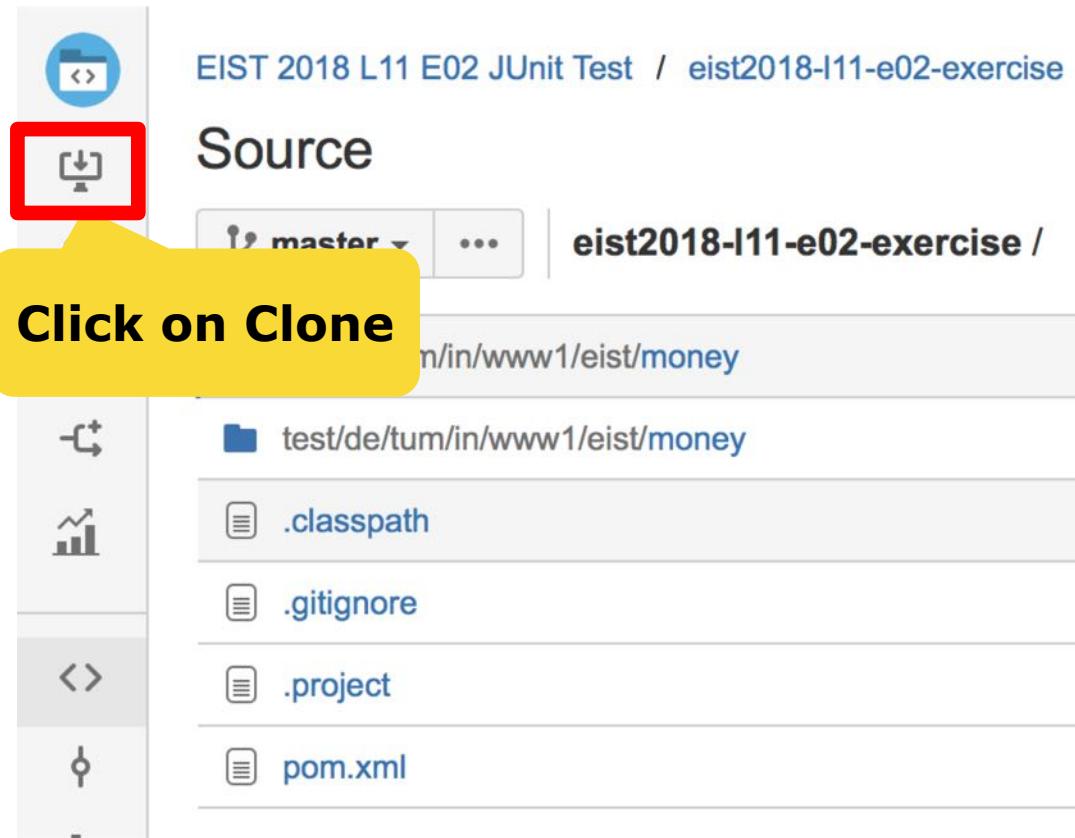
Edsger W Dijkstra:
Testing shows the presence, not the absence of faults ("bugs")

Tasks:

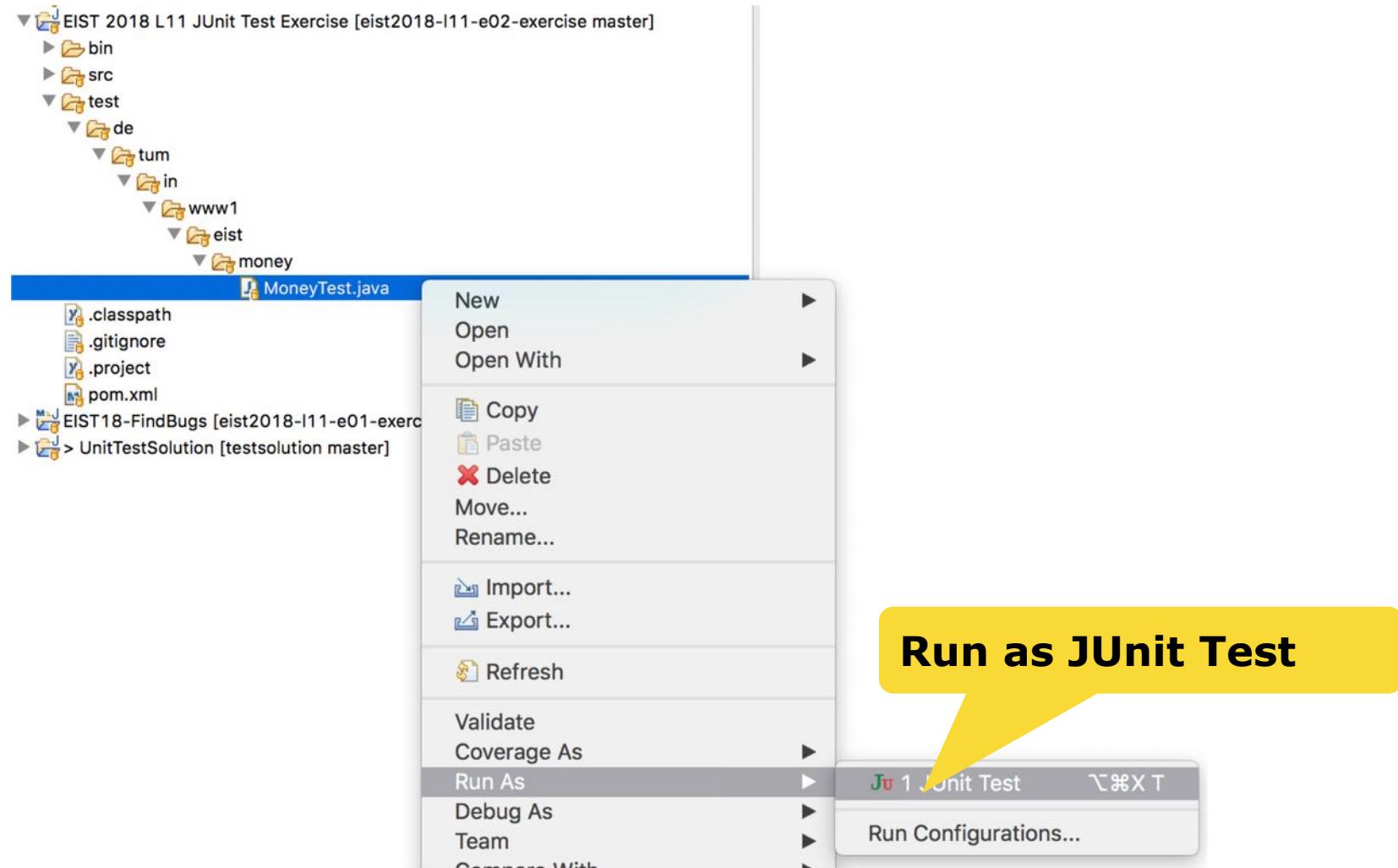
1. Clone the exercise repository from
<https://repobruegge.in.tum.de/projects/EIST2018L11E02/repos/eist2018-l11-e02-exercise/>
2. Run the JUnit test
3. Follow the demo
4. Complete the implementation of the Money class
5. Complete the test cases

Task 1: Clone the exercise repository from

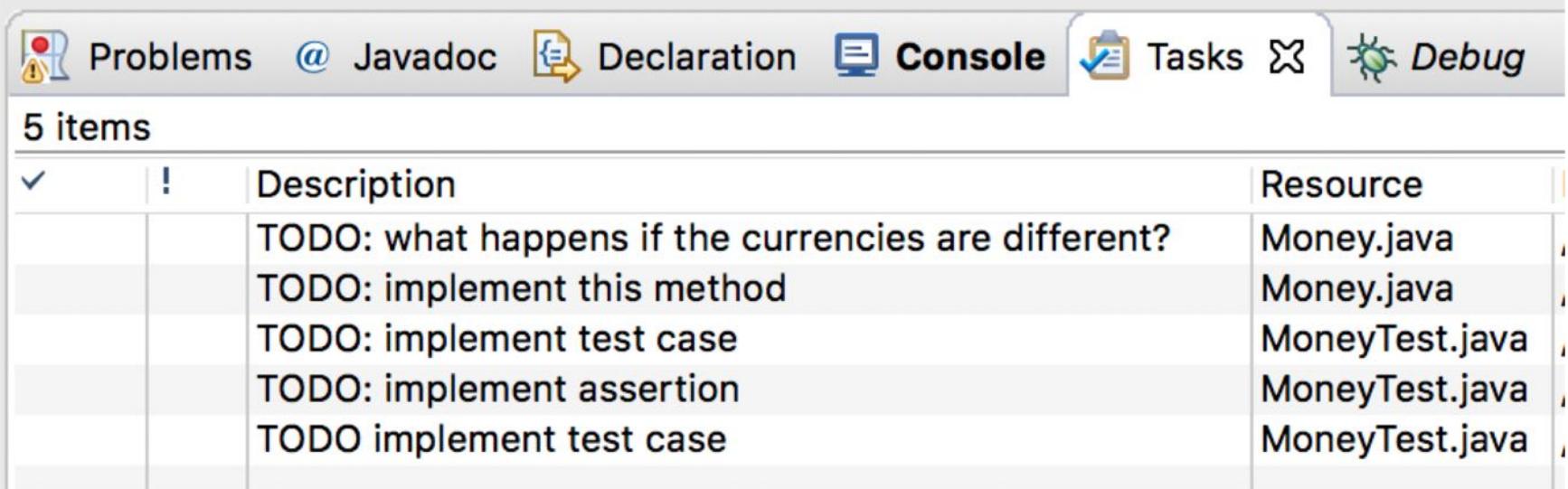
- Go to <https://repobruegge.in.tum.de/projects/EIST2018L11E02/repos/eist2018-l11-e02-exercise/>



Task 2: Run the Junit Test



Task 3: Follow the Demo



The screenshot shows the Eclipse IDE's Tasks view. The title bar includes tabs for Problems, Javadoc, Declaration, Console, Tasks, and Debug. The Tasks tab is selected, indicated by a blue border and bold text. Below the tabs, the message "5 items" is displayed. A table lists five TODO items:

✓	!	Description	Resource
		TODO: what happens if the currencies are different?	Money.java
		TODO: implement this method	Money.java
		TODO: implement test case	MoneyTest.java
		TODO: implement assertion	MoneyTest.java
		TODO implement test case	MoneyTest.java

Task 4: Complete the implementation of the Money class

```
public class Money {  
    ...  
    public Money add(Money money) {  
        if(currency != money.getCurrency()) {  
            throw new IllegalArgumentException("Different currencies not  
                supported!");  
        }  
        return new Money(amount() + money.amount(), getCurrency());  
    }  
}
```

What happens if the currencies are different?

Task 4: Complete the implementation of the Money class (ctd.)

```
public class Money {  
    ...  
    public Money add(Money money) {  
        if(currency != money.getCurrency()) {  
            throw new IllegalArgumentException("Different currencies not  
                supported!");  
        }  
        return new Money(amount() + money.amount(), getCurrency());  
    }  
    public Money subtract(Money money) {  
        if(currency != money.getCurrency()) {  
            throw new IllegalArgumentException("Different currencies not  
                supported!");  
        }  
        return new Money(amount() - money.amount(), getCurrency());  
    }  
}
```

What happens if the currencies are different?

Task 4: Complete the implementation of the Money class (ctd.)

```
public class Money {  
    ...  
    public Money add(Money money) {  
        ...  
    }  
    public Money subtract(Money money) {  
        ...  
    }  
    public boolean equals(Object object) {  
        if (object instanceof Money) {  
            Money money = (Money) object;  
            return money.getCurrency().equals(currency) &&  
                   money.amount() == amount;  
        }  
        return false;  
    }  
}
```

Task 5: Complete the test cases

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;  
  
public class MoneyTest {  
    @Test  
    public void simpleAdd() {  
        Money m12CHF = new Money(12, Currency.CHF);  
        Money m14CHF = new Money(14, Currency.CHF);  
        Money expected = new Money(26, Currency.CHF);  
        Money observed = m12CHF.add(m14CHF);  
        assertEquals(expected, observed);  
    }  
}
```

Check if the expected amount is the same as the observed amount

Task 5: Complete the test cases

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MoneyTest {
    @Test
    public void simpleAdd() {
        Money m12CHF = new Money(12, Currency.CHF);
        Money m14CHF = new Money(14, Currency.CHF);
        Money expected = new Money(26, Currency.CHF);
        Money observed = m12CHF.add(m14CHF);
        assertEquals(expected, observed);
    }
    @Test
    public void simpleSubtract() {
        Money m14CHF = new Money(14, Currency.CHF);
        Money m12CHF = new Money(12, Currency.CHF);
        Money expected = new Money(2, Currency.CHF);
        Money observed = m14CHF.subtract(m12CHF);
        assertEquals(expected, observed);
    }
}
```

Check if the expected amount is the same as the observed amount

Task 5: Complete the test cases

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MoneyTest {
    @Test
    public void simpleAdd() {
        Money m12CHF = new Money(12, Currency.CHF);
        Money m14CHF = new Money(14, Currency.CHF);
        Money expected = new Money(26, Currency.CHF);
        Money observed = m12CHF.add(m14CHF);
        assertEquals(expected, observed);
    }
    @Test
    public void simpleSubtract() {
        Money m14CHF = new Money(14, Currency.CHF);
        Money m12CHF = new Money(12, Currency.CHF);
        Money expected = new Money(2, Currency.CHF);
        Money observed = m14CHF.subtract(m12CHF);
        assertEquals(expected, observed);
    }
    @Test(expected = IllegalArgumentException.class)
    public void invalidAdd() {
        Money m12EUR = new Money(12, Currency.EUR);
        Money m14USD = new Money(14, Currency.USD);
        m12EUR.add(m14USD);
    }
}
```

Check if the expected amount is the same as the observed amount

If the currencies are not the same, throw an Exception

Types of Testing Activities

- **Unit Testing**
 - Individual components (class or subsystem) are tested
 - Carried out by developers
 - Goal: Confirm that the component is correctly coded and carries out the intended functionality
- **System Testing**
 - The entire system is tested in the development environment
 - Carried out by developers
 - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- **Acceptance Testing**
 - The system is tested in the target environment
 - Carried out by the client. May involve executing typical transactions on site on a trial basis
 - Goal: Demonstrate that the system meets the requirements and is ready to use.



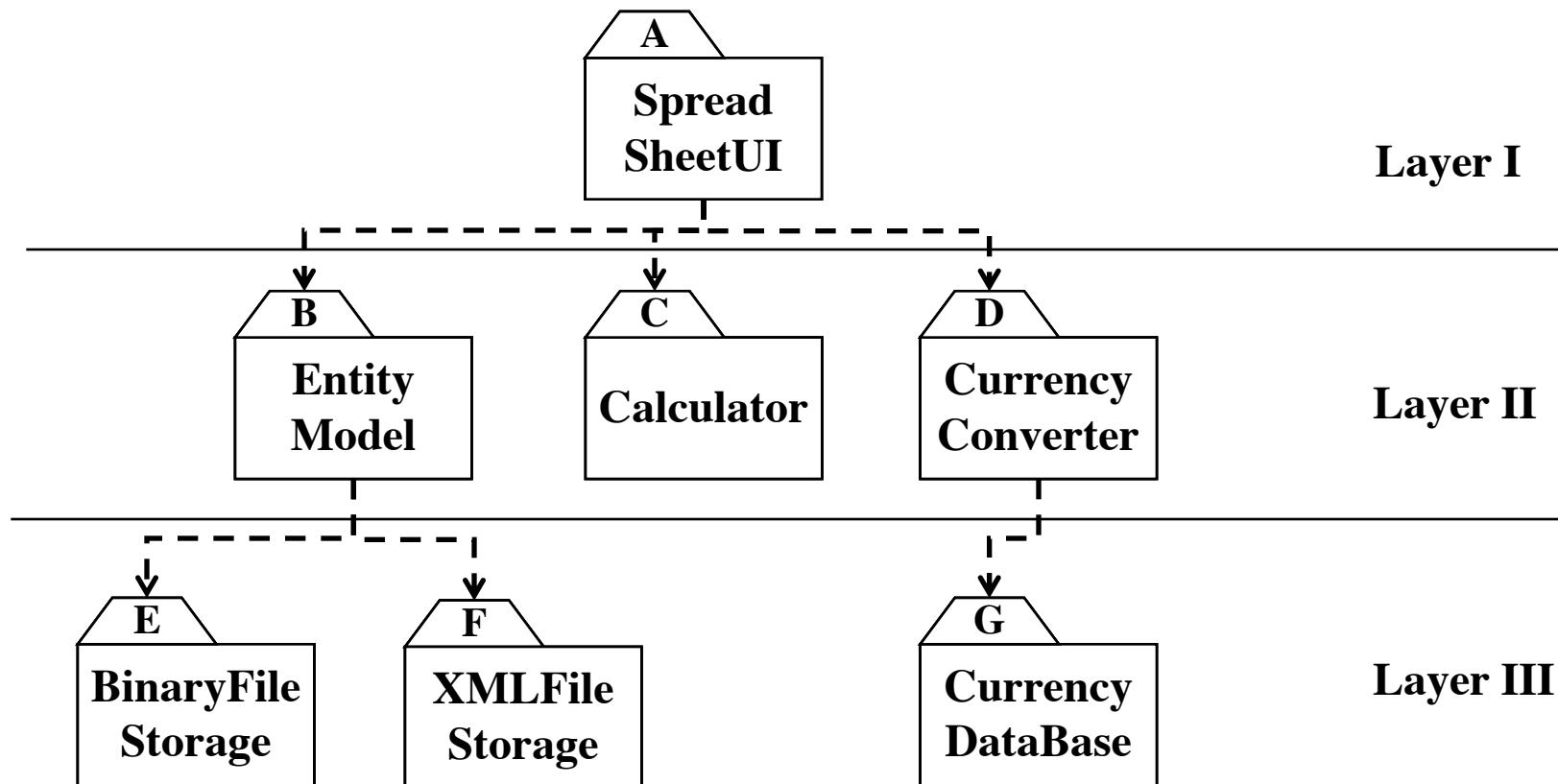
Integration Testing

- Groups of subsystems are tested
- Carried out by developers
- Goal: Test the subsystem interfaces among the subsystems.

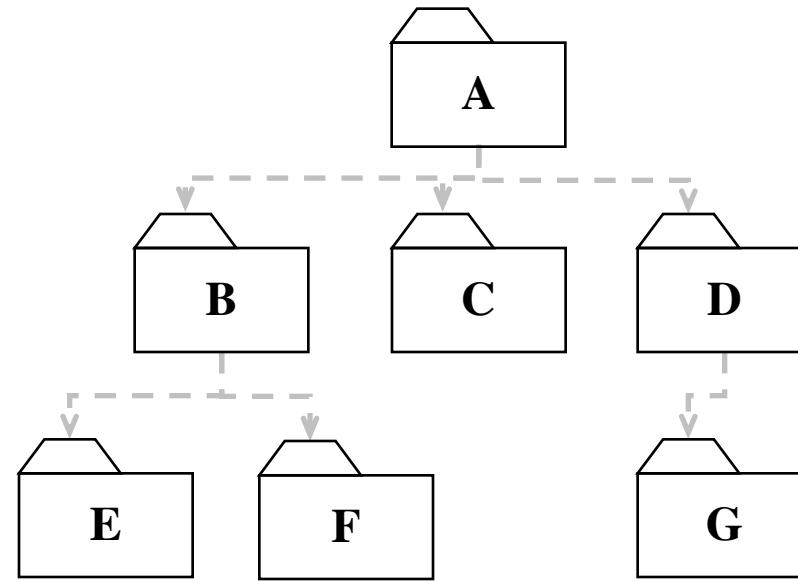
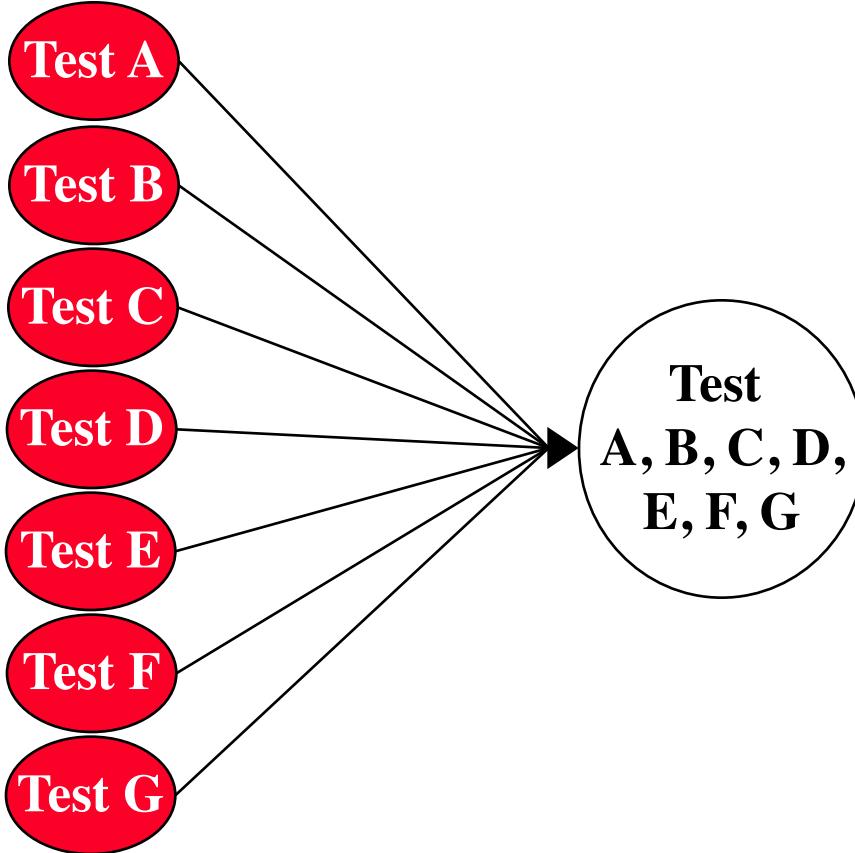
Integration Testing

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The **integration testing strategy** determines the order in which the subsystems are selected for testing and integration
 - Big Bang integration
 - Bottom Up testing
 - Top Down Testing
 - Vertical Integration.

Example: Integration Testing for a 3-Layer-Design



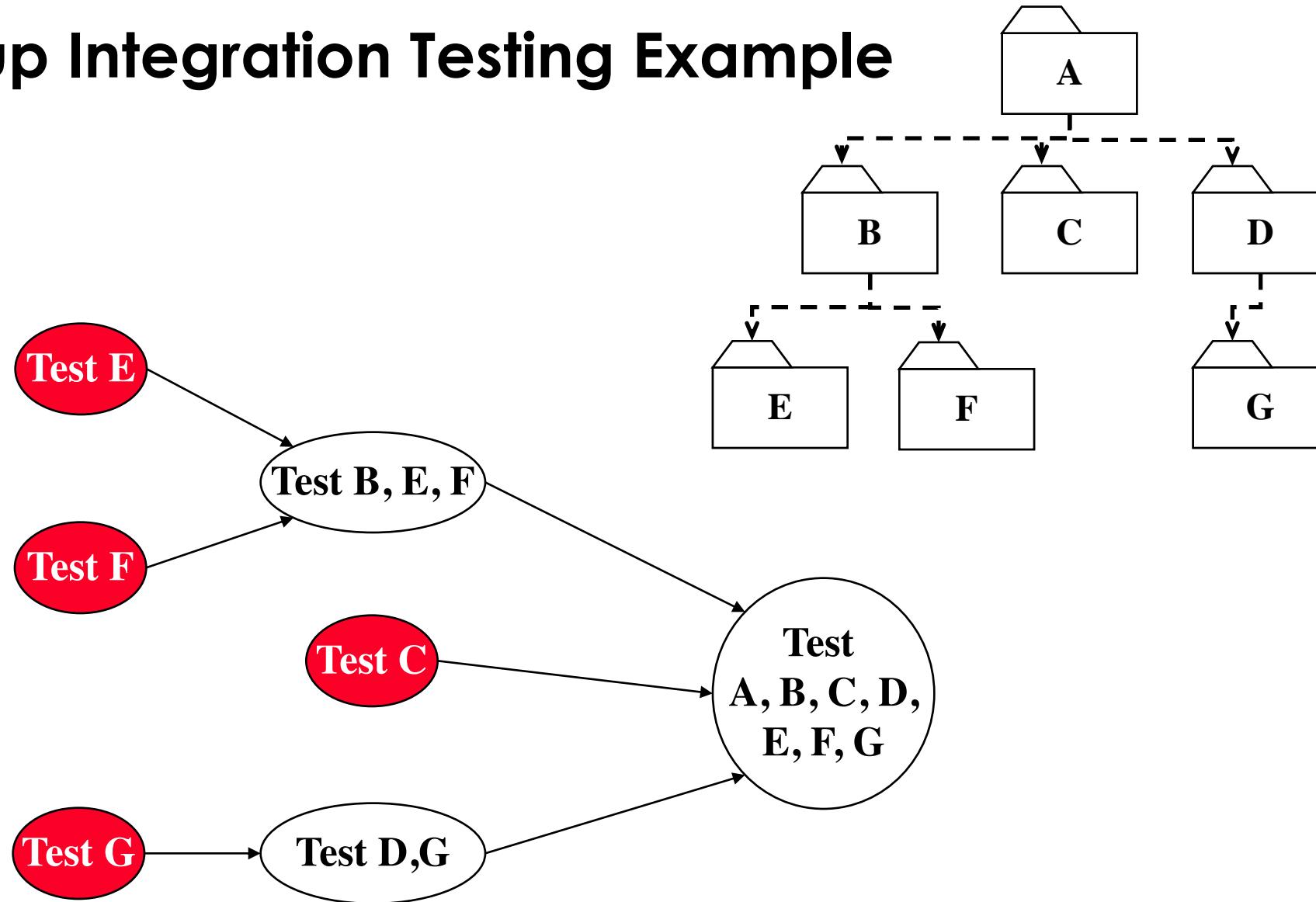
Big-Bang Approach



Bottom-up Testing Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the subsystems above this layer are tested that call the previously tested subsystems
- This is repeated until all subsystems are included.

Bottom-up Integration Testing Example



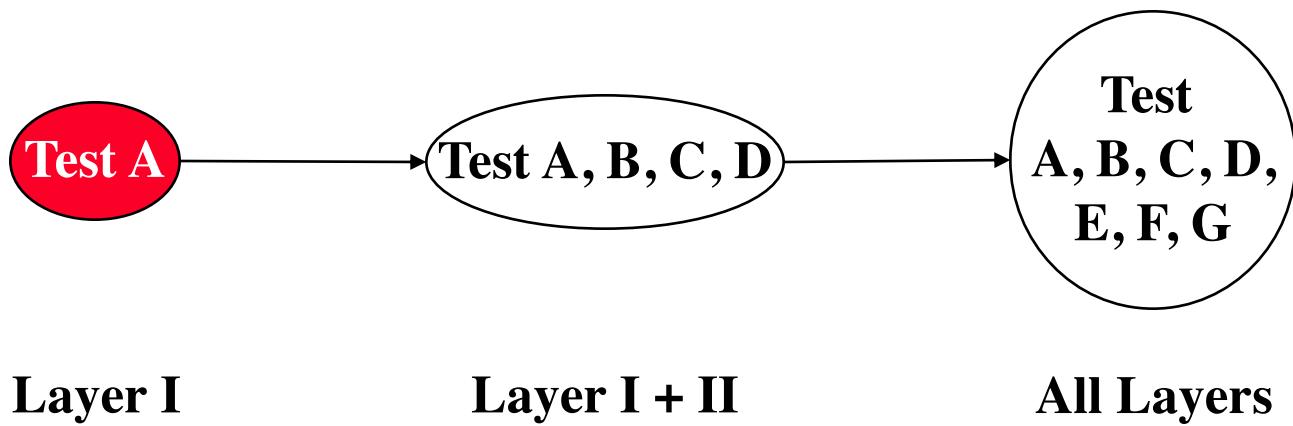
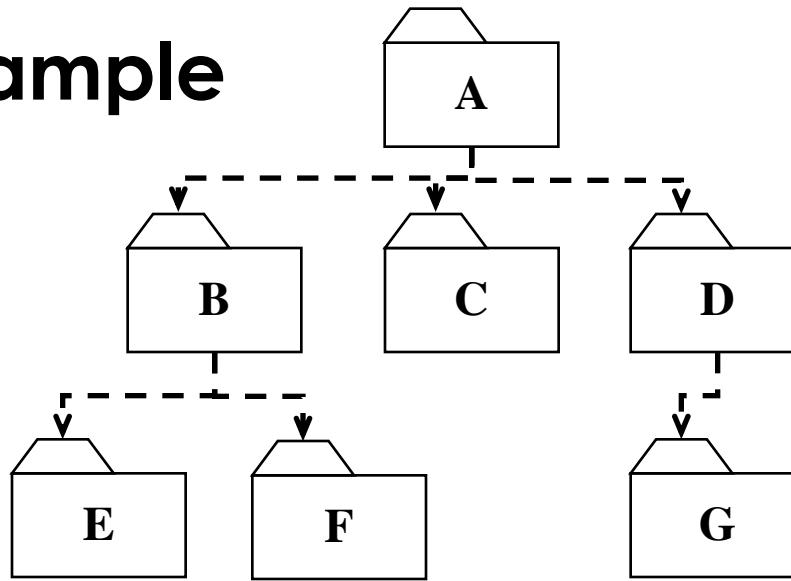
Pros and Cons: Bottom-Up Integration Testing

- Pros
 - No doubles needed
 - Useful for integration testing of the following systems
 - Object-oriented systems
 - Systems with strict performance requirements, e.g. real-time systems
- Cons:
 - Tests an important subsystem (the user interface) last
 - Test drivers are needed.

Top-down Testing Strategy

- Test the subsystems in the top layer first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the tests.

Top-down Integration Testing Example



Pros and Cons: Top-Down Integration Testing

Pros:

- Test cases can be defined in terms of the functional requirements of the system
- No test drivers needed

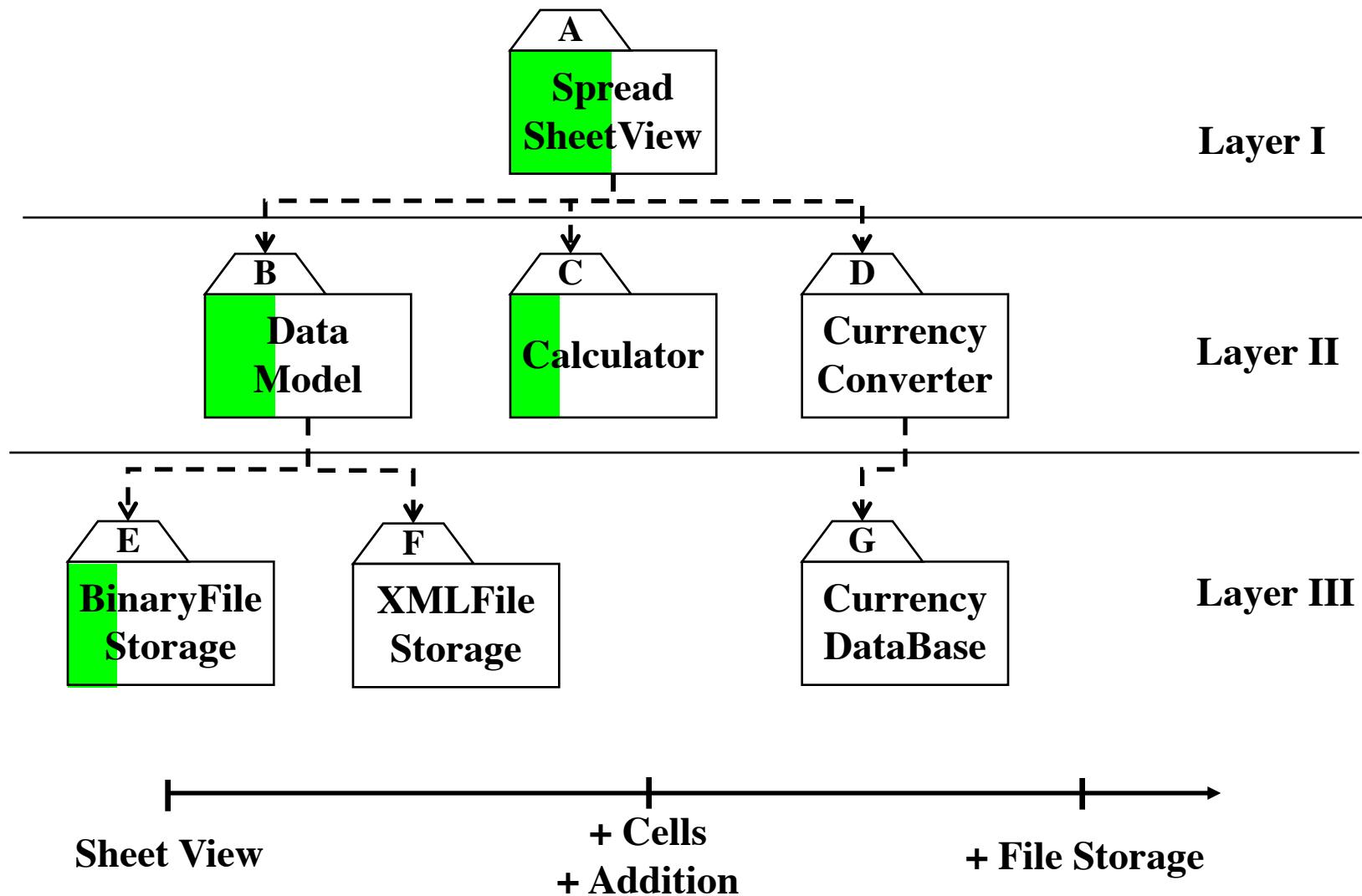
Cons:

- Doubles are needed
- Writing doubles is difficult: Doubles must allow all possible conditions to be tested
- Large number of doubles may be required, especially if the lowest level of the system contains many methods
- Some interfaces are not tested separately
- One solution to avoid too many doubles: *Modified top-down testing strategy*
 - Test each layer of the system decomposition individually before merging the layers
 - Disadvantage of modified top-down testing: Both, doubles and test drivers are needed

Horizontal Integration Testing Risks

- Risk #1: The higher the complexity of the software system, the more difficult is the integration of its components
- Risk #2: The later integration occurs in a project, the bigger is the risk that unexpected failures occur
- Horizontal integration strategies (Bottom up, top down, sandwich testing) don't do well with risk #2
- **Vertical integration** addresses these risks by building as early and frequently as possible
 - Used in Scenario-driven design: Scenarios are used to drive the integration
 - Used in Scrum: User stories are used to drive the integration. Potential deliverable product increment
- Advantages of vertical integration:
 - There is always an executable version of the system
 - All the team members have a good overview of the project status.

Vertical Integration Testing



Horizontal vs Vertical Integration Testing

Test Cases	Requirements				
	R1	R2	R3	R4	
Bottom-up integration tests					User Interface
Top-down integration tests					Middleware
Sandwich integration tests ¹					Database
Vertical integration tests					

¹Not covered in today's class. Find the definition in the textbook

Outline of the Lectures on Testing

- ✓ Terminology
 - ✓ Failure, Error, Fault
- ✓ Test Model
- ✓ Model-based testing
- ✓ Object-Oriented testing
- ✓ Testing activities
 - ✓ Static Analysis
 - ✓ Unit testing
 - ✓ Integration testing
- ➡ System Testing
 - Function testing
 - Performance testing
 - Acceptance Testing.

System Testing

- Functional Testing
 - Validates functional requirements
- Structure Testing
 - Validates the subsystem decomposition
- Performance Testing
 - Validates non-functional requirements
- Acceptance Testing
 - Validates clients expectations

Impact of requirements on system testing

- The more explicit the requirements, the easier they are to test
- Quality of use cases determines the ease of functional testing and acceptance testing
- Quality of subsystem decomposition determines the ease of structure testing
- Quality of nonfunctional requirements and constraints determines the ease of performance tests.

Functional Testing

Essentially the same as black box testing

Goal: Test functionality of system

- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box
- Unit test cases can be reused, but new test cases have to be developed as well.

Structure Testing

Essentially the same as white box testing

Goal: Cover all paths in the system design

- Exercise all input and output parameters of each component.
- Exercise all components and all calls (each component is called at least once and every component is called by all possible callers.)

Performance Testing

Goal: Try to violate non-functional requirements

- Test how the system behaves when overloaded
 - Can bottlenecks be identified?
 - If yes, these are the first candidates for redesign in the next iteration
- Try unusual orders of execution
 - Call a receive() before send()
- Check the system's response to large volumes of data
 - If the system can handle 1000 items, try it with 1001 items
- What is the amount of time spent in different use cases?
 - Are typical cases executed in a timely fashion?

Acceptance Testing

Goal: Demonstrate system is ready for operational use

- Choice of tests is made by client/sponsor
- Many tests can be taken from integration testing
- Acceptance test is performed by the client, not by the developer.
- Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:

- ***Alpha test:***

- Sponsor uses the software at the *developer's site*.
- Software used in a controlled setting, with the developer always ready to fix bugs

- ***Beta test:***

- Conducted at *sponsor's site* (developer is not present)
- Software gets a realistic workout in target environment
- Potential customer might get discouraged
- Software is used in an uncontrolled setting.

Types of Performance Testing

- Stress Testing
 - Stress limits of system
- Volume testing
 - Test what happens if large amounts of data are handled
- Configuration testing
 - Test the various software and hardware configurations
- Compatibility test
 - Test backward compatibility with existing systems
- Timing testing
 - Evaluate response times and time to perform a function
- Security testing
 - Try to violate security requirements
- Environmental test
 - Test tolerances for heat, humidity, motion
- Quality testing
 - Test reliability, maintainability & availability
- Recovery testing
 - Test system's response to presence of errors or loss of data
- Human factors testing
 - Test with end users.

Summary

- Testing is still a black art, but many rules and heuristics are available
- Object-Oriented Testing
- Model Based Testing
- Testing Activities
 - Unit testing
 - Integration testing
 - System testing
 - Acceptance testing
- Testing has its own lifecycle



<http://www.youtube.com/watch?v=bzBkSDb07iA&feature=related>

Readings

Wikipedia Unit Testing https://en.wikipedia.org/wiki/Unit_testing

Kent Beck, Erich Gamma, *Junit Cookbook*

<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>

JUnit Source Forge: <http://sourceforge.net/projects/junit/files/junit/>

Latest JUnit Source on github: <https://github.com/kentbeck/junit/wiki>

Junit Fixture: <http://www.informit.com/articles/article.aspx?p=101374&seqNum=5>

Martin Fowler, Mocks are not Stubs: <http://martinfowler.com/articles/mocksArentStubs.html>

Brown & Tapolcsanyi: Mock Object Patterns. In Proceedings of the 10th Conference on Pattern Languages of Programs, 2003. <http://hillside.net/plop/plop2003/papers.html>

Herman Bruyninckx, Embedded Control Systems Design, WikiBook, Learning from Failure:
http://en.wikibooks.org/wiki/Embedded_Control_Systems_Design/Learning_from_failure

Joanne Lim, An Engineering Disaster: Therac-25

- <http://www.bowdoin.edu/~allen/courses/cs260/readings/therac.pdf>
- Peter G. Neumann, Computer-Related Risks, Addison-Wesley, ACM Press, 384 pages, 1995.

Morning Quiz 12

- Start Time: **8:00**
- End Time: **8:10**
- To participate in the quiz, use ArTEMiS
- Click on Open Quiz or Start Quiz
- The Lecture starts at 8:10

Introduction to Software Engineering

Exercise	Due date	Actions
Show 2 overdue exercises		
EIST 2018 Lecture 02 Bumpers Sprint 03	in 4 days	Start exercise
Good Morning Quiz 12		Open Quiz

Remaining Time: **46 s**

Saved: never

● Connected

Submit

**Only click on Submit when
you have entered all answers!**

A photograph of a massive, light-colored rock formation, possibly limestone, with several vertical fissures. Two climbers are visible on the slope: one in a red jacket near the bottom left and another in a blue jacket further up towards the top right. A red rope extends from the climber in red towards the climber in blue.

Project Communication and Management

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

12 July 2018

Roadmap for Today's Lecture

- **Context and Assumptions**

- We have completed Chapter 1-11, 13 and 15 in the text book

- **Content of this lecture: 2 Chapters**

- Chapter 3 Project Organization and Communication
 - Chapter 14 Project Management

- **Objective:** At the end of the lecture you understand

- The difference between communication events and communication mechanisms
 - Communication activities to start a project
 - The difference between responsibility, authority, accountability and delegation
 - States of a software project
 - Project organization forms.

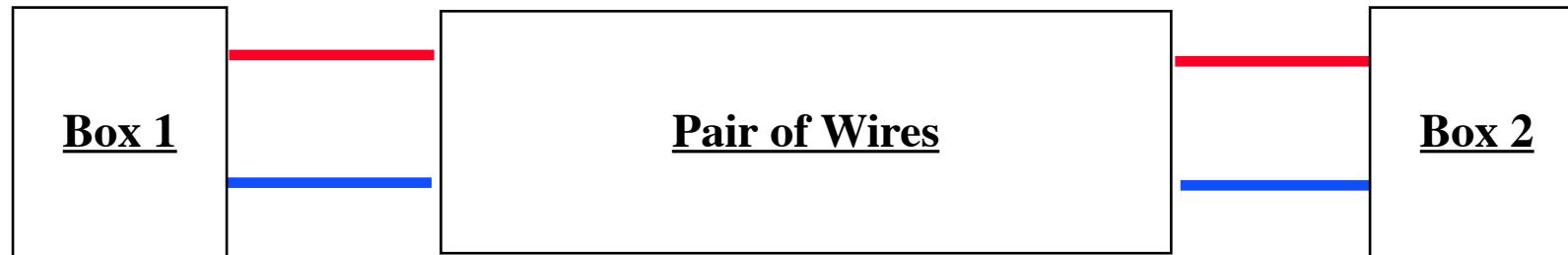
Outline of Today's Lecture

- Communication events
- Communication mechanisms
- Communication activities
- Project definition
- Typical project management issues
- Organization forms
- Mapping roles to people in organizations
- Tasks & Activities, Work Products & Deliverables
- UML Model of a Project
- Communication Structure

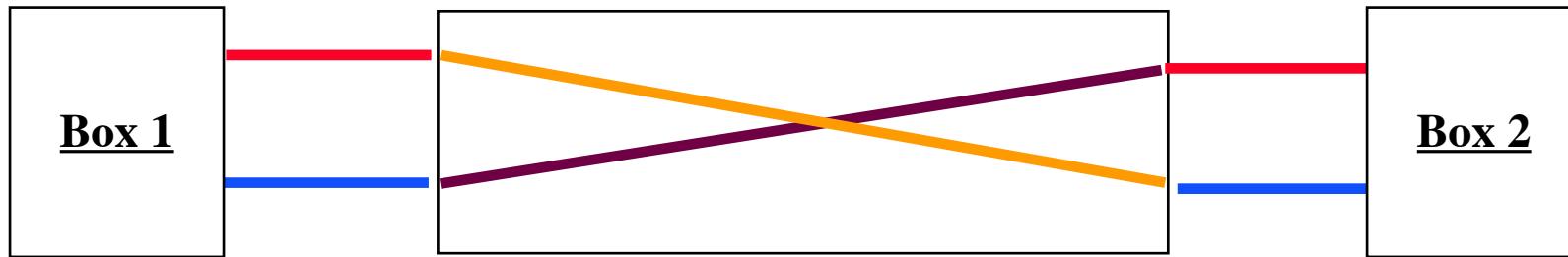
Why is Communication Important?

From an Airplane Crash report:

"Two missile electrical boxes manufactured by different contractors were joined together by a pair of wires."



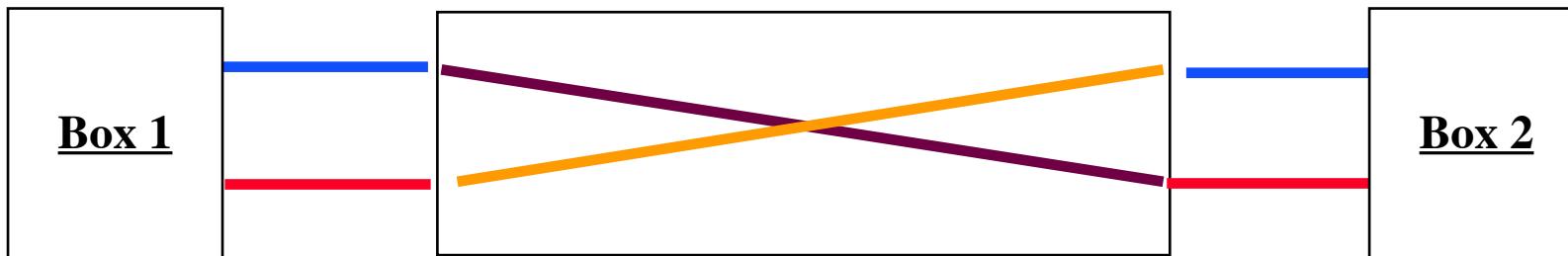
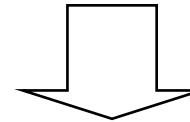
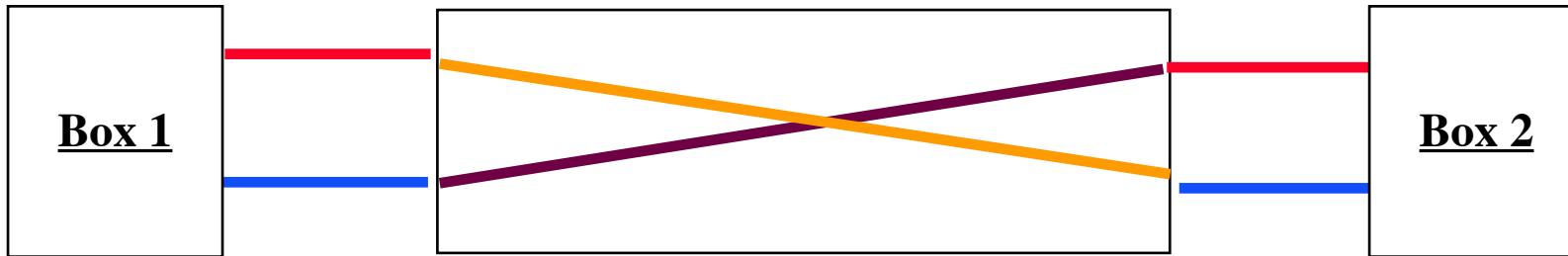
Thanks to a particular thorough preflight check, it was discovered that the wires had been reversed."



...

"The *postflight analysis after the crash* revealed that the contractors had indeed corrected the reversed wires as instructed."

"In fact, both of them had."



Communication is critical

- In large system development efforts, you will spend more time communicating than coding
- A software engineer needs to learn the so-called soft skills:
 - **Collaboration**
 - Negotiate requirements with the client and with members from your team and other teams
 - **Presentation**
 - Present a major part of the system during a review
 - **Management**
 - Facilitate a team meeting
 - **Technical writing**
 - Model the system, participate in the project documentation.

Communication Event vs. Mechanism

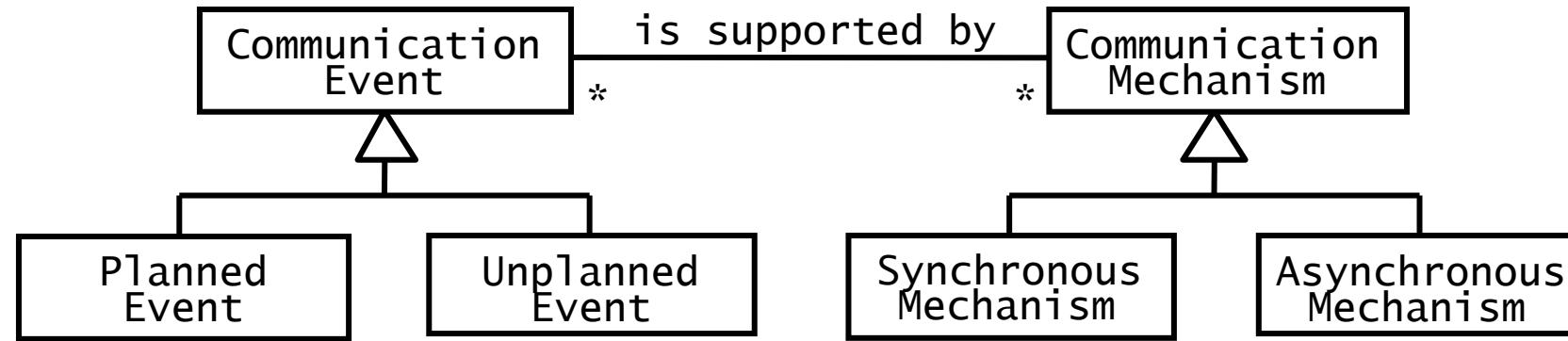
Communication event

- Information exchange with defined objectives and scope
- *Scheduled*: Planned communication
 - Examples: weekly team meeting, review
- *Unscheduled*: Event-driven communication
 - Examples: problem report, request for change, clarification

Communication mechanism

- Tool or procedure that can be used to transmit information
- *Synchronous*: Sender and receiver are communicating at the same time
- *Asynchronous*: Sender and receiver are not communicating at the same time.

Modeling Communication



Planned Communication Events

Problem Definition

- Objective: Present goals, requirements and constraints
- Example: Client presentation
- Usually scheduled at the beginning of a project

Project Review: Focus on system models

- Objective: Assess status and review the system model
- Examples: Analysis review, system design review
- Scheduled around project milestones and deliverables

Client Review: Focus on requirements

- Objective: Brief the client, agree on requirements changes
- The first client review is usually scheduled after analysis phase.

Planned Communication Events (2)

Walkthrough (Informal)

- Objective: Increase quality of subsystem
- Example
 - Developer informally presents subsystem to team member (“peer-to-peer”)
- Scheduled by each team

Inspection (Formal)

- Objective: Compliance with requirements
- Example
 - Demonstration of final system to customer (Client acceptance test)
- Scheduled by project management

Unplanned Communication Events

Request for change

- A participant reports a problem and proposes a solution
- Change requests are often formalized in large projects
- Example: Request for additional functionality

Report number: 1291 **Date:** 5/3 **Author:** Dave

Synopsis: The STARS form should have a galaxy field.

Subsystem: Universe classification

Version: 3.4.1

Classification: missing functionality

Severity: severe

Proposed solution: ...

Unplanned Communication Event (2)

Issue resolution

- Selects a single solution to a problem for which several solutions have been proposed
- Uses issue tracker to collect problems and proposals.

The screenshot shows a Jira interface. On the left, there is a sidebar with icons for file, folder, search, and refresh. Below it is a list of issues:

- Order by Priority ↓ | ▾
- [IOS1415ALLIANZ-43](#) Add sensors to the system
- [IOS1415ALLIANZ-47](#) Burglary detection
- [IOS1415ALLIANZ-70](#) [F] Notify user of succesful calibration
- [IOS1415ALLIANZ-49](#) Enduser activating Calibration mode
- [IOS1415ALLIANZ-68](#) [F] Activate Calibration Mode
- [IOS1415ALLIANZ-52](#) [F] Notify Enduser of new sensor
- [IOS1415ALLIANZ-464](#) Review Class diagram SSD
- [IOS1415ALLIANZ-294](#) Everyone: Think about the demo scenario
- [IOS1415ALLIANZ-161](#) Bundle on openHAB that triggers the Apple ...

On the right, a detailed view of the first issue is shown:

iOS1415 Allianz / IOS1415ALLIANZ-43
Add sensors to the system

Edit **Comment** **Assign** **More** **Start Progress** **Close** **Admin**

Details

Type:	User Story	Status:	OPEN (View Workflow)
Priority:	Critical	Resolution:	Unresolved
Difficulty:	L		
User Story Role:	User		
User Story Feature:	add sensors to the system		
User Story Reason:	the system can detect a new sensor and inform the user about it		

Description
Click to add description

Attachments
Drop files to attach, or browse.

Synchronous Communication Mechanisms

- Smoke signals
- Hallway conversation
 - Supports: Unplanned conversations, request for clarification, request for change
 - + Cheap and effective for resolving simple problems
 - Information loss, misunderstandings are frequent
- Meeting (face-to-face, phone, video conference)
 - Supports: Planned conversations, client review, project review, status review, brainstorming, issue resolution
 - + Effective for issue resolution and consensus building
 - High cost (people, resources), low bandwidth.

Asynchronous Communication Mechanisms

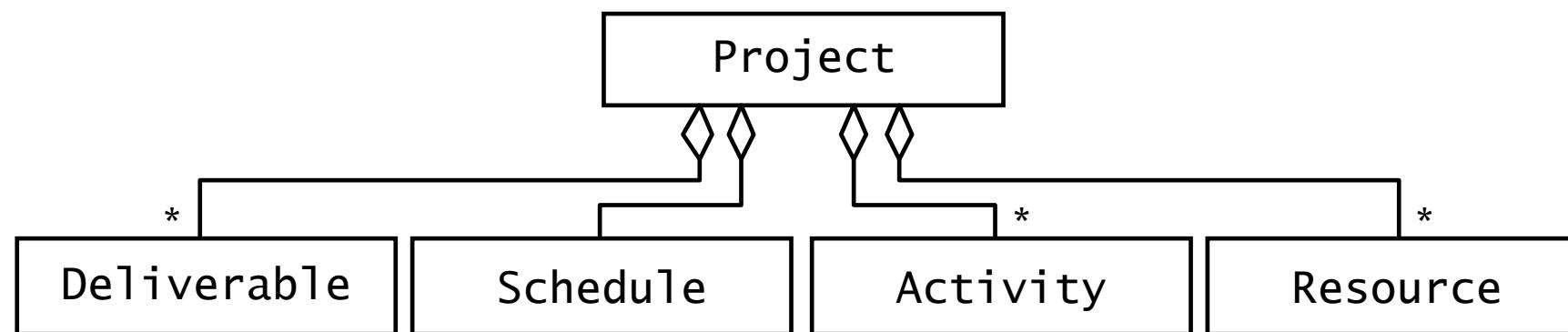
- E-Mail
 - Supports: Release, change request
 - + Ideal for planned communication and announcements
 - E-mail out of context can be misunderstood, sent to the wrong person, or lost
- Newsgroup (Chat tools: Slack, Hipchat, ...)
 - Supports: Release, change request, brainstorming
 - + Suited for discussion among people who share a common interest; cheap (shareware available)
 - Primitive access control (often, you are either in or out)
- Portal (Portal tools: Wikis, Confluence,...)
 - Supports: Release, change request, inspections
 - + Uses the hypertext metaphor: Documents links to other documents.
 - Does not easily support rapidly evolving documents.

Outline of Today's Lecture

- ✓ Communication events
- ✓ Communication mechanisms
- ✓ Communication activities
- Project definition
 - Typical project management issues
 - Organization forms
 - Mapping roles to people in organizations
 - Tasks & Activities, Work Products & Deliverables
 - UML Model of a Project
 - Communication Structure

Project Definition

- A **project** is an undertaking, limited in time, to achieve a set of goals that require a concerted effort
- **A project includes**
 - A set of deliverables to a client
 - A schedule
 - Technical and managerial activities to produce and deliver the deliverables
 - Resources consumed by the activities (people, budget)

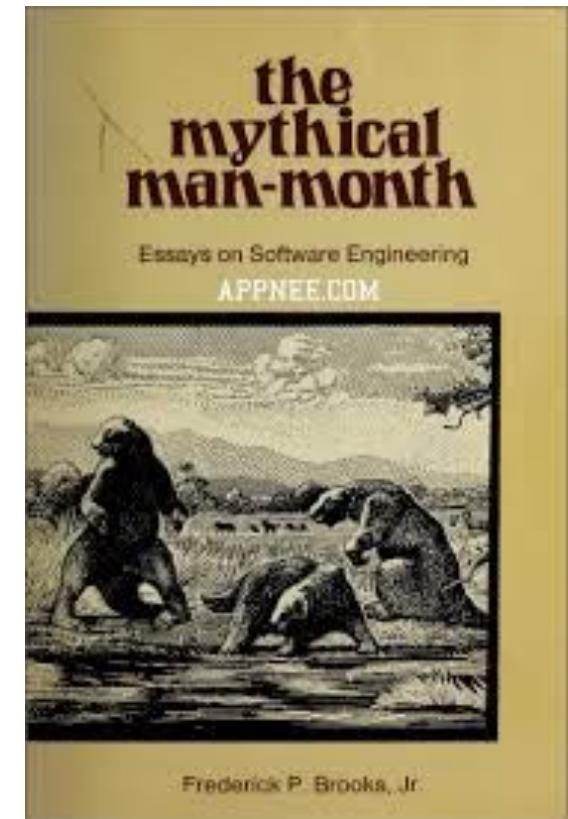


Laws of Project Management

- Projects progress quickly until they are 90% complete
 - Then they remain at 90% complete forever
- If project content is allowed to change freely, the rate of change will exceed the rate of progress
- Project teams detest progress reporting because it manifests their lack of progress
- Murphy's law:
 - "When things are going well, something will go wrong"
 - "When things just can't get worse, they will"
 - "When things appear to be going better, you have overlooked something."

[wikipedia.org/wiki/Murphy's law](https://en.wikipedia.org/wiki/Murphy%27s_law)

- Adding manpower to a late software project makes it later
-- Frederick Brooks



Typical Project Management Issues

- How should the project be organized?
- Who should be part of it?
- Who should do what?
- How do we break down the overall work to be done?
- How do we schedule the work?
- What are the deliverables?

Outline of Today's Lecture

- ✓ Communication events
- ✓ Communication mechanisms
- ✓ Communication activities
- ✓ Project definition
- ✓ Typical project management issues
- Organization forms
 - Mapping roles to people in organizations
 - Tasks & Activities, Work Products & Deliverables
 - UML Model of a Project
 - Communication Structure

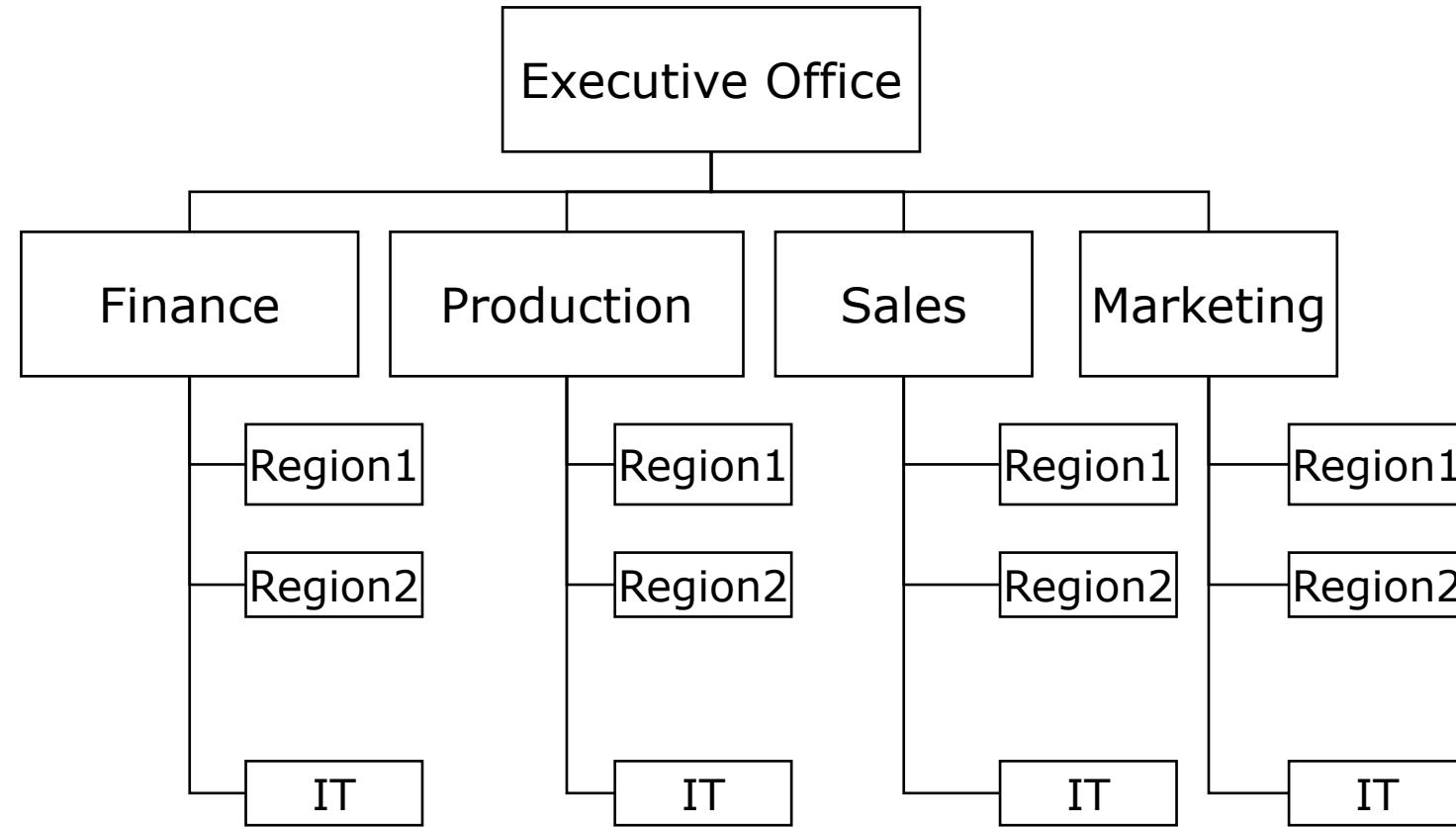
Project Organization

- A **project organization** defines the relationships among resources, in particular the participants, in a project
- 3 Types
 - Functional Organization
 - Project-based Organization
 - Matrix Organization

Functional Organization

- In a **functional organization** people are grouped into departments, each addressing an activity ("function")
- Examples of departments
 - Traditional company: Research, development, production, sales, marketing, finance.
 - Software company: Analysis, design, integration, implementation, testing, delivery, maintenance
- Properties of functional organizations
 - Projects are pipelined through the departments.
 - Example: The project starts in research, moves to development, then moves to production
 - Different departments often have identical needs
 - Example: Communication, regular backups, configuration management support, IT infrastructure.

Example of a Functional Organization



Line organization of a „traditional business“

Properties of Functional Organizations

- **Advantages:**
 - Members of a department have a good understanding of the functional area they support
- **Disadvantages:**
 - It is difficult to make major investments in equipment and facilities
 - There is a high chance of overlap or duplication of work among departments.

Project-based Organization

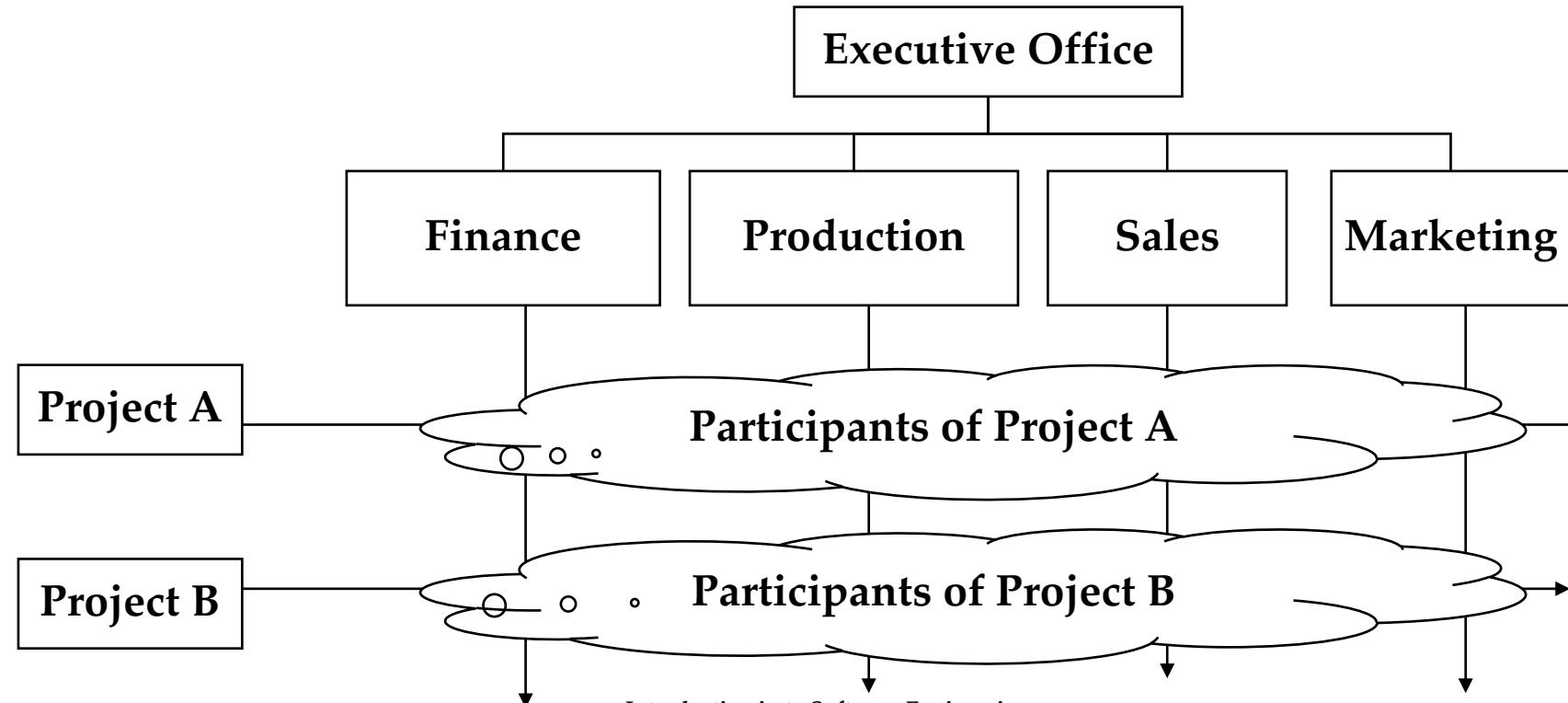
- In a **project-based organization** people are assigned a project, addressing a problem to be solved within a specified time and budget
- Key properties of project-based organizations
 - Teams are assembled for a project as it is created
 - Each project has its own leader (project manager)
 - Teams are disassembled when the project terminates.

Properties of Project-based Organizations

- **Advantages**
 - Very responsive to new requirements (because the project is newly established and can be tailored specifically to address the problem)
 - New people can be hired who are familiar with the problem or who have special capabilities
- **Disadvantages**
 - Teams cannot be assembled rapidly. Often it is difficult to manage the staffing/hiring process
 - Because there are „no predefined lines“ (as in functional organizations), roles and responsibilities need to be defined at the beginning of the project.

Matrix Organization

- In a **matrix organization**, people from different departments of a functional organization are assigned to work on one or more projects
- Participants are usually assigned to a project < 100 % of their time.



Properties of Matrix Organizations

- **Advantages**
 - Teams for projects can be assembled rapidly from the existing line organization
 - Rare expertise can be applied to different projects as needed
 - Consistent reporting and decision procedures can be used for projects of the same type
- **Disadvantages**
 - Participants are usually not familiar with each other
 - Participants have different working styles
 - Participants must get used to each other.

Challenges in Matrix Organizations

- Team members work on multiple projects which have competing demands for their time
- Team members work for **two bosses** with different focus:
 - Focus of the functional manager:
 - Assignments to different projects, performance appraisal
 - Focus of the project manager:
 - Work assignments to team members, support of the project team
- Multiple work procedures and reporting systems are used by different team members.

When to use a Functional Organization

- Projects with high degree of certainty, stability, uniformity and repetition
 - Requires little communication
 - Role definitions are clear
- The more people on a project, the more the need for a formal structure

When to use a Project-based Organization

- The project has high degree of uncertainty
 - Open communication needed among members
 - Roles are defined on project basis
 - Requirements change during development
 - New technology appears during project.

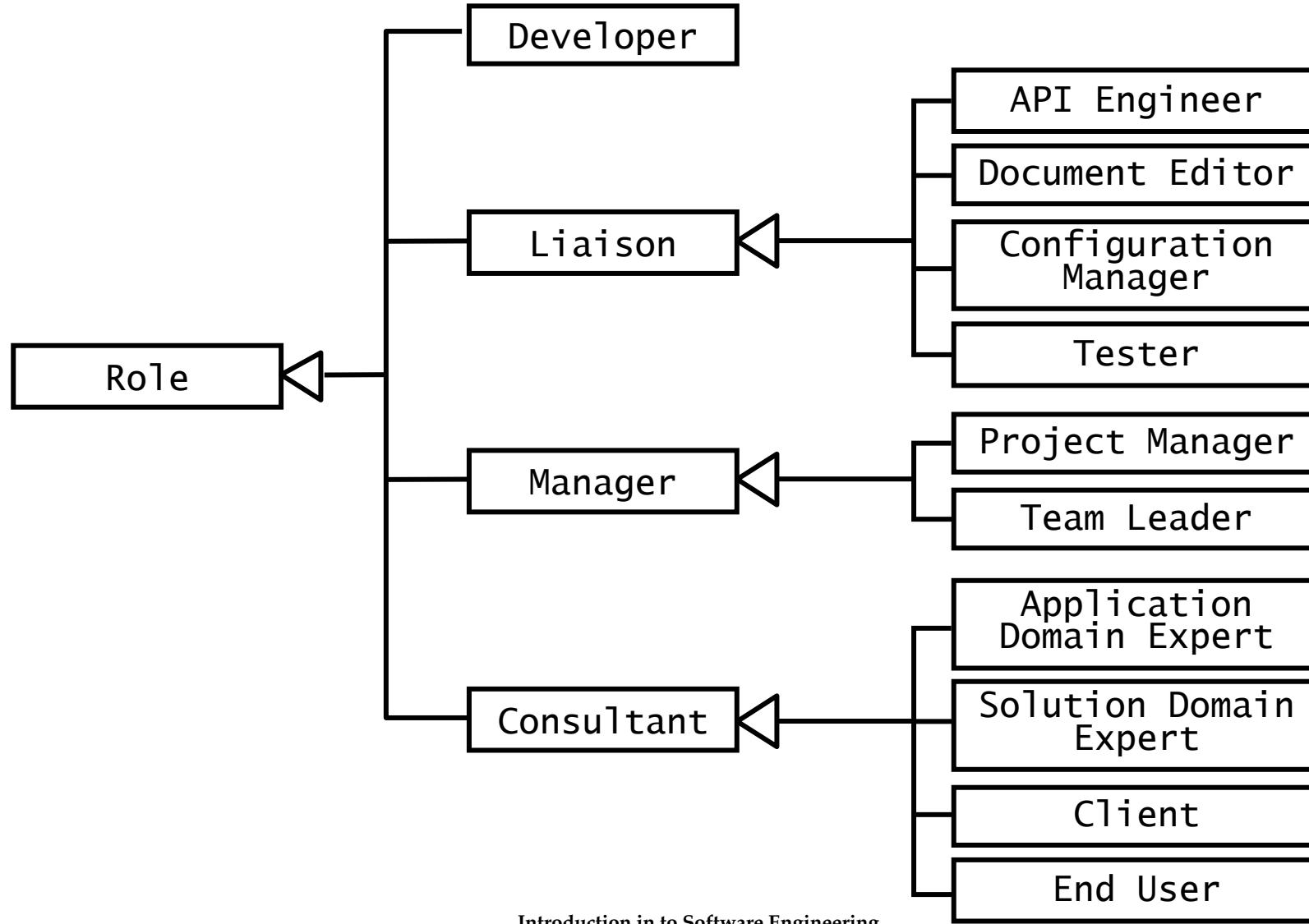
Outline of Today's Lecture

- ✓ Communication events
- ✓ Communication mechanisms
- ✓ Communication activities
- ✓ Project definition
- ✓ Typical project management issues
- ✓ Organization forms
- Mapping roles to people in organizations
 - Tasks & Activities, Work Products & Deliverables
 - UML Model of a Project
 - Communication Structure

Role

- A **role** defines a set **responsibilities** ("to-dos")
- **Role: Tester**
 - Responsibilities:
 - Write tests
 - Report failures
 - Check if bug fixes address a specific failure
- **Role: System architect**
 - Responsibilities:
 - Ensure consistency in design decisions and define subsystem interfaces
 - Formulate system integration strategy
- **Role: Liaison**
 - Responsibilities:
 - Negotiate API with other teams.

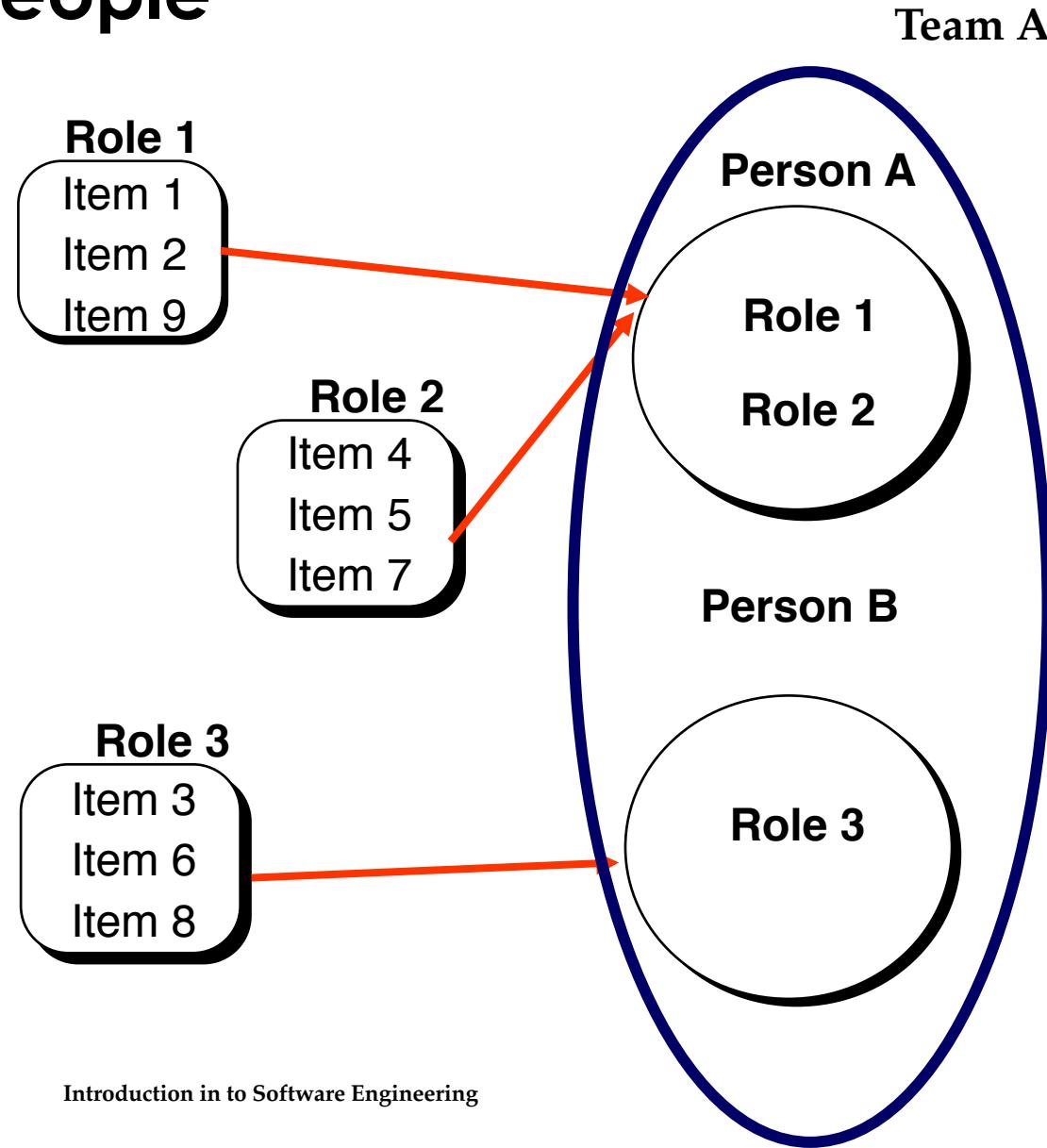
Types of Roles in Software Organizations



Responsibilities are assigned to Roles, Roles are assigned to People

“To Do” List for the Project

- Item 1
- Item 2
- Item 3
- Item 4
- Item 5
- Item 6
- Item 7
- Item 8
- Item 9



Possible Mappings of Roles to Participants

- One-to-One
 - Ideal but rare
- Many-to-Few
 - Each project member assumes several "hats"
 - Danger of over-commitment
 - Need for load balancing
- Many-to-"Too-Many"
 - Some people don't have significant roles
 - Lack of accountability
 - Loosing touch with project.

Key Concepts for Mapping Roles to People

- **Authority:**
 - The ability to make binding decisions between people and roles
- **Responsibility:**
 - The commitment of a role to achieve specific results
- **Accountability:**
 - Tracking a task performance to a specific person
- **Delegation:**
 - Binding a responsibility assigned to one person (including yourself) to another person.

Delegation

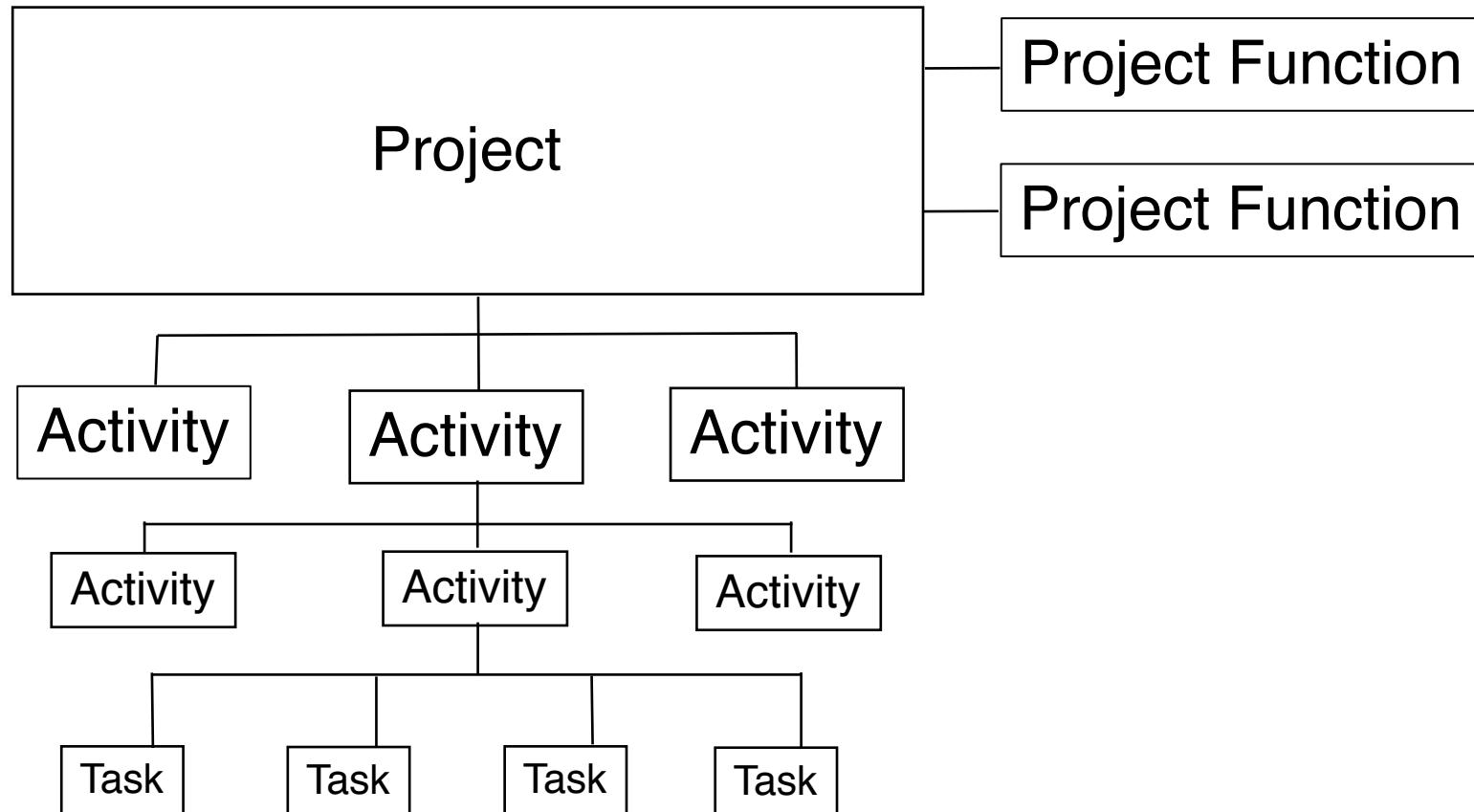
- **Delegation:** Binding a responsibility assigned to one person to another person
- Three reasons for delegation:
 - **Time Management:** To free yourself up for other tasks
 - **Expertise:** The most qualified person makes the decision
 - **Training:** To develop another person's ability to handle additional assignments
- You can delegate authority, but you cannot delegate responsibility. You can only share responsibility.

Outline of Today's Lecture

- ✓ Communication events
- ✓ Communication mechanisms
- ✓ Communication activities
- ✓ Project definition
- ✓ Typical project management issues
- ✓ Organization forms
- ✓ Mapping roles to people in organizations
- Tasks & Activities, Work Products & Deliverables
 - UML Model of a Project
 - Communication Structure

Project: Activities, Tasks and Project Functions

- A project has a duration and consists of project functions, activities and tasks



Activities

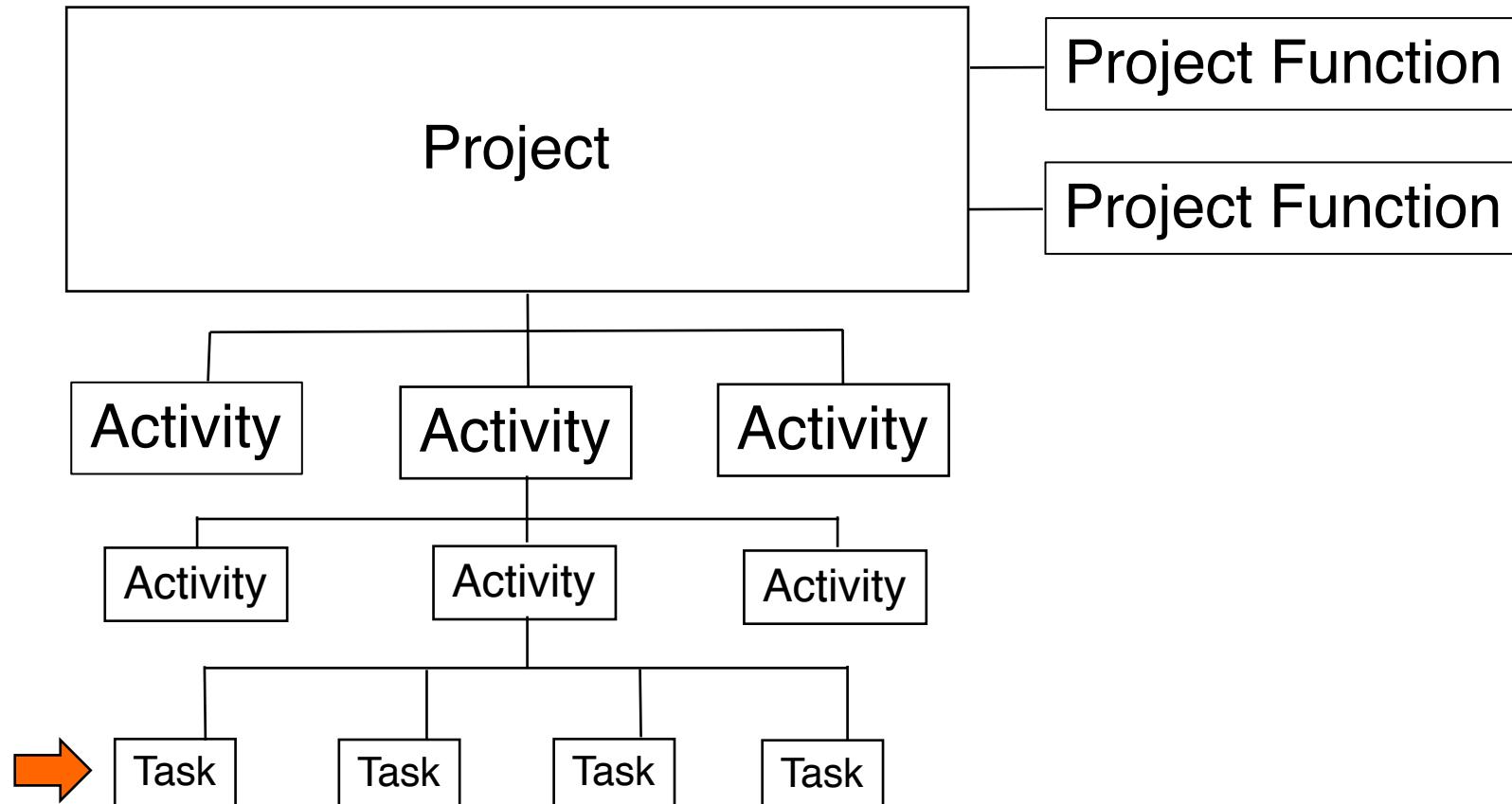
- Major unit of work
- Culminates in a major project milestone:
 - Scheduled event used to measure progress
 - Internal checkpoints should not be externally visible
 - A project milestone usually produces a baseline
- Activities are often grouped again into higher-level activities with different names:
 - Phase 1, Phase 2 ...
 - Step 1, Step 2 ...
- Allows separation of concerns
- Precedence relations can exist among activities
 - Example: “A1 must be executed before A2”

Examples of Software Engineering Activities

- Planning
- Requirements Elicitation
- Analysis
- System Design
- Object Design
- Implementation
- Testing
- Delivery

Project: Activities, Tasks and Project Functions

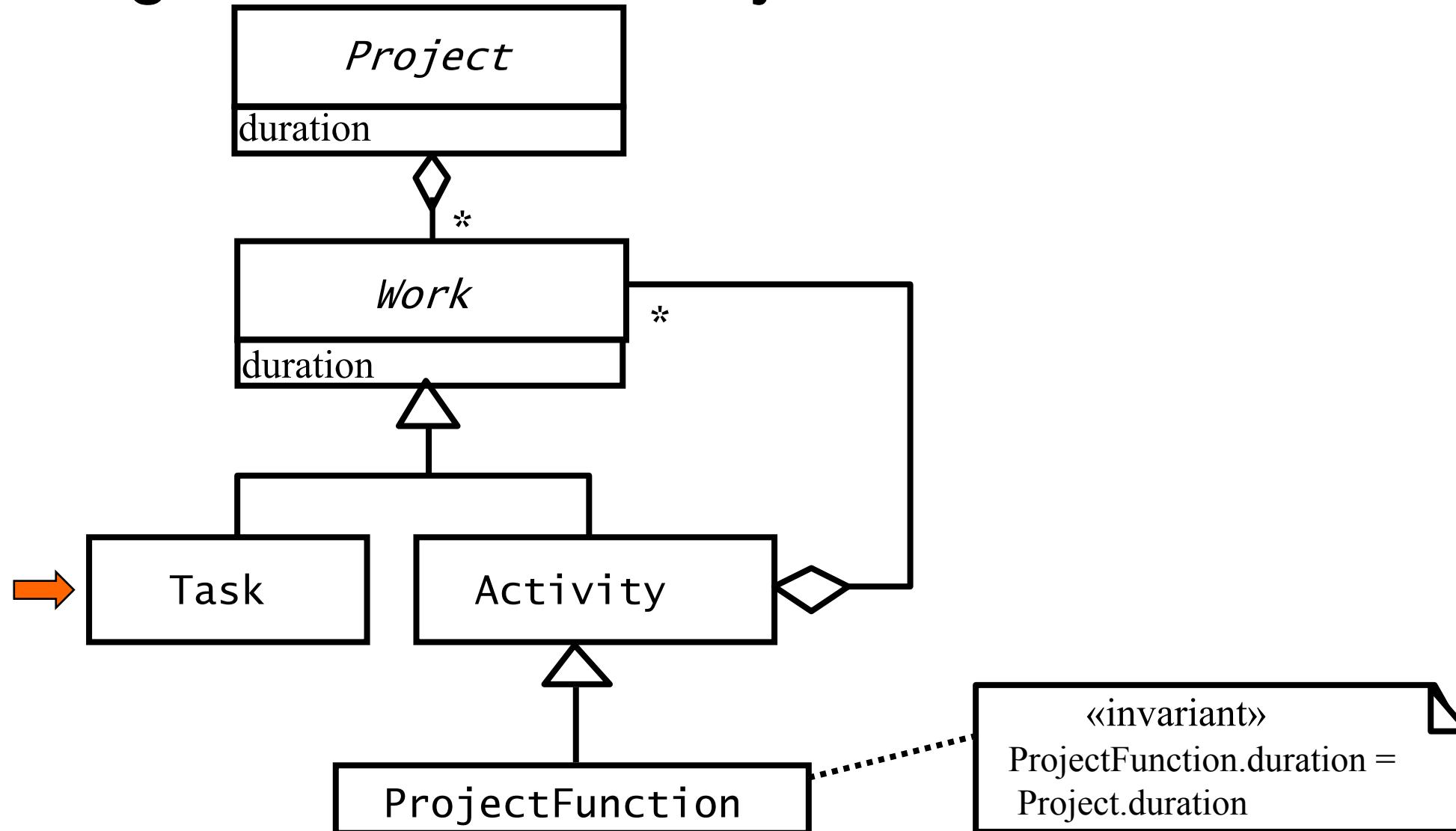
- A project has a duration and consists of project functions, activities and tasks



Tasks and Work Packages

- A task is specified by a **work package**
 - Description of work to be done
 - Preconditions for starting, duration, required resources
 - Work products to be produced, acceptance criteria for it
 - Risks involved
- A task must have **completion criteria**
 - Includes the acceptance criteria for the work products (deliverables) produced by the task.

Modeling Tasks, Activities, Project Functions in UML



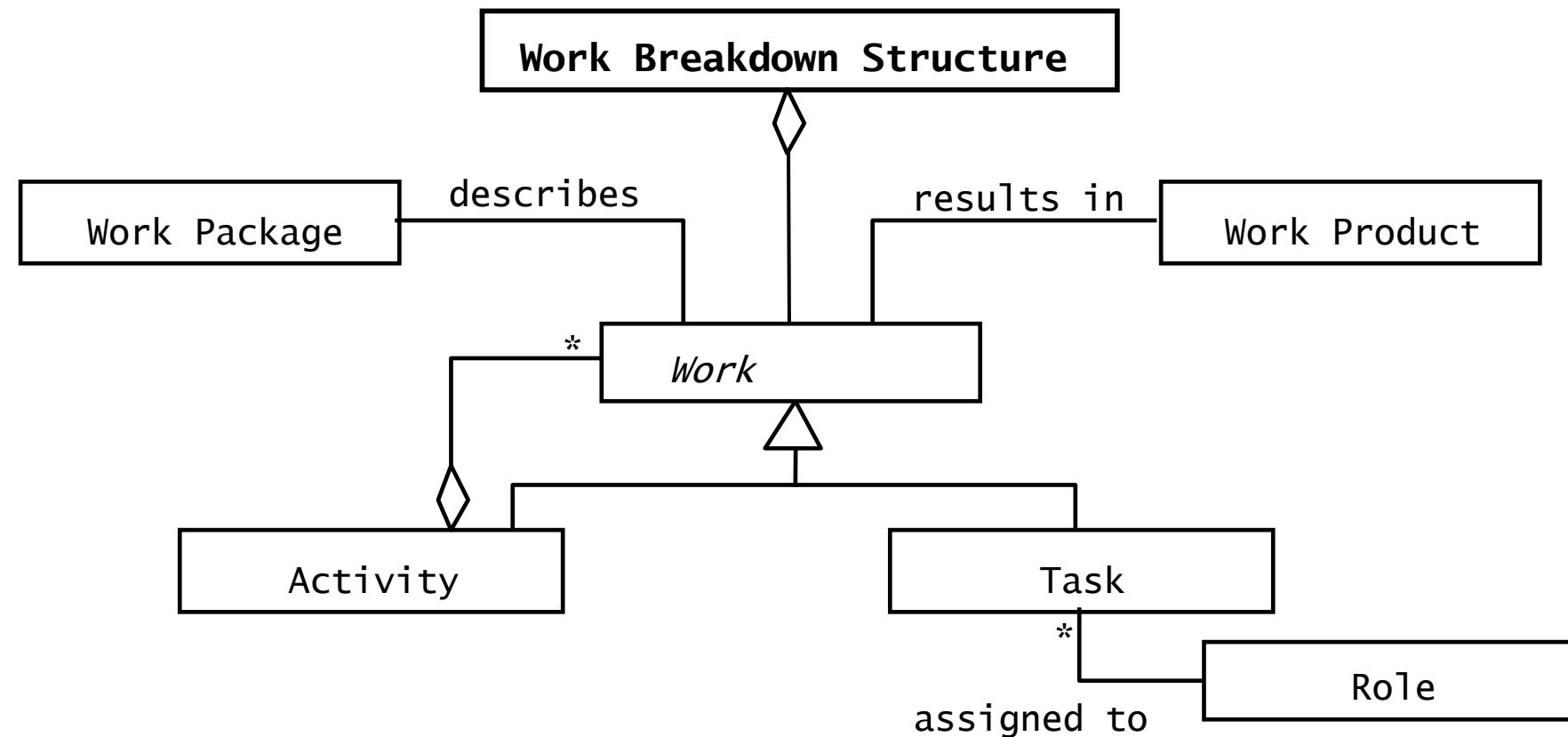
Examples of Project Functions

- Configuration Management
 - Documentation
 - Quality Control (V&V: verification and validation)
 - Training
 - Testing
 - Project management
-
- Project functions are called Integral processes in the IEEE 1074 standard for Software Life Cycle Processes (See Lecture on Lifecycle Modeling)
 - Sometimes they are also called cross-development processes.

Work Products

- A work product is the visible outcome of a task
- Examples
 - A document
 - A review of a document
 - A presentation
 - A piece of code
 - A test report
- Work products delivered to the customer are called **deliverables**.

Modeling Work Products, and Work Packages



Work Breakdown Structure: The aggregation of the work to be performed in a project. Often called **WBS** (in traditional projects) or **Epics** (in agile projects)

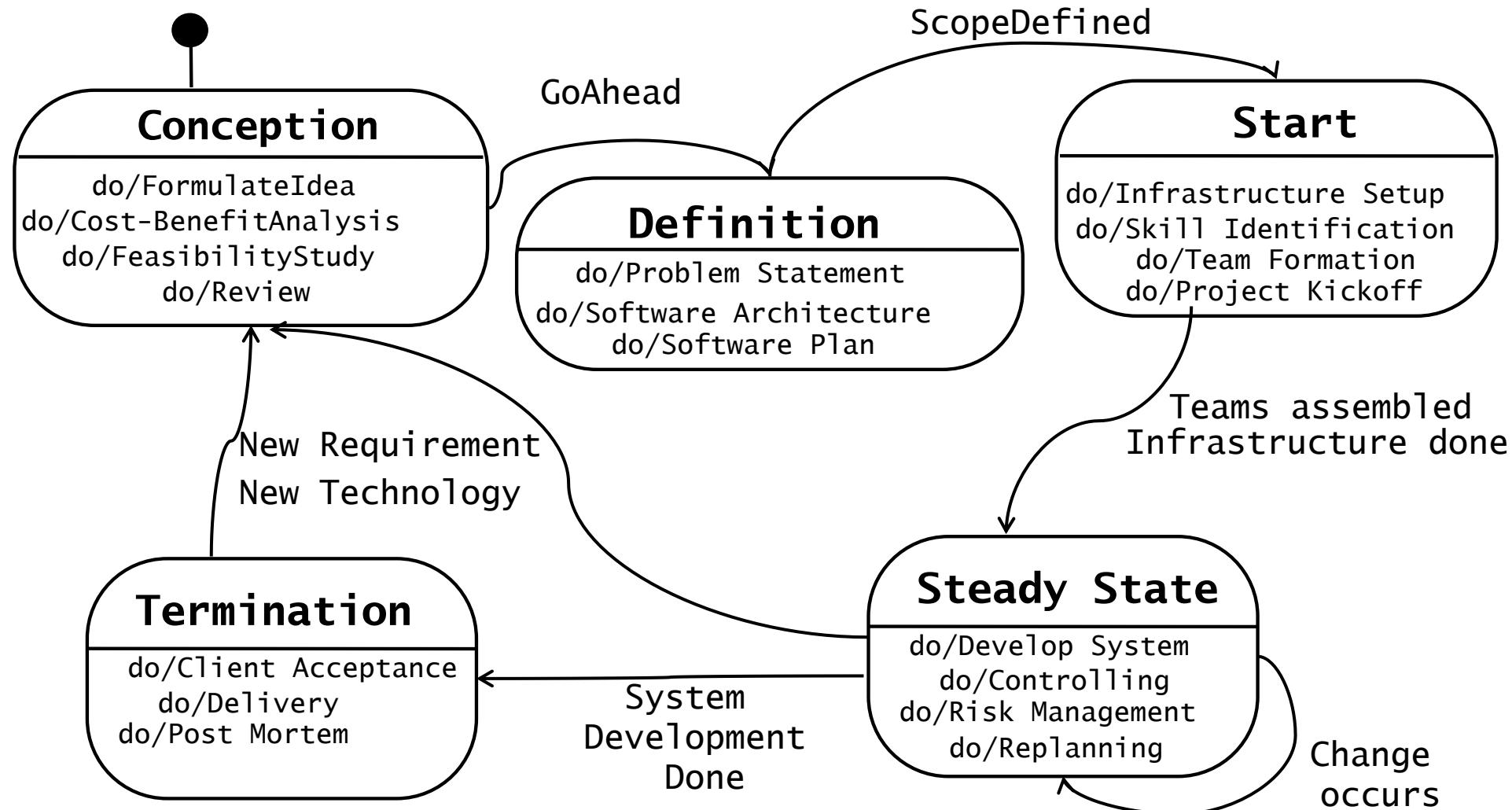
Outline of Today's Lecture

- ✓ Communication events
- ✓ Communication mechanisms
- ✓ Communication activities
- ✓ Project definition
- ✓ Typical project management issues
- ✓ Organization forms
- ✓ Mapping roles to people in organizations
- ✓ Tasks & Activities, Work Products & Deliverables
- UML Model of a Project
- Communication Structure

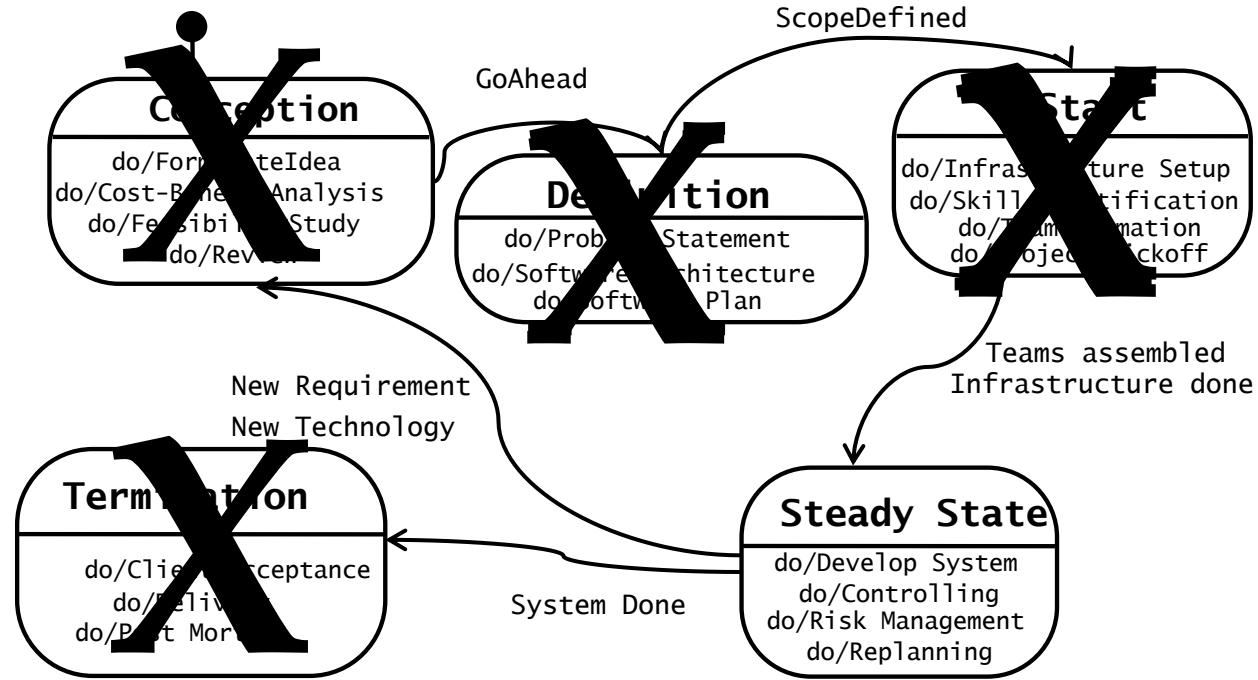
Simple Dynamic Model of a Project

- Every project has at least 5 states
 - Conception: The idea is born
 - Definition: A plan is developed
 - Start: Teams are formed
 - Steady State: The work is being done
 - Termination: The project is being finished.

States of a Software Project



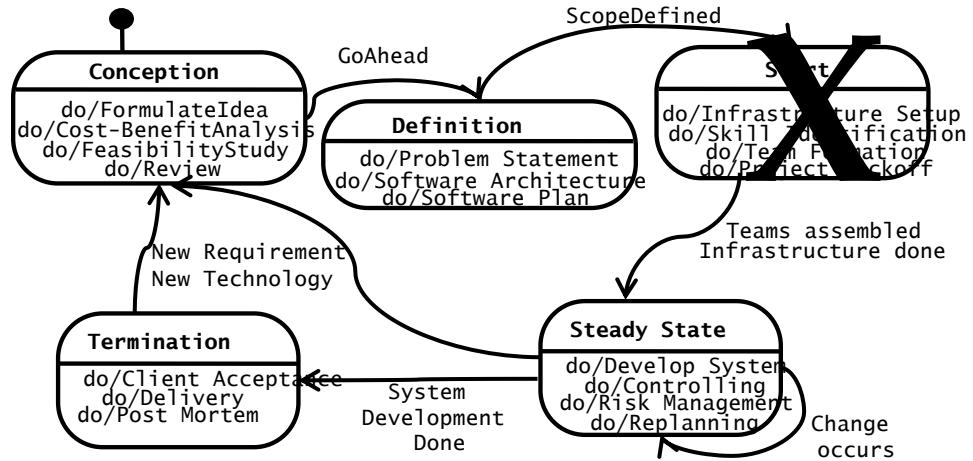
Software Project Management Mistakes



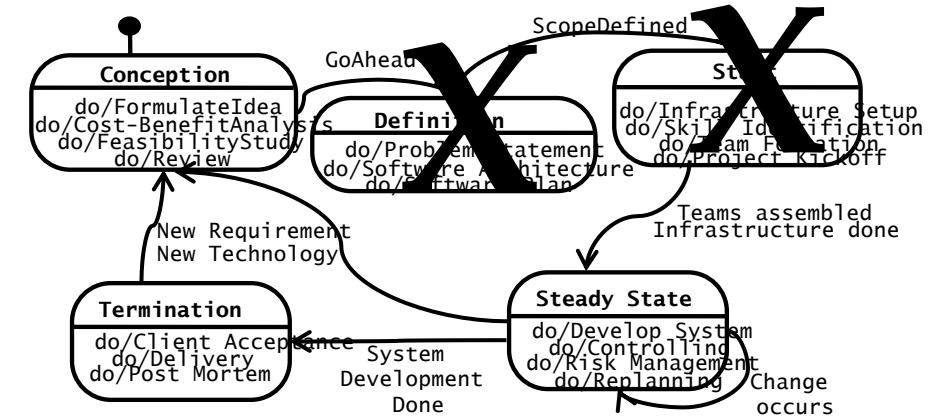
- Project manager skips the start phase
- Project manager skips the definition and start phase
- Project manager jumps straight to the steady state phase after joining the project late
- Project manager cancels the termination phase.

Mistake: Skipping the Start Phase

- Main reason: Time pressure
 - “I have no time to get to know the teams and check their skills, they should just do their work, we have no money for additional tutorials anyway”
- Reasons for not skipping the start phase
 - Inform stakeholders that the project has been approved and a schedule has been set
 - Confirm that stakeholders are able to support the project
 - Reevaluate and reconfirm work packages with developers.



Mistake: Skipping Definition and Start Phase



- Known territory argument
 - “I have done this before, no need to waste time”
- Unknown territory argument
 - “My project is different from anything I have ever done before, so what good is it to plan?”
- Reasons for not skipping these phases:
 - Even though a project may be similar to an earlier one, some things are always different
 - It is better to create a map if you are attempting to travel into unknown territory.

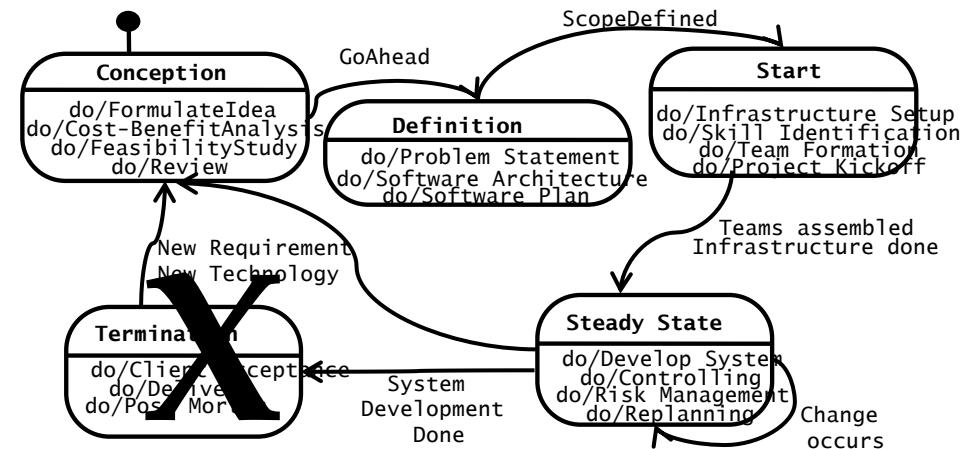
Mistake: Skipping Termination Phase

- Main reasons:

- You leave a project to move on right to the next one
- Why? You just showed you are a successful manager:-)
- Limited resources, ambitious deadlines
- A new project is always more challenging than wrapping up an old one

- Reasons not to skip the termination phase:

- You would never really know how successful or unsuccessful your project was
 - Take the time to ensure that all tasks are completed or identified as open issues
 - Learn from your mistakes ("Post Mortem", "lessons learned"):
 - If you don't, you will make the same mistakes again and again and you may even fail.

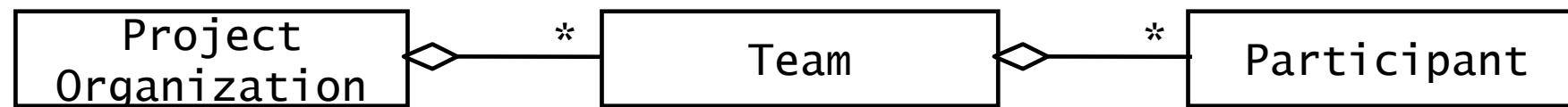


Outline of Today's Lecture

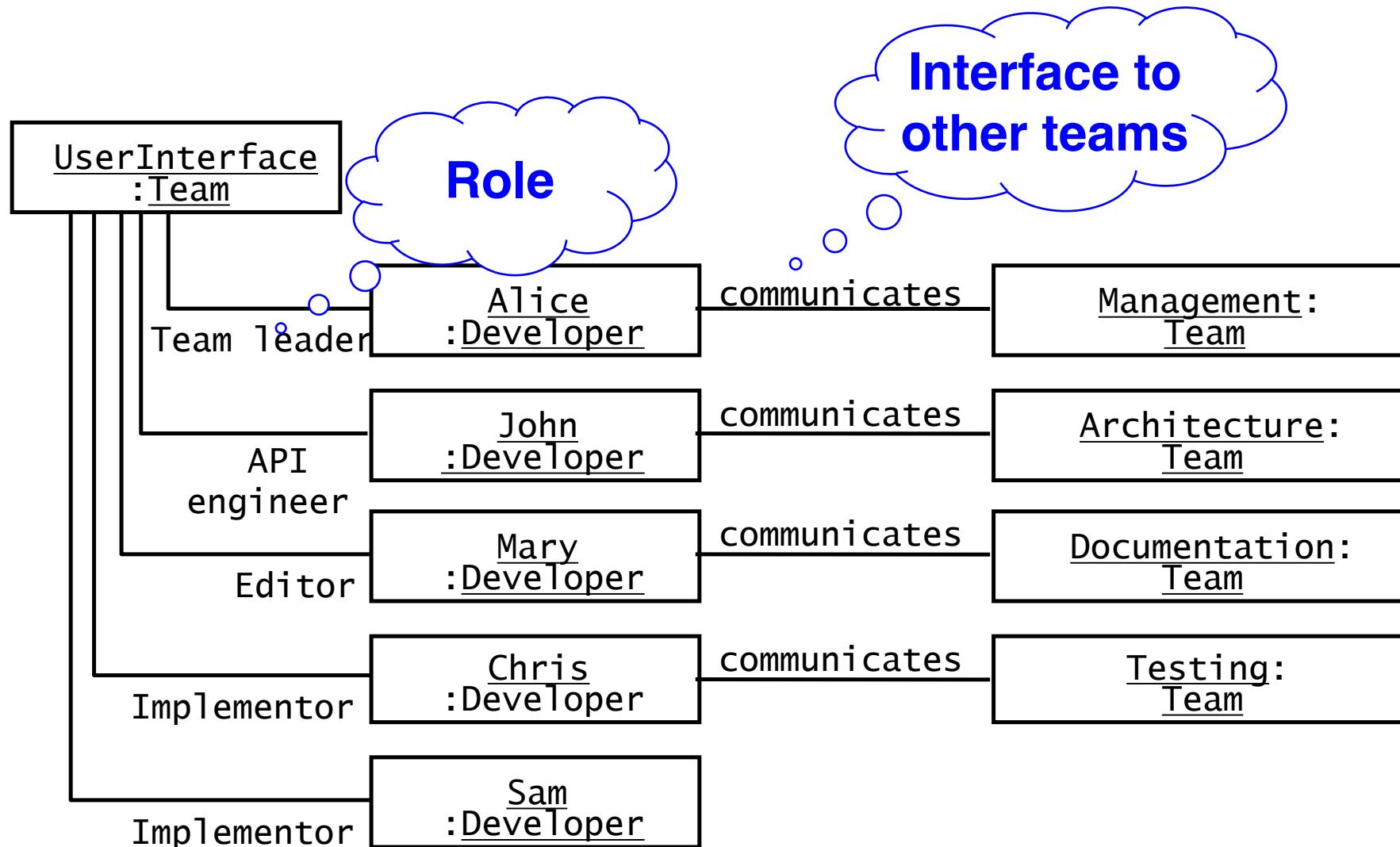
- ✓ Communication events
- ✓ Communication mechanisms
- ✓ Communication activities
- ✓ Project definition
- ✓ Typical project management issues
- ✓ Organization forms
- ✓ Mapping roles to people in organizations
- ✓ Tasks & Activities, Work Products & Deliverables
- ✓ UML Model of a Project
- Communication Structure

Project Organization Revisited

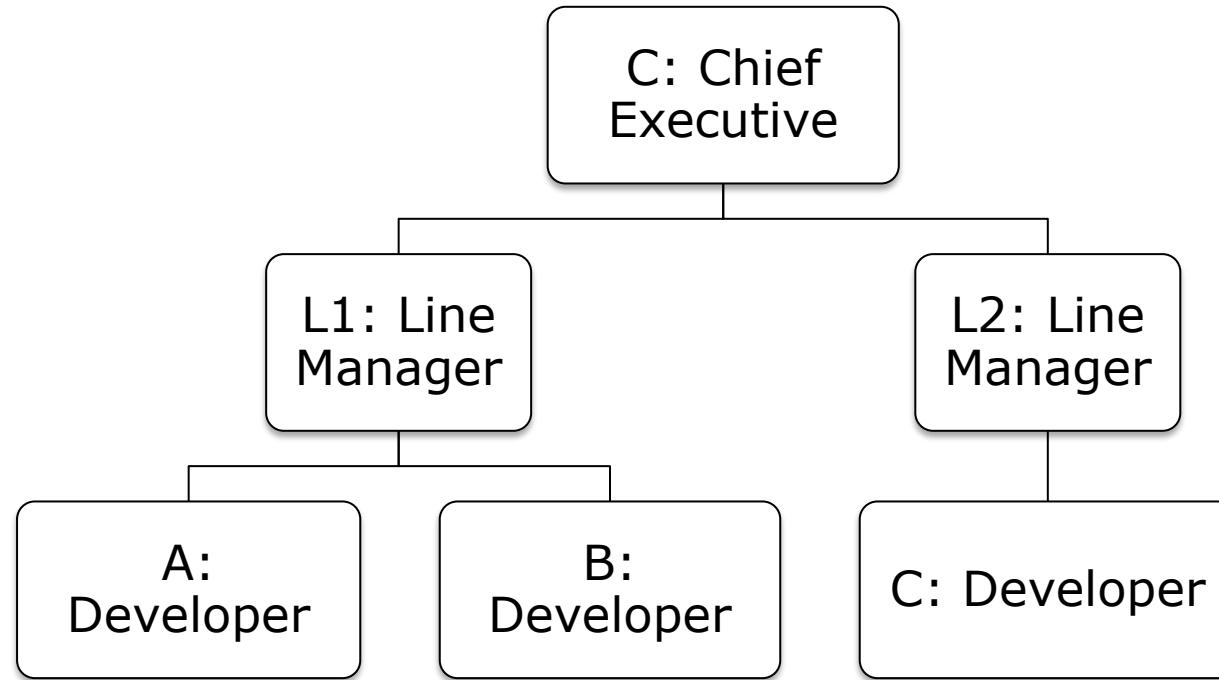
- A **project organization** defines the relationships among resources in a project, in particular the roles of the participants
- A project organization should define
 - Who decides (**decision structure**)
 - Who reports their status to whom (**reporting structure**)
 - Who communicates with whom (**communication structure**)



Example of a Communication Structure



Organization Chart for Many Organizations

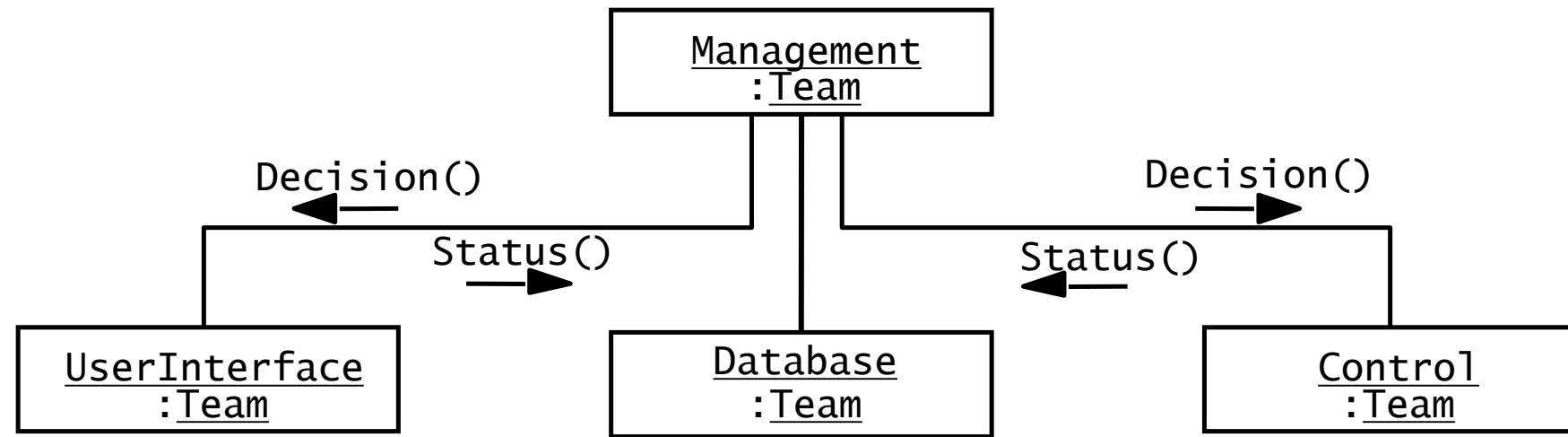


**No distinction between decision structure,
reporting structure, communication structure!**

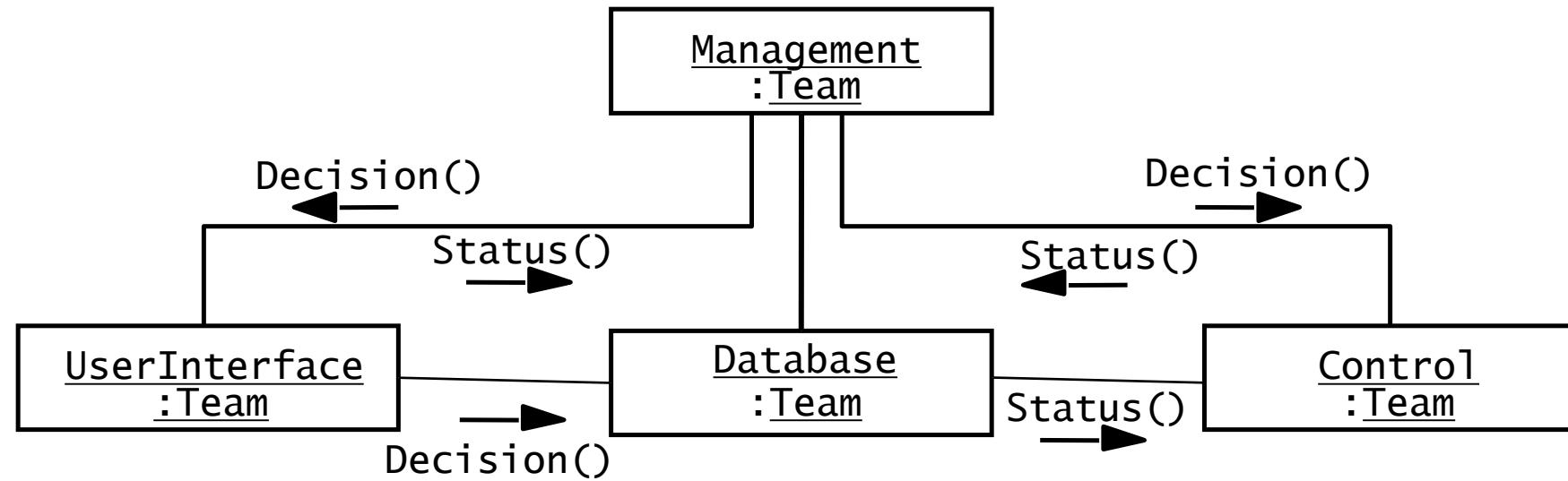
Reporting Structure vs. Communication Structure

- Reporting supports project management in tracking project status
 - What work has been completed?
 - What work is behind schedule?
 - What issues threaten project progress?
- Reporting along the hierarchy is not sufficient when two teams need to communicate
 - Slows down communication
 - A communication structure different from the reporting structure should be set up
 - A participant from each team is responsible for facilitating communication between both teams
 - Such a participant is called a **liaison**.

Distinguishing between Decision and Reporting Structure



Modified Decision and Reporting Structure





Research Study

Stack Overflow Considered Helpful?

- Help us to investigate the impact of **Stack Overflow** on **Android** application development
- Participate in a **user study** conducted by the Chair of Cyber Trust (www.cybertrust.in.tum.de)
- €10 compensation. How?
 - Solve a programming task with help from Stack Overflow in one hour
- **Programming skills in Java are required:**
 - Participation in at least one Java course
 - Or half a year of practical experience off-campus
- **Register by sending email to flx.fischer@tum.de**



Summary

- Projects are concerted efforts towards a goal that take place within a limited time
- Project participants are organized in terms of teams, roles, control relationships, and communication relationships
- An individual can fill more than one role
- Work is organized in terms of activities, tasks assigned to roles and producing work products.

Additional References

- [Brooks, 1995] F. P. Brooks
 - The Mythical Man Month: Anniversary Edition: Essays on Software Engineering. Addison-Wesley, Reading, MA, 1995
 - Also available as PDF File:
https://www.researchgate.net/publication/220689892_The_Mythical_Man-Month_Essays_on_Software_Engineering
- [Weinberg, 1971] G. M. Weinberg
 - The Psychology of Computer Programming, Van Nostrand, New York, 1971
- [Paulish, 2001] D. J. Paulish
 - Architecture-centric Software Project Management, SEI Series in Software Engineering, Addison-Wesley, 2001
- [Raymond, 1998] E. Raymond
 - The Cathedral and the bazaar, 1998
 - <http://manybooks.net/titles/raymondericother05cathedralandbazaar.html>

Course Review

Bernd Bruegge

Chair for Applied Software
Engineering

Technische Universität München

12 July 2018

Roadmap for Today's Lecture

- **Context and Assumptions**

- You have reviewed the slides and films of the lectures 1-12

- You have read the OOSE text book by Bruegge and Dutoit

- **Content of this lecture**

- Repetitorium

- **Objective:** At the end of the lecture you understand 42.

Examination Information

- **Seating List:** A seating list will be handed out to you in time to inform you about your personal examination location
- **Dictionary:** You are allowed to bring an English - German dictionary
- **Cheat Sheet:** You can bring a **handwritten** DIN A4 sheet (both sides)
- **Additional Resources:** No other resources are allowed – this is a closed book examination.



How to take the exam

- The exam consists of questions worth 60 points
- Since the exam is 90 minutes long, 1 point to 1 1/2 minute of work on the average
 - Important: answer only the question and nothing more
- Use the amount of points and the text of the question to determine the appropriate length of your answer
- The questions of the exam will be in English. You can answer exam questions either in German or English. Once you have selected a language, you cannot change it.

General Information

- The exam consists of two main parts:
 - Multiple Choice questions (with single choice answer)
 - Problem solving questions (including programming and modeling)
- Distribution in the exam:
 - About 20% MC questions
 - About 80% problem solving questions
- General info exam questions:
 - **Explain** – A concept should be explained in detail in your own words.
A 1-to-1 repetition of the definition in the lecture is **not** sufficient.
 - **What would you choose** -> Justify your answer
 - **Difference between...Justify your answer** – Explain the difference between two or more concepts.

Exam Areas

0. Software Engineering is a Problem Solving Activity
1. Abstraction and Modeling
2. Analysis and System Design
3. Project Management Basics
4. Configuration and Release Management
5. Lifecycle Modeling
6. Object Design and Design Patterns
7. OCL and Contracts
8. Agile Methods
9. Testing
10. Methodologies

Software Engineering is a Problem Solving Activity

- **Analysis:** Understand the problem nature and break the problem into pieces
- **Synthesis:** Put the pieces together into a large structure that prepares for the solution
- Problem solving is a creative activity
- **George Polya**, 1887–1985, *How to Solve It*, Princeton University Press, 2nd ed., 1957
- Summary of the book:
 - math.utah.edu/~pa/math/polya.html

Once a problem is solved, it is often not hard anymore.

Example: Egg of Columbus.

- https://en.wikipedia.org/wiki/Egg_of_Columbus
- Now a metaphor for a brilliant idea or discovery that seems simple or easy after the fact.



Problem Solving Heuristics

Polya:

First: *understand the problem*

Second: *devise a plan*

Third: *carry out your plan*

Fourth: *examine the solution obtained.*

Feynman's Algorithm:

1. Write down the problem
2. Think very hard
3. Write down the solution

Bad Algorithm:

1. Do not write down the problem
2. Look at the sample solution
3. Copy the sample solution

Polya's Step 2 with Software Engineering Terms

- *Have you seen the problem before?*
 - Or have you seen the same problem in a slightly different form?
- *Do you know a related problem?*
 - Do you know an algorithm that could be useful?
- *There is a problem related to yours and solved before*
 - Could you reuse it? Could you use its result? Could you use its method?
Should you introduce an auxiliary element to make its use possible?
- *Could you restate the problem?*
 - Could you restate it differently in a different domain? Go back to definitions
- If you cannot solve the problem, try to solve first some related problem
 - Could you imagine a more accessible related problem?
 - A more general problem? A more special problem?
 - Could you solve a part of the problem? Keep only a part of the condition,
drop the other 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 is the *resulting idea* of a thought process where an idea has been distanced from an object
 - **Abstraction as entity**
- Ideas can be expressed by models



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
- **Functional model:** Scenarios, use case diagrams
- **Structural model:** Class diagrams, instance diagrams, communication diagrams, deployment diagrams
- **Dynamic model:** Sequence diagrams, statechart and activity diagrams

We can use Models to Describe Differences in Philosophies

Plato

Plato's description of reality:

- Reality consists of many things
- A thing can be either a particular thing or a form
- **Beauty is not a particular thing, but it is real and exists**

Aristotle

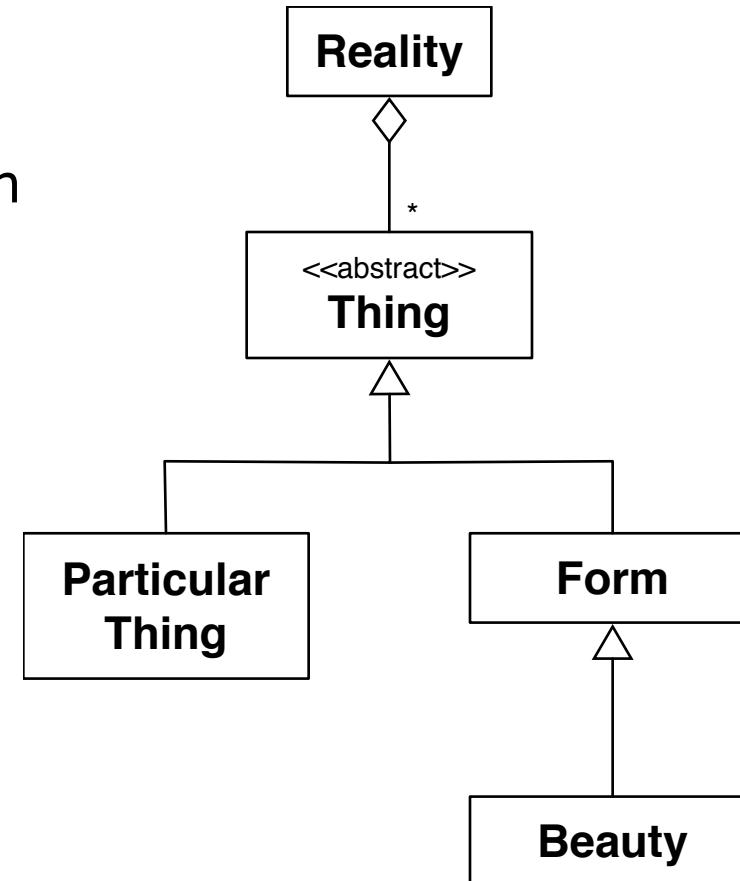
Aristotle's description of reality:

- Reality consists of many particular things called substances
- Each substance is composed of matter and form
- **Beauty is real, but it does not exist on its own, it is a form that is always part of a substance.**

How do the corresponding UML models look like?

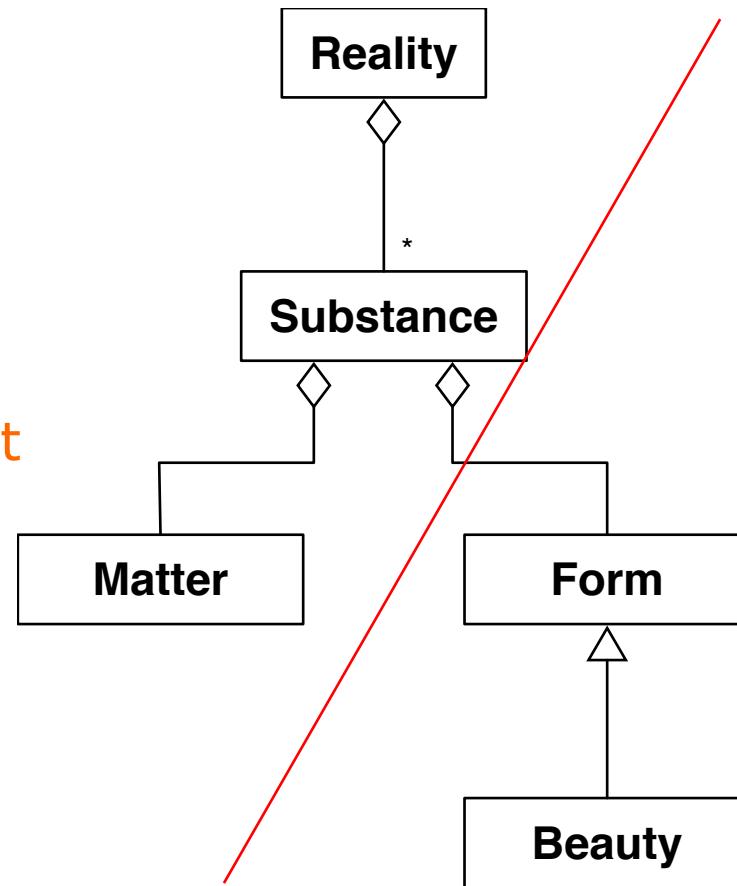
Plato's Model of Reality

- Reality consists of many things
- A thing can be either a particular thing or a form
- **Beauty is not a particular thing, it is a form**

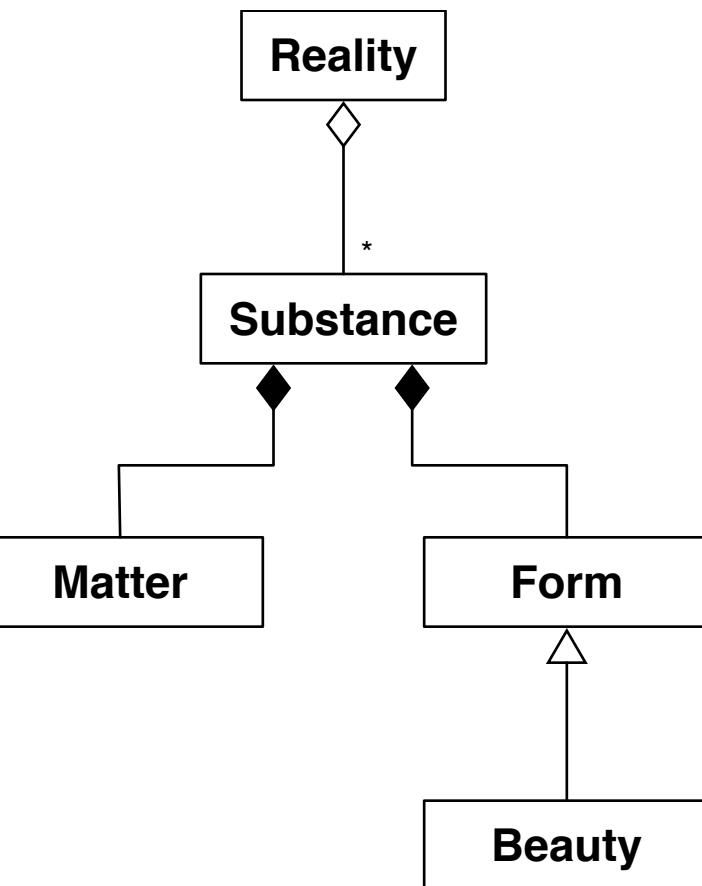


Aristotle's Model of Reality

- Reality consists of many particular things called substances
 - Each substance is composed of matter and form
 - Beauty is real, but it does not exist on its own, it is a form that is always part of a substance.

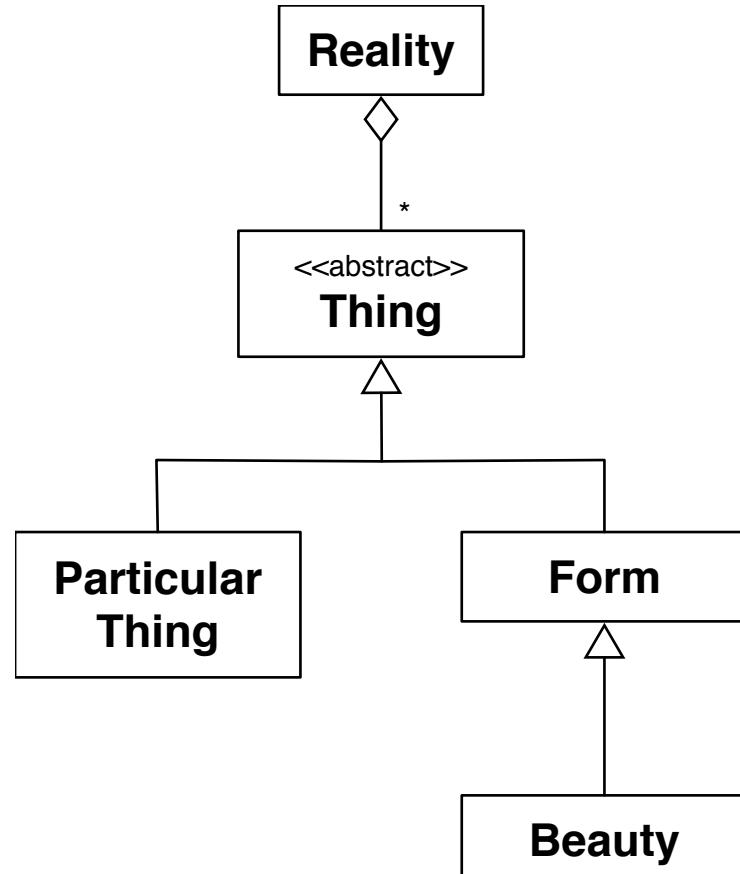


Aggregation is wrong

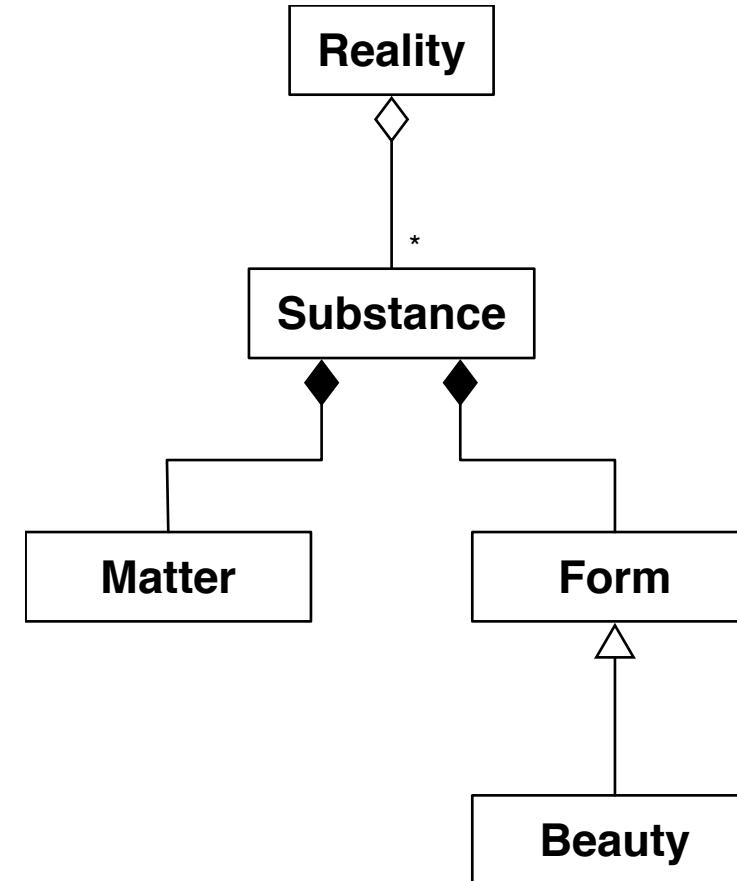


Composition is right

Comparison of Plato's and Aristotle's Philosophies



Plato



Aristotle

Why Modeling?

- We use models
 - To abstract away from details in the application domain, so we can draw complicated conclusions in the reality by performing simple steps in the model
 - To get insights into the past or presence
 - To make predictions about the future
- We can model all kinds of systems
 - Natural systems (Astronomy, Astrophysics)
 - Human beings (Psychology, Sociology, HCI, CSCW)
 - Artificial Systems (Computer science)
 - Even philosophical ideas can be modeled

Functional vs Object-oriented decomposition

- Functional decomposition
 - The system is decomposed into functions
 - Functions can be decomposed into smaller smaller functions
- Object-oriented decomposition
 - The system is decomposed into classes ("objects")
 - Classes can be decomposed into smaller classes.

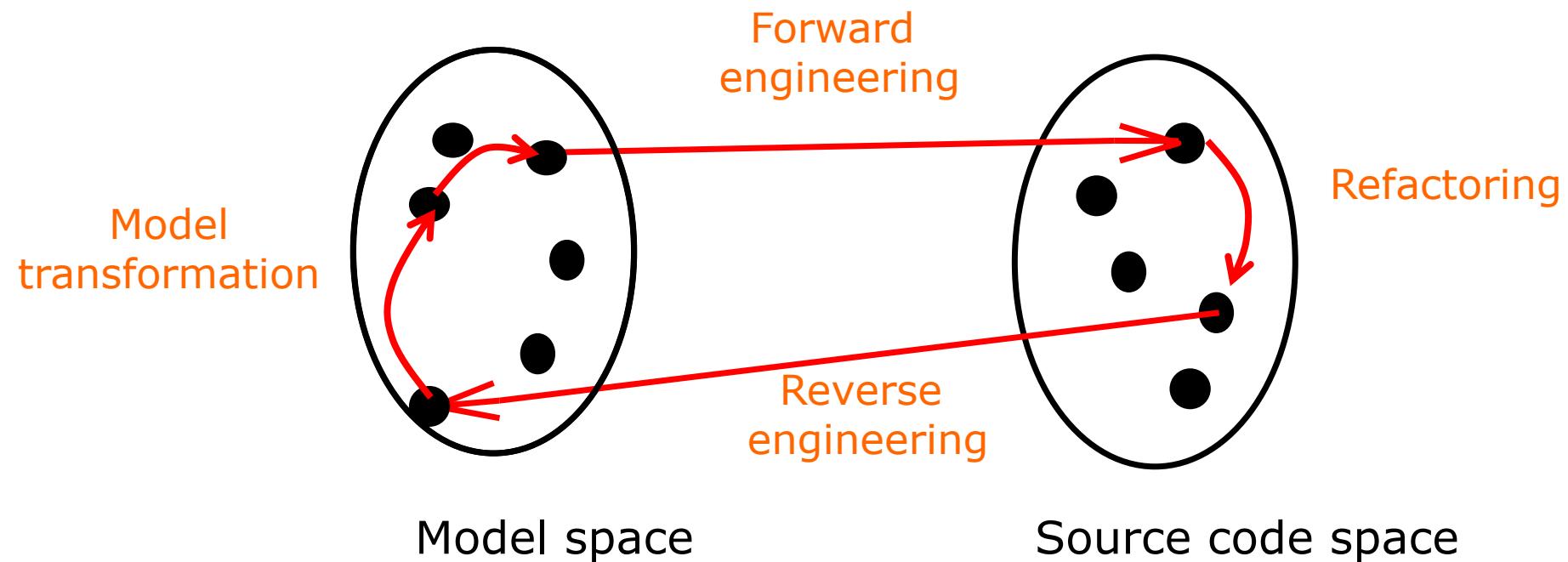
Which decomposition is the right one?

Models are not the Truth

- 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.

Model Transformations

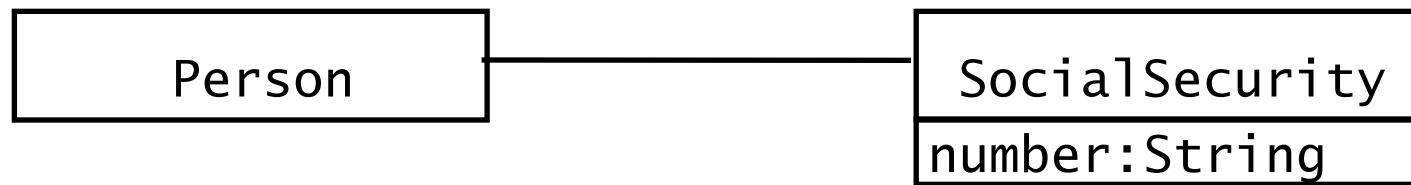
- Object-Oriented modeling is the continued transformation of a system model throughout the software development process
- We distinguish 4 types of transformations



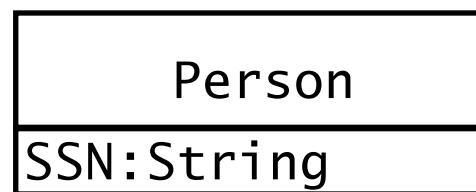
Details in Lecture 7 Mapping Models to Code on June 7, 2018

Example of a Model Transformation: Collapsing Objects

Object design model before model transformation:



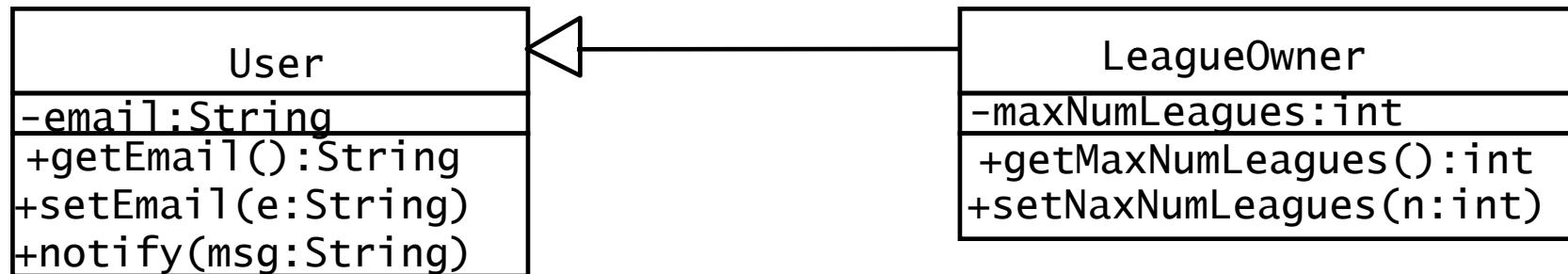
Object design model after model transformation:



Turning an object into an attribute of another object is usually done when the object does not have any interesting dynamic behavior (only get and set operations).

Mapping UML Inheritance into Java Inheritance

Object design:



Source code:

```
public class User {
    private String email;
    public String getEmail() {
        return email;
    }
    public void setEmail(String value){
        email = value;
    }
    public void notify(String msg) {
        // ...
    }
}
```

```
public class LeagueOwner extends User {
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
        return maxNumLeagues;
    }
    public void setMaxNumLeagues(int value) {
        maxNumLeagues = value;
    }
}
```

Example: Realizing Inheritance in Java

- Realization of specialization and generalization
 - Definition of subclass
 - Java keyword: **extends**
- Realization of simple inheritance
 - Overwriting of methods is not allowed
 - Java keyword: **final**
- Realization of implementation inheritance
 - Overwriting of methods
 - No keyword necessary:
 - Overwriting of methods is default in Java
- Realization of specification inheritance
 - Specification of an interface
 - Java keywords: **abstract**, **interface**

Use of Models

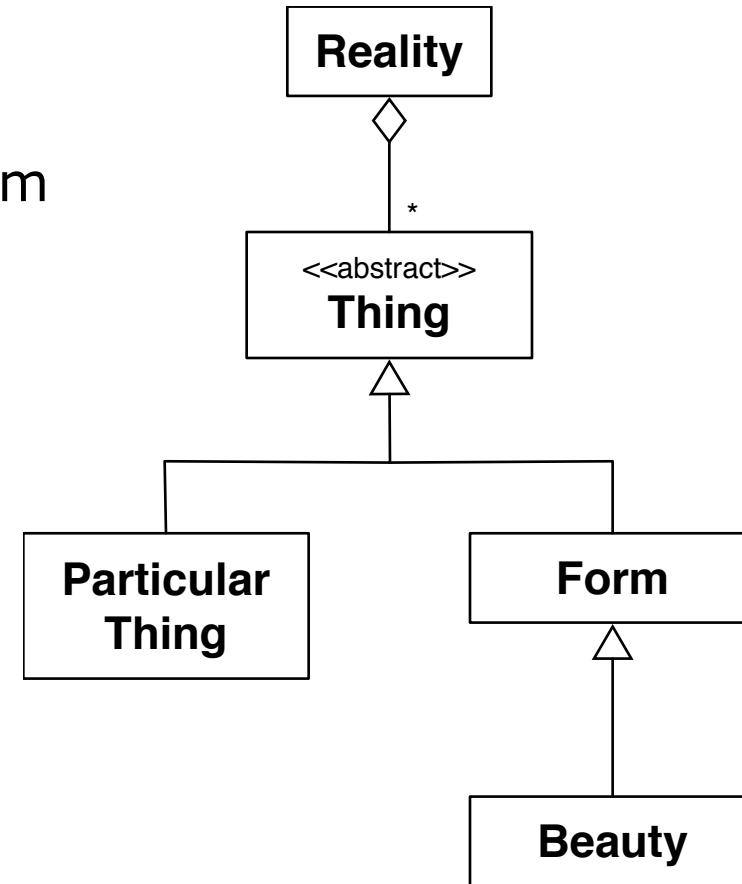
- **Communication:** The model provides a common vocabulary. A model is communicated informally (“conceptual model”)
 - Target is a human being (developer, end user)
- **Analysis and Design:** Models enable developers to specify and reason about a future system
 - Target is a tool (CASE tool, compiler)
- **Archival:** Compact representation for storing the design and rationale of an existing system
 - Target is the human (analysis, project manager).

Be able to Apply Abbots' Technique

- From the Problem Statement to the System Model
- Analyze the problem statement
 - Identify functional requirements (verbs)
 - Identify nonfunctional requirements
 - Identify constraints (pseudo requirements)
 - Identify participating objects (nouns)
- 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.

Review: Plato's Model of Reality

- Reality consists of many things
- A thing can be either a particular thing or a form
- Beauty is not a particular thing, it is a form



Problem Statement

Object Model

Abbotts' Technique applied in the Plato Example

- **From the Problem Statement to the System Model**
- Analyze the problem statement
 - Identify functional requirements (verbs)
 - Identify nonfunctional requirements
 - Identify constraints (pseudo requirements)
 - **Identify participating objects (nouns)**
- 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.**

Transform a Problem Statement into a UML class diagram

- Problem Statement:
 - In a software system managed by a configuration management system there are many controlled items
 - Each controlled item can have many versions. A configuration item is a controlled item
 - A CM Aggregate (configuration management aggregate) consists of many controlled items
 - A version can be either a release or a promotion. A release is a version that has been delivered to a customer. A promotion is a version used internally by the developers.
 - Promotions are stored in a master directory. Releases are copied into a separate directory, called the repository.

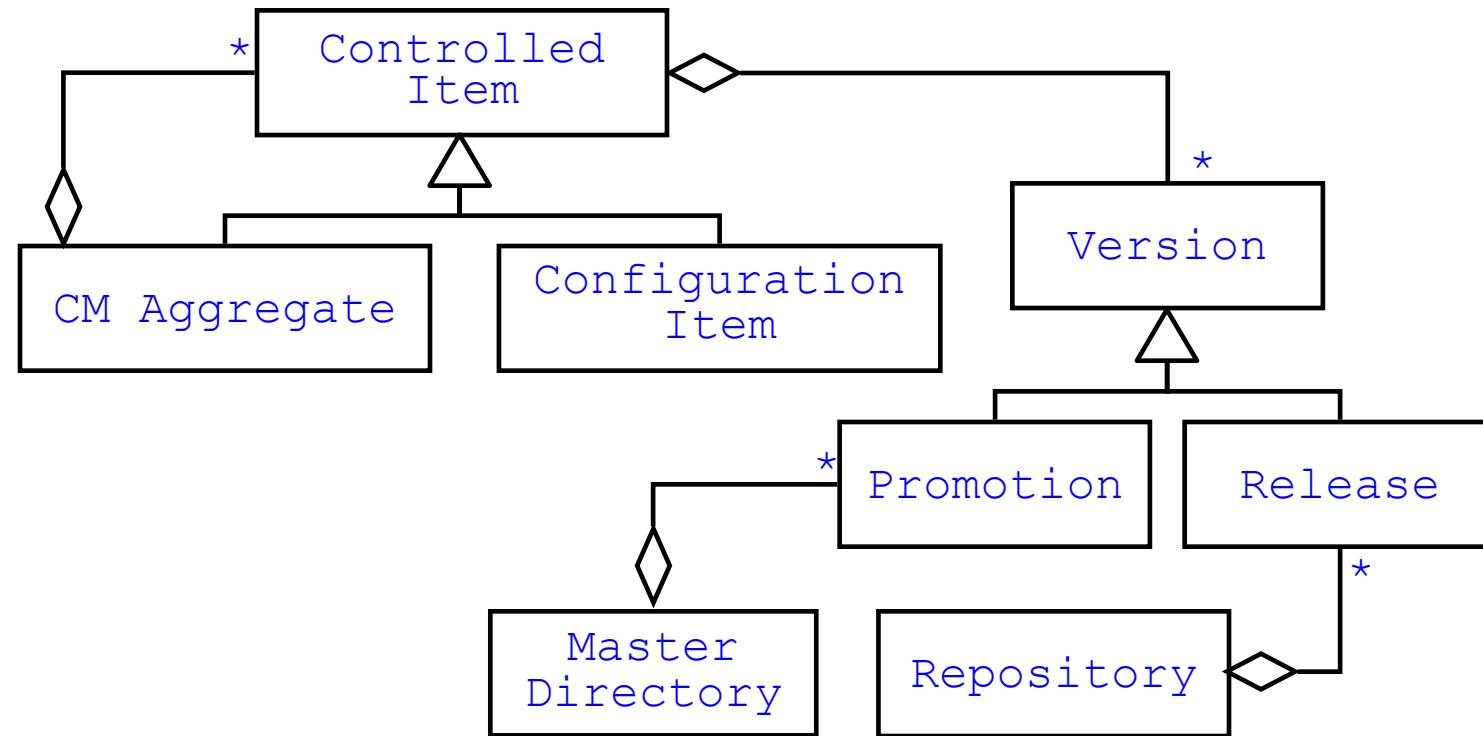
Possible Exam Question: Create a UML Class diagram from a Problem Statement

Transform a Problem Statement into a UML class diagram

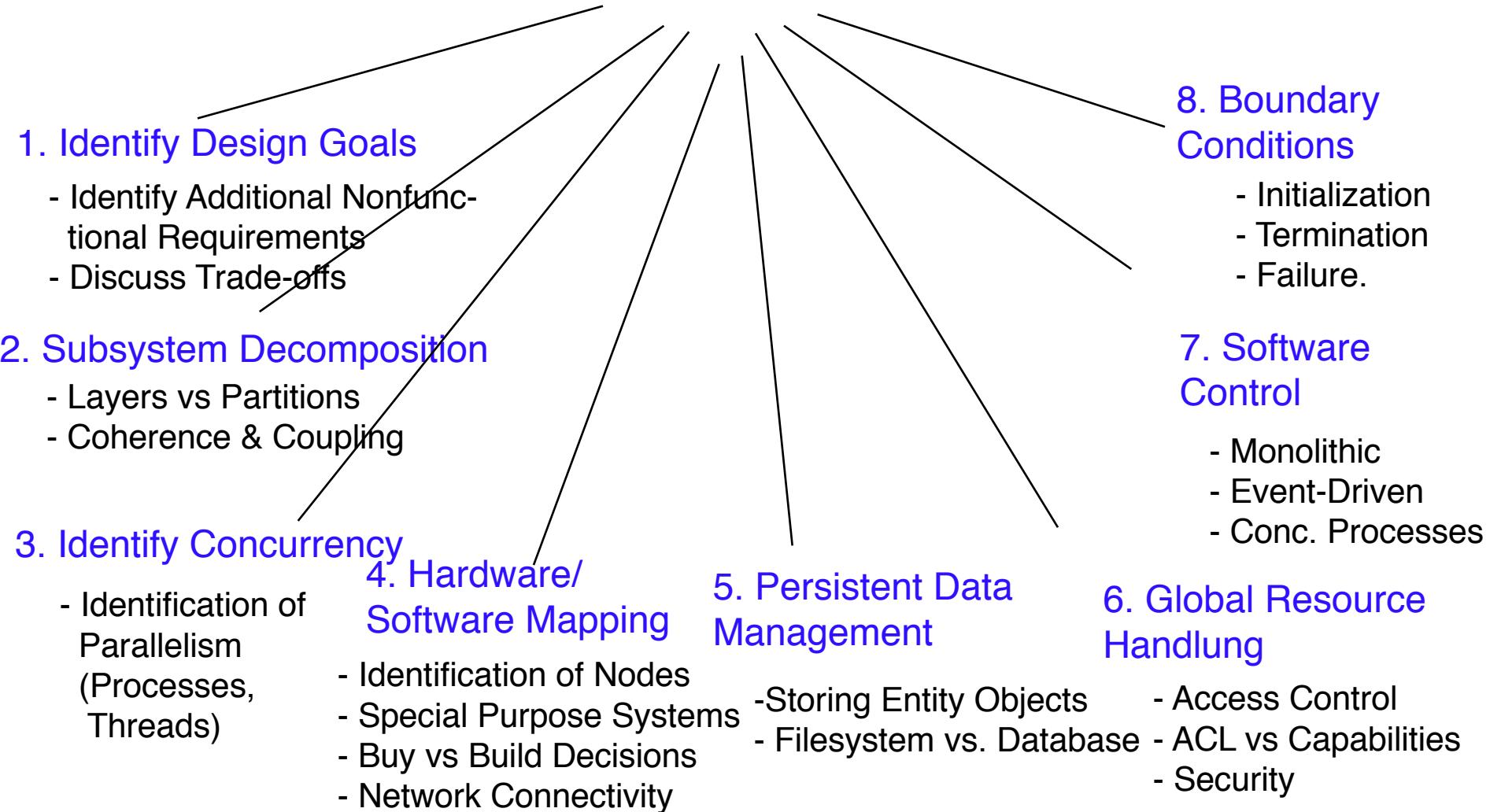
- Problem Statement:
 - In a software system managed by a configuration management system there are many **controlled items**
 - Each controlled item can have many **versions**. A configuration item is a controlled item
 - A **CM Aggregate** (configuration management aggregate) consists of many controlled items
 - A version can be either a **release** or a **promotion**. A release is a version that has been delivered to a customer. A promotion is a version used internally by the developers.
 - Promotions are stored in a **master directory**. Releases stored in a separate directory, called the **repository**.

Possible Exam Question: Create a UML Class diagram from a Problem Statement

Solution: Configuration Management System Model



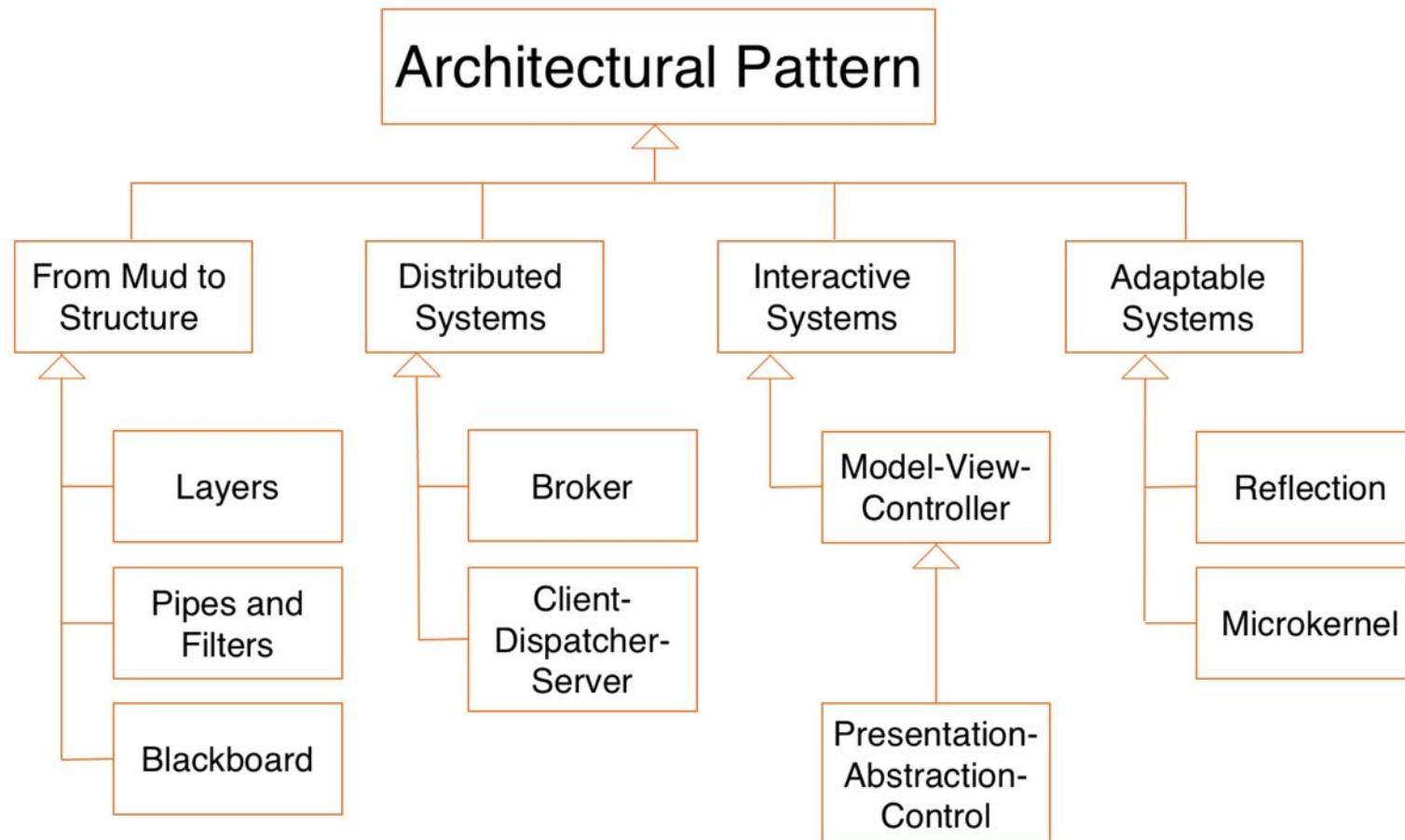
System Design



Miscellaneous System Design Topics

- Examples of Design Goals
- Architectural Styles and Architectures
- Open and closed Architectures
- Layering vs Partitioning
- Coupling and Coherence
- Hardware Software Mapping
- Mapping object models to relational databases
- Access Matrix and Access control list

Architectural Patterns

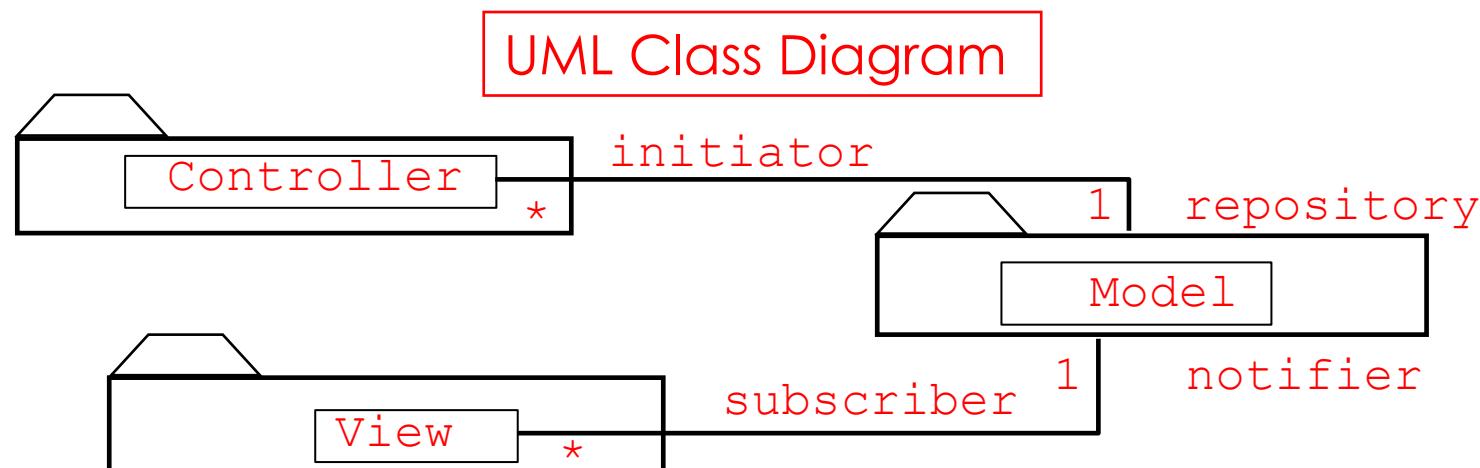


Design Pattern vs. Architectural Style vs Architecture

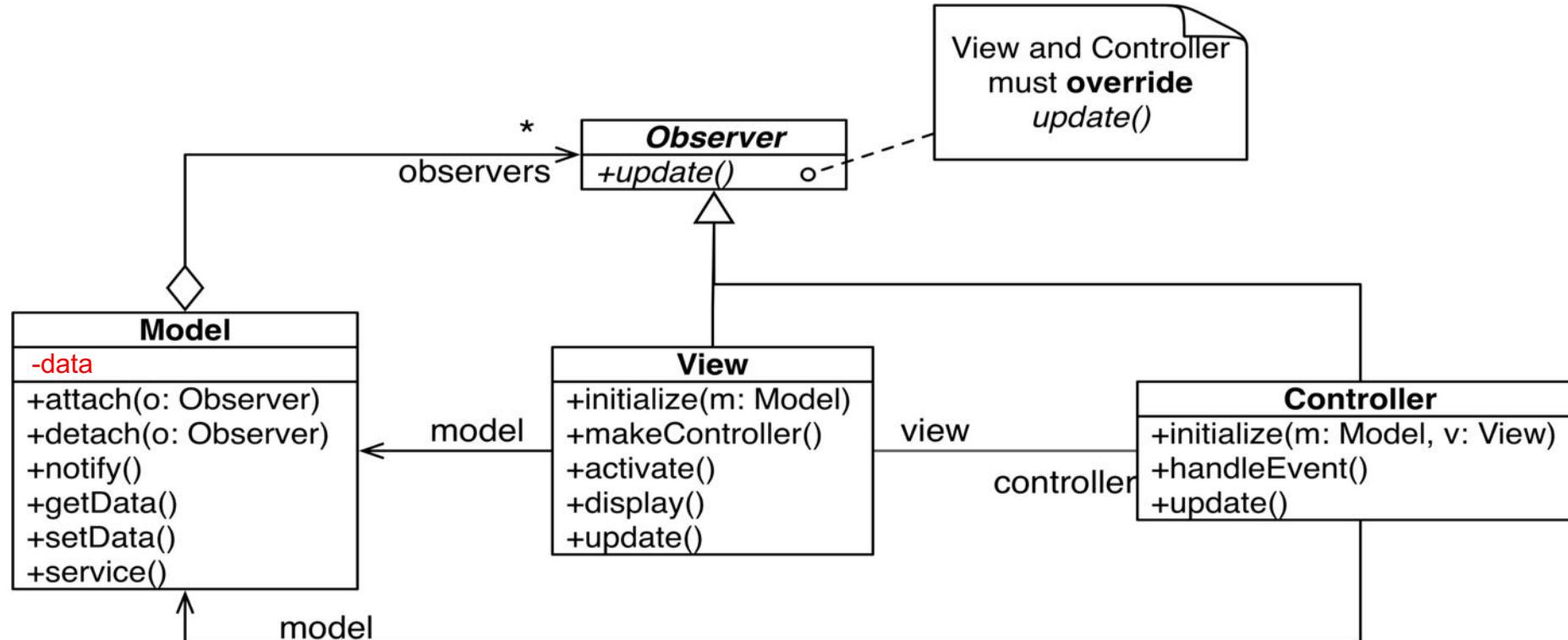
- **Design Pattern:** Describe associations and collaborations of a set of classes (during Object Design)
- **Architectural Style:** A pattern for a subsystem decomposition, i.e. describes relationships and collaborations of different subsystems (during System Design)
- **Software Architecture:** Instance of an architectural style.

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 the sequence of interactions with the user and notifying views of change.



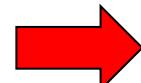
Model View Controller Architectural Style



Adapted from: [Buschmann et. al. 1996].

Instantiating the Model View Controller Architectural Style

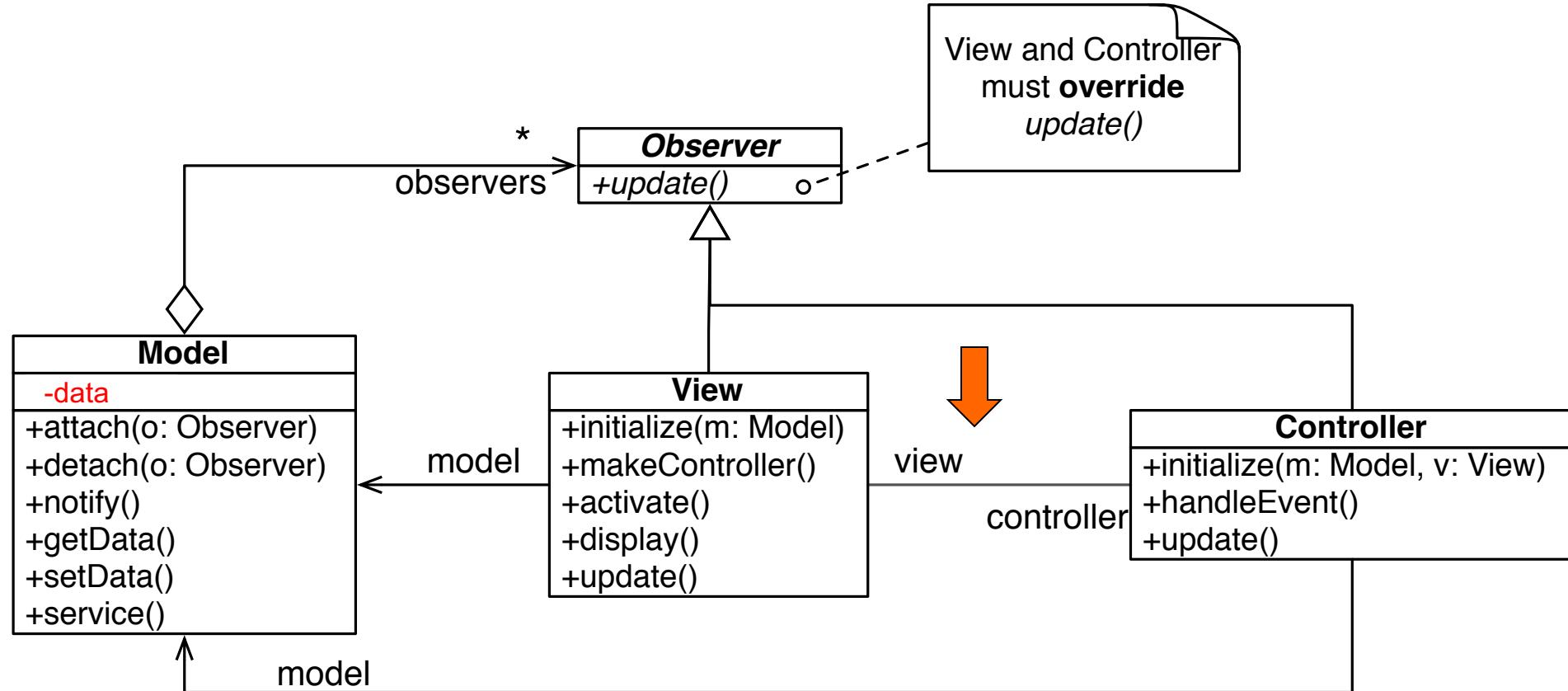
2 Choices:



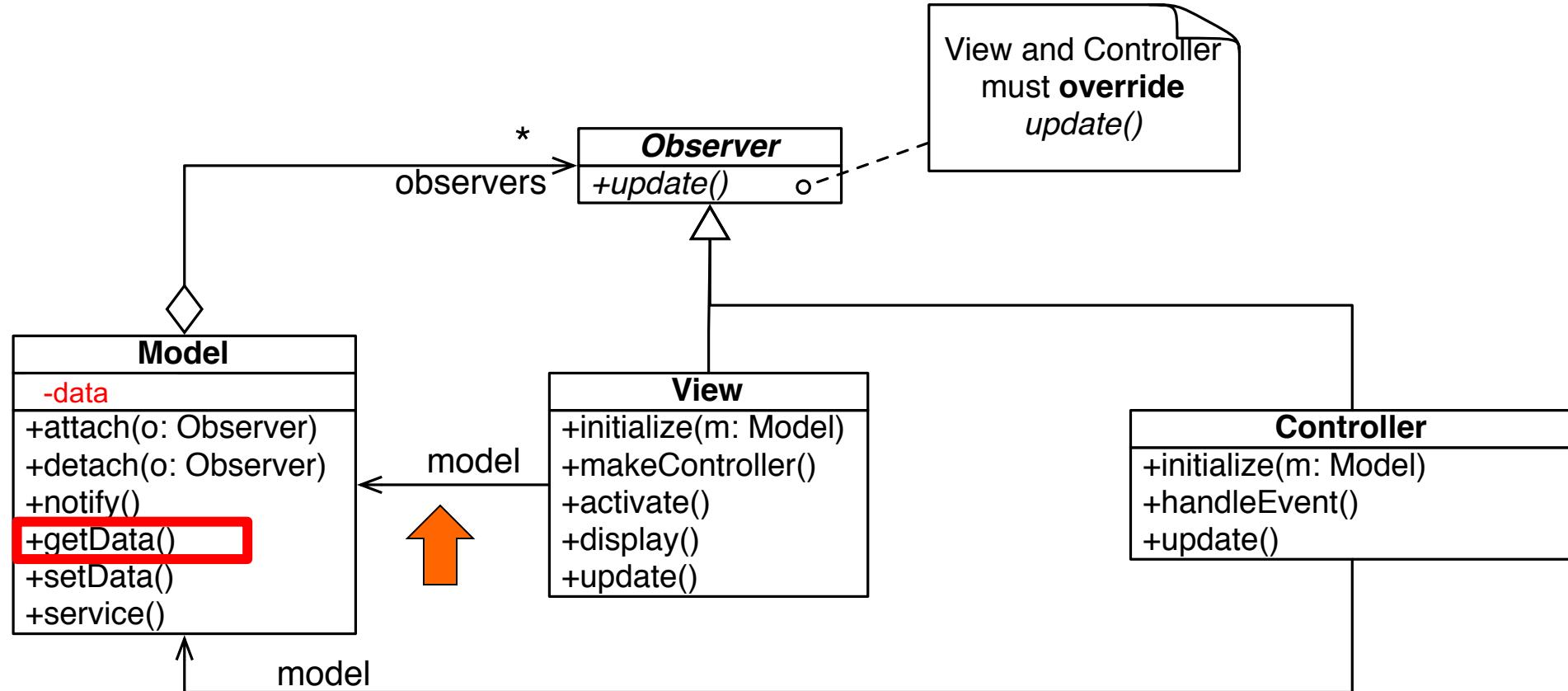
Instantiating the Model View Controller Architectural Style:
Pull Notification Variant

- Instantiating the Model View Controller Architectural Style:
Push Notification Variant

Pull Notification Variant



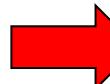
Pull Notification Variant



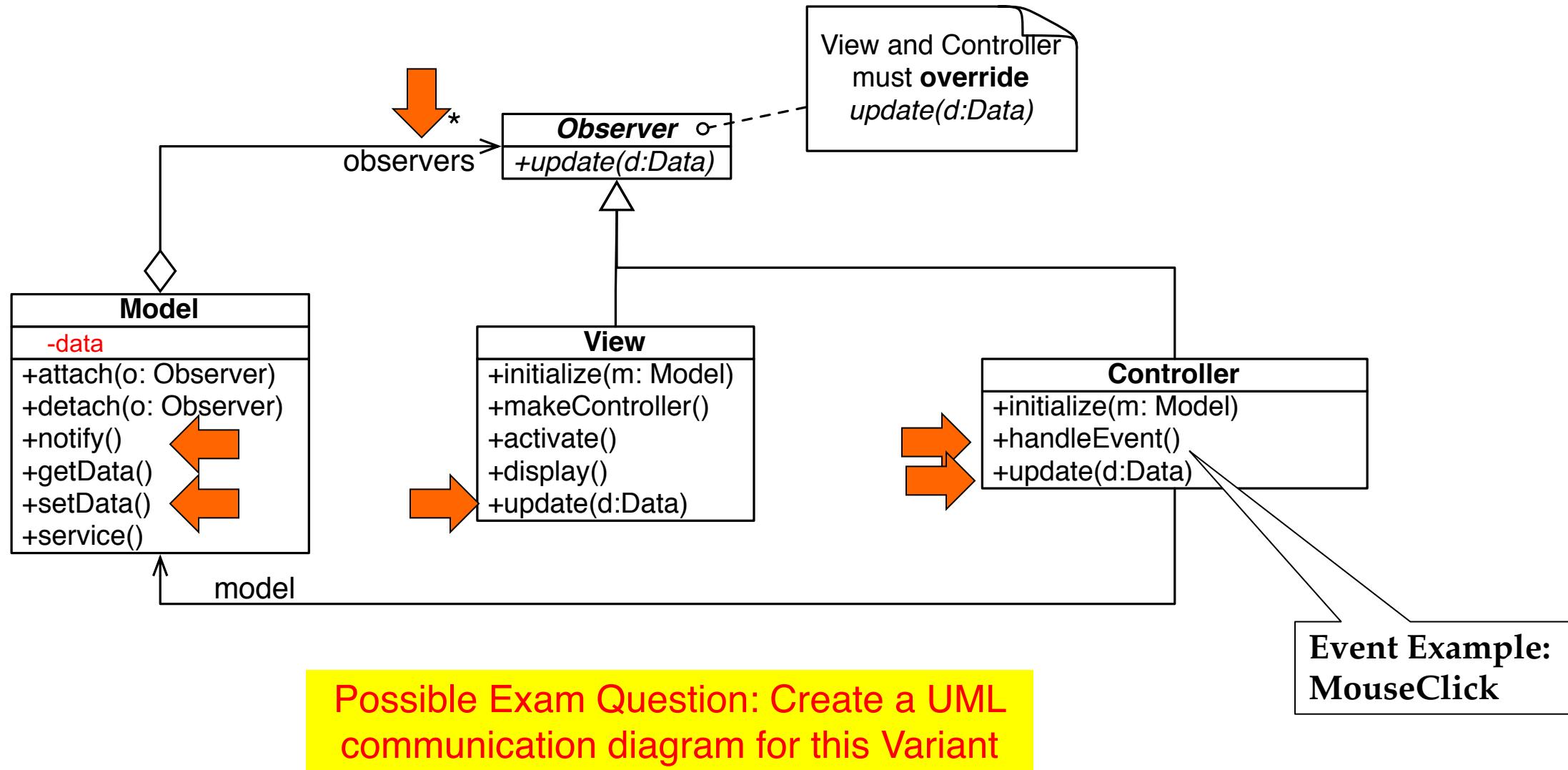
Instantiating the Model View Controller Architectural Style

2 Choices:

- Instantiating the Model View Controller Architectural Style:
Pull Notification Variant

 Instantiating the Model View Controller Architectural Style:
Push Notification Variant

Push Notification Variant



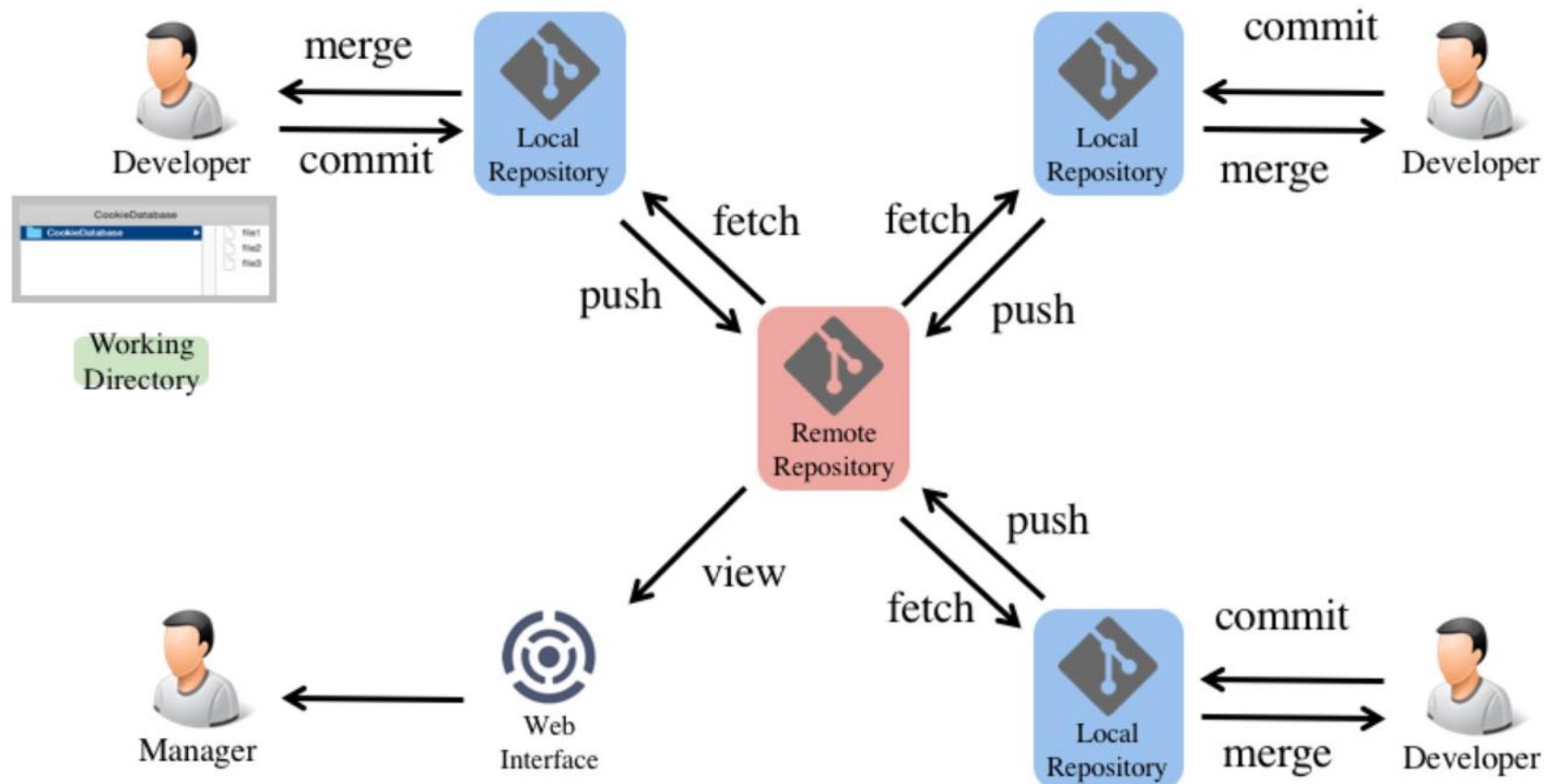
Configuration Management Terminology

- Configuration Item
- Baseline
- SCM Directories
- Version
- Revision
- Release
- Feature Branch
- Continuous Integration
- Continuous Delivery

Configuration Management

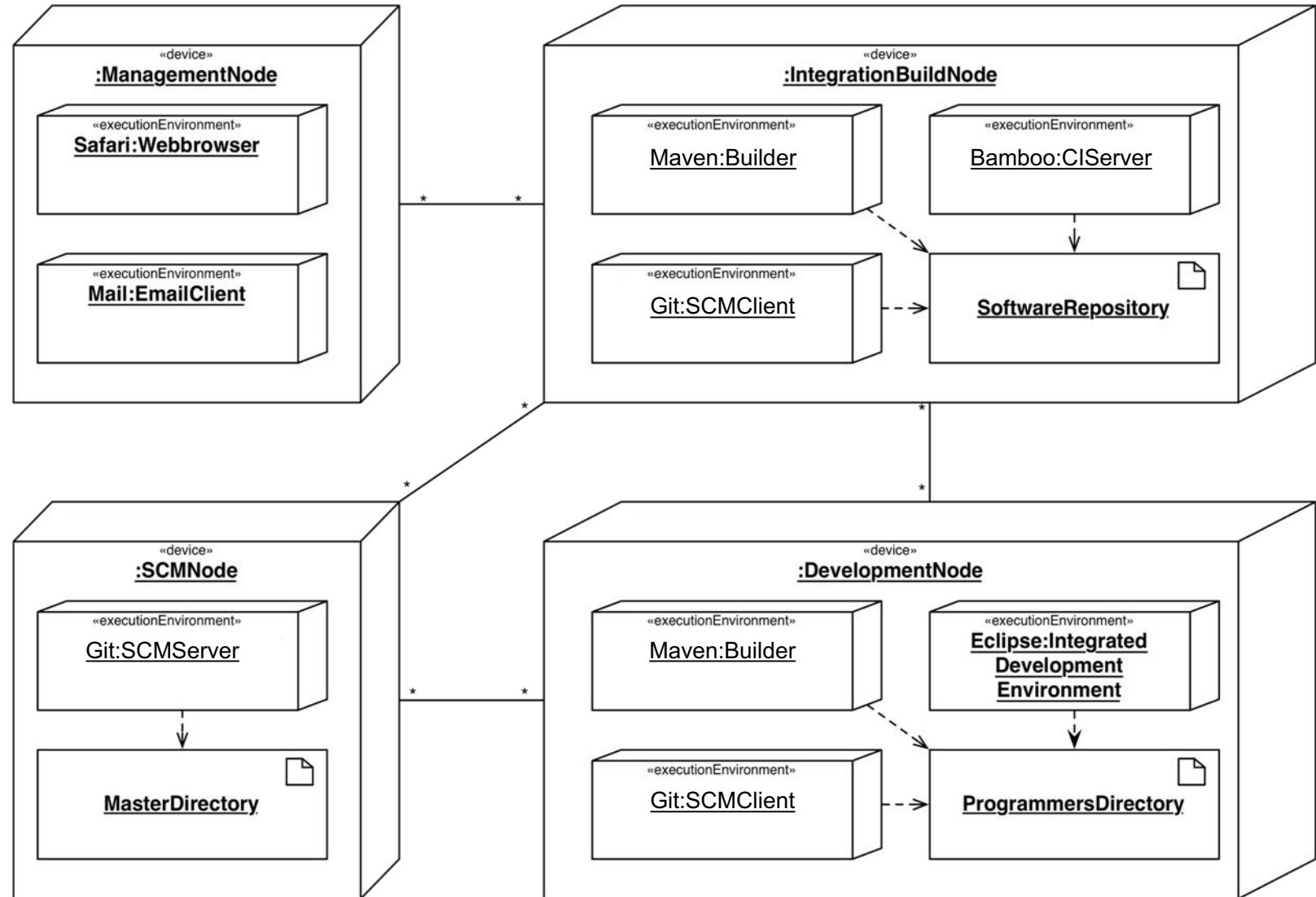
- Configuration Item Identification
- Configuration Control
- Configuration Status Accounting
- Configuration Audits and Reviews

Converting an Informal Model into a UML Diagram



Possible Exam Question: Create a UML Sequence diagram for this Informal Model

Deployment Diagram of a Continuous Integration System



Modeling the Software Lifecycle

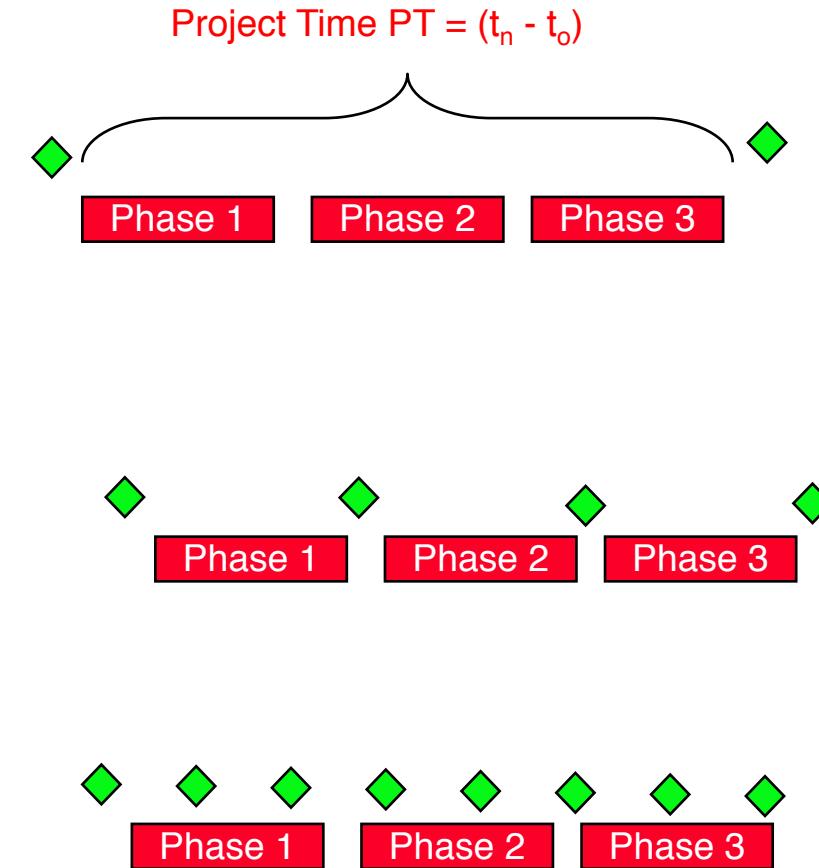
- IEEE Standard 1074-2006 for Software Lifecycles
- Software life cycle models
 - **Sequential models**
 - Waterfall model
 - V-model
 - **Iterative models**
 - Boehm's spiral model
 - V-Model XT
 - Unified Process
 - **Agile models**
 - Scrum
 - Kanban

Software Lifecycle Terminology

- Software Lifecycle
- Software Lifecycle Model
- IEEE Std 1074 Standard
- Tailoring
- Activity-centered vs. entity-centered views
- Iterative vs. incremental development
- Types of prototypes, types of prototyping
- Unified Process

Change influences Choice of the Lifecycle Model

- No change during project
 - MTBC » Project Time PT
 - Linear Model: Waterfall model, V-model
- Infrequent changes during the project
 - MTBC ≈ Duration of Phase
 - Iterative Model: Spiral model, Unified Process
- Changes are frequent
 - MTBC < Project Time PT
 - Agile Model: Scrum.



Incremental vs Iterative vs Adaptive

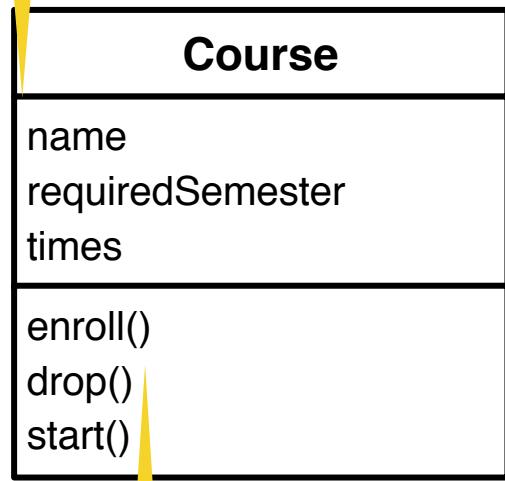
- **Incremental** means to “**add onto something**”
 - Incremental development improves your **process**
- **Iterative** means to “**re-do something**”
 - Iterative development improves your **product**
- **Adaptive** means “**react to changing requirements**”
 - Adaptive development improves the reaction to changing customer needs.

Object Design Topics

- Requirements Analysis vs. Object Design
- Component-Based Software Engineering
- COTS-Development
- Reuse (White box vs black box reuse)
- Generalizations and Specification
- Delegation vs Inheritance
- The use of inheritance in object design
 - Implementation Inheritance vs Specification Inheritance

Distinction between Analysis Object Model and Object Design Model

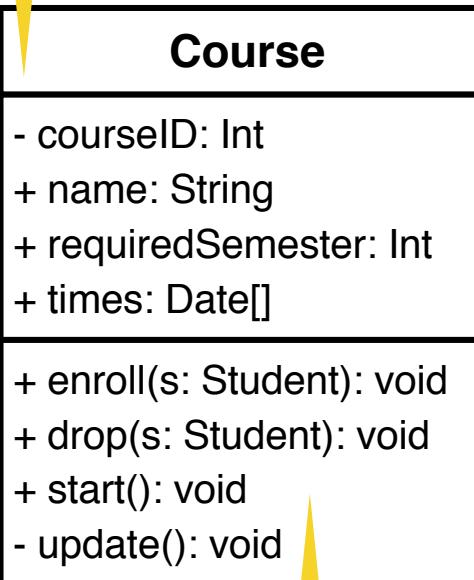
During **analysis**: attributes and methods **without** visibility information



During **analysis**: methods **without** signature

Analysis Object Model
(Application Domain Language)

During **object design**: we specify the visibility for each attribute and method



During **object design**: specify the signature of each method

Object Design Model
(Solution Domain Language)

- public class Course {
- private Integer courseId;
- public String name;
- public Integer requiredSemester;
- public Date[] times;
- }
- public void enroll(Student s)
- {...}
- public void drop(Student s) {...}
- public void start() {...}
- private void update() {...}
- }

type

signature

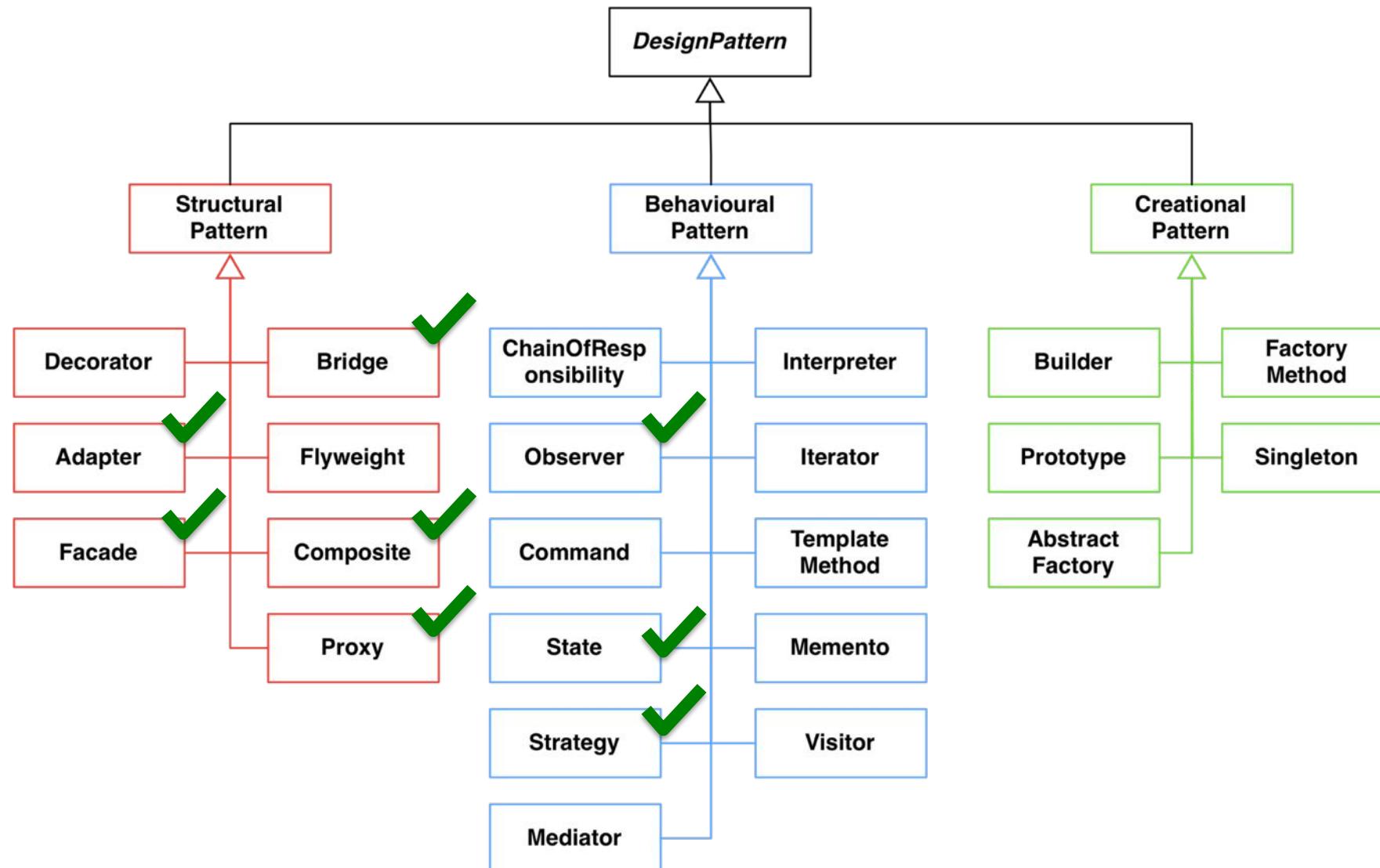
visibility

Implementation
(Java)

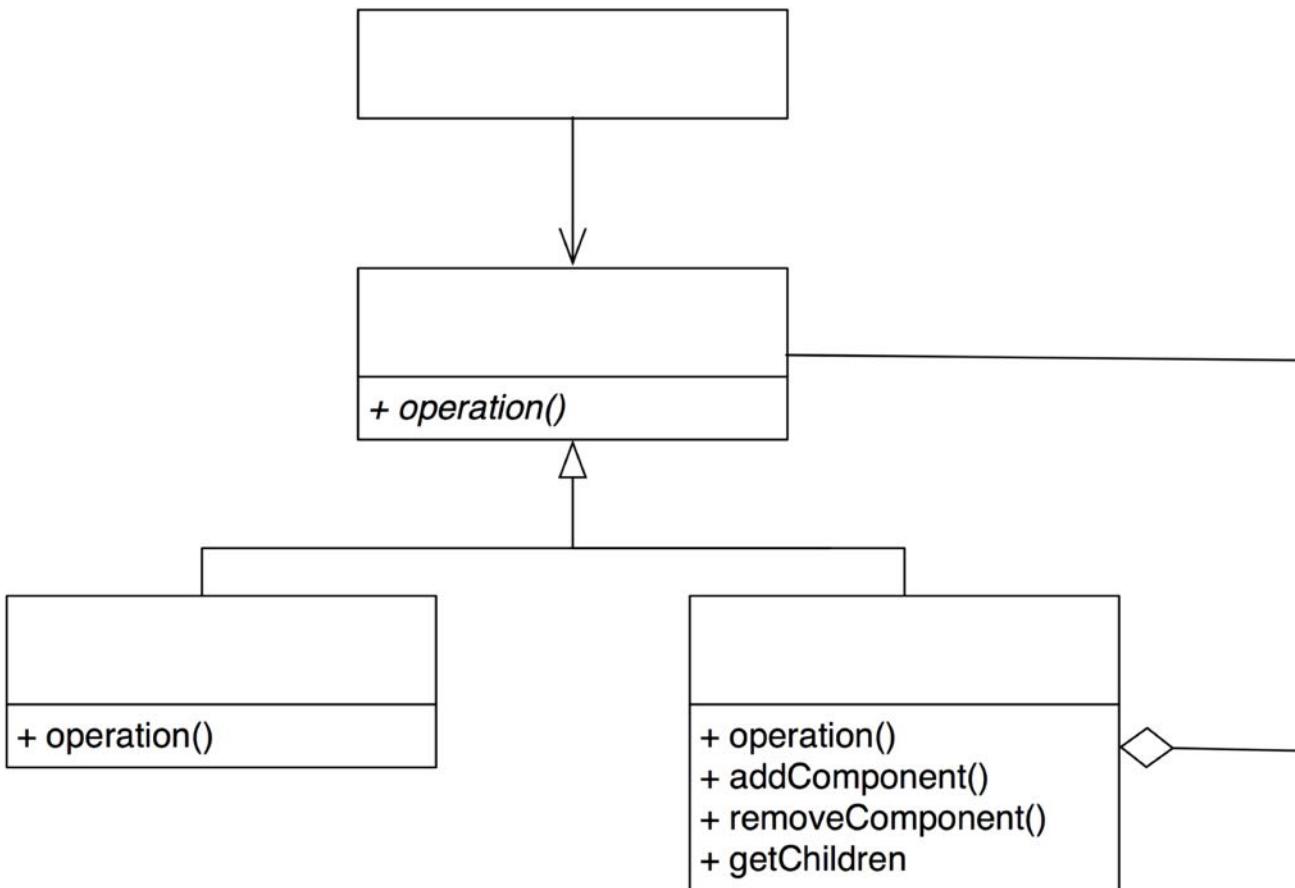
Design Patterns

- **Structural Patterns**
 - Focus: Composing objects to form larger structures
 - Problems solved:
 - Realize new functionality from old functionality
- **Behavioral Patterns**
 - Focus: Assignment of responsibilities to objects
 - Problem solved: Tight coupling to particular algorithms
- **Creational Patterns**
 - Focus: Creation of complex objects
 - Problems solved: Hide how objects are created or put together

Overview of Design Patterns covered in EIST



Know all the Patterns Covered in Class



Possible Exam Questions:

- 1) Name the Pattern
- 2) Fill in the blanks spaces in the Class Diagram

Typical Design Goals Addressed by Design Patterns

- **Extensibility (Expandability)**
 - A system is extensible, if new functional requirements can easily be added to the existing system
- **Customizability**
 - A system is customizable, if new nonfunctional requirements can be addressed in the existing system
- **Scalability**
 - A system is scalable, if existing components can easily be multiplied in the system
- **Reusability**
 - A system is reusable, if it can be used by another system without requiring major changes in the existing system model (design reuse) or code base (code reuse).

Controlling Software Development

How do we control software development? 2 opinions:

- Through **organizational maturity**
 - Repeatable process
- Through **agility**
 - Large parts of software development is empirical in nature; they cannot be modeled with a defined process
- How can software development best be described?
 - with a defined process control model
 - with an empirical process control model.

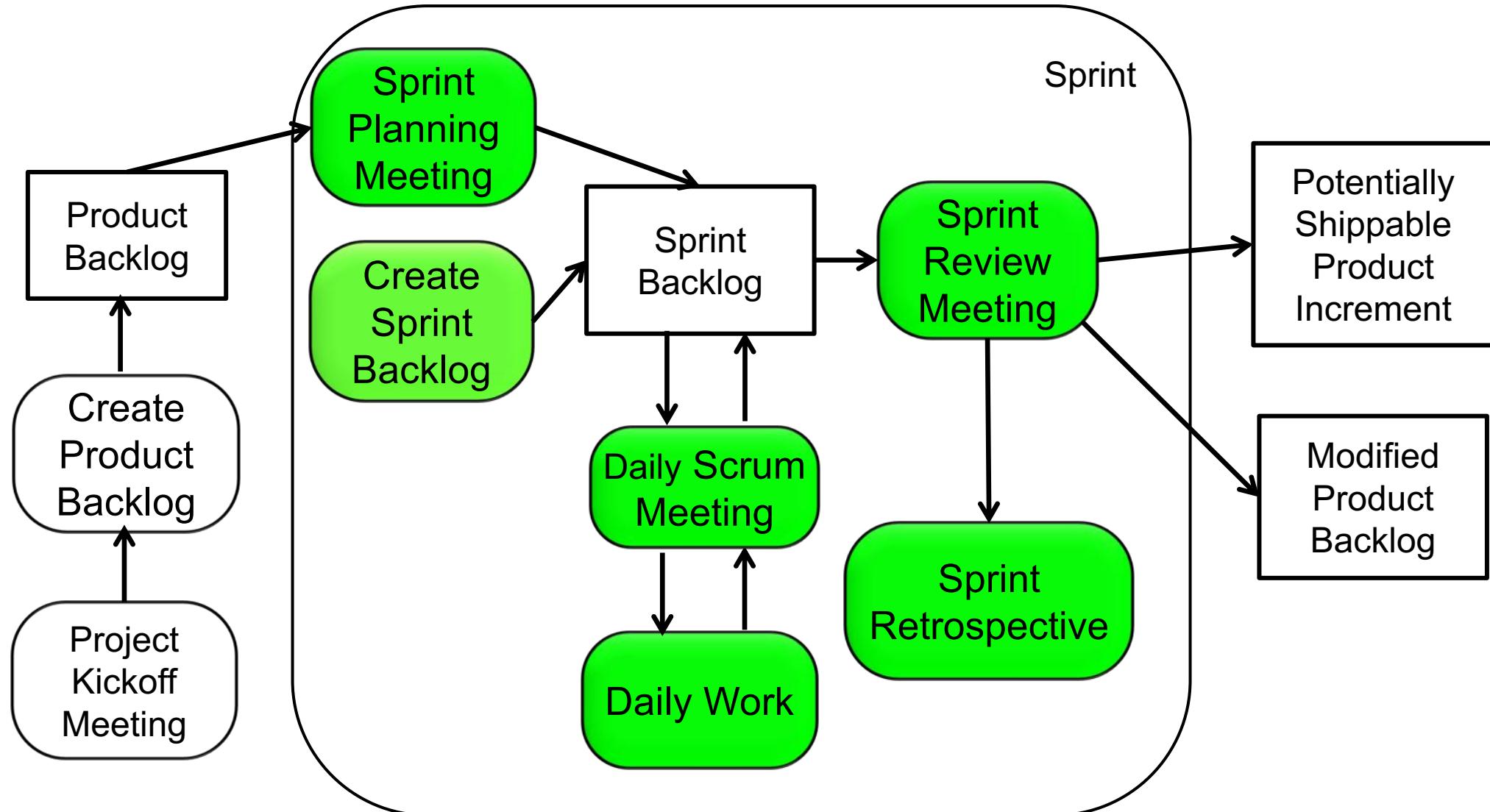
Defined Process Control Model

- Requires that every piece of work is completely understood
- Deviations are seen as errors that need to be corrected
- Given a well-defined set of inputs, the same outputs are generated every time
- Precondition to apply this model:
 - All the activities and tasks are well defined to provide repeatability and predictability
- If the preconditions are not satisfied:
 - Lot of surprises, loss of control, incomplete or wrong work products.

Empirical Process Control Model

- Empirical process
 - An imperfectly defined process, not all pieces of work are completely understood
- Deviations, errors and failures are seen as opportunities that need to be investigated
 - The empirical process “expects the unexpected”
- Control and risk management is exercised through frequent inspection
- Condition when to apply this model:
 - Change is frequent and cannot be ignored
 - Change of requirements, change of technology, change in the organization, people change too.

Overview of Scrum Activities and Entities



Agile Manifesto: Dealing with Uncertainty, Complexity and Change

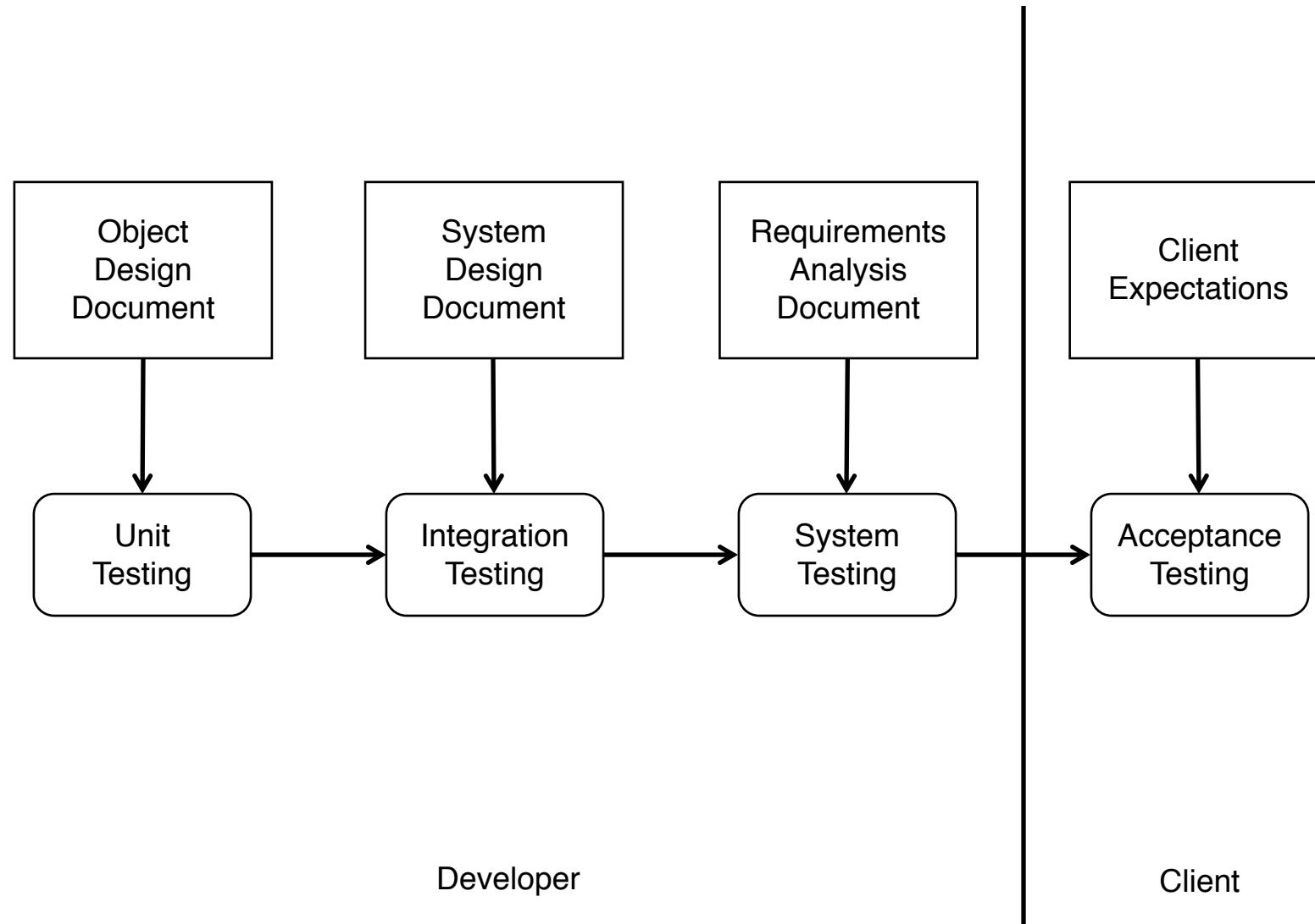
Individuals and Interactions	Processes and Tools
Working Software	Comprehensive Documentation
Customer Collaboration	Contract Negotiation
Responding to Change	Following a Plan

Testing Terminology

- **Failure:** Any deviation of the observed behavior from the specified behavior (also often called “crash”)
- **Fault:** The mechanical or algorithmic cause of an error (often called “bug”)
- **Error** (also called erroneous state): The system is in a state such that further processing by the system can lead to a failure
- **Verification:** Activity that checks if the observed behavior complies with the specified behavior of a system
- **Validation:** Activity that checks if the observed behavior meets the needs informally expressed by a stakeholder (customer, end user)

- Unit testing
- Types of Integration testing
- System testing
- Continuous integration.

Model-based Testing Activities



Unit Testing

Problem Statement: Storing, adding, and subtracting **money** in a computer currently is not possible. We can do these operations only on **integers**. As a result we might accidentally add 5 Euros (€) and 7 US Dollars (\$), which is of course invalid

Solution: Create a Money class that provides the currency abstraction (encapsulating amount and type of currency)

Functional Requirements:

1. Store an amount and currency
2. Add money with the same currency
3. Return **Null** if the addition is invalid (e.g. 6 Euro + 5 US Dollars is invalid).

A Unit Test for the add() method

```
import org.junit.Test;  
  
public class MoneyTest {  
    @Test public void simpleAdd() {  
        Money m12CHF= new Money(12, "CHF");  
        Money m14CHF= new Money(14, "CHF");  
        Money expected= new Money(26, "CHF");  
        Money observed= m12CHF.add(m14CHF);  
    }  
}
```

@Test annotation:
simpleAdd() is the
name of the test
case in the test
driver

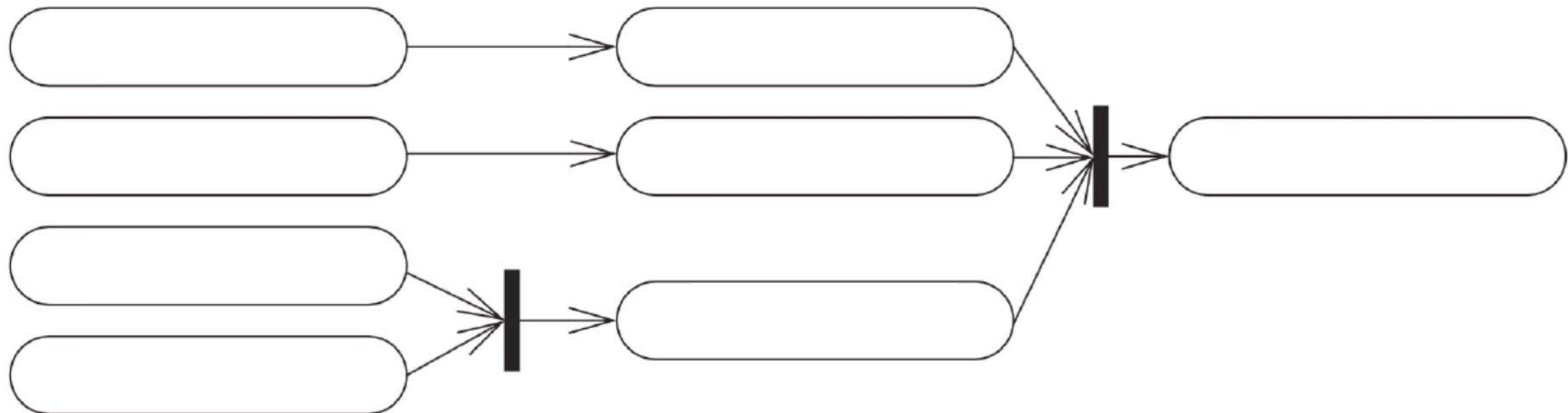
**Study Java annotations
Java assertions!**

MoneyTest is a test
driver in the Test
Model

This is still an incomplete unit
test. The Java annotation is
missing!

Here we are calling the
SUT Method add() from
the System Model

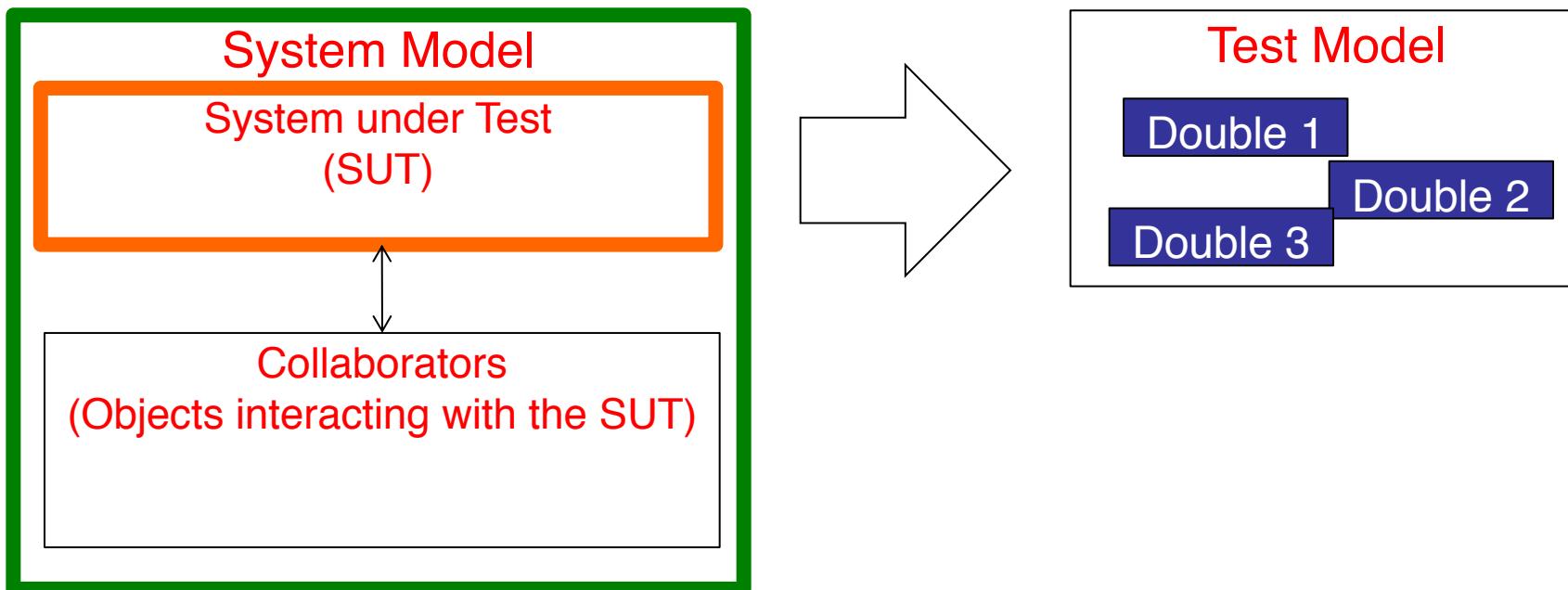
Integration Strategies



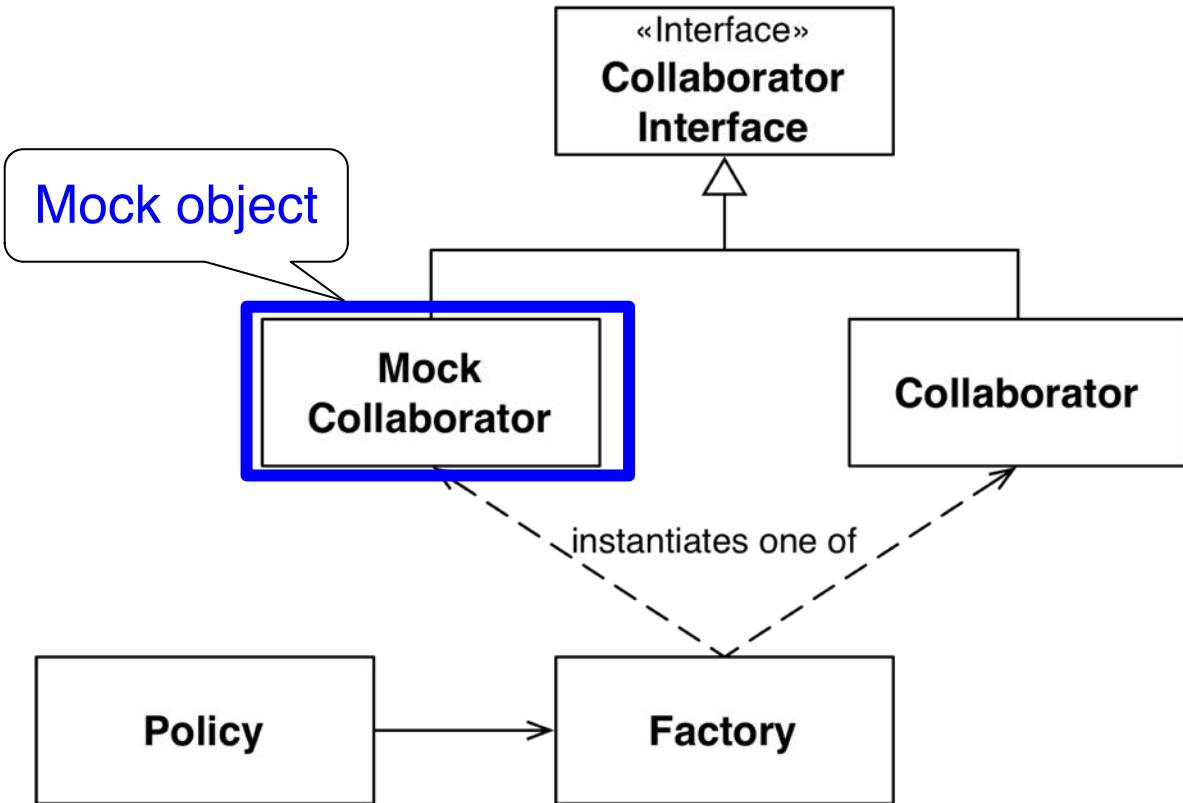
Possible Exam Question: Given this UML Activity Diagram, name the Integration Strategy

Object-Oriented Model-Based Testing

- We start with the **system model**
- The system contains the **SUT** (the unit we want to test)
- The SUT does not exist in isolation, it collaborates with other objects in the system model (called Collaborators)
- The test model is derived from the SUT
- To be able to interact with collaborators, we add *objects to the test model*
- These objects are called **test doubles**



Mock-Object Pattern



- In the mock object pattern a **mock object** replaces the *behavior* of a real object called the collaborator and returns hard-coded values
- A mock object can be created at startup-time with the factory pattern (Look it up in Gamma's book)
- Mock objects can be used for testing *state of individual objects* as well as the *interaction between objects*, that is, to validate that the interactions of the SUT with its collaborators behave as expected
- The use of Mock objects is based on the **Record-Play Metaphor**.

Methodologies

- Software methodologies provide
 - Guidance and general principles for dealing with complexity, change and uncertainty
 - Strategies for selecting methods and tools in a given project environment
 - Guidance what to do when things go wrong
- Key questions in a methodology
 - How much involvement of the customer?
 - How much planning?
 - How much reuse?
 - How much modeling?
 - How much process?
 - How much control and monitoring?

Model-driven Software Development

Some acronyms for model-driven software development

- **MDE (Model-Driven Engineering)**
 - A development methodology which focuses on creating and exploiting domain models (application domain models, solution domain models). Usually used when the development consists of hardware and software components
- **MDD (Model-Driven Development)**
 - It is impossible to keep a model consistent automatically after a manual change of the generated code. Manual changes to generated code should be avoided. MDD forbids round-trip engineering
- **MDA (Model-Driven Architecture)**
 - The system functionality is defined using a platform-independent model (PIM) and an appropriate domain-specific language (DSL). The MDA acronym is used mainly by the Object Management Group (OMG).

Possible Questions

- Project Definition
- Roles and Binding Roles to People
- Tasks, Activities and Projection Functions
- Project Organizations
- Project Organization Structures
- Communication Event vs. Mechanism
- Modeling a Project with a UML State Chart
- Typical Mistakes when starting a Project
- Difference between Delegation and Responsibility

Final Thoughts and Suggestions

- If a topic is not mentioned here, that does not imply that it might not appear in the exam
- Form a study group
 - Post problems that others in the group must solve
 - Rotate questioner and answerer
- Questions can have one of these two forms:
 - Here is a problem. Analysis and propose a solution from your perspective of a manager
 - What is X? Name the advantages and disadvantages of X
 - One more thing

What is necessary to pass the exam?

- 4 Types of Exam takers
 - Perfectionist
 - Winger
 - Fatalist
 - Clever exam taker

What is necessary to pass the exam?

- Exam Taking Approaches
 - **Perfectionist Approach**
 - I will be prepared for any type of question, I want to be a master of the field
 - **Fatalist Approach**
 - I don't need to study. I will be able to answer the questions, because I know am strong
 - **Winger Approach**
 - I will deal with unforeseen questions as they come.

Perfectionist

- I will not sleep until I take the exam
- I have to know every possible move the exam can take



Fatalist

- I don't know what questions will come. I will just wait for the questions and answer them as they come towards me.



Winger

- I can deal with any type of question.



What is necessary to pass the exam?

- Exam Taking Approaches
 - **Perfectionist:**
 - I will be prepared for any type of question, I want to be a master of the field
 - **Fatalist:**
 - I don't need to study. I will be able to answer the question, because I know am strong
 - **Winger:**
 - I will deal with unforeseen questions as they come
- Best approach: Seasoned exam taker



Final Thoughts

If you know the solution to a problem make sure you do not apply the solution to the wrong object

Good Luck!