



# Software Engineering

0. Course Organization | Thomas Thüm | October 7, 2021

# About Me

- My name is Thomas  
(avoid Prof./Mr. Thüm please)
- Programming since 1998
- Contributing to FeatureIDE since 2007
- 2004–2010: studied Computer Science  
(minor subject Mathematics)
- 2010–2015: PhD student in Magdeburg
- 2015–2019: PostDoc in Braunschweig
- Since 2020: Professor for the Construction  
and Analysis of Secure Software Systems



# Institute of Software Engineering and Programming Languages

## Professors



Prof. Dr. Matthias Tichy

Professor  
E-Mail: matthias.tichy@uni-sie.de  
URL zur Website  
Raum: Q27 420  
Telefon: +49(2150)24.900



Prof. Dr. Thoralf Röhrwirth

Professor  
E-Mail: thoralf.roehrwirth@uni-sie.de  
URL zur Website  
Raum: Q27 418  
Telefon: +49(2150)24.903



Prof. Dr. Thomas Thüm

Professor  
E-Mail: thomas.thuem@uni-sie.de  
URL zur Website  
Raum: Q27 416  
Telefon: +49(2150)24.904

## Scientific Employees



M.Sc. Paul-Axelmann Wöhrel  
Research Assistant  
E-Mail: paul.axelmann@uni-sie.de  
URL zur Website  
Raum: Q27 v12  
Telefon: +49(2150)24.900



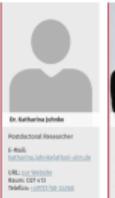
Florian Stoeni Egz  
Research Assistant  
E-Mail: florian.stoeni@uni-sie.de  
URL zur Website  
Raum: Q27 v10  
Telefon: +49(2150)24.900



M.Sc. Hoffmeier Gruner  
Research Assistant  
E-Mail: hoffmeier.gruner@uni-sie.de  
URL zur Website  
Raum: Q27 v12  
Telefon: +49(2150)24.900



M.Sc. Tobias Höfl  
Research Assistant  
E-Mail: tobias.hoefl@uni-sie.de  
URL zur Website  
Raum: Q27 v12  
Telefon: +49(2150)24.900



Dr. Katharina Jähnle  
Postdoctoral Researcher  
E-Mail: katharina.jaehnle@uni-sie.de  
URL zur Website  
Raum: Q27 v12  
Telefon: +49(2150)24.900



M.Sc. Dennis Heimiller  
Research Assistant  
E-Mail: dennis.heimiller@uni-sie.de  
URL zur Website  
Raum: Q27 v10  
Telefon: +49(2150)24.900



M.Sc. Jakob Pfeifer  
Research Assistant  
E-Mail: jakob.pfeifer@uni-sie.de  
URL zur Website  
Raum: Q27 v12  
Telefon: +49(2150)24.900



Dr. Alexander Rischke  
Research Associate  
E-Mail: alexander.rischke@uni-sie.de  
URL zur Website  
Raum: Q27 v10  
Telefon: +49(2150)24.900

## Secretary & Technician



Carolin Göring

Secretary  
E-Mail: carolin.goering@fakultu.uni-sie.de  
URL zur Website  
Raum: Q27 419  
Telefon: +49(2150)24.900



Kathrin Denßler

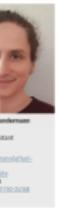
Secretary  
E-Mail: kathrin.denessler@uni-sie.de  
URL zur Website  
Raum: Q27 419  
Telefon: +49(2150)24.903



M.Sc. Sascha Reichenberger  
Research Assistant  
E-Mail: sascha.reichenberger@uni-sie.de  
URL zur Website  
Raum: Q27 v10  
Telefon: +49(2150)24.900



M.Sc. Michael Voigtmeier  
Research Assistant  
E-Mail: michael.voigtmeier@uni-sie.de  
URL zur Website  
Raum: Q27 v10  
Telefon: +49(2150)24.900



M.Sc. Chico Sonnenburg  
Research Assistant  
E-Mail: chico.sonnenburg@uni-sie.de  
URL zur Website  
Raum: Q27 v12  
Telefon: +49(2150)24.900



M.Sc. Thomas Wölfe  
Research Assistant  
E-Mail: thomas.woelfe@uni-sie.de  
URL zur Website  
Raum: Q27 v12  
Telefon: +49(2150)24.900

# Bachelor Courses of Our Institute

Key	<input checked="" type="checkbox"/> Lecture	<input checked="" type="checkbox"/> (Pro-)Seminar	<input checked="" type="checkbox"/> Project	<input checked="" type="checkbox"/> Thesis		
Lecturer	<input checked="" type="checkbox"/> Tichy	<input checked="" type="checkbox"/> Frühwirth	<input checked="" type="checkbox"/> Thüm	<input checked="" type="checkbox"/> Raschke	<input checked="" type="checkbox"/> Juhnke	<input checked="" type="checkbox"/> misc
	Winter Term	Summer Term	infrequent	always		
Master & Bachelor	Functional Programming RASCHKE	Constraint Programming FRÜHWIRTH	Rule-based and Constraint Programming FRÜHWIRTH			
	Management of Software Projects HOUDEK					
	Rule-based and Constraint Programming FRÜHWIRTH					
	Web Engineering TICHY					
Bachelor	Software Engineering I+II THÜM		Research Trends in Software Engineering Logic-based Programming Languages		Bachelor Thesis	
	Software Project THÜM		Software Engineering Anwendungsprojekt VMA			
	Introduction to Computer Science	Programming Paradigms RASCHKE	Anwendungsprojekt MDSD			
		Interactive Systems Programming TICHY				

# Master Courses of Our Institute

Key	<input checked="" type="checkbox"/> Lecture	<input checked="" type="checkbox"/> (Pro-)Seminar	<input checked="" type="checkbox"/> Project	<input checked="" type="checkbox"/> Thesis
Lecturer	<input checked="" type="checkbox"/> Tichy	<input checked="" type="checkbox"/> Frühwirth	<input checked="" type="checkbox"/> Thüm	<input checked="" type="checkbox"/> Raschke
	Winter Term	Summer Term	infrequent	always
Master	Compiler Construction Thüm	Parallel Programming (in german) RASCHKE	Hot Software Engineering Topics in Research and Practice Model-Driven Software Engineering TICHY	Master Thesis SE-Projekt A SE-Projekt B
	Empirische Forschungsmethoden der Informatik (in german) JUHNKE	Software-Produktlinien (in german) THÜM	Software Quality Assurance TICHY	
	Principles of Program Analysis FRÜHWIRTH		Application of modern principles of software engineering in software development Individual Project Rule-based and Constraint Programming Research Trends in Software Engineering	

# Current Courses of Our Institute

## Bachelor

- Software Engineering I+II  
(Softwaretechnik I+II)
- Software Project  
(Softwaregrundprojekt)
- Proseminar on Logic-Based Programming Languages  
(Logikbasierte Programmiersprachen)
- Seminar on Rule-Based and Constraint Programming  
(Regelbasierte und Constraint-Programmierung )
- Numerous Projects

## Master

- Compiler Construction  
(Compilerbau)
- Empirical Research Methods in Computer Science  
(Empirische Forschungsmethoden der Informatik)
- Functional Programming  
(Funktionale Programmierung)
- Management of Software Projects  
(Management von Softwareprojekten)
- Hot Software Engineering Topics in Research and Practice  
(Aktuelle Themen der Softwaretechnik aus Forschung und Praxis)
- Web Engineering
- Rule-Based Programming  
(Regelbasierte Programmierung)

# Software Engineering I + II

## The Facts

- German title: Softwaretechnik I + II
- Abbreviation: SE (aka. SWT)
- Credits: 6 ECTS (60h+120h=180h)
- Semester hours: 4
- Requirements (contentual):  
Interactive Systems Programming  
[\(Programmierung von Systemen\)](#)
- Courses of studies:
  - ▶ B.Sc.: Informatik, Medieninformatik,  
Software Engineering,  
Informationssystemtechnik
  - ▶ M.Ed.: Informatik Lehramt
  - ▶ ...

## Course Organization

- Weekly lecture over winter and summer term
- Written exam at the end of the summer term
- No formal exercises, but small tasks within the lecture
- In parallel: Software Project  
[\(Softwaregrundprojekt\)](#)

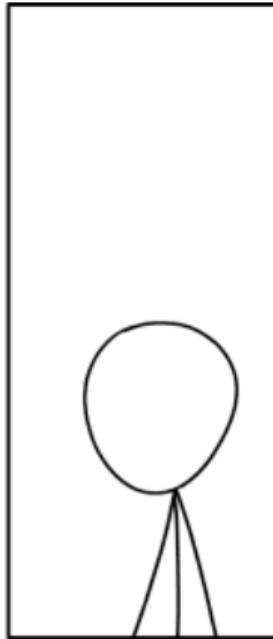
HI THERE! WE'RE THE VIRUSES THAT CAUSE THE COMMON COLD.  
THIS HANDWASHING...  
IT STOPS WHEN THIS IS ALL OVER, RIGHT?



IT'S JUST, IT'S MAKING THINGS REALLY HARD FOR US, TOO.  
MAYBE WE COULD MAKE A DEAL?



WE WON'T KILL YOU!  
WE JUST WANT TO GET BACK IN YOUR THROAT AND MAKE YOU FEEL GROSS NOW AND THEN.



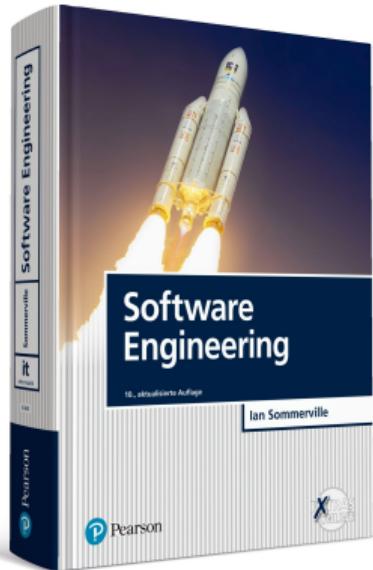
# Corona-Update

## Online-Format in Winter Term

- no physical lecture
- videos available in Moodle and on Youtube
- extensive use of Moodle:  
small tasks, your questions, your answers
- use tutorial groups of software project for  
questions
- watch videos together!
- do interactive parts together!



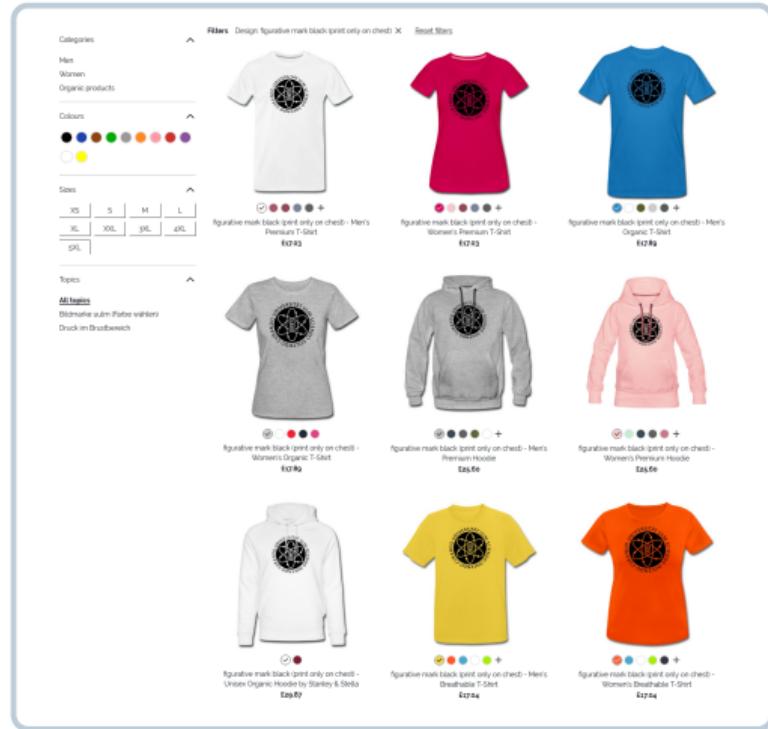
# Literature



[Sommerville]

- Ian Sommerville.  
Software Engineering, 10.  
Edition, Pearson, 2018.
  - ▶ German, English,  
and earlier versions
  - ▶ Videos by Ian  
Sommerville and  
others available  
online
- More literature  
announced in each lecture

# Bug Bounty Program



## Bug Bounty Program

- Lecture created from scratch in 2020/2021
- Expect inconsistencies and errors
- Please report problems in Moodle:  
<https://moodle.uni-ulm.de/mod/moodleoverflow/view.php?id=420071>
- Bug Bounty: significant contributions honored
- Most active students in Moodle also honored

# Questions?

## How to Find Answers

1. Check information in Moodle:  
<https://moodle.uni-ulm.de/course/view.php?id=26062>
2. Check already answered questions in Moodle overflow
3. Ask your own questions and answer questions of fellow students
4. Meet me in my consultation hour:  
Wednesday 1–2pm in Zoom <https://uni-ulm.zoom.us/my/thomas.thuem>
5. Contact me via [thomas.thuem@uni-ulm.de](mailto:thomas.thuem@uni-ulm.de)

## Foren



Organisatorische Fragen



Interaktive Aufgaben und inhaltliche Fragen



Feedback und Bug Bounty





# Software Engineering

1. Introduction | Thomas Thüm | October 20, 2021

# Lecture Overview

1. What is Software?
2. How Relevant is Software?
3. What is Software Engineering Good For?

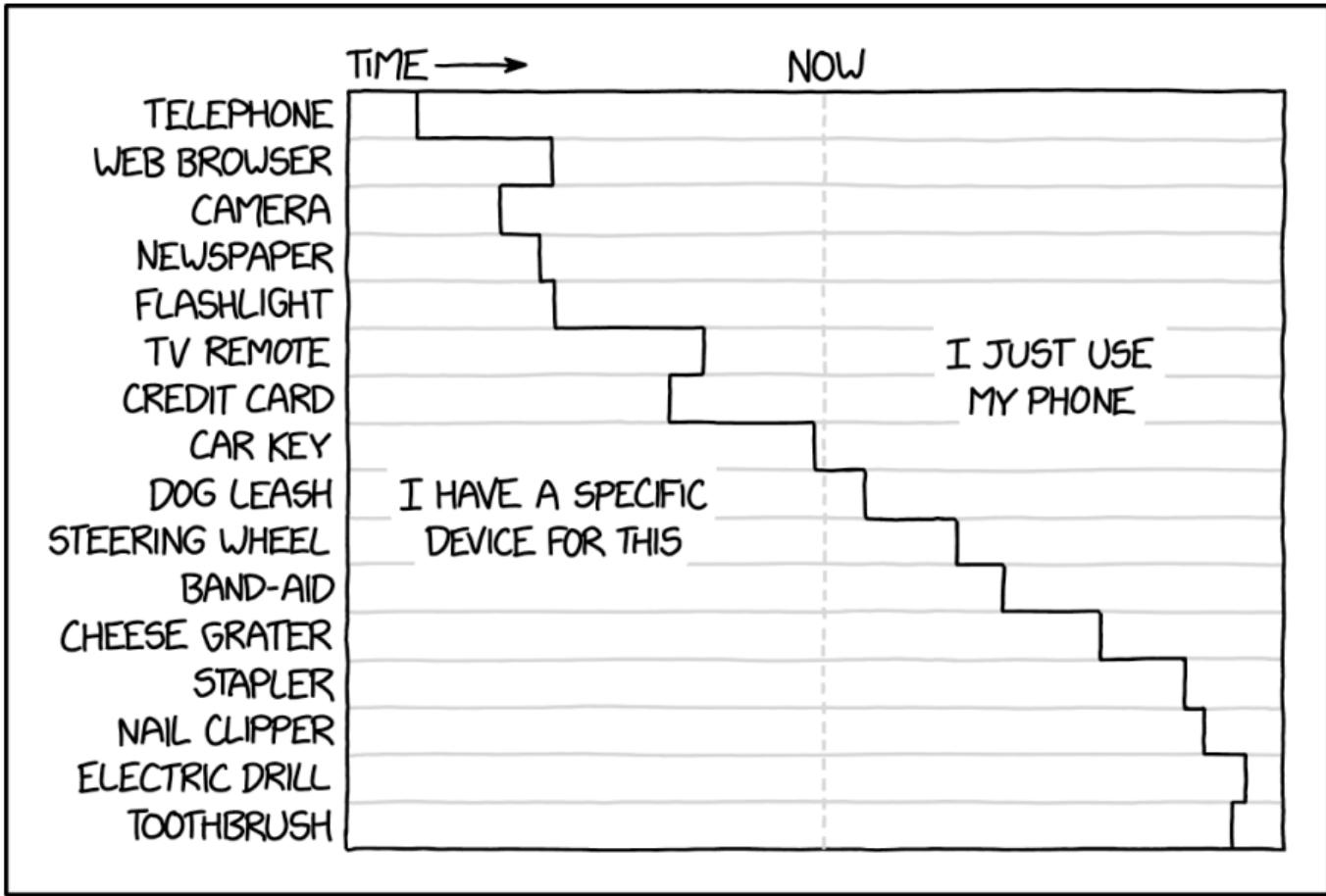
# Lecture Contents

## 1. What is Software?

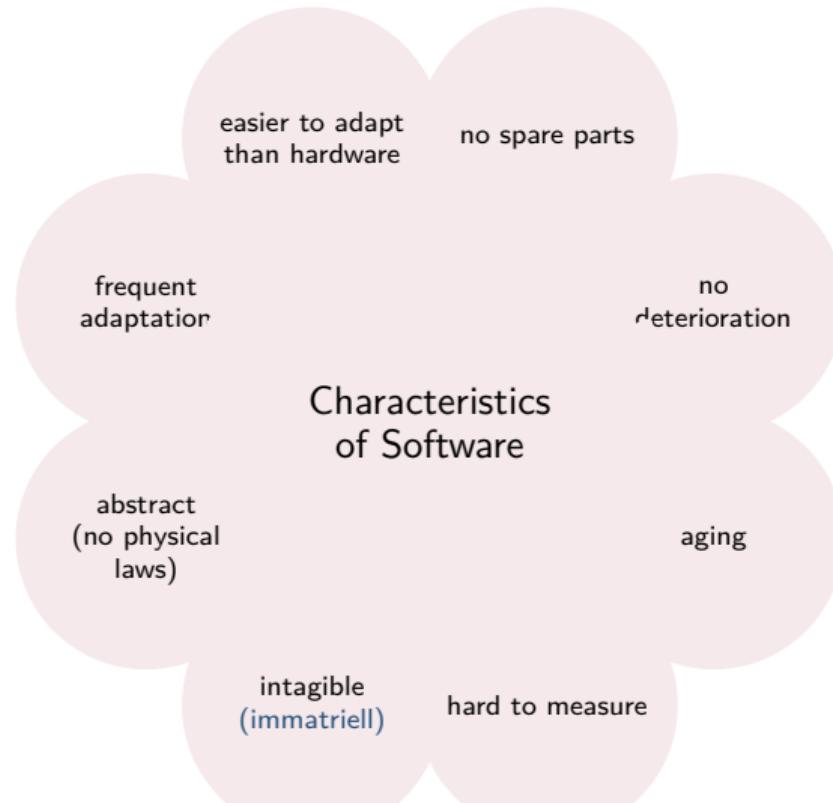
- Characteristics of Software
- Software
- Software Products
- Application and System Software
- Application Software
- Properties of Software
- Lessons Learned

## 2. How Relevant is Software?

## 3. What is Software Engineering Good For?



# Characteristics of Software



# Software

## Software

[adapted from **Sommerville**]

Software stands for one or several computer programs and all associated documentation, libraries, support websites, and configuration data that are needed to make these programs useful.

## Explanation

The term program is used in a broader sense here. Software may also include source code, software models, or binaries.

# Software Products

## Software Product and Professional Software

A **software product** is a software that can be sold to a customer. **Professional software** is software intended for use by someone apart from its developer and it is usually developed by teams rather than individuals.

[adapted from [Sommerville](#)]



# Application and System Software

## Application Software or Application

Software that is designed for end users and applied for certain purposes.  
(Anwendungssoftware oder Anwendung)

### Examples

web browsers, media players, email or chat clients, text or photo editors, games

## System Software

Software that is not application software and typically being designed to provide a platform for other software.

### Examples

operating systems, game engines, GUI frameworks

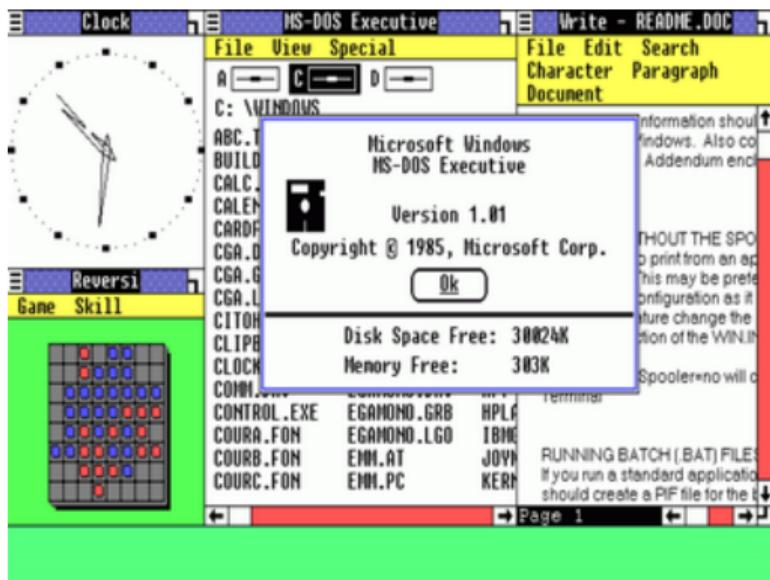
## Classification Not Always Unique

e.g., web browsers and chat clients take over more and more features of operating systems

# Application Software

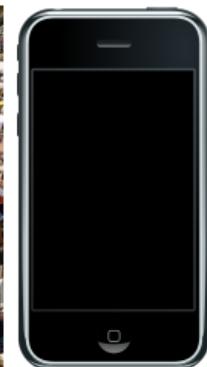
## Desktop Application or Desktop App

Windows 1.0 released in 1985



## Web Application or Web App

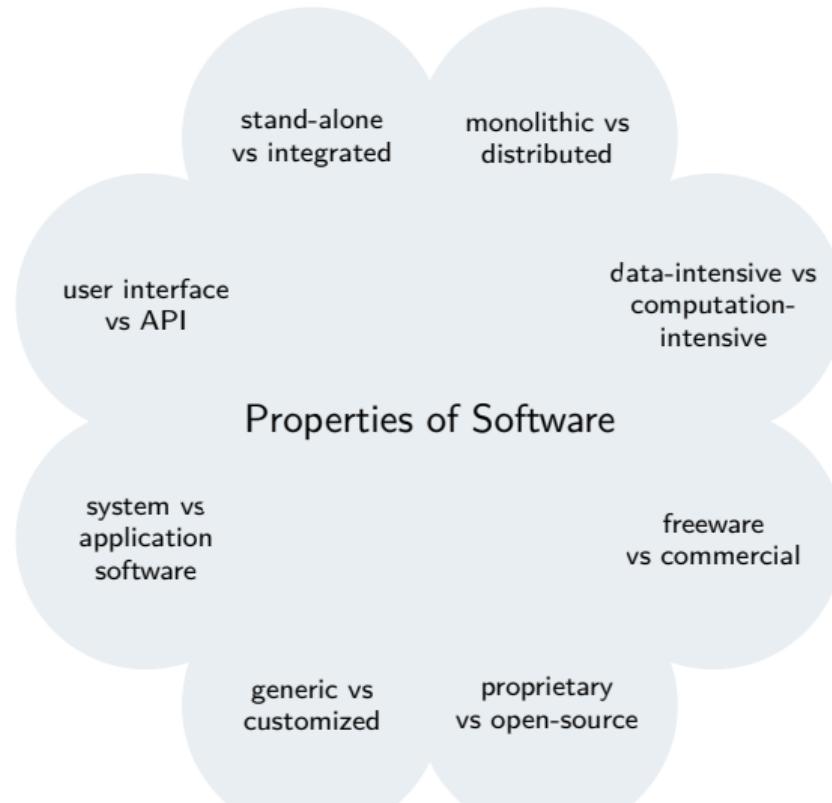
Ebay was born in 1995



## Mobile Application or Mobile App or App

First iPhone released in 2007

# Properties of Software



# What is Software?

## Lessons Learned

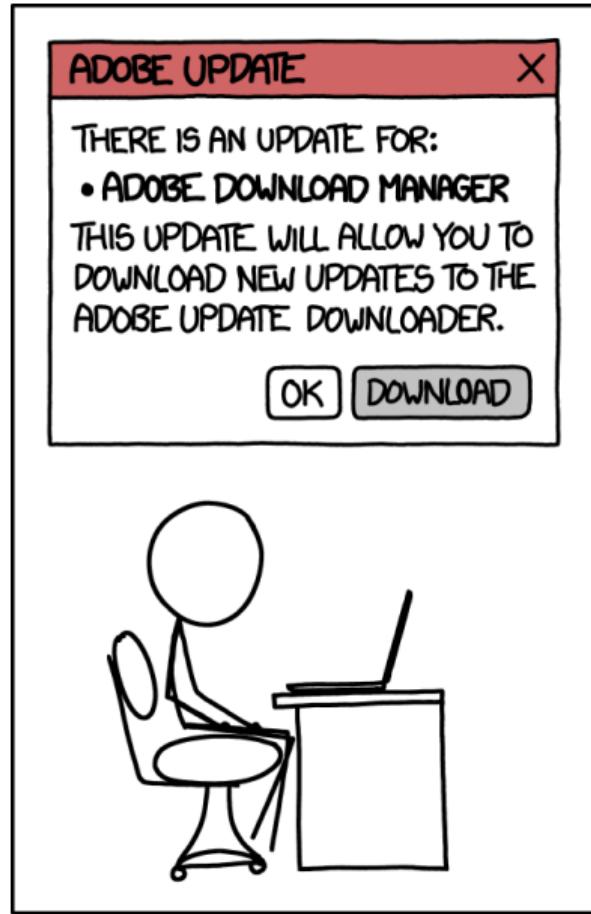
- What is software?
- What is the difference between program, software product, professional software, desktop/web/mobile app?
- Next: How does software influence our life?

## Practice

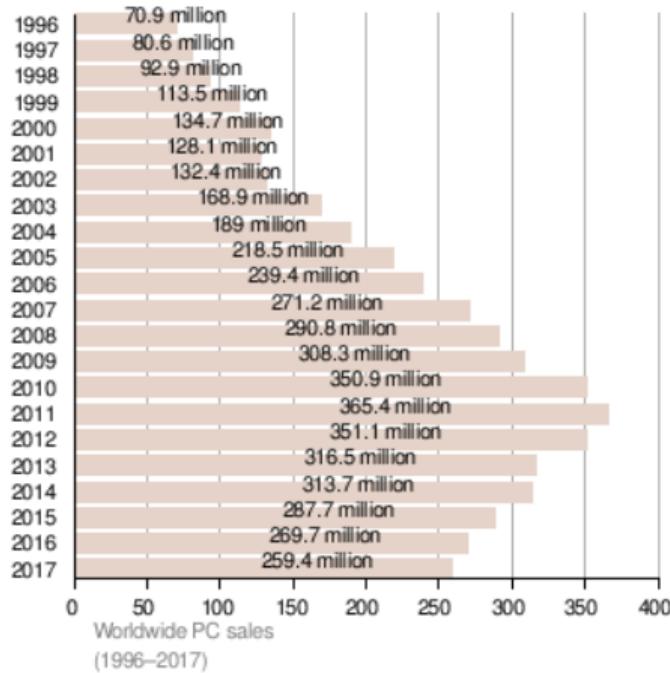
- See Moodle:  
<https://moodle.uni-ulm.de/mod/moodleoverflow/discussion.php?d=3667>
- Questionnaire: What is your experience with software?
- Read about **Korean Air Flight 801** and **Ariane 5 Flight 501** in Wong et al.'s article on “Recent Catastrophic Accidents: Investigating How Software Was Responsible”

# Lecture Contents

1. What is Software?
2. How Relevant is Software?
  - World-Wide PC Sales
  - World-Wide Mobile Phone Subscriptions
  - Downloads of Android Apps
  - Relevance of Software for Me
  - Lessons Learned
3. What is Software Engineering Good For?

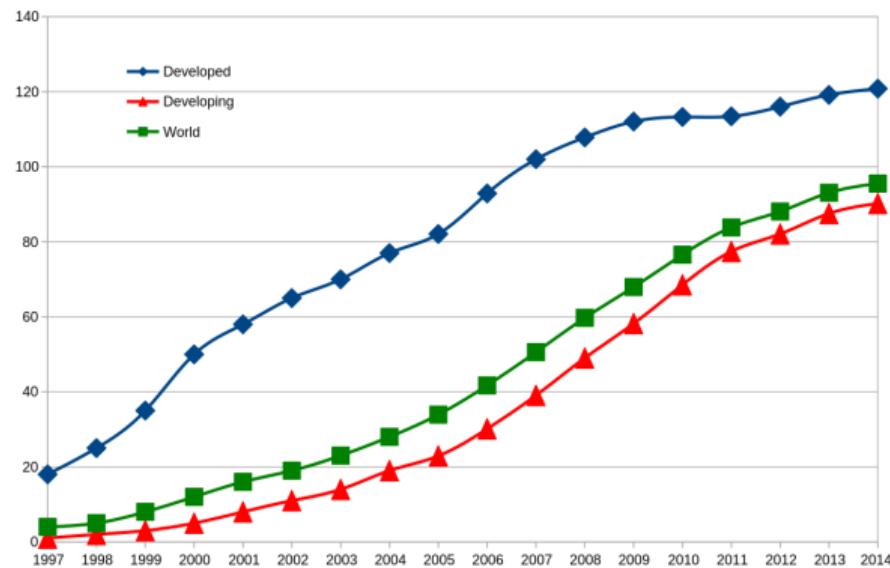


# World-Wide PC Sales

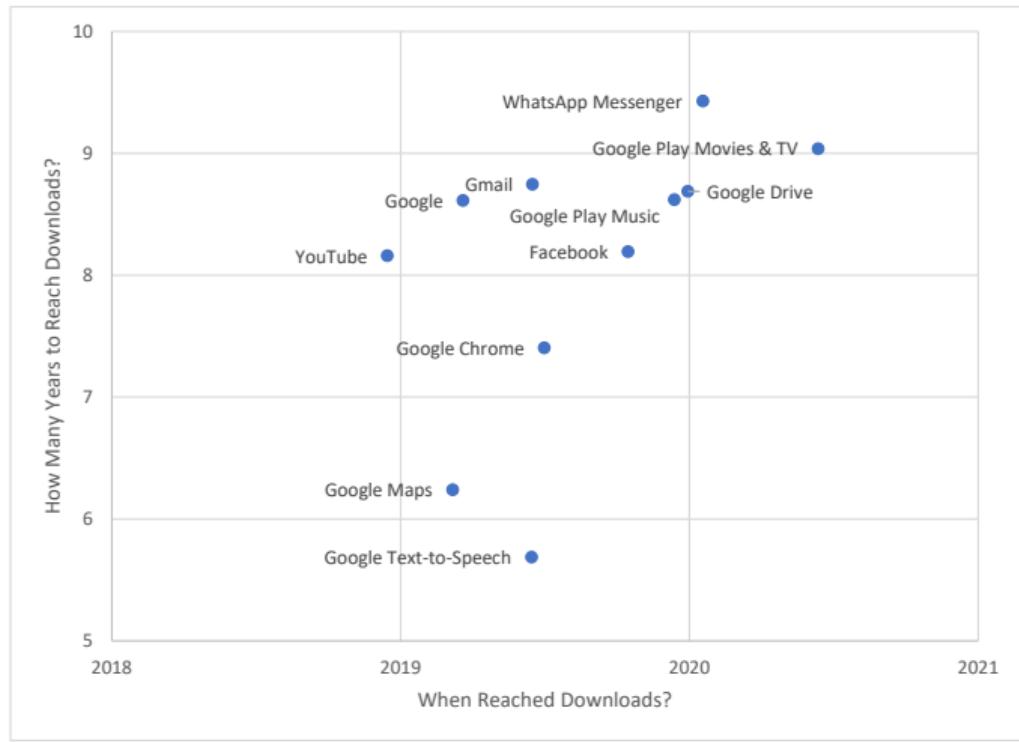


# World-Wide Mobile Phone Subscriptions

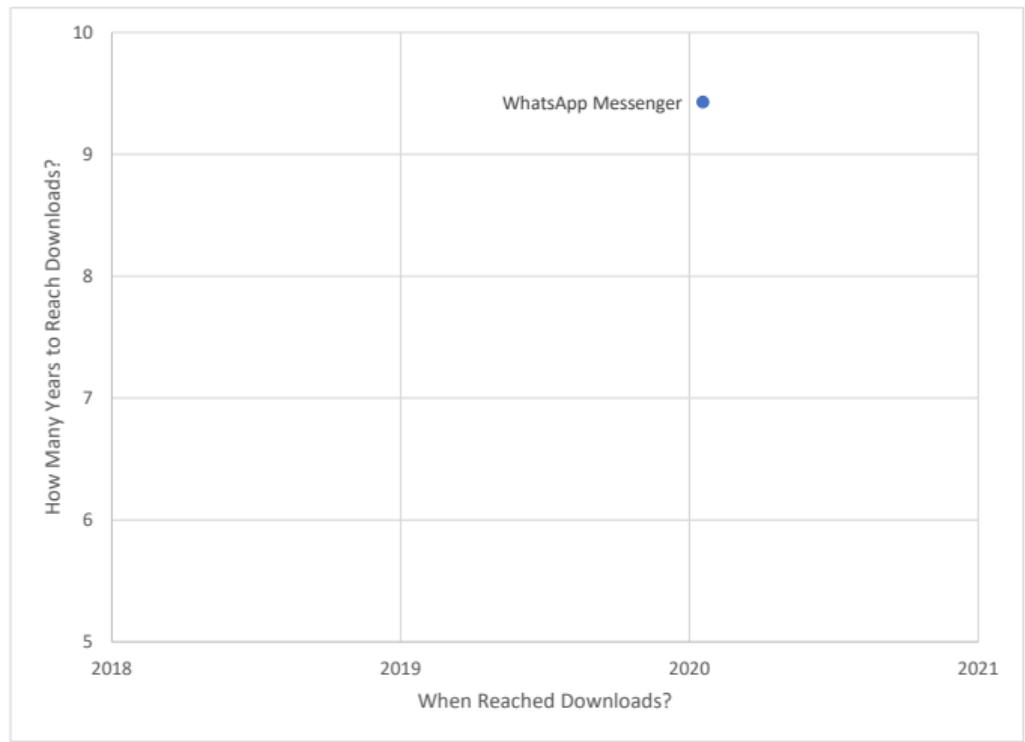
Mobile phone subscribers per 100 inhabitants 1997-2014



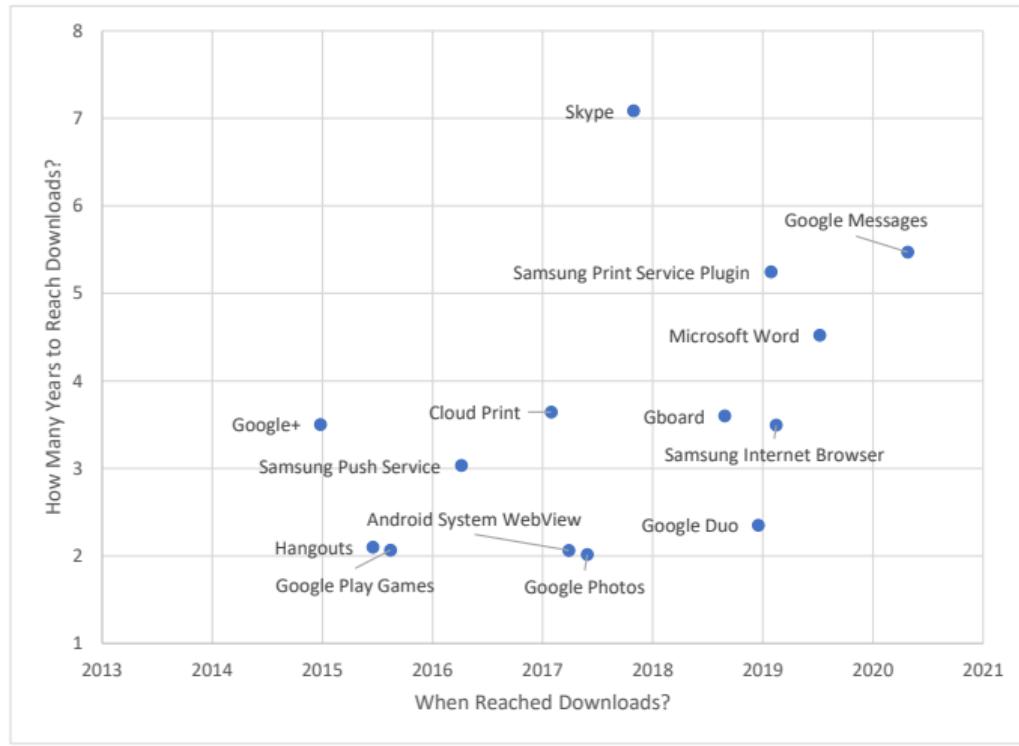
# 5 Billion Downloads of Android Apps



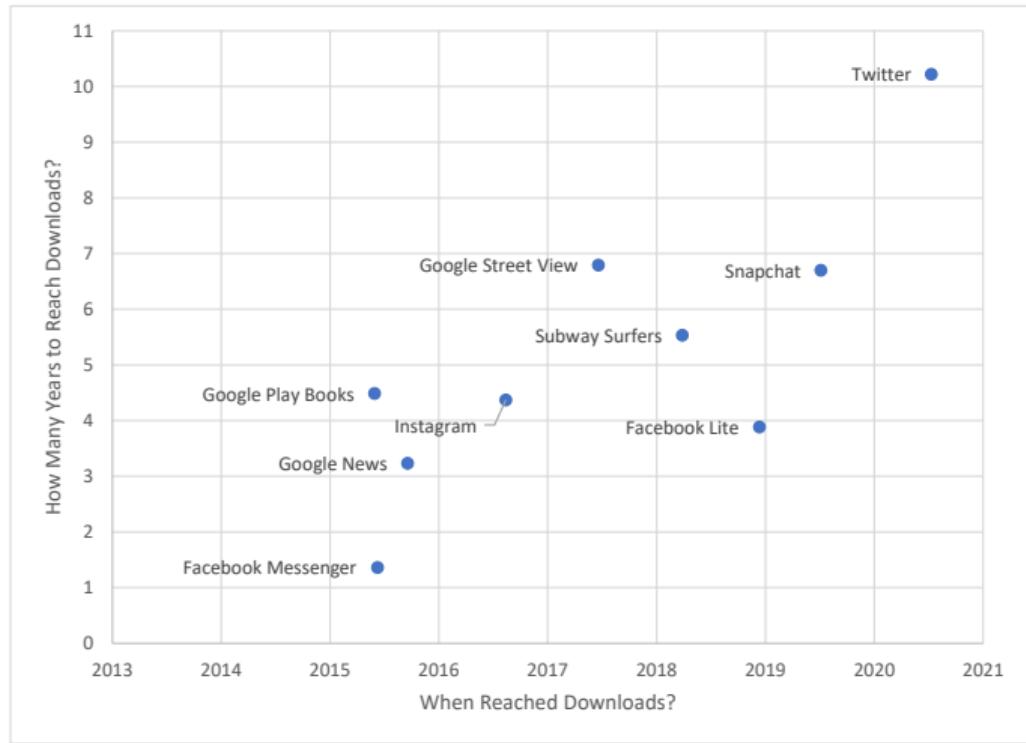
# 5 Billion Downloads of Android Apps



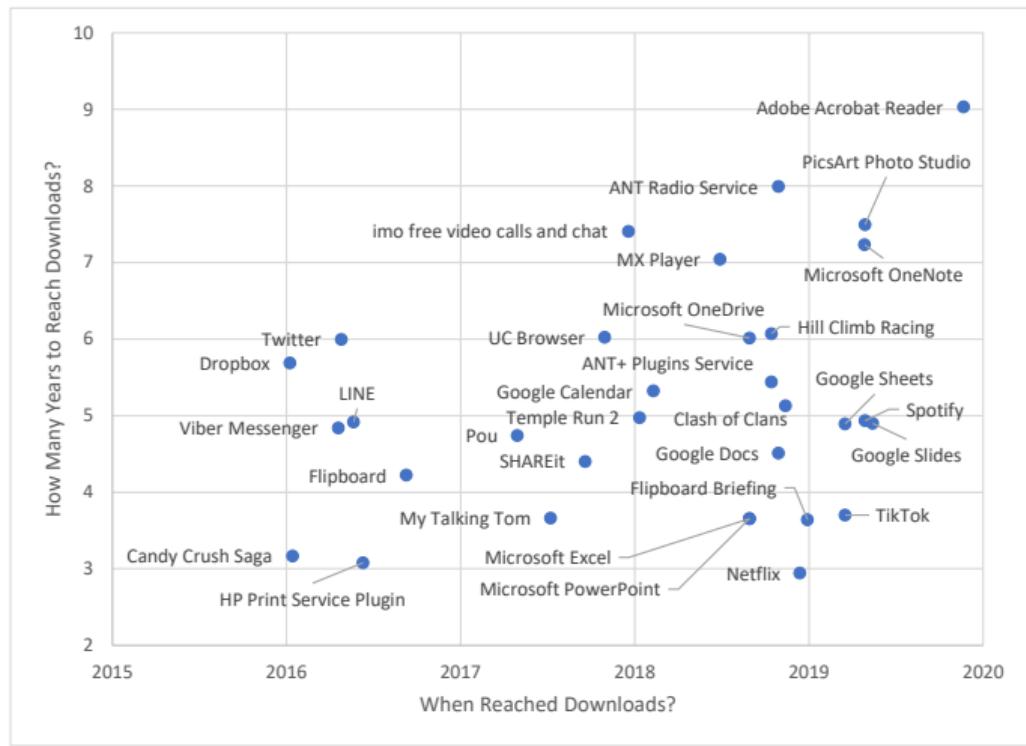
# 1 Billion Downloads of Android Apps



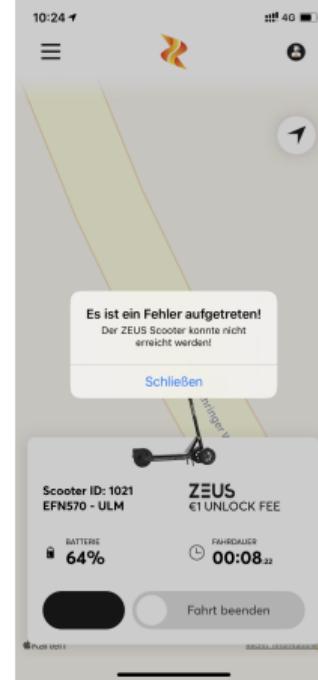
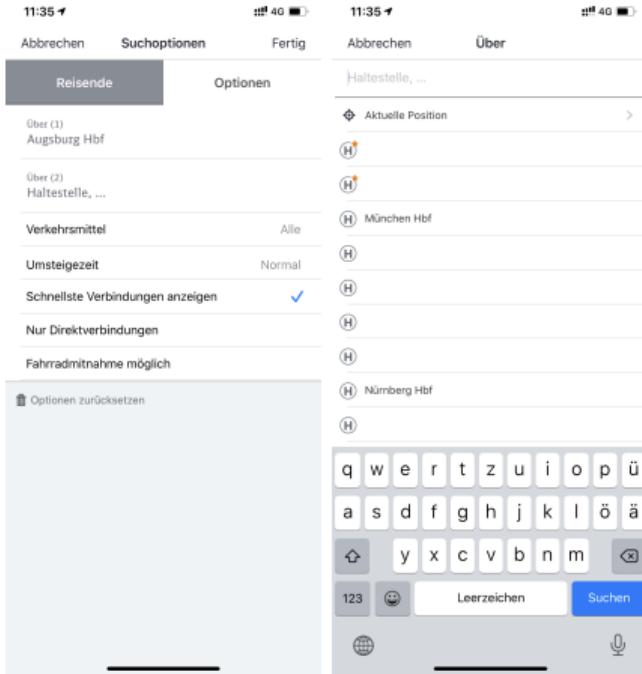
# 1 Billion Downloads of Android Apps



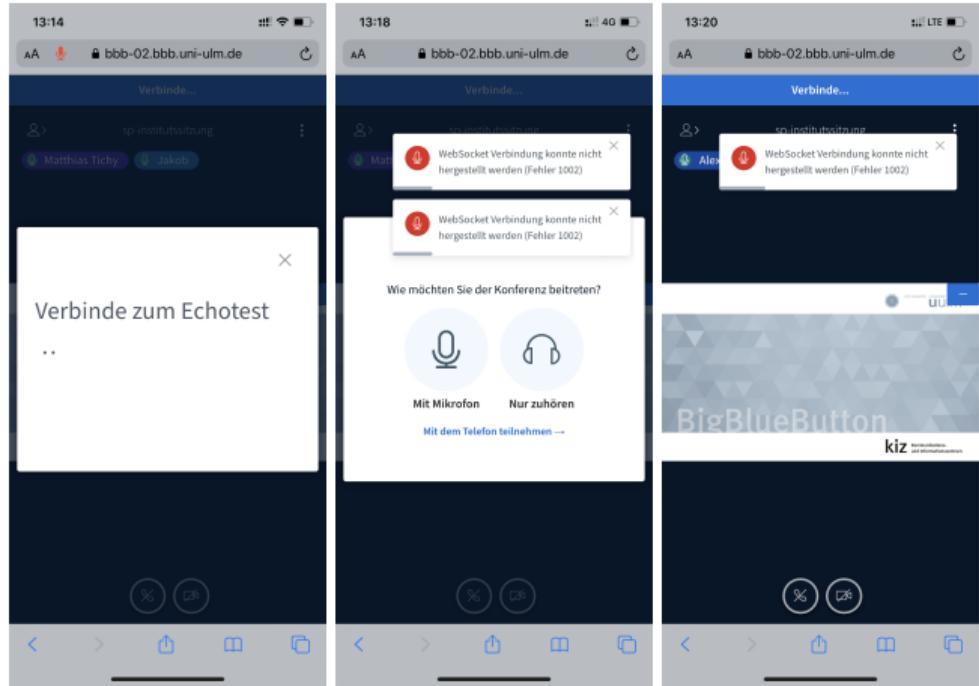
# 500 Million Downloads of Android Apps



# Relevance of Software for Me



# Relevance of Software for Me



# Relevance of Software for Me

BigBlueButton.

ulm university

kiz Kommunikations- und Informationszentrum

Slide 1

# Relevance of Software for Me

Zoom Webinar You are viewing Jackson Prado Lima's screen View Options ▾

The screenshot shows a Zoom webinar interface. At the top, a video feed of Jackson Prado Lima is displayed, with the status "You are viewing Jackson Prado Lima's screen". Below the video, there is a slide titled "COLEMAN interacting with the CI Pipeline". The slide contains a diagram illustrating the interaction between Developers, a Source Control Server, and a CI Pipeline (represented by a box labeled "BUILD", "Test Set T available", "MAIL", and "Rewards"). A red arrow points from the "MAIL" section of the CI Pipeline box to a "COLEMAN" component within it. Below the diagram, there are three buttons: "check-in", "fetch changes", and "result". At the bottom of the slide, there are icons for Chat (with 3 notifications), Raise Hand, and Q&A. To the right of the slide, a separate "Zoom Cloud Meetings" window is open, showing a "Connecting..." message and a form to "Enter your email and name". The "Your email" field contains "Thomas Thüm" and the "Remember my name for future meetings" checkbox is checked. At the bottom of the Zoom Cloud Meetings window are "Join Webinar" and "Cancel" buttons.

COLEMAN interacting with the CI Pipeline

```
graph TD; Developers[Developers] -- "check-in" --> SourceControl[Source Control Server]; SourceControl -- "fetch changes" --> CI[CI Pipeline]; CI -- "result" --> Developers;
```

Developers

Source Control Server

CI Pipeline

check-in

fetch changes

result

COLEMAN

Build

Test Set T available

MAIL

Rewards

Your email

Thomas Thüm

Remember my name for future meetings

Join Webinar

Cancel

Audio Settings ▾

Chat 3

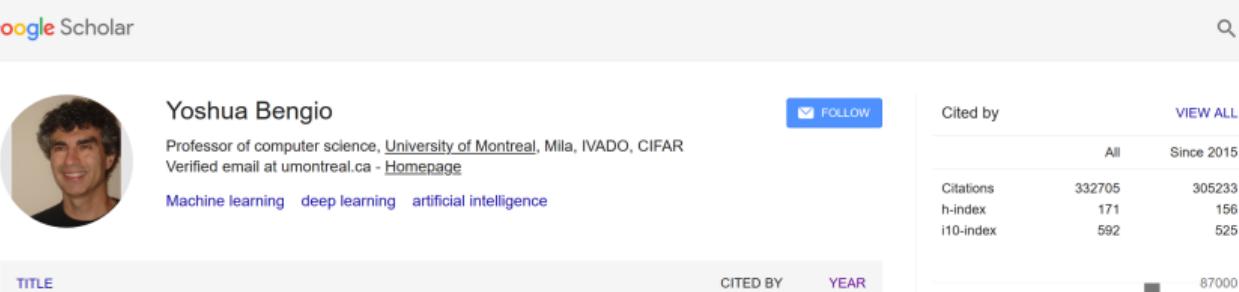
Raise Hand

Q&A

Privacy & Legal Policies

# Relevance of Software for Me

Google Scholar



Yoshua Bengio  
Professor of computer science, University of Montreal, Mila, IVADO, CIFAR  
Verified email at umontreal.ca - Homepage  
Machine learning deep learning artificial intelligence

TITLE CITED BY YEAR

TITLE	CITED BY	YEAR
Deep learning Y LeCun, Y Bengio, G Hinton nature 521 (7553), 436-444	31	2015
Gradient-based learning applied to document recognition Y LeCun, L Bottou, Y Bengio, P Haffner Proceedings of the IEEE 86 (11), 2278-2324	30	2015
Generative adversarial nets I Goodfellow, J Pouget-Abadie, M Mirza, B Xu, D Warde-Farley, S Ozair, ... Advances in neural information processing systems, 2672-2680	25	2015
Deep learning I Goodfellow, Y Bengio, A Courville, Y Bengio MIT press 1, 2	20	2015
Neural machine translation by jointly learning to align and translate D Bahdanau, K Cho, Y Bengio arXiv preprint arXiv:1409.0473	14	2015

Cited by  
VIEW ALL

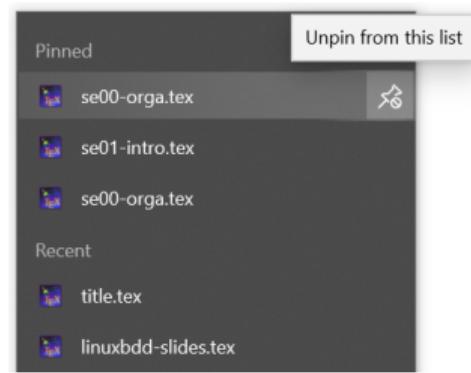
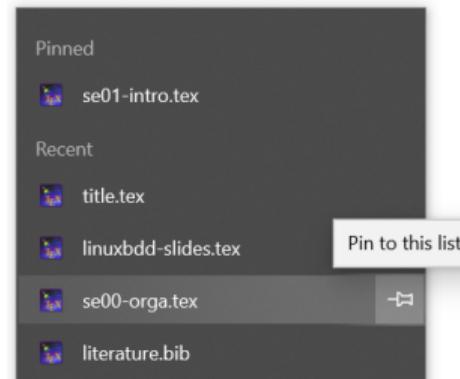
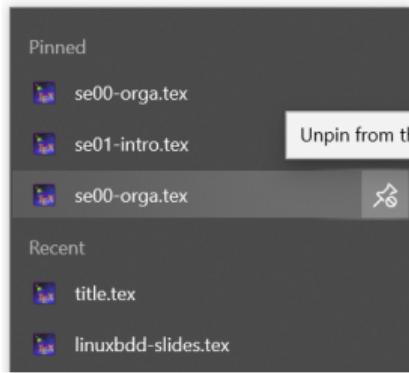
All	Since 2015
Citations 332705	305233
h-index 171	156
i10-index 592	525

87000



31  
**Google**  
500. That's an error.  
The server encountered an error and could not complete your request.  
If the problem persists, please report your problem and mention this error message and the query that caused it.  
That's all we know.

# Relevance of Software for Me



# Relevance of Software for Me

The screenshot shows a double-page spread in Adobe Acrobat Pro 2020. Both pages display the same content: a title slide with the text "Many Queries on BDDs are fast." and a diagram of a Binary Decision Diagram (BDD) labeled "BDDs". The BDD consists of nodes labeled 1 through 6, connected by directed edges. The left page also features a sidebar with a list of topics related to BDDs and SAT-solving.

**Title:** Many Queries on BDDs are fast.

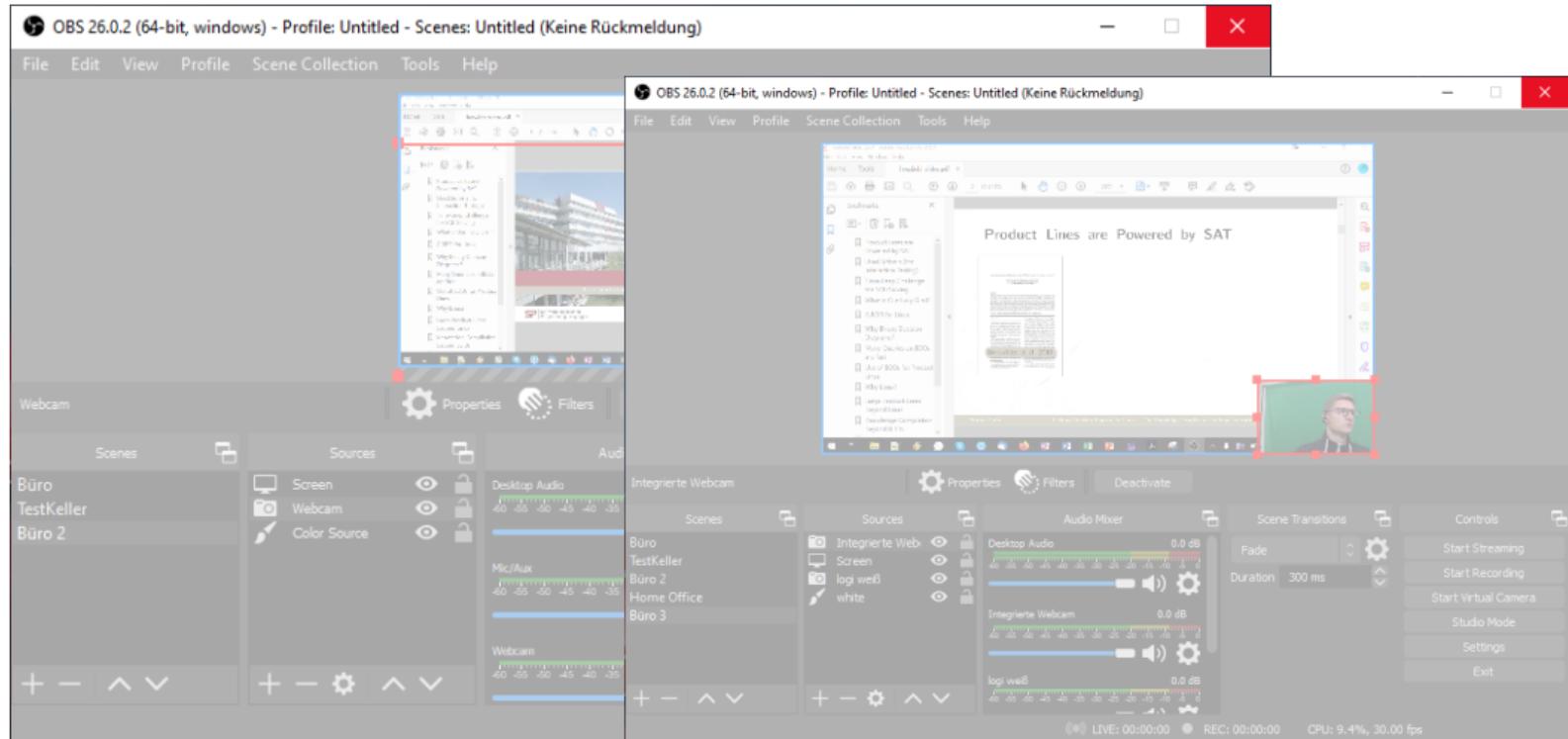
**Diagram:** A BDD diagram with nodes labeled 1 through 6. The nodes are arranged in layers, with node 1 at the top, followed by nodes 2 and 3, then 4, 5, and 6 at the bottom. Directed edges connect nodes between adjacent layers. The label "BDDs" is placed near the bottom of the diagram.

**Reference:** Bryant 1986

**Topics Sidebar:**

- Product Lines are Powered by SAT
- Used Solvers (for Interaction Testing)
- Time-Leap Challenge for SAT-Solving
- What is Our Holy Grail?
- A BDD for Linux.
- Why Binary Decision Diagrams?
- Many Queries on BDDs are fast.
- Use of BDDs for Product Lines
- Why Linux?
- Large Product-Lines Beyond Linux
- Knowledge Compilation Beyond BDDs
- What is the Goal of this Challenge?
- What is the Scope of this Challenge?
- Knowledge Compilation Challenge for Variability

# Relevance of Software for Me



# How Relevant is Software?

## Lessons Learned

- What is the impact of software?
- How relevant is software for us?
- Next: What has software to do with engineering?

## Practice

- See Moodle:  
<https://moodle.uni-ulm.de/mod/moodleoverflow/discussion.php?d=3668>
- What were reasons and consequences of the Facebook outage on October 4th, 2021?

# Lecture Contents

1. What is Software?
2. How Relevant is Software?
3. What is Software Engineering Good For?

Software Engineering

Software Engineering vs Programming

Software Engineering vs Computer Science

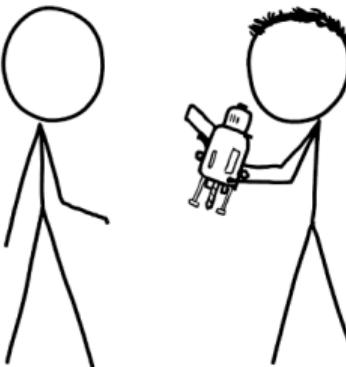
Software and System Engineering

(Software) Engineering

Lessons Learned

WE NEED TO MAKE 500 HOLES IN THAT WALL,  
SO I'VE BUILT THIS AUTOMATIC DRILL. IT USES  
ELEGANT PRECISION GEARS TO CONTINUALLY  
ADJUST ITS TORQUE AND SPEED AS NEEDED.

GREAT, IT'S THE PERFECT WEIGHT!  
WE'LL LOAD 500 OF THEM INTO  
THE CANNON WE MADE AND  
SHOOT THEM AT THE WALL.



HOW SOFTWARE DEVELOPMENT WORKS

# Software Engineering

## Software Engineering

[Sommerville]

“Software engineering is an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance. [...] Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software development.”

# Software Engineering vs Programming



# Software Engineering vs Computer Science

## SE vs CS

[Sommerville]

“Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software. [...] Computer science theory, however, is often most applicable to relatively small programs. Elegant theories of computer science are rarely relevant to large, complex problems that require a software solution.”

# Software and System Engineering

## System Engineering

[Sommerville]

“System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.”



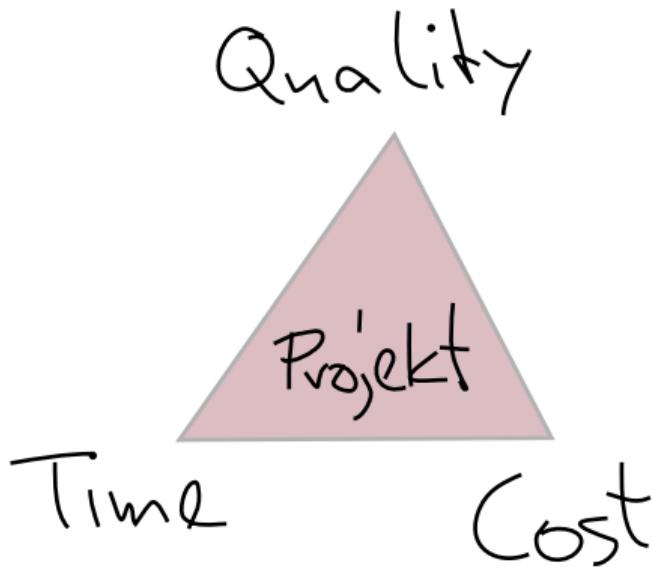
# (Software) Engineering

## Engineering

[Sommerville]

"Engineering is about getting results of the required **quality** within **schedule** and **budget**. [...] Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work within organizational and financial constraints, and they must look for solutions within these constraints."

Spannungsdruck



# What is Software Engineering Good For?

## Lessons Learned

- What is software engineering?
- Which trade-off is crucial to software engineering?
- Further Reading: **Sommerville**, Chapter 1.1, p. 19–28
- Next: How to develop the right thing?

## Practice

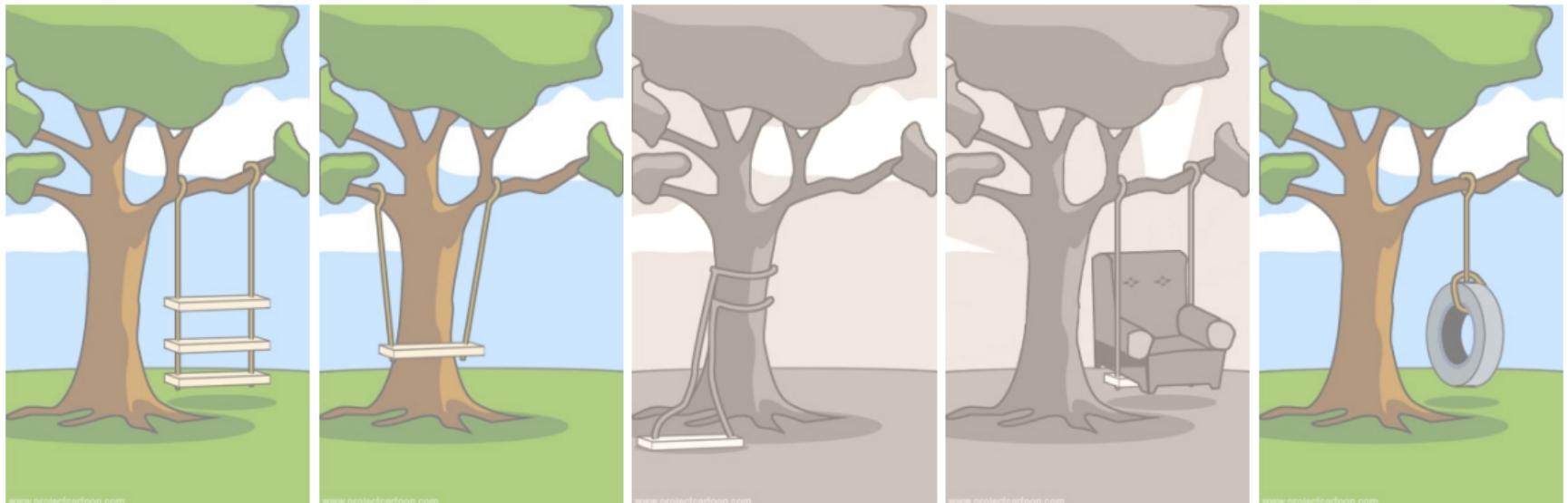
- See Moodle:  
<https://moodle.uni-ulm.de/mod/moodleoverflow/discussion.php?d=3669>
- What is your connection to software engineering in 10 years?
- Read about ethics in software engineering: **Sommerville**, Chapter 1.2, p. 28–31



# Software Engineering

2. Requirements | Thomas Thüm | October 27, 2021

# Software Engineering Projects



how the customer  
explained it

how the project leader  
understood it

how the programmer  
implemented it

how the business  
consultant described it

what the customer  
really needed

# Lecture Overview

1. What are Requirements?
2. How to Elicit Good Requirements?
3. How to Document Requirements?

# Lecture Contents

## 1. What are Requirements?

User Requirements

System Requirements

Functional Requirements

Non-Functional Requirements

Structure of a Requirements Document

Lessons Learned

## 2. How to Elicit Good Requirements?

## 3. How to Document Requirements?

# User Requirements (Benutzeranforderungen)

## User Requirement

[Sommerville]

"User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality."

## User Requirements Document (Lastenheft)

System description from user's point of view.  
What? For what? (Was? Wofür?)

## Example

The Corona app should track contacts by means of random IDs and should store them locally for two weeks.

## Example

If a user has a positive test result, he or she can inform tracked contacts about their increased risk by means of a QR code.



### **Tony Hoare (1969):**

**“The most important property of a program is whether it accomplishes the intention of its user.”**

# System Requirements (Systemanforderungen)

## System Requirement

[adapted from [Sommerville](#)]

System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (aka. functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

## System Requirements Document (Pflichtenheft)

System description from technical point of view.  
How? Whereby? ([Wie?](#) [Womit?](#))

## Example

If two devices are within a distance of 2m for at least 15 minutes they exchange their IDs via Bluetooth.

## Example

After IDs have been exchanged, they are stored for two weeks.

## Example

A new ID is generated every 24 hours and old IDs are stored for two weeks.

# Functional Requirements

## Functional Requirement

[Sommerville]

“Functional requirements are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.”

# Non-Functional Requirements

## Non-Functional Requirement

[Sommerville]

"Non-functional requirements are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole rather than individual system features or services."

## Note

often more critical than functional requirements

## Example

The app consumes less than 10MB of RAM.

## Example

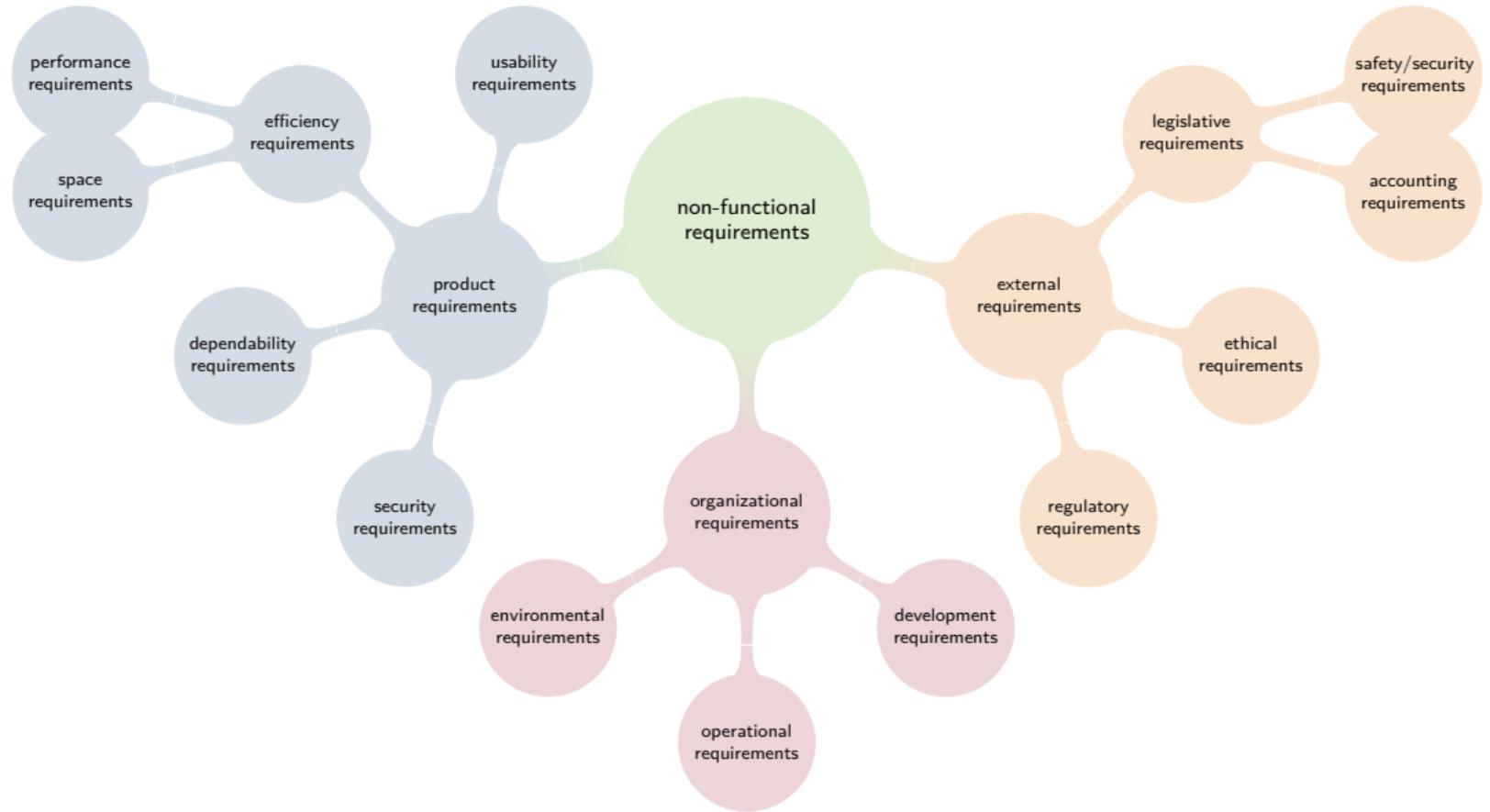
The app has no access to private data of the user.

## Example

The app conforms to the GDPR. ([DSGVO](#))

## Example

The source code of the app is open source.



# Structure of a Requirements Document

## Typical Structure

[Sommerville]

Preface expected readers, version history

**Introduction** motivation/needs, collaboration with other system, strategic objectives

Glossary technical terms

**User Requirements Definition** functional and non-functional user requirements

**System Architecture** distribution of functions across system modules, potential reuse

**System Requirements Specification** detailed requirements, interfaces to other systems

**System Models** graphical models illustrating the system with its environment

**System Evolution** anticipated changes due to hardware evolution or changing needs

**Appendices** hardware and database requirements, minimal/optimal system configuration

**Index** index of diagrams/functions/terms/...

# What are Requirements?

## Lessons Learned

- What kind of requirements exist and why?
- User and System Requirements Document ([Lasten- und Pflichtenheft](#))
- Further Reading: [Sommerville](#), Chapter 2.2.1 (p. 54f), Chapter 4.1 (p. 101–111), Chapter 4.4.4 (p. 126–128)

## Practice

- See [Moodle](#)
- Give examples for functional, product, organizational, and external requirements
- Classify requirements of colleagues

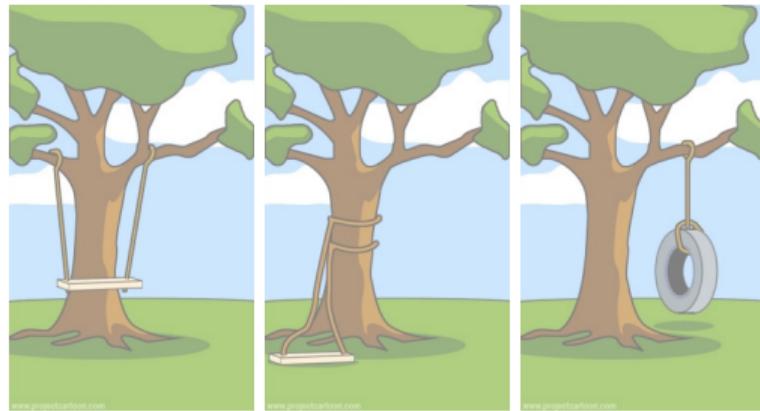
# Lecture Contents

1. What are Requirements?
2. How to Elicit Good Requirements?
  - Imprecise Requirements
  - Complete and Consistent Requirements
  - Requirements Engineering Process
  - Why is Requirements Elicitation so Hard?
  - Example Dialog by Ludewig and Licher
  - Requirements Elicitation Techniques
  - Requirements Validation
  - Lessons Learned
3. How to Document Requirements?

# Imprecise Requirements

## Sommerville:

"Imprecision in the requirements specification can lead to disputes between customers and software developers. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs."



how the project  
leader  
understood it

how the  
programmer  
implemented it

what the  
customer really  
needed

# Complete and Consistent Requirements

## Completeness and Consistency

[Sommerville]

“Ideally, the functional requirements specification of a system should be both complete and consistent. **Completeness** means that all services and information required by the user should be defined. **Consistency** means that requirements should not be contradictory.”

## Further Desired Properties

clear, easy to understand, unambiguous, correct, verifiable, prioritized, changeable, traceable

## Often not Feasible in Practice

mistakes, omission, implicit knowledge, many stakeholders (**Akteure**) with different backgrounds / expectations / inconsistent needs



**Fred Brooks (1987):**

“Much of the essence of building a program is in fact the debugging of the specification.”

# Requirements Engineering Process

## Requirements Elicitation and Analysis

Requirements elicitation and analysis is the process of deriving the system requirements through observation of existing systems, discussions with potential users, or development of prototypes. (Anforderungsermittlung und -analyse)

[adapted from Sommerville]

## Requirements Specification

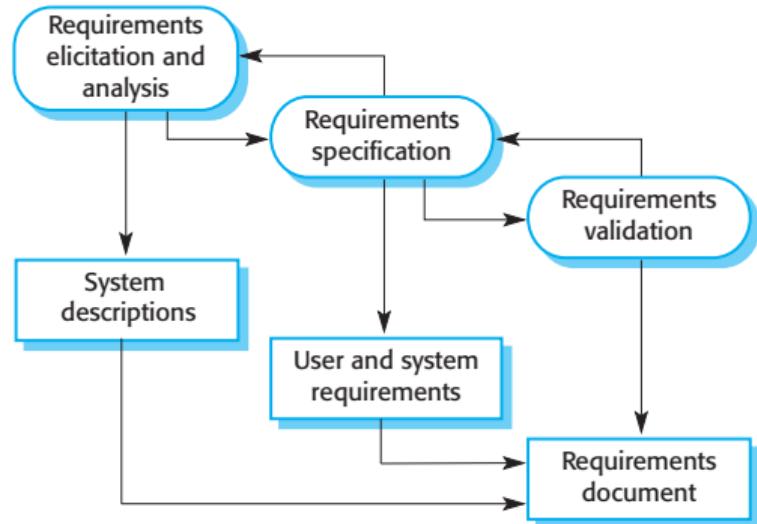
[Sommerville]

“Requirements specification is the process of writing down the user and system requirements in a requirements document (aka. requirements specification).” (Anforderungsspezifikation)

## Requirements Validation

[Sommerville]

“Requirements validation is an activity that checks the requirements for realism, consistency, and completeness.” (Anforderungsvalidierung)



# Why is Requirements Elicitation so Hard?

- Stakeholders have difficulties to articulate what they want
- Stakeholders don't know what is (not) feasible
- Implicit knowledge and jargon in the customer's domain
- Dynamic business environment (e.g., new stakeholders)

# Example Dialog by Ludewig and Licher

- At the morning you unlock the door at the main entrance?
- Yes, as I said.
- Every morning?
- Of course.
- Even at the weekend?
- No, at weekend the entrance remains closed.
- And during the plant shutdown?
- Clearly, it is then closed as well.
- And if you are sick or in holidays?
- Mr. X opens the door in this case.
- And if Mr. X is off?
- Then, a client knocks at the window to tell that the door is closed.
- What does “morning” mean?
- ...

Source: Ludewig and Licher

# Requirements Elicitation Techniques

- Open interviews: talk to users what they do
- Closed interviews: stakeholders answer predefined questions
- Ethnography: observation of stakeholder's work
- Prototyping
- Feedback loops

## Sommerville:

"You need to spend time understanding how people work, what they produce, how they use other systems, and how they may need to change to accommodate a new system."

## In Practice

Combinations of numerous techniques used

# Requirements Validation

Motivation: the later problems with requirements are detected, the more costly it will be

**Validity checks** do requirements (still) reflect real needs?

**Consistency checks** are there contradictory/redundant requirements?

**Completeness checks** are all functions and constraints documented?

**Realism checks** is it feasible within budget and schedule?

**Verifiability** can we test the requirements?

Checked by reviews, prototyping, and test-case creation

# How to Elicit Good Requirements?

## Lessons Learned

- What are desired and undesired properties for requirements?
- What is the requirements engineering process?
- What are techniques for requirements elicitation and why is it hard?
- Further Reading: [Sommerville](#), Chapter 4.2–4.3 (p. 111–119), Chapter 4.5 (p. 129f)

## Practice

- See [Moodle](#)
- Give example for a bad requirement
- Improve a requirement of a colleague

# Lecture Contents

1. What are Requirements?
2. How to Elicit Good Requirements?
3. How to Document Requirements?
  - Natural Language Specification
  - Structured Specifications
  - CoronaWarnApp User Stories
  - Use Case Diagrams
  - Include and Extend Relationships
  - Lessons Learned

# Natural Language Specification

- Used since 1950s
- Pros: expressive, intuitive, universal
- Cons: vague, ambiguous, interpretation depends on reader's background

## Style Guidelines

- one or two short sentences of natural language
- one message per sentence
- use active voice (e.g., “the system . . . ”)
- consistent use of language (i.e., avoid synonyms)
- use shall for mandatory and should for desirable requirements
- use text highlighting
- avoid jargon, abbreviations, acronyms
- provide a rationale

# Structured Specifications

- Use of templates rather than free-form text
- Distinguishes between function, description, input (origin), output (destination), pre- and postconditions, side effects, rationale, dependencies to other requirements, ... or any subset thereof
- Pros: same as before
- Cons: similar as before, but less variability

## Example

Function Exchange of IDs

Description Two devices send their IDs to each other.

Precondition Devices have been within a distance of 2m for at least 15 minutes.

Postcondition IDs are stored in the local database.

Rationale Contact tracing requires to temporarily identify the device of contacts.

# CoronaWarnApp User Stories

Show the number of stored keys in the local database #65

 Closed asdil12 opened this issue on May 31 · 1 comment



asdil12 commented on May 31

...

## Feature description

The app receives keys from other smartphones via bluetooth and stores the keys for the last 14 days (IIRC) in the local database. Even though the app doesn't know if multiple keys in the local db correspond to one single person in real life, it would be nice to have this number as it could be used as a rough indicator if the app is working at all and give some transparency how well it is performing.

Also it could be helpful by providing some feedback to the user how well he is keeping distance to other people eg. by looking at the counter before and after going to the supermarket.

The counter could be put into a debug submenu to prevent confusing users with less technical knowledge.

## Problem and motivation

As a technically interested person I would like to gain some insights und transparency how the app is performing.

## Is this something you're interested in working on

No

# CoronaWarnApp User Stories

## Enhance Logging With Timber #24

 Closed Magoli1 opened this issue on May 30 · 1 comment



Magoli1 commented on May 30



...

### Current Implementation

Currently the app uses the native android logging framework. To do that, every class holds a reference to a dedicated logging tag, which is given the logger, in case it is used. Example:

cwa-app-android/Corona-Warn-App/src/main/java/de/rki/coronawarnapp/receiver/ExposureStateUpdateReceiver.kt  
Line 38 in 43412bd

```
38     private val TAG: String? = ExposureStateUpdateReceiver::class.simpleName
```

### Suggested Enhancement

There is a lightweight library called [Timber](#), which automatically sets the logging tags to the calling class. We suggest to change the implementation for all logging calls towards Timber.

### Expected Benefits

Timber optimizes the developer experience in several cases, e.g. it automatically makes sure that the tag is at most 23 characters long (more details [here](#)). We could also get rid of the tag in every class that is currently hardcoded.

# CoronaWarnApp User Stories

## Do not log in production #235

 Closed IndianaDschnones opened this issue on Jun 7 · 4 comments



IndianaDschnones commented on Jun 7

Contributor  ...

### Current Implementation

The `release` build contains `Log`-statements of all levels.

### Suggested Enhancement

A production build should not contain any `Log`-statements, see for example the [Android Prepare for release guide](#).

`Log` statements can be removed using ProGuard rules.

### Expected Benefits

- Compliance to Android's suggestions for `release` builds
- Security: logs can contain sensitive data which should never be logged
- Behaves like the iOS application will do, see [corona-warn-app/cwa-app-ios#22](#)

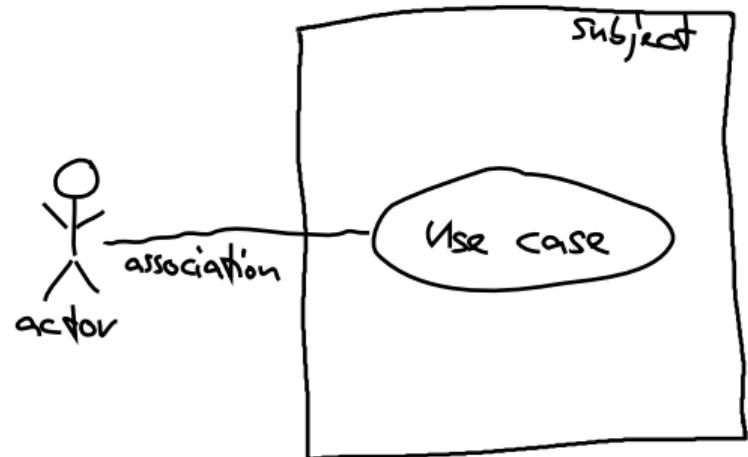
# Use Case Diagrams (since 1993)

## Use Case Diagram (Anwendungsfalldiagramm)

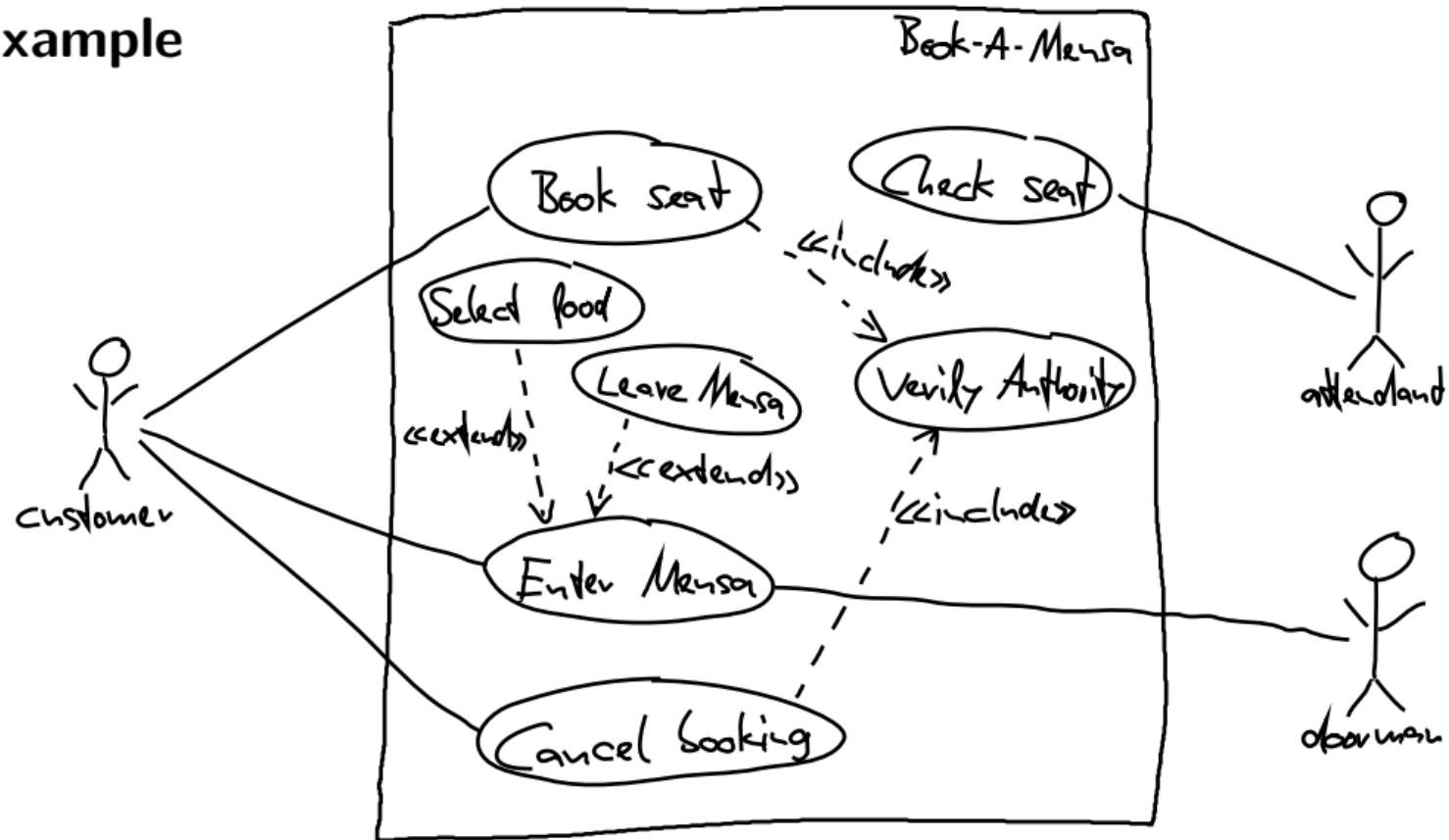
Use case diagrams are a means to capture the requirements of systems, i.e., what systems are supposed to do. The key concepts specified in this diagram are actors, use cases, and subjects:

- Each **subject** represents a system under consideration to which the use case applies. (**System**)
- Each user and any other system that may interact with a subject is represented as an **actor**. (**Akteur**)
- A **use case** is a specification of behavior in terms of verb and noun. (**Anwendungsfall**)

[adapted from UML 2.5.1]



## Example



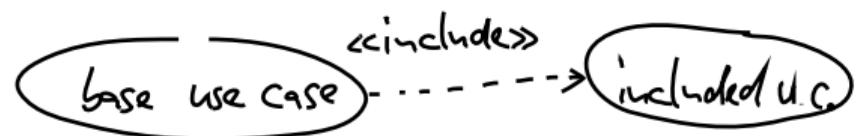
# Include and Extend Relationships

## Include Relationship

**Motivation:** make common parts of multiple use cases explicit

**Relationship:** a **base use case** may define an include relationship to an **included use case**

**Meaning:** included use case is always executed when the base use case is

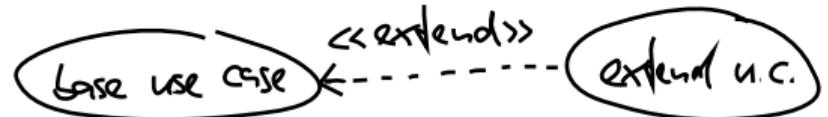


## Extend Relationship

**Motivation:** make explicit that some use cases only happen under certain circumstances

**Relationship:** an **extend use case** may define an extend relationship to a **base use case**

**Meaning:** when the base use case is executed, the extend use case may or may not be executed



# How to Document Requirements?

## Lessons Learned

- Natural language and structured specification of requirements
- Graphical specification with use case diagrams (*Anwendungsfalldiagramme*)
- Further Reading: *Sommerville*, Chapter 4.4 (p. 120–126) and Chapter 5.2.1 (p. 144–146)
- Further Reading: *UML 2.5.1*, Chapter 18
- Next: How to model the system behavior in more detail?

## Practice

- See [Moodle](#)
- Draw a use case diagram related to your requirements and upload it to Moodle
- Optional: watch a great tutorial on use case diagrams: <https://www.youtube.com/watch?v=zid-MVo7M-E>  
(contains 15s advertisement for a tool)  
(generalization and extension points are not needed in this course, can stop at 10:50)



**Edward V. Berard (1993):**

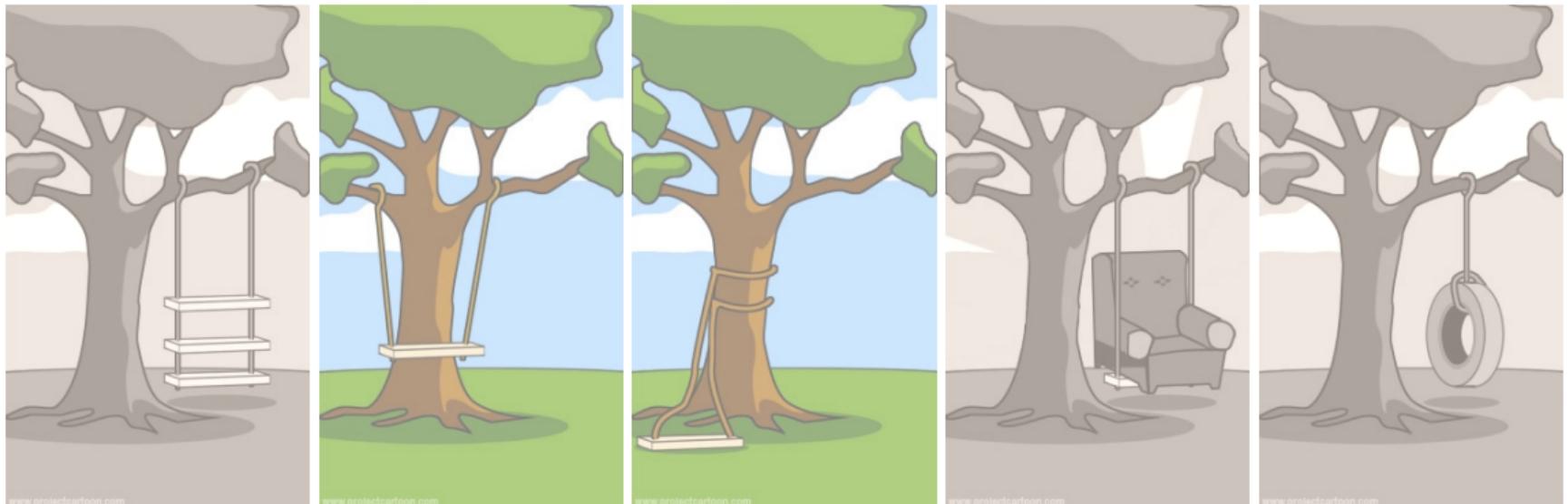
“Walking on water and developing software from a specification are easy if both are frozen.”



# Software Engineering

3. System Modeling | Thomas Thüm | November 3, 2021

# Why System Modeling?



how the customer  
explained it

how the project leader  
understood it

how the programmer  
implemented it

how the business  
consultant described it

what the customer  
really needed

# Lecture Overview

1. Why to Model Systems?
2. Modeling Behavior with Activity Diagrams
3. Modeling Behavior with State Machine Diagrams

# Lecture Contents

## 1. Why to Model Systems?

Motivation for Modeling

Recap: Software Engineering vs Programming

What is System Modeling?

What is a Model?

What Language to Use for Modeling?

The Unified Modeling Language

Different Kinds of UML Diagrams

14 Types of UML Diagrams

Lessons Learned

## 2. Modeling Behavior with Activity Diagrams

## 3. Modeling Behavior with State Machine Diagrams

# Motivation for Modeling

## UML User Guide:

"A successful software organization is one that consistently deploys **quality software** that meets the needs of its users. An organization that can develop such software in a **timely and predictable** fashion, with an **efficient and effective use of resources**, both human and material, is one that has a sustainable business.

[...]

Modeling is a central part of all the activities that lead up to the deployment of good software. We build models to **communicate** the desired structure and behavior of our system. We build models to **visualize and control** the system's architecture. We build models to better **understand** the system we are building, often exposing opportunities for **simplification and reuse**. And we build models to **manage risk**."

## UML User Guide:

"We build models of complex systems because we cannot comprehend such a system in its entirety."

# Recap: Software Engineering vs Programming





**Bjarne Stroustrup (2000):**

"The most important single aspect of software development is to be clear about what you are trying to build."

# What is System Modeling?

## System Modeling

[Sommerville]

“**System modeling** is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. [...] Models are used during the requirements engineering process to help derive the **detailed requirements** for a system, during the design process to **describe the system to engineers** implementing the system, and after implementation to **document the system's** structure and operation.”

# What is a Model?

## UML User Guide:

"A **model** is a simplification of reality."

## Sommerville:

"A **model** is an abstract view of a system that deliberately ignores some system details."

## Goals of Models

[UML User Guide]

- visualize a system as it is (wanted)
- specify the structure or behavior of a system
- template to guide construction of a system
- document the decisions we have made

## Sommerville:

"It is important to understand that a system model is **not a complete representation** of system. It purposely leaves out detail to make it **easier to understand**. A model is an abstraction of the system being studied rather than an alternative representation of that system. A representation of a system should maintain all the information about the entity being represented. An abstraction **deliberately simplifies a system** design and picks out the most salient characteristics."

# What Language to Use for Modeling?

## Towards a Common Language

- Natural language? hard to abstract from details, already used in requirements
- Programming language? unfamiliar to people without programming skills in that language, too early to decide for the programming language
- Textual language? harder to understand
- Graphical language? makes use of our visual abilities, requires common understanding
- Problem: engineers need to be aware of all languages being used
- Solution: use a graphical language independent of company and domain

# HOW STANDARDS PROLIFERATE:

(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)

SITUATION:  
THERE ARE  
14 COMPETING  
STANDARDS.

14?! RIDICULOUS!  
WE NEED TO DEVELOP  
ONE UNIVERSAL STANDARD  
THAT COVERS EVERYONE'S  
USE CASES.



YEAH!

SOON:

SITUATION:  
THERE ARE  
15 COMPETING  
STANDARDS.

# The Unified Modeling Language

## UML

[UML Reference Manual]

“The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system.”

## UML User Guide:

“Modeling yields an understanding of a system. No one model is ever sufficient. Rather, you often need multiple models that are connected to one another [...].”

# Different Kinds of UML Diagrams

## Structure Diagrams (Strukturdiagramme)

“**Structure diagrams** show the static structure of the objects in a system. That is, they depict those elements in a specification that are irrespective of time. The elements in a structure diagram represent the meaningful concepts of an application, and may include abstract, real-world and implementation concepts.”

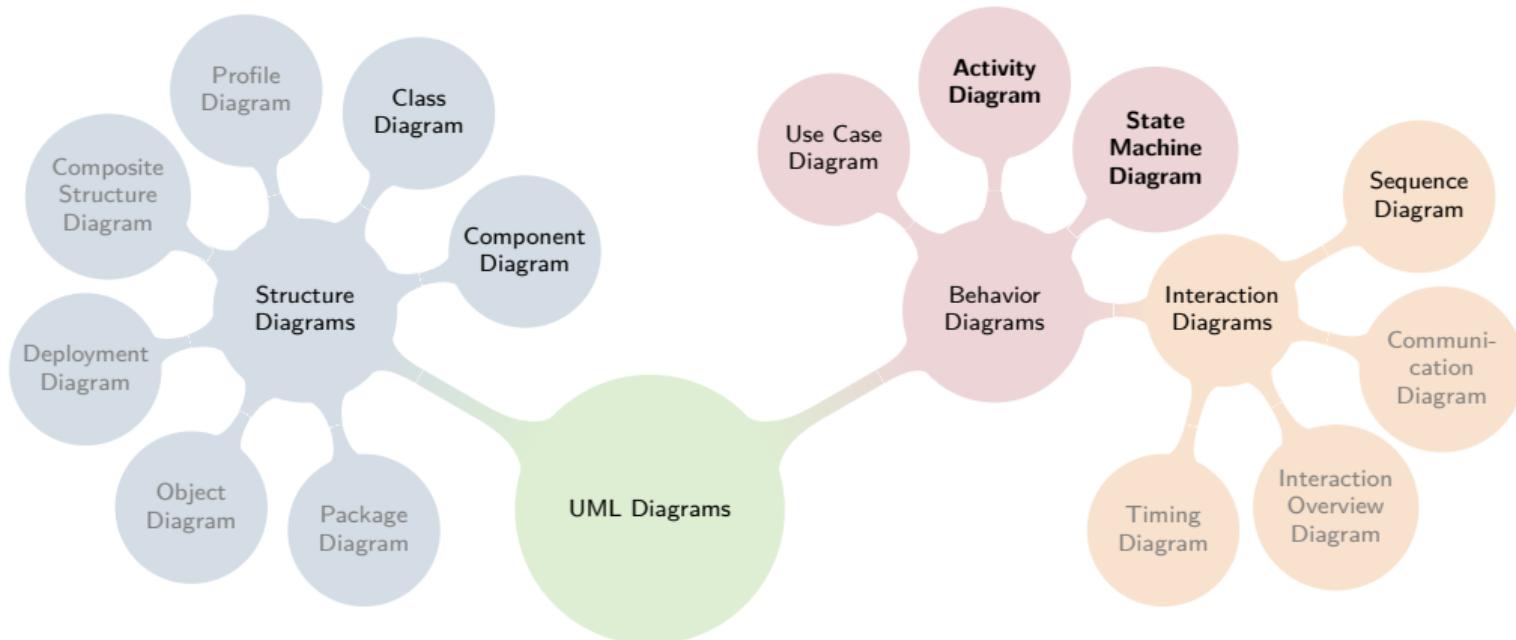
[UML 2.5.1]

## Behavior Diagrams (Verhaltensdiagramme)

“**Behavior diagrams** show the dynamic behavior of the objects in a system, including their methods, collaborations, activities, and state histories. The dynamic behavior of a system can be described as a series of changes to the system over time.”

[UML 2.5.1]

# 14 Types of UML Diagrams [UML 2.5.1]



Six most important UML diagrams\* discussed in this course

\*John Erickson and Keng Siau. 2007. Theoretical and practical complexity of modeling methods. Commun. ACM 50, 8 (August 2007), 46–51.

# Why to Model Systems?

## Lessons Learned

- What is the motivation for system modeling?
- What are models and what is UML?
- Which (kinds of) UML diagrams exist?
- Further Reading: [UML User Guide](#), Chapter 1 — great introduction to modeling

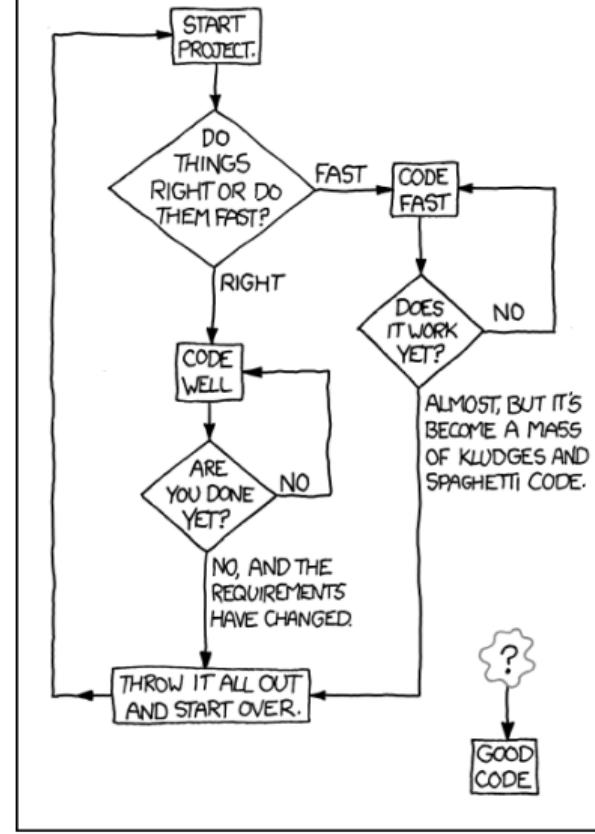
## Practice

- See [Moodle](#)
- Practice abstraction by explaining a room that you know well to a good friend. Use 1 minute (not more!) to create sketch of the room and upload that to Moodle.
- Look at another sketch submitted and create an answer in Moodle with a list of ten things that the sketch abstracts from (i.e., what details are not shown in the visualization).

# Lecture Contents

1. Why to Model Systems?
2. Modeling Behavior with Activity Diagrams
  - Activity Diagrams
  - Branching and Merging in Activity Diagrams
  - Forking and Joining in Activity Diagrams
  - Swimlanes in Activity Diagrams
  - Lessons Learned
3. Modeling Behavior with State Machine Diagrams

## HOW TO WRITE GOOD CODE:



# Activity Diagrams



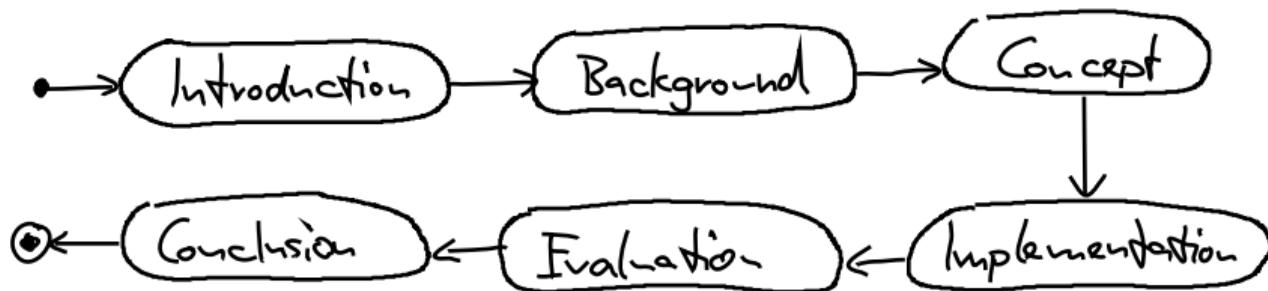
## Activity Diagram (Aktivitätsdiagramm)

An **activity diagram** is a diagram visualizing activities and their order of execution. An activity diagram contains **activities** (rounded box) that are connected by means of **flows** (solid arrows). The execution begins at the **initialization** (filled circle) and ends with the **completion** node (bull's eye). (Aktivität, Fluss, Startzustand, Endzustand)

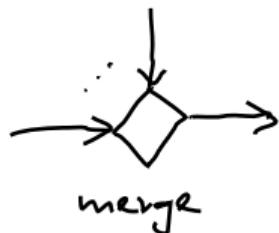
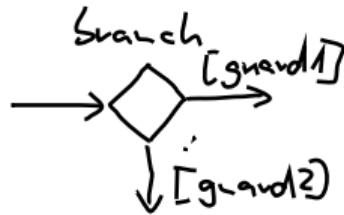
## Rules for Activity Diagrams

- exactly one initialization/completion node
- at least one activity
- every activity has one incoming and one outgoing flow
- every activity is reachable from initialization
- completion is reachable from every activity

# Example of Sequential Activities



# Branching and Merging in Activity Diagrams



## Branching and Merging

[UML User Guide]

**Motivation:** model control flow that depends on certain conditions (i.e., actions that may happen)

**Branching:** A **branch** has exactly one incoming and two or more outgoing flows. Each outgoing flow has a Boolean expression called **guard**, which is evaluated on entering the branch.

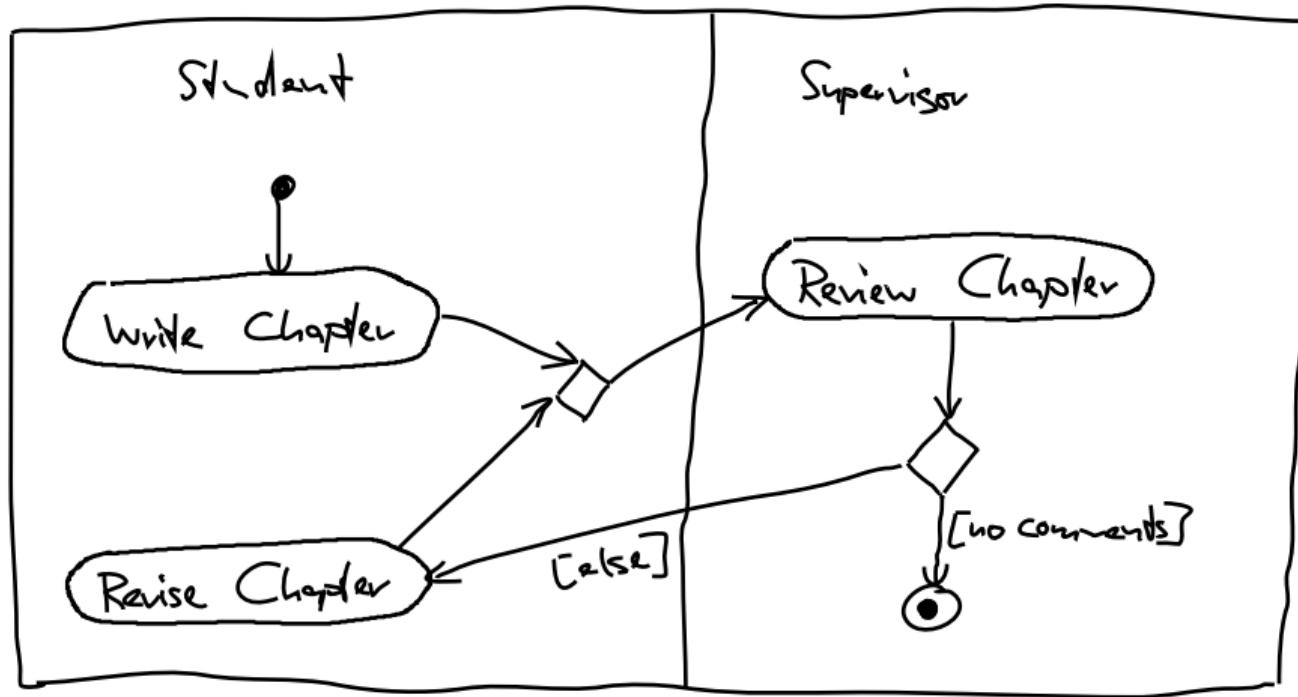
(Verzweigung)

**Merging:** A **merge** has two or more incoming and exactly one outgoing flow. (Zusammenführung)

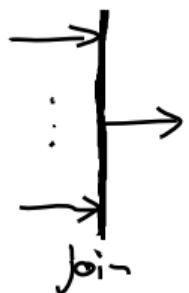
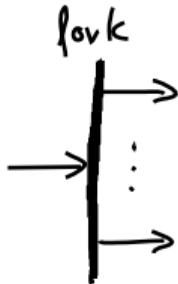
## Further Rules for Activity Diagrams

- guards on outgoing flows should not overlap (flow of control is unambiguous)
- guards should cover all possibilities (flow of control does not freeze)
- keyword **else** possible for one guard (**sonst**)

# Example of Conditional Activities



# Forking and Joining in Activity Diagrams



## Forking and Joining

[UML User Guide]

**Motivation:** model concurrent control flows (i.e., activities that run in parallel)

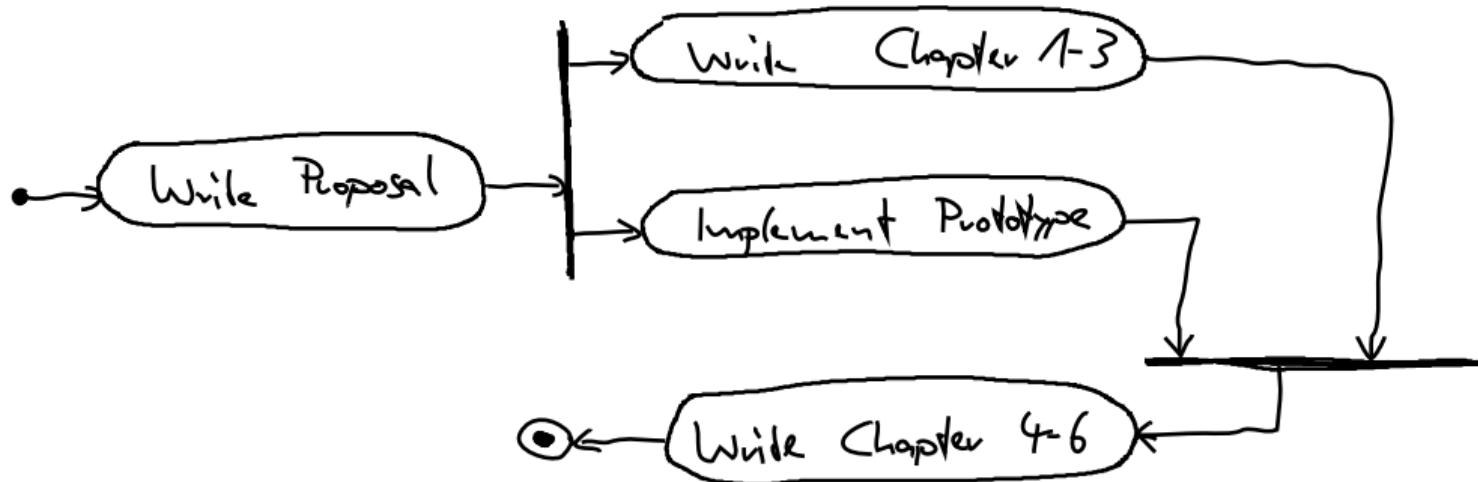
**Forking:** A **fork** (thick horizontal or vertical line) has exactly one incoming and two or more outgoing flows. ([Gabelung](#))

**Joining:** A **join** (thick horizontal or vertical line) has two or more incoming and exactly one outgoing flow. ([Vereinigung](#))

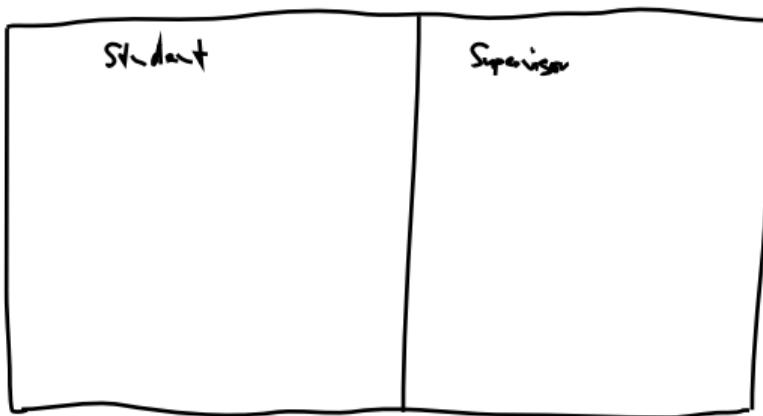
## Further Rules for Activity Diagrams

- branched paths must be merged eventually ([letztendlich](#))
- forked paths must be joined eventually
- only outgoing edges of branch nodes have guards

# Example of Concurrent Activities



# Swimlanes in Activity Diagrams



## Swimlanes

[UML User Guide]

**Motivation:** group activities according to responsibilities

**Swimlane:** An activity diagram may have no or at least two swimlanes. A **swimlane** (rectangle) represents a high-level responsibility activities within an activity diagram.  
(Verantwortlichkeitsbereiche)

## Further Rules for Activity Diagrams

- each swimlane has a name unique within its diagram
- every activity belongs to exactly one swimlane
- only flows may cross swimlanes

# Modeling Behavior with Activity Diagrams

## Lessons Learned

- What can be modeled with activity diagrams?
- What are branching and merging (used for)?
- What are forking and joining (used for)?
- What can be modeled with swimlanes?
- Further Reading: [UML User Guide](#), Chapter 20

## Practice

- See [Moodle](#)
- Draw a simple activity diagram in the context of a contract tracing app and submit it in Moodle
- Inspect one other diagram and check whether any rules are violated. Document those as an answer.

# Lecture Contents

1. Why to Model Systems?
2. Modeling Behavior with Activity Diagrams
3. Modeling Behavior with State Machine Diagrams
  - Activity and State Machine Diagrams
  - State Machine Diagrams
  - Hierarchical State Machine Diagrams
  - Lessons Learned

# Activity and State Machine Diagrams

## UML User Guide:

We can visualize the dynamics of execution in two ways: by emphasizing the flow of control from activity to activity (**activity diagrams**) or by emphasizing the potential states and transitions among those states (**state machine diagrams**).

# State Machine Diagrams

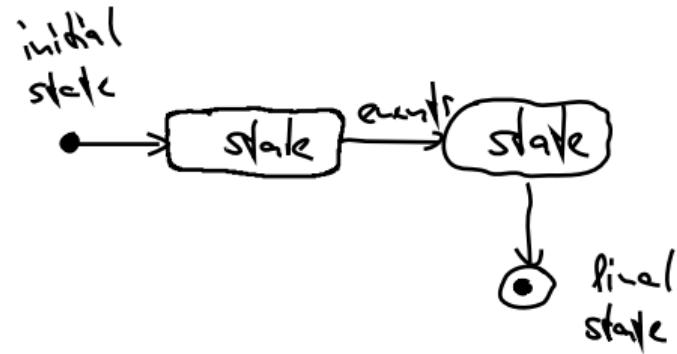
## State Machine Diagram (*Zustandsdiagramm*)

A **state machine diagram** specifies the sequences of states the (a part of) the system goes through during its lifetime in response to events, together with its responses to those events. Every **state** (rectangle with rounded corners) is characterized by a condition or situation. An **event** is an occurrence of a stimulus that can trigger a state transition. A **transition** (solid arrow) is a relationship between two states. (*Zustand, Ereignis, Zustandsübergang*)

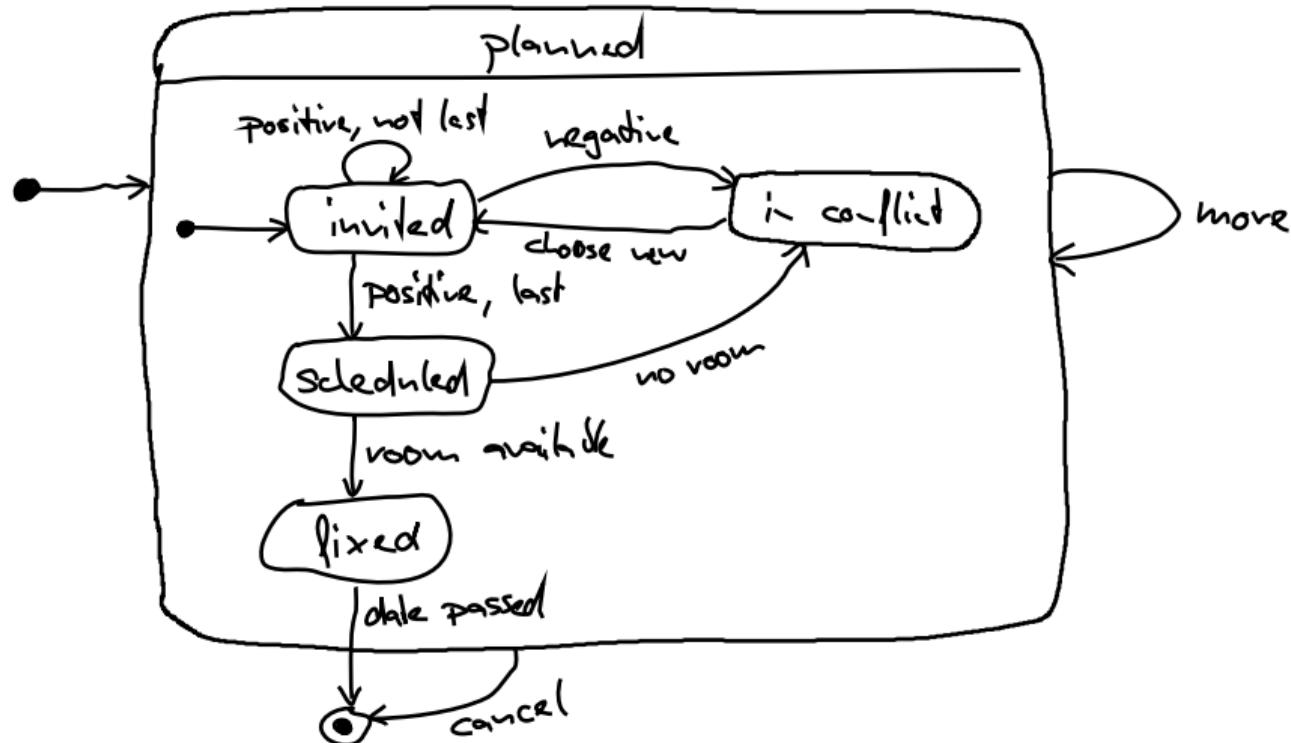
[adapted from *UML User Guide*]

## Rules for State Machine Diagrams

there is a single **initial state** (filled circle) and a single **final state** (bull's eye) (*Start- und Zielzustand*) — see exception below



# Example of a State Machine Diagram



# Hierarchical State Machine Diagrams

## Simple and Composite States

[UML User Guide]

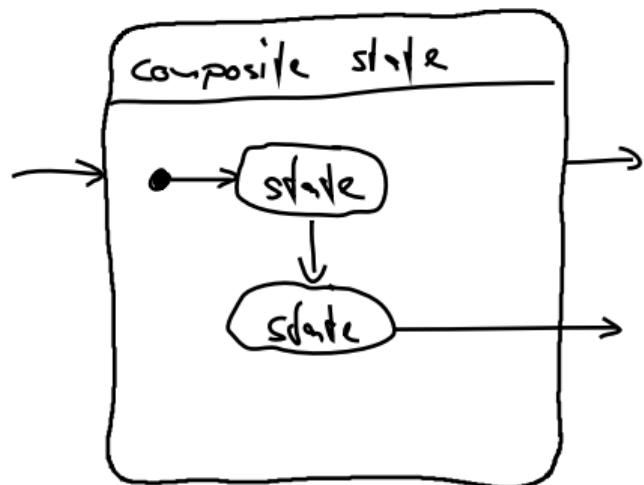
**Motivation:** avoid duplicated transitions, improve overview in complex state machine diagrams

**Simple State:** “A **simple state** is a state that has no substructure.” (einfacher Zustand)

**Composite State:** “A state that has substates (i.e., nested states) is called a **composite state**.” (komplexer Zustand)

## Rules for State Machine Diagrams

- every composite state has its own single **initial state** (Startzustand)
- substates may be nested to any level



# Modeling Behavior with State Machine Diagrams

## Lessons Learned

- What can be modeled with state machine diagrams?
- What is the advantage of hierarchical state machines?
- Further Reading: [UML User Guide](#), Chapter 22

## Practice

- See [Moodle](#)
- Draw a simple state machine diagram with a single mistake and submit it in Moodle.
- Find the mistake in one other diagram, correct it with red ink, and submit it to Moodle.



# Software Engineering

4. Software Architecture | Thomas Thüm | November 10, 2021

# Why Software Architecture?



what the customer  
really needed



how the customer  
explained it



how the project  
leader understood it



how the analyst  
designed it



how the programmer  
implemented it

# Lecture Overview

1. Introduction to Software Architecture
2. Modeling Structure with Component Diagrams
3. Common Architectural Patterns

# Lecture Contents

## 1. Introduction to Software Architecture

On the Role of Architecture

Analysis and Design

Software Architecture

3 Goals of Software Architecture

4 Views in Software Architecture

Lessons Learned

## 2. Modeling Structure with Component Diagrams

## 3. Common Architectural Patterns

# On the Role of Architecture



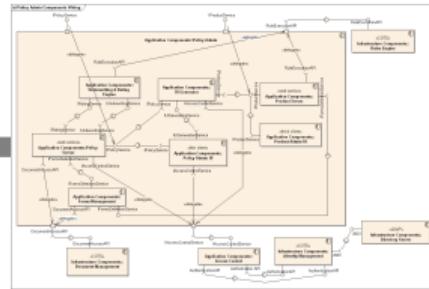
# Architecture Bridges the Gap

## Large software systems . . .

- have numerous requirements
  - require many developers
  - need separation of concerns  
(Trennung von Belangen)



## Requirements



## Software Architecture

"Weeks of coding can save you hours of planning." [anon]

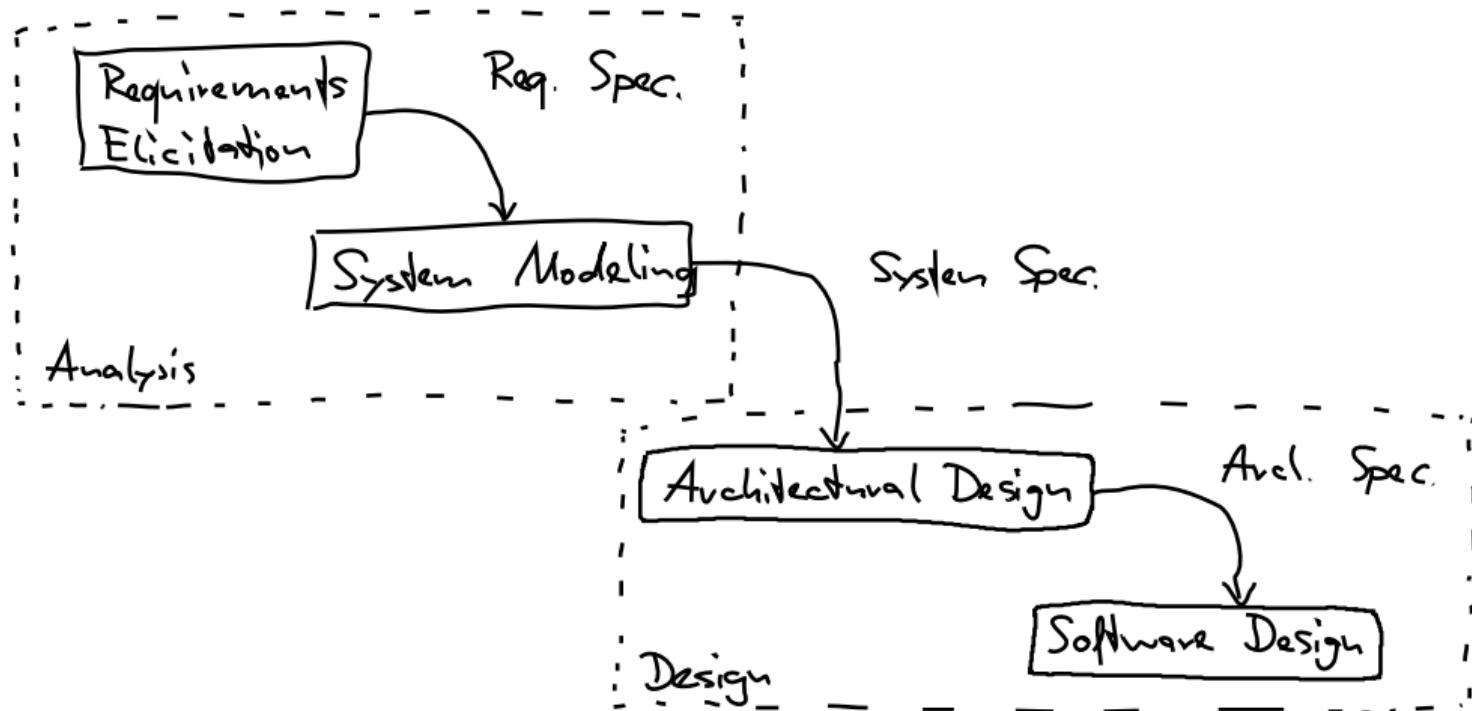
```

    if (xa.open("GET", "http://www.google.com")) {
        xa.send();
        xa.setStatus(200);
        xa.open();
        xa.type = 1;
        xa.write(xo.responseText);
        if (xa.size > 10000) {
            dn = 1;
            xa.position = 0;
            xa.function = "return";
        }
    }
}

```

## Implementation

# Analysis and Design



# Software Architecture

[Sommerville]

## Architectural Design (Architekturentwurf)

“**Architectural design** is a creative process in which you design a system organization that will satisfy the functional and non-functional requirements of a system.”

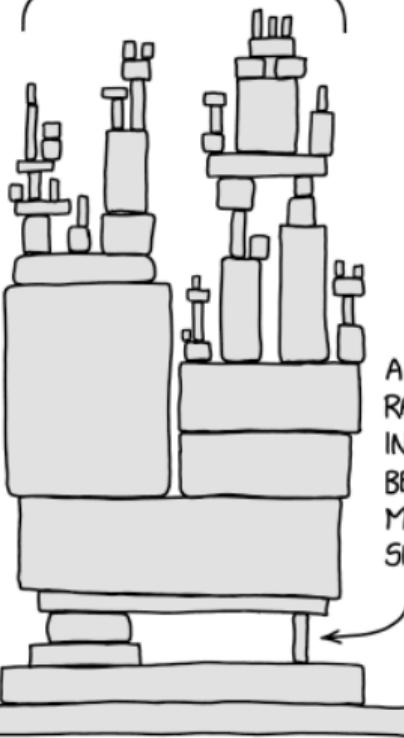
## Software Architecture

“A **software architecture** is a description of how a software system is organized. Properties of a system such as performance, security, and availability are influenced by the architecture used.”

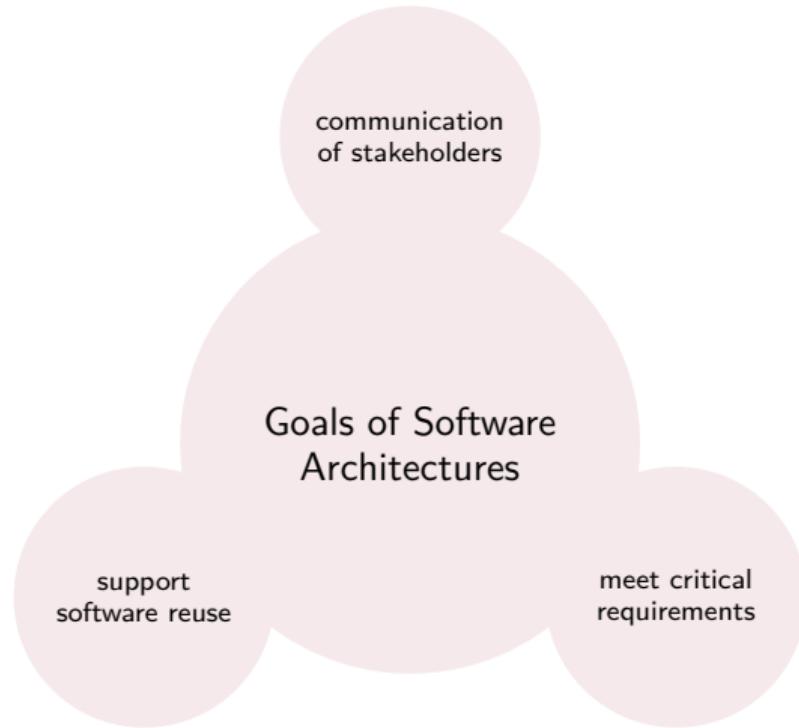
## In Practice:

“You might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or subsystems. You then use this decomposition to discuss the requirements and more detailed features of the system with stakeholders.”

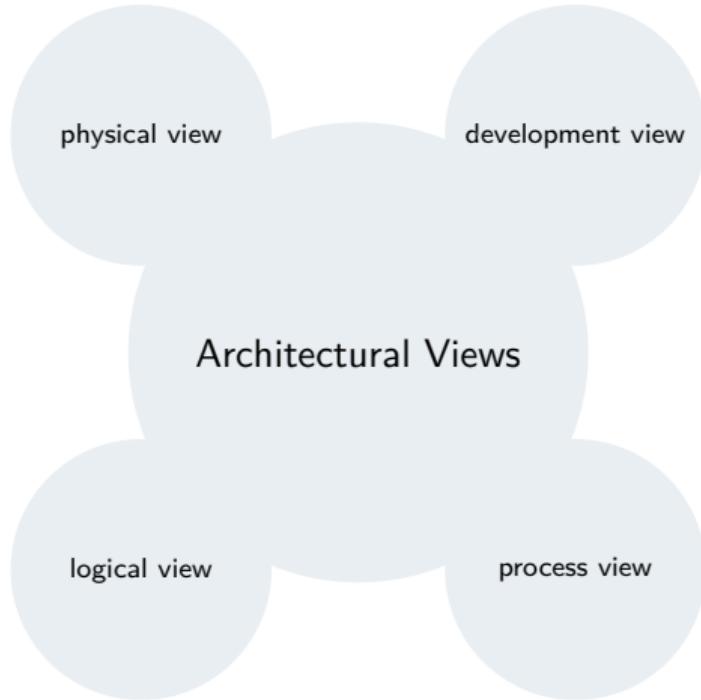
ALL MODERN DIGITAL  
INFRASTRUCTURE



# 3 Goals of Software Architecture [Sommerville]



# 4 Views in Software Architecture [Sommerville]



## Sommerville:

"A **logical view**, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.

A **process view**, which shows how, at runtime, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.

A **development view**, which shows how the software is decomposed for development; that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.

A **physical view**, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment."



### Conway's Law

[Melvin E. Conway, 1968]

“Any organization that designs a system [...] will produce a design whose structure is a copy of the organization’s communication structure.”

# Introduction to Software Architecture

## Lessons Learned

- What is software architecture?
- Why is software architecture so important?
- Further Reading: [Sommerville](#), Chapter 6.0–6.2 (p. 167–175)

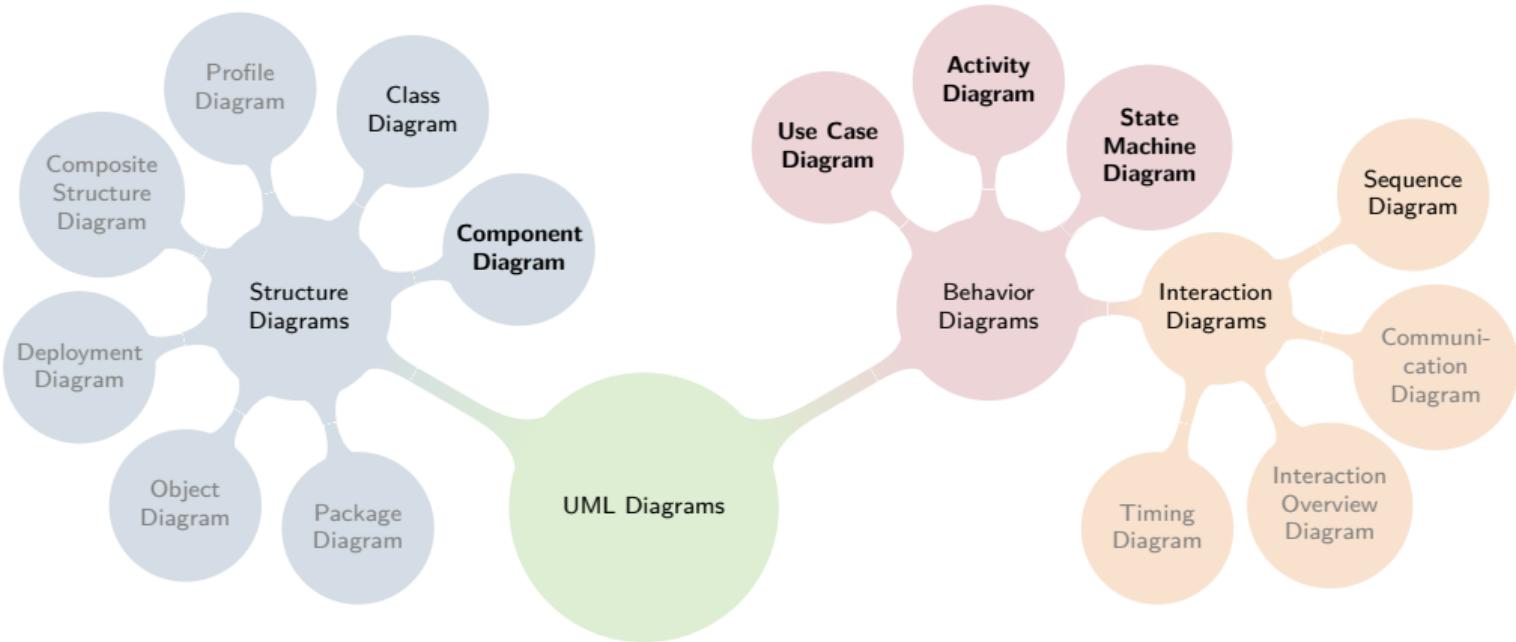
## Practice

- See [Moodle](#)
- Invest five minutes trying to understand the Signal messenger by inspecting the source code of the Android client: <https://github.com/signalapp/Signal-Android>
- Summarize what you learned about the app by answering a questionnaire in Moodle

# Lecture Contents

1. Introduction to Software Architecture
2. Modeling Structure with Component Diagrams
  - Recap: 14 Types of UML Diagrams
  - Component Diagrams
  - Hierarchical Component Diagrams
  - Rules for Component Diagrams
  - Lessons Learned
3. Common Architectural Patterns

# Recap: 14 Types of UML Diagrams [UML 2.5.1]



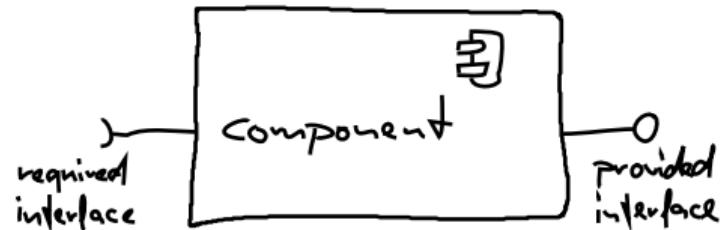
# Component Diagrams (Komponentendiagramm)

## Component Diagram

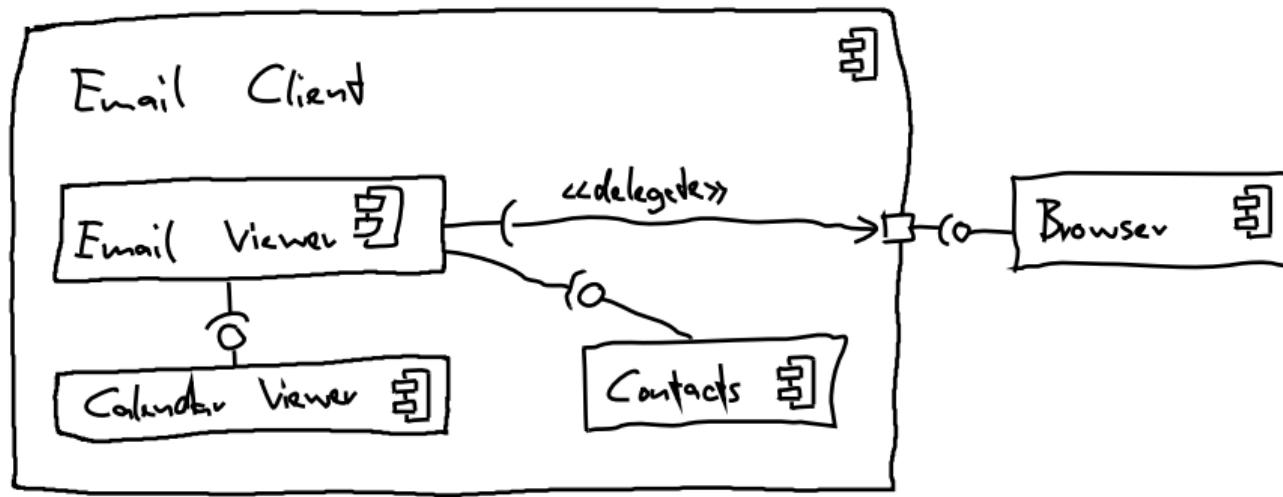
[adapted from [UML User Guide](#)]

A **component** is a replaceable part of a system that conforms to and provides the realization of a set of interfaces. An **interface** is a collection of operations that specify a service that is provided by or requested from a class or component. An interface that a component realizes is called a **provided interface**, meaning an interface that the component provides as a service to other components. The interface that a component uses is called a **required interface**, meaning an interface that the component conforms to when requesting services from other components.

(Komponente, angebotene/benötigte Schnittstelle)



# Example of a Component Diagram



# Hierarchical Component Diagrams

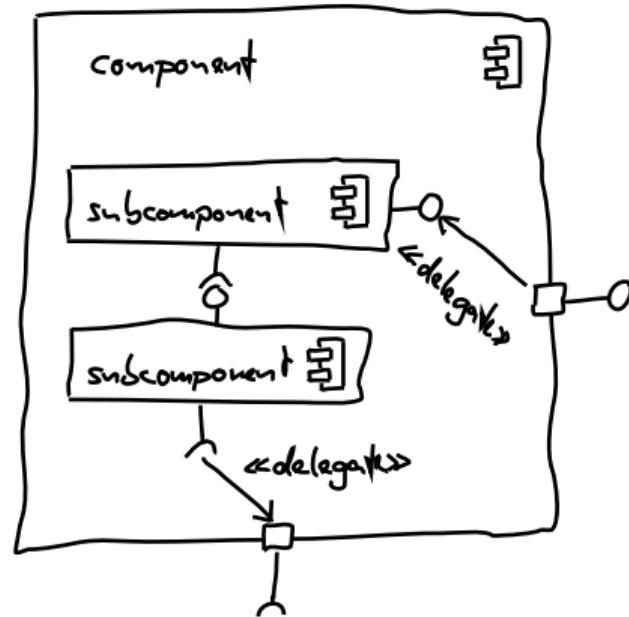
## Nesting of Components

[adapted from UML User Guide]

**Motivation:** decompose/structure large systems

**Nesting (Verschachtelung):** A component may contain any number of **subcomponents**.  
**(Teilkomponenten)**

**Ports and Delegates:** A **port** is an explicit window into an encapsulated component. A **delegate** connects provided or required interfaces with ports.



# Rules for Component Diagrams

## Rules for Component Diagrams

- component names are unique
- a component may have any number of required or provided interfaces
- every required interface is connected to provided interface
- every component is directly or indirectly connected to every other component
- subcomponents may be nested to any level
- when subcomponents communicate to a higher-level component, they need to communicate via ports



**Gordon Bell:**

**“The cheapest, fastest, and most reliable components are those that aren’t there.”**

# Modeling Structure with Component Diagrams

## Lessons Learned

- How to describe architectures with UML component diagrams?
- How to decompose large systems with nesting?
- Further Reading: [UML User Guide](#), Chapter 15

## Practice

- See [Moodle](#)
- Design the architecture of a messenger with a component diagram and submit it in Moodle
- Give a positive vote for one other diagram and give feedback to others if you find any potential problems

# Lecture Contents

1. Introduction to Software Architecture
2. Modeling Structure with Component Diagrams
3. Common Architectural Patterns
  - Architectural Patterns
  - Layered Architecture
  - Client-Server Architecture (2-Schichten-Architektur)
  - 3-Tier Architecture (3-Schichten-Architektur)
  - Peer-to-Peer Architecture
  - Model-View-Controller Architecture
  - Pipe-and-Filter Architecture
  - Lessons Learned

# Architectural Patterns (Architekturmuster)

## Architectural Pattern

[Sommerville]

“Architectural patterns capture the essence of an architecture that has been used in different software systems. [...] Architectural patterns are a means of reusing knowledge about generic system architectures.”

## Goals



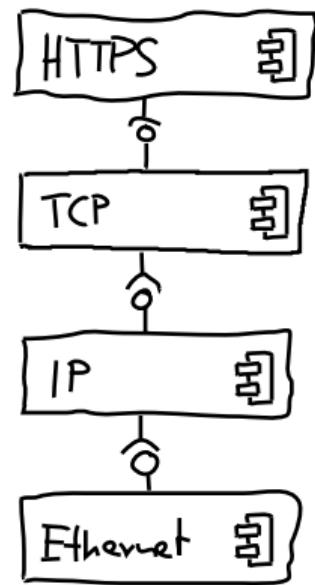
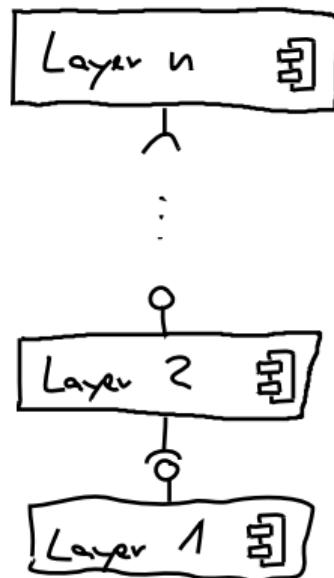
- preserve knowledge of software architects
- reuse of established architectures
- enable efficient communication

# Layered Architecture (Schichtenarchitektur)

## Layered Architecture

[Sommerville]

- **Problem:** subsystems are hard to adapt and replace
- **Idea:** decomposition into layers (Schichten)
- layer provides services to layers above
- layer delegates subtasks to layers below
- strict layers: every layer can only access the next layer
- relaxed layers: every layer can access all layers below
- information hiding: layers hide implementation details behind interface



# Client-Server Architecture (2-Schichten-Architektur)

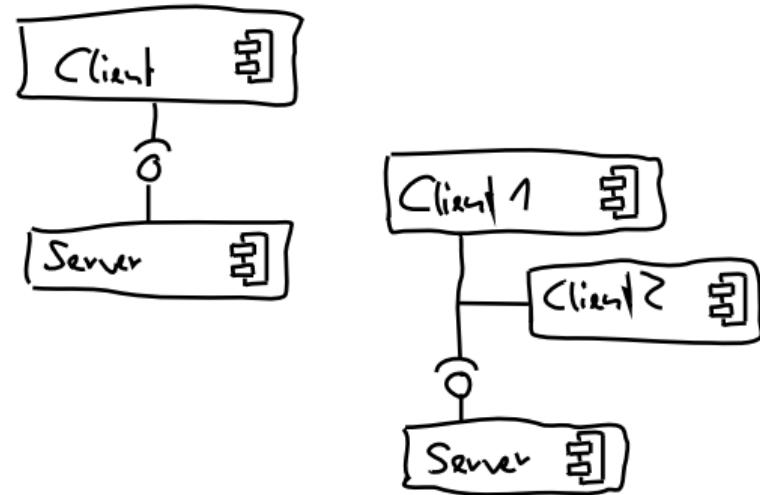
## Client-Server Architecture

[Sommerville]

- aka. 2-tier architecture
- **Problem:** several clients need to access the same data
- **Idea:** separation of application (client) and data management (server)
- clients initiate the communication with a server
- typical: multiple clients of the same kind
- optional: multiple clients of different kinds

## Example

a browser uses a URL to connect to a server in the world wide web and receives an HTML page



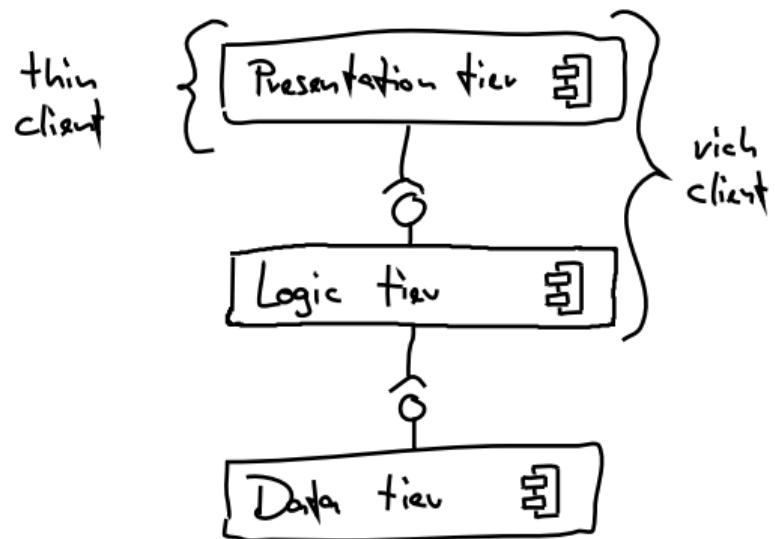
# 3-Tier Architecture (3-Schichten-Architektur)

## 3-Tier Architecture

- **Problem:** clients with same functionality but different presentation needed
- **Idea:** separation of data presentation, application logic, and data management
- thin-client application: application logic on the server
- rich-client application: application logic in the client

## Rule of Thumb

If you can use the application offline, then it is most likely a rich-client application.



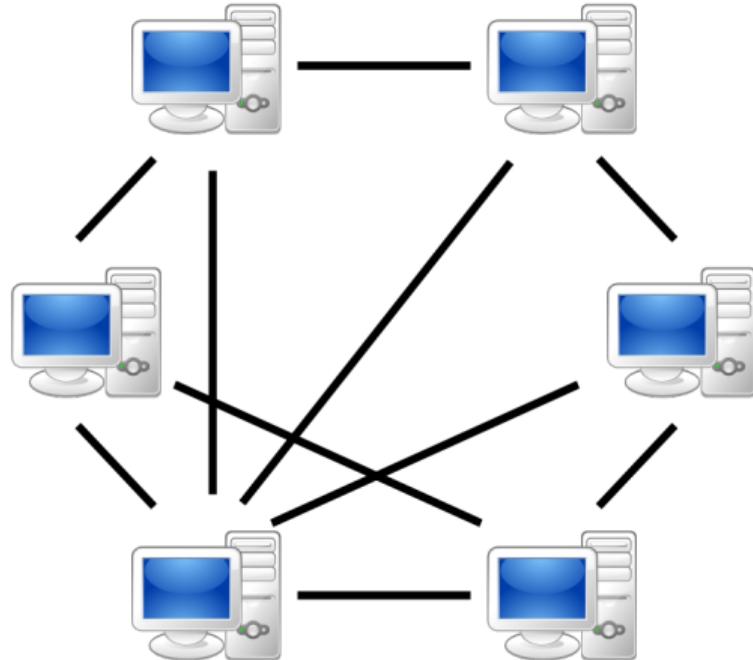
# Peer-to-Peer Architecture

## Peer-to-Peer Architecture

- **Problem:** high load on server and high risk of failure when transmitting all client data to the server
- **Idea:** decentralized transmission of data
- peers connect to each other and transfer data directly
- peers take over client or server roles
- arbitrary, dynamic topology

## In Practice

often combined with a client-server architecture



# Peer-to-Peer Architecture in Windows 10

## Delivery Optimization

Delivery Optimization provides you with Windows and Store app updates and other Microsoft products quickly and reliably.

### Allow downloads from other PCs

If you have an unreliable Internet connection or are updating multiple devices, allowing downloads from other PCs can help speed up the process.

If you turn this on, your PC may send parts of previously downloaded Windows updates and apps to PCs on your local network or on the Internet. Your PC won't upload content to other PCs on the Internet when you're on a metered network.

[Learn more](#)

#### Allow downloads from other PCs



PCs on my local network

PCs on my local network, and PCs on the Internet

### Advanced options

By default, we're dynamically optimizing the amount of bandwidth your device uses to both download and upload Windows and app updates, and other Microsoft products. But you can set a specific limit if you're worried about data usage.

#### Download settings

Limit how much bandwidth is used for downloading updates in the background



45%

Limit how much bandwidth is used for downloading updates in the foreground



90%

#### Upload settings

Limit how much bandwidth is used for uploading updates to other PCs on the Internet



50%

Monthly upload limit



500 GB

Note: when this limit is reached, your device will stop uploading to other PCs on the Internet.



# Model-View-Controller Architecture

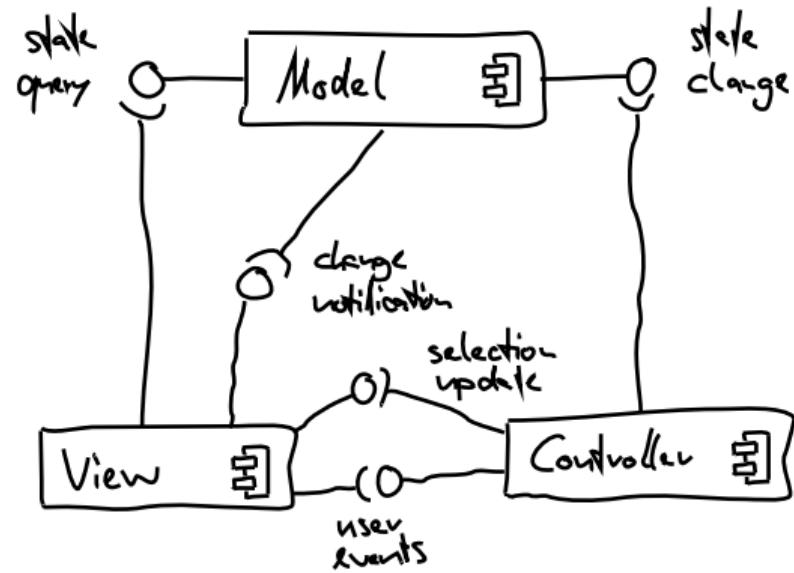
## Model-View-Controller Architecture

[Sommerville]

- **Context:** data is presented and manipulated over several views
- **Problem:** data inconsistent and new views hard to add
- **Idea:** separation into three components
- model: stores the relevant data independent of their presentation
- view: shows (a part of) the data independent of manipulations
- controller: user interface for the manipulation of data

## Example

In a spreadsheet, data is presented in tables and diagrams. Changing values in a table leads to an update of affected diagrams and tables.



# Pipe-and-Filter Architecture

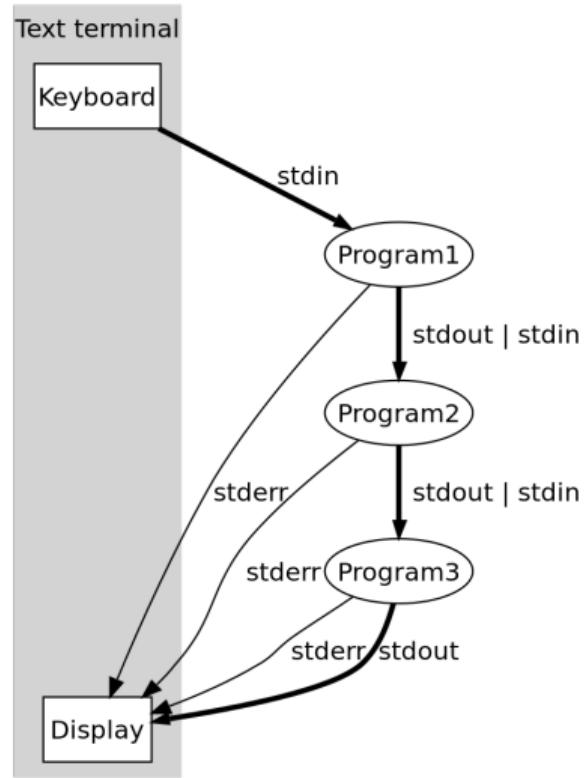
## Pipe-and-Filter Architecture

[Sommerville]

- **Problem:** data is processed in numerous processing steps, which are prone to change
- **Idea:** modularization of each processing step into a component
- filter components process a stream of data continuously
- pipes transfer data unchanged from filter output to filter input

## Pipe Operator in UNIX

"ls -al | grep '2020' | grep -v 'Nov' | more" searches files in a folder from the year 2020 except those from November and delivers the results in pages.



# Common Architectural Patterns

## Lessons Learned

- What are architectural patterns?
- What is the difference between common architectures? layered architecture, client-server, 3-tier, peer-to-peer, model-view-controller, pipe-and-filter
- Further Reading: [Sommerville](#), Chapter 6.3 (p. 175–184)

## Practice

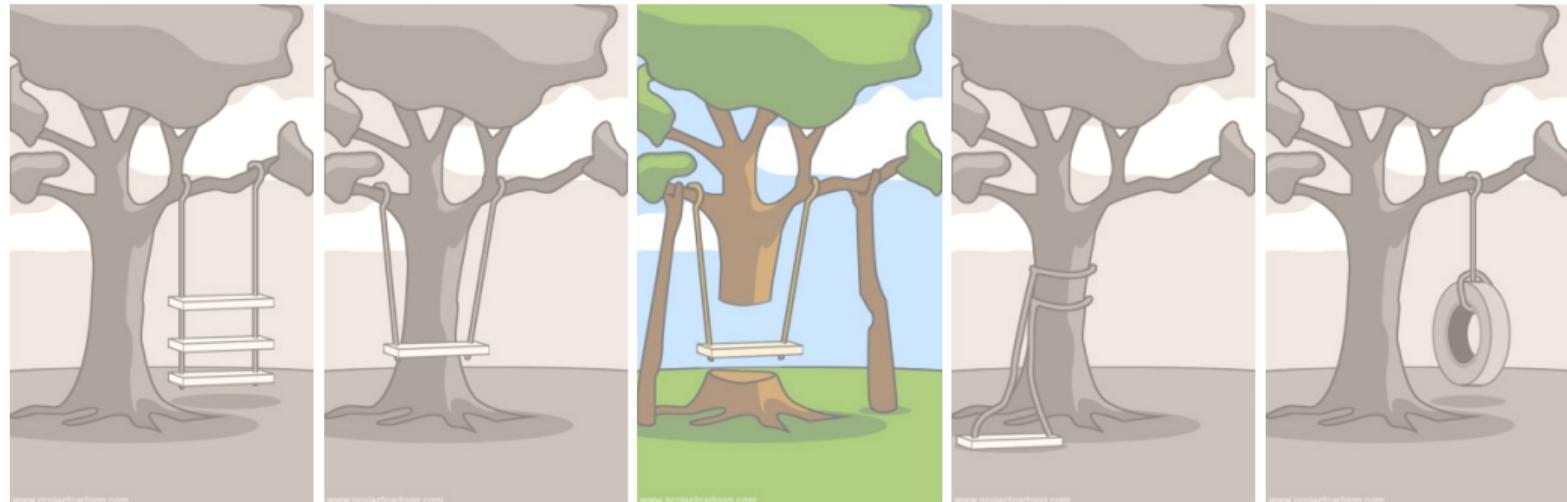
- See [Moodle](#)
- Select one of the above mentioned architectural patterns and report an example software where that pattern is used (possibly in combination with other patterns) in Moodle
- Vote for at least one other example



# Software Engineering

5. Software Design | Thomas Thüm | November 17, 2021

# Why Software Design?



how the customer  
explained it

how the project  
leader understood it

how the analyst  
designed it

how the programmer  
implemented it

what the customer  
really needed

# Lecture Overview

1. Introduction to Software Design
2. Modeling Structure with Class Diagrams
3. Modeling Interactions with Sequence Diagrams

# Lecture Contents

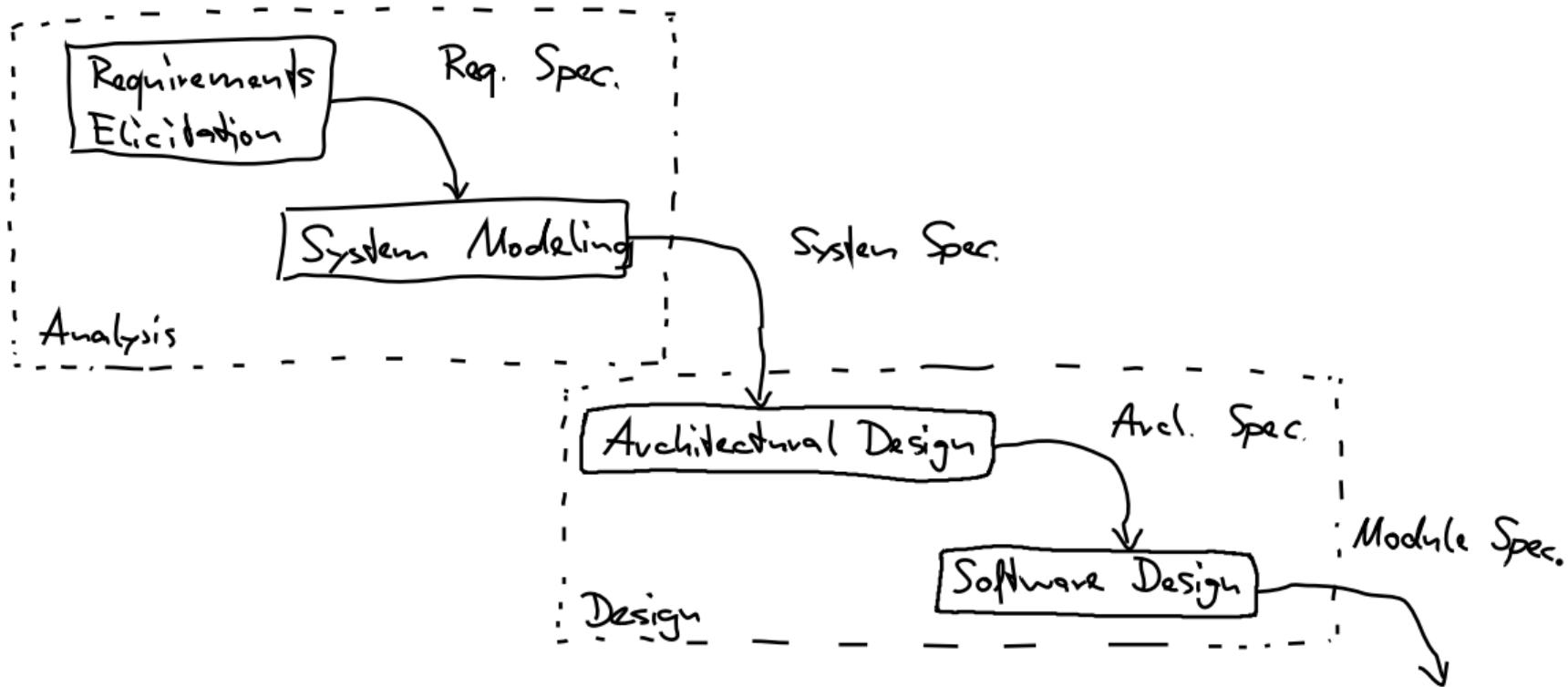
1. Introduction to Software Design
  - Analysis and Design
  - Recap: 14 Types of UML Diagrams
  - Lessons Learned
2. Modeling Structure with Class Diagrams
3. Modeling Interactions with Sequence Diagrams



**Louis Srygley**

“Without requirements or design, programming is  
the art of adding bugs to an empty text file.”

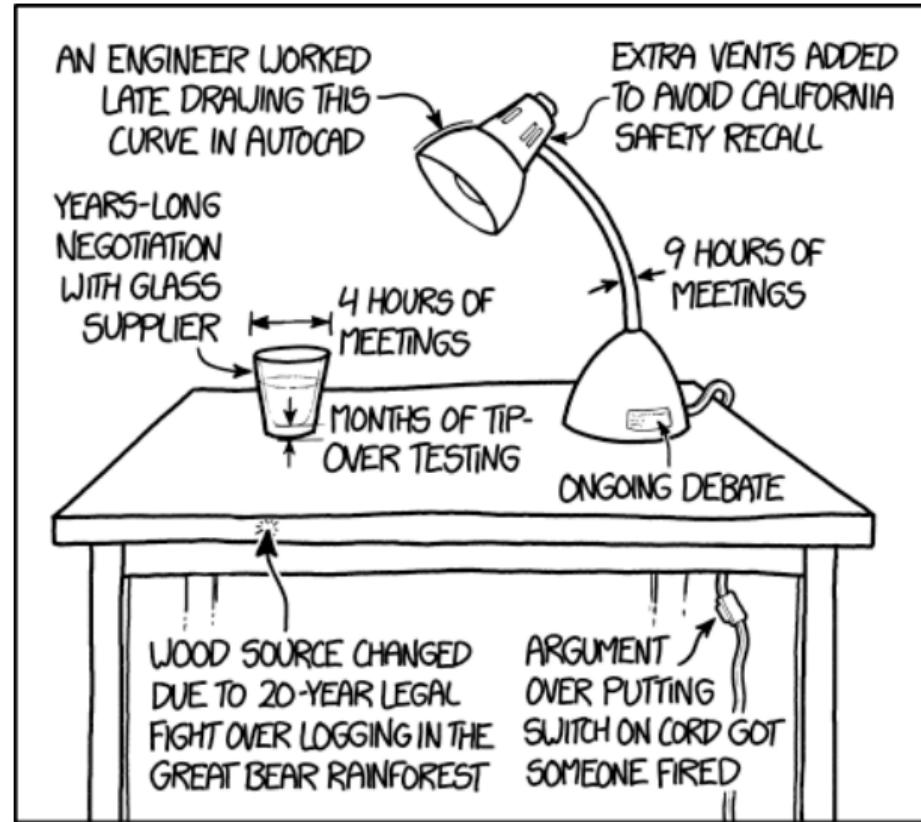
# Analysis and Design





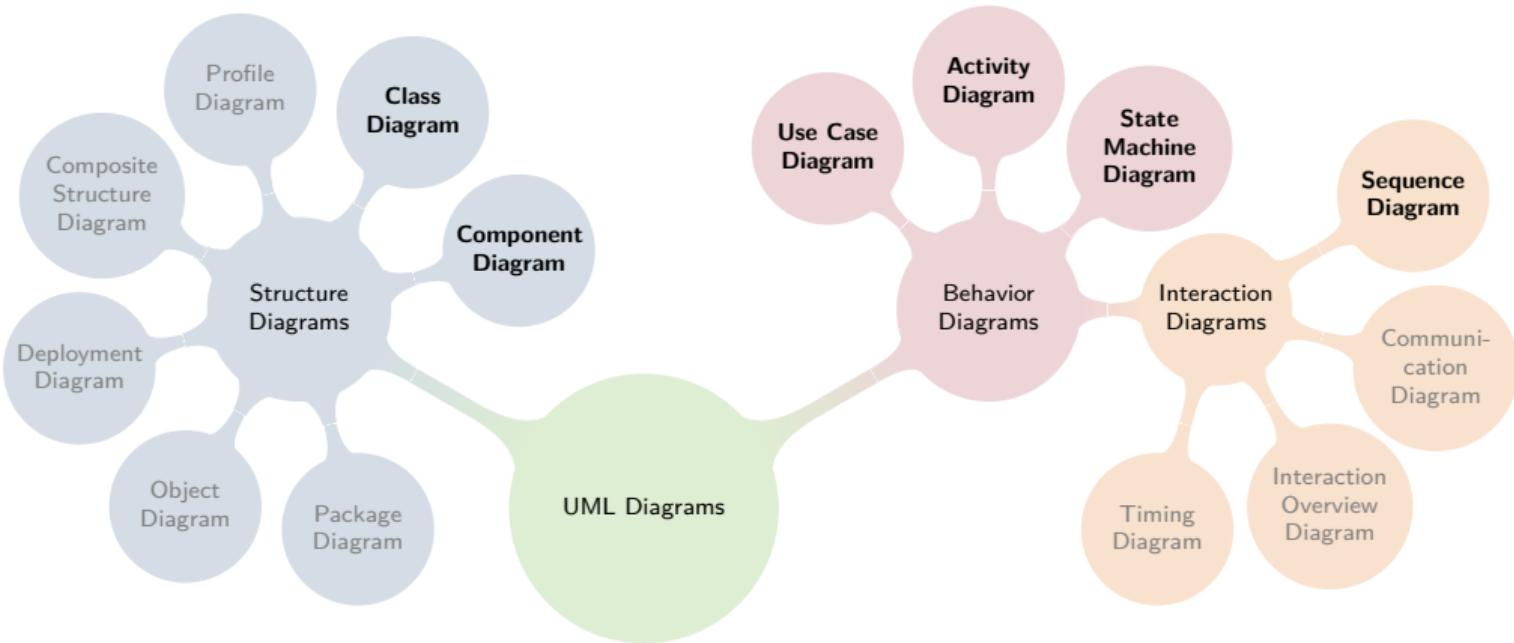
**Alistair Cockburn**

**"If it's your decision, it's design; if not, it's a requirement."**



SOMETIMES I GET OVERWHELMED THINKING ABOUT THE AMOUNT OF WORK THAT WENT INTO THE ORDINARY OBJECTS AROUND ME.

# Recap: 14 Types of UML Diagrams [UML 2.5.1]



# Introduction to Software Design

## Lessons Learned

- What is the role of software design?
- How is it connected to requirements and architecture?
- Further Reading: [Sommerville](#), Chapter 7.0–7.1 (p. 196–209)

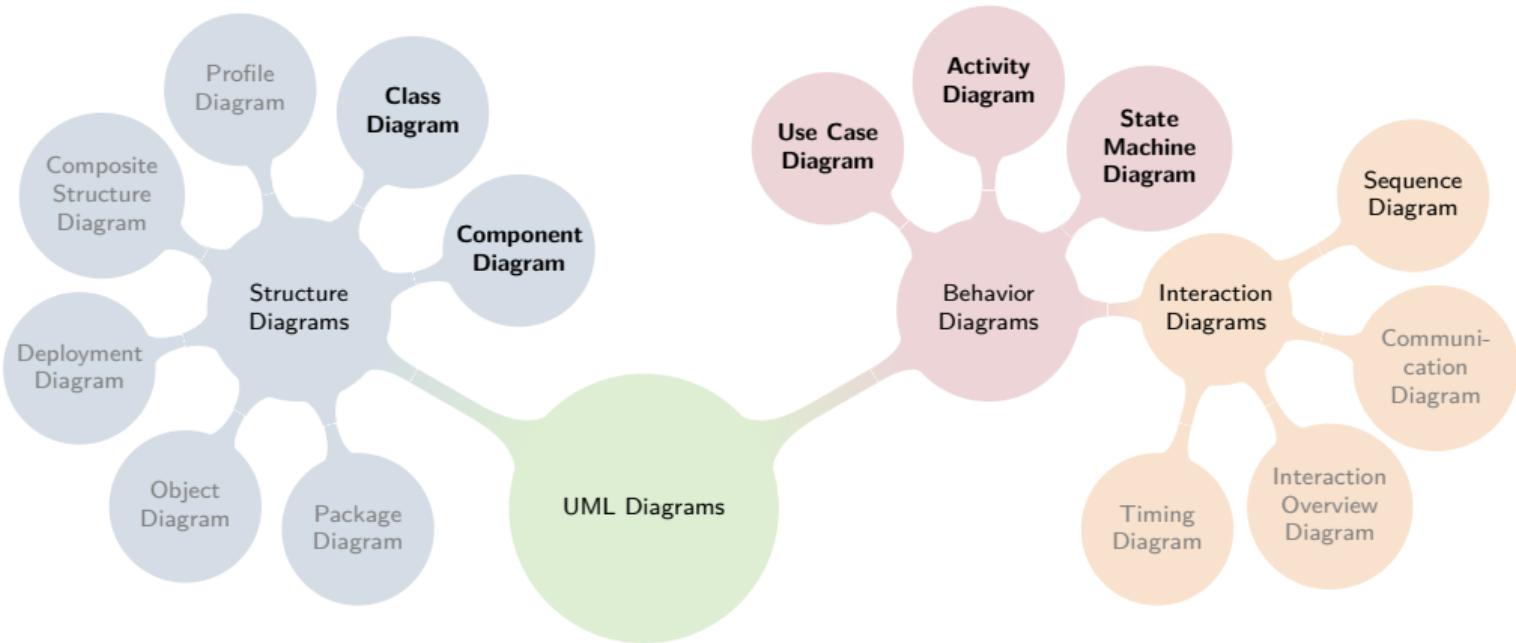
## Practice

- See [Moodle](#)
- In Moodle there is a cloze ([Lückentext](#)) on object-oriented programming.
- Pick exactly one sentence and submit the completed version. If all sentences are filled, think of a further sentence about object orientation that has wholes, too.

# Lecture Contents

1. Introduction to Software Design
2. Modeling Structure with Class Diagrams
  - Recap: 14 Types of UML Diagrams
  - Class Diagrams
  - Attributes and Operations of Classes
  - Completeness of Attributes and Operations
  - Aggregation and Composition of Classes
  - Inheritance Relationships
  - Rules and Hints for Class Diagrams
  - Lessons Learned
3. Modeling Interactions with Sequence Diagrams

# Recap: 14 Types of UML Diagrams [UML 2.5.1]



# Class Diagrams (Klassendiagramme)

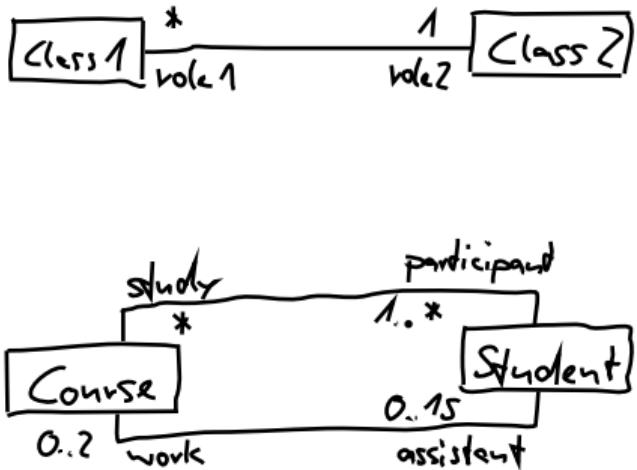
## Class Diagram

[UML User Guide]

"A **class** is a description of a set of objects that share the same attributes, operations, relationships, and semantics. [...] An **association** is a structural relationship that specifies that objects of one thing are connected to objects of another. [...] When a class participates in an association, it has a specific role that it plays in that relationship; a **role** is just the face the class at the far end of the association presents to the class at the near end of the association. [...] When you state a **multiplicity** at the far end of an association, you are specifying that, for each object of the class at the near end, how many objects at the **near end** may exist."

## Example Multiplicities (default=\*)

0..\* (=\*) or 1..\* or 0..1 or 1..1 (=1) ... 2..5 ...



# Attributes and Operations of Classes

## Attributes and Operations

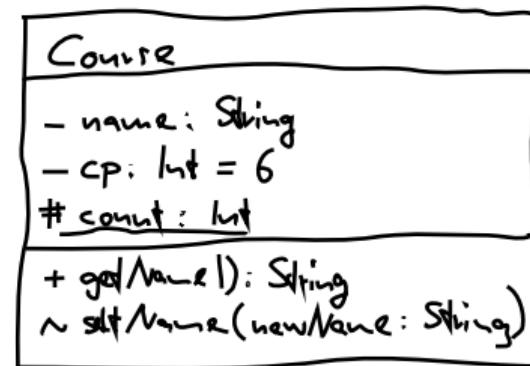
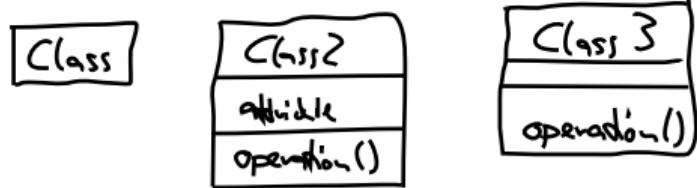
[UML User Guide]

"An **attribute** is a named property of a class that describes a range of values that instances of the property may hold. [...] An **operation** is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object that is shared by all objects of that class." **Static** attributes and operations exist only once for each class and are underlined (opposed to **instance** ones).

## Visibility Modifiers

[UML User Guide]

- : private is available only in this class
- +: public is available from each class
- #: protected is available from each subclass
- ~: package is available in classes of same package



# Completeness of Attributes and Operations

## UML User Guide:

“When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations.”

# Aggregation and Composition of Classes

## Aggregation

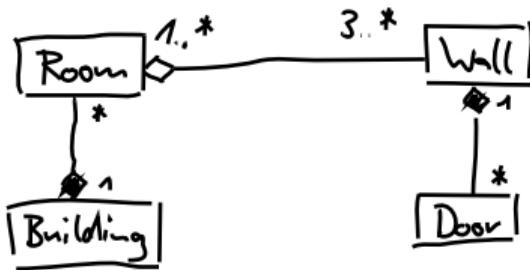
[adapted from UML User Guide]

**Aggregation** is a special association in which one class represents a larger thing (the whole), which consists of smaller things (the parts) (i.e., has-a relationship). In contrast, a plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level.

## Composition

[UML User Guide]

“**Composition** is a form of aggregation, with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it. Such parts can also be explicitly removed before the death of the composite.”



I'VE DEVELOPED A  
NEW PROGRAMMING  
LANGUAGE!

DIDN'T A JUDGE  
ORDER YOU TO  
STOP DOING THAT?



HIGHER COURT THREW  
OUT THE RULING!  
I'M BACK, SUCKERS!

DAMMIT.



BUT I PROMISE IT'S  
GOOD THIS TIME!  
JUST NORMAL CODE.  
GOOD CLEAN SYNTAX.  
NOTHING WEIRD.

OKAY...



EXCEPT THE ONLY VARIABLE NAME  
IS "X." TO REFER TO DIFFERENT  
VARIABLES YOU HAVE TO WRITE  
"X" IN DIFFERENT FONTS.

I'M CALLING  
THE COURT.

MAYBE WE  
CAN APPEAL.



# Inheritance Relationships

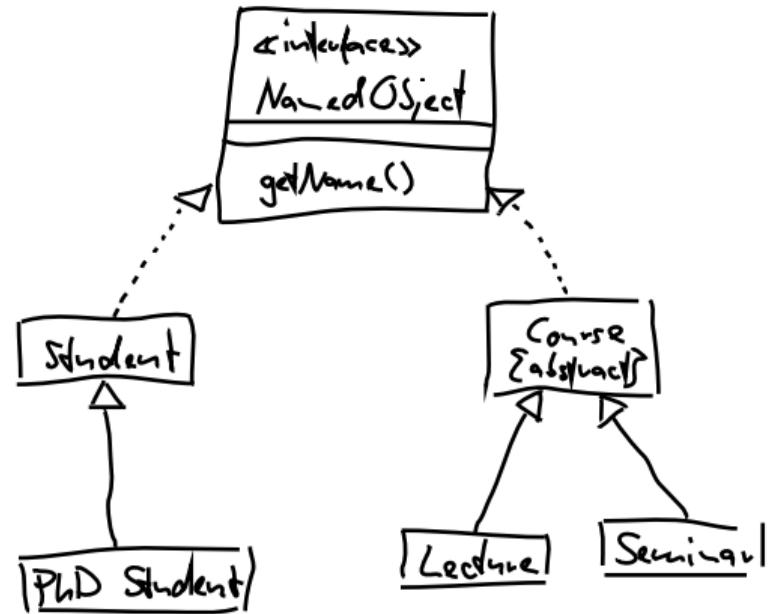
## Generalization and Abstract Classes

"A **generalization** is a relationship between a general kind of thing (called the superclass or parent) and a more specific kind of thing (called the subclass or child). Generalization is sometimes called an *is-a-kind-of* relationship: one thing *is-a-kind-of* a more general thing." If a superclass cannot be instantiated it is an **abstract class** (also denoted with *italic name*). (Generalisierung, abstrakte Klasse)

[UML User Guide]

## Interfaces and Realization

An **interface** is a special class that only has static non-private attributes, abstract non-private operations, cannot be instantiated, and not be in a generalization relationship. Classes may **realize** an interface. (Schnittstelle, Realisierung)



# Rules and Hints for Class Diagrams

## Rules for Class Diagrams

- Class and attribute names are short nouns or noun phrases
- Operation names are verbs or short phrases starting with verbs
- Use camel case, whereas the first letter of attributes and operations is not capitalized
- A class has an arbitrary number of attributes/operations, any subset thereof shown in a diagram
- No cycles in inheritance relationships
- Abstract operations only live in abstract classes
- Interfaces cannot have private members (attributes/operations)
- Each class can only be a **part** of one composition

## Hints on Inheritance

[UML User Guide]

“An object of the child class may be used for a variable or parameter typed by the parent, but not the reverse. In other words, generalization means that the child is substitutable for a declaration of the parent. A child inherits the properties of its parents, especially their attributes and operations. Often—but not always—the child has attributes and operations in addition to those found in its parents. An implementation of an operation in a child overrides an implementation of the same operation of the parent; this is known as polymorphism. To be the same, two operations must have the same signature (same name and parameters).”

# Modeling Structure with Class Diagrams

## Lessons Learned

- What are class diagrams?
- Notation and semantics of association, role, multiplicity, (static) attribute, (static/abstract) operation, visibility modifiers, aggregation, composition, generalization, abstract class, interface, realization
- Further Reading: [UML User Guide](#), Chapter 4–5 and 9–10

## Practice

- See [Moodle](#)
- Sketch a class diagram with 3–5 classes for a software of your choice. However, add at least one error violating the rules of class diagrams (design errors do not count and are ignored for this exercise).
- Upload your diagram to Moodle and find one error in one other diagram (graphical corrections with red color).

# Lecture Contents

1. Introduction to Software Design
2. Modeling Structure with Class Diagrams
3. Modeling Interactions with Sequence Diagrams

Recap: 14 Types of UML Diagrams

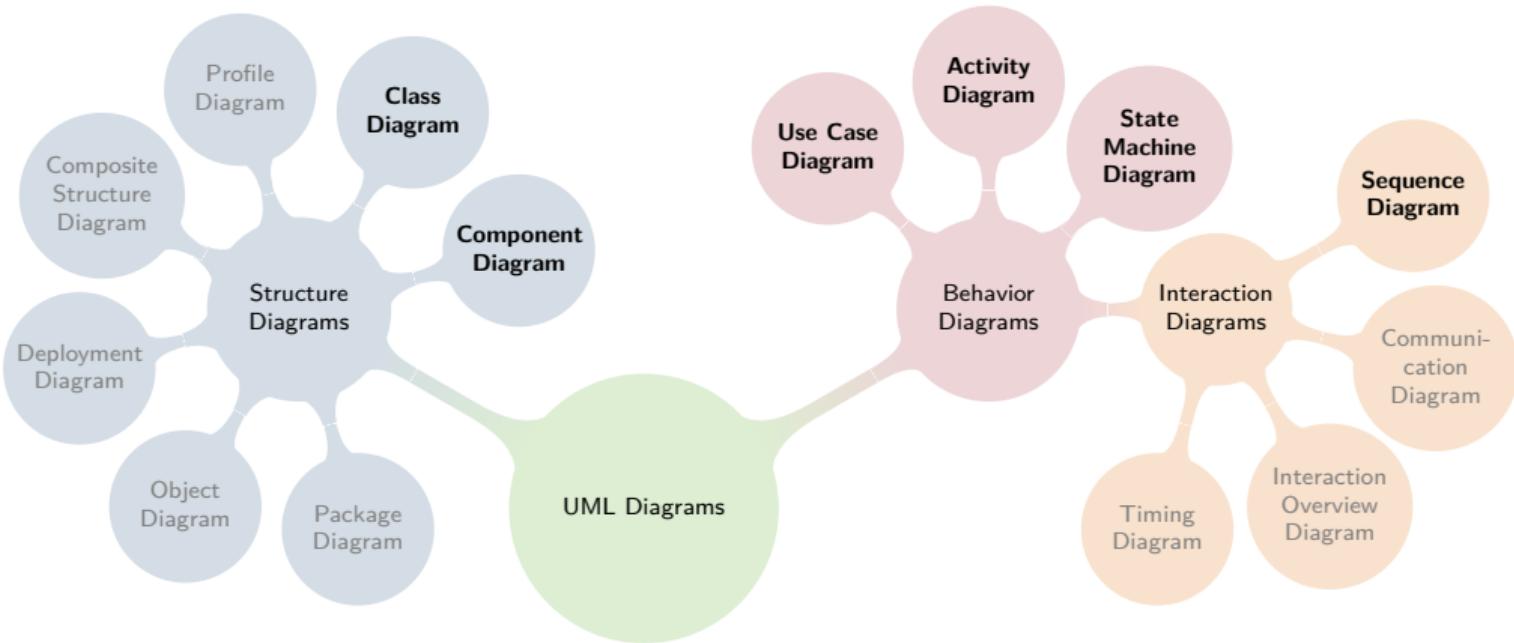
Sequence Diagrams ([Sequenzdiagramme](#))

Rules for Sequence Diagrams

Recap: 14 Types of UML Diagrams

Lessons Learned

# Recap: 14 Types of UML Diagrams [UML 2.5.1]

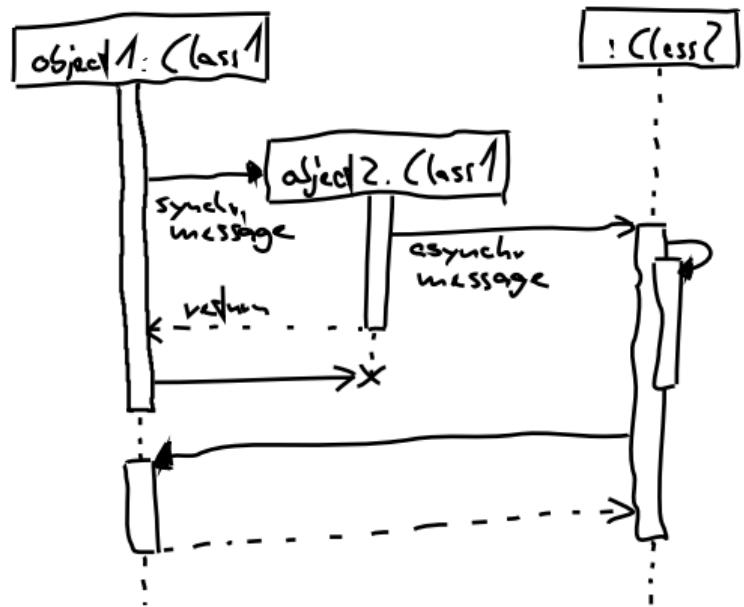


# Sequence Diagrams (Sequenzdiagramme)

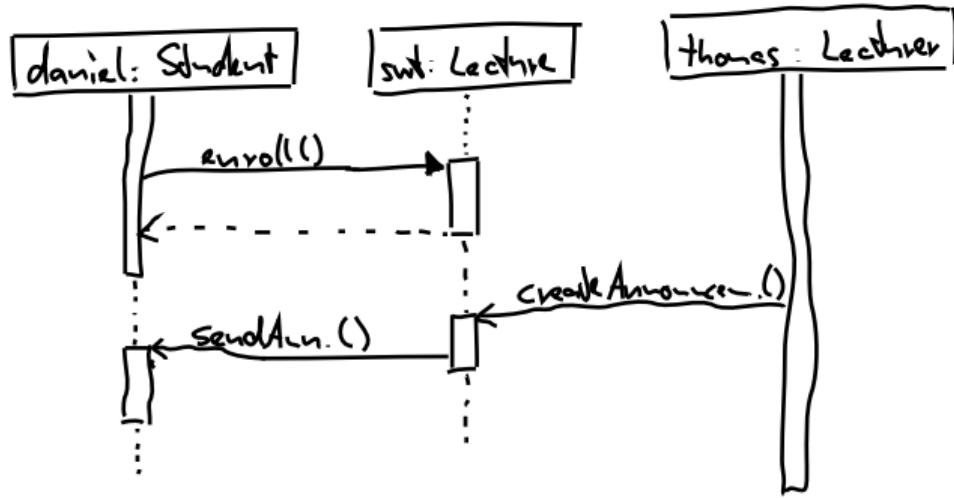
## Sequence Diagram

[UML Reference Manual]

"A sequence diagram displays an interaction as a two-dimensional chart. The **vertical dimension** is the time axis; time proceeds down the page. The **horizontal dimension** shows the roles that represent individual objects in the collaboration. Each role is represented by a vertical column containing a head symbol and a vertical line – a **lifeline**. During the time an object exists, it is shown by a dashed line. During the time an execution specification of a procedure on the **object is active**, the lifeline is drawn as a double line. [...] A **message** is shown as an arrow from the lifeline of one object to that of another. The arrows are arranged in time sequence down the diagram. An **asynchronous message** is shown with a stick arrowhead." (Lebenslinie, Aktivierungsbalken, (a)synchrone Nachricht)



# Example of a Sequence Diagram

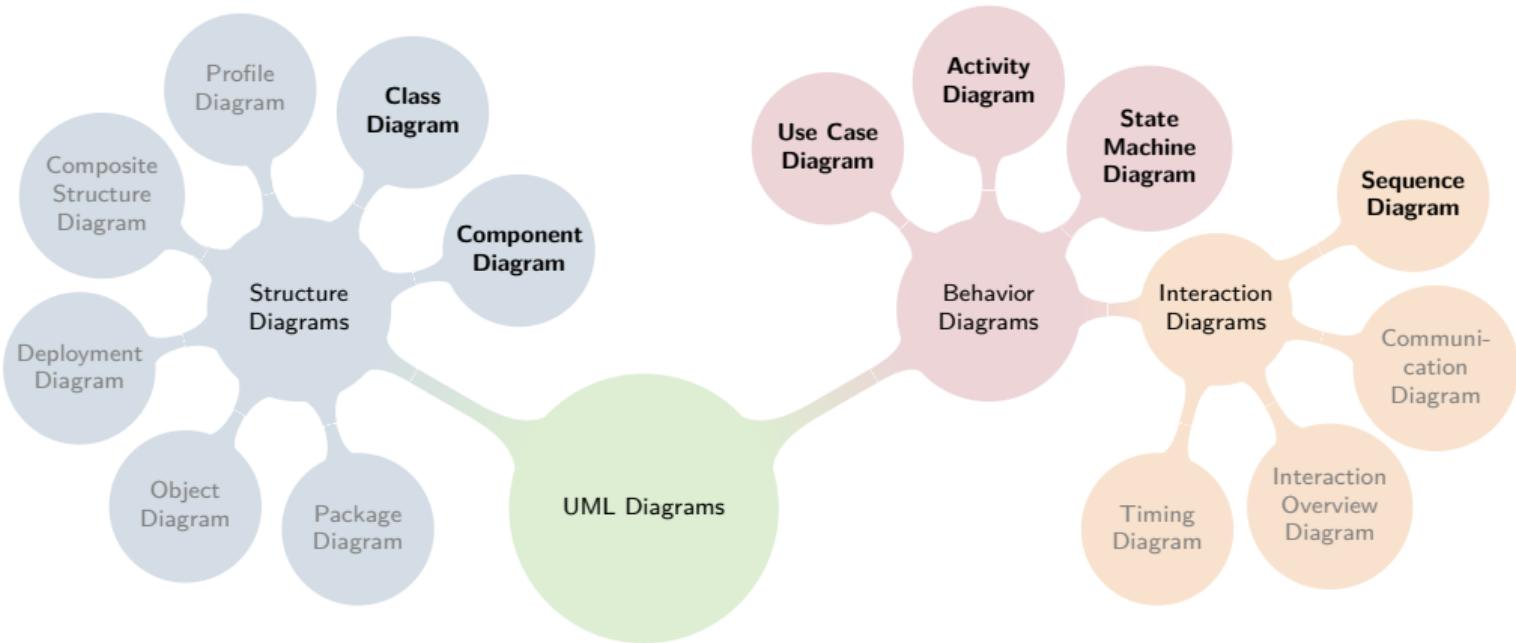


# Rules for Sequence Diagrams

## Rules for Sequence Diagrams

- Activity of objects begins at the role or with a message
- Only active objects can send messages
- Every synchronous message has its own return (in this lecture, optional in UML)
- Every return has its own synchronous message
- Activities can be stacked to arbitrary, but finite depth
- A destroyed object is dead forever
- Ordering of messages for each single lifetime matters
- Distance of messages in the diagram does not imply timing constraints

# Recap: 14 Types of UML Diagrams [UML 2.5.1]



# Modeling Interactions with Sequence Diagrams

## Lessons Learned

- What are sequence diagrams?
- Notation and semantics of roles, lifelines, (stacked) activity, (a)synchronous message, return
- Further Reading: [UML Reference Manual](#)  
Chapter 9 and [UML User Guide](#) Chapter 19

## Practice

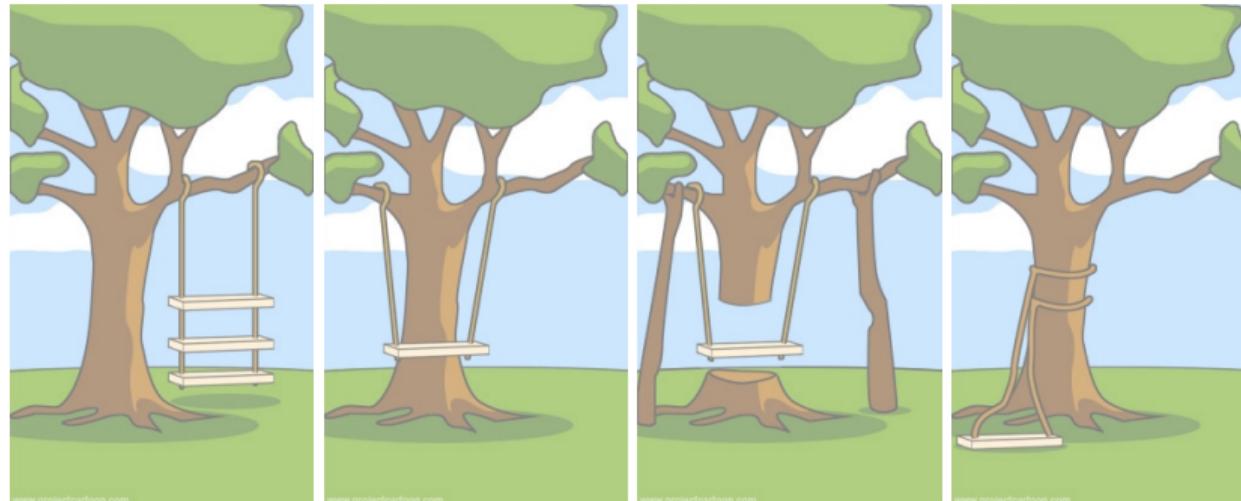
- See [Moodle](#)
- Sketch a sequence diagram with 2–4 roles for a messenger app.
- Upload your diagram to Moodle and correct other solutions if you find errors.



# Software Engineering

6. Implementation | Thomas Thüm | November 24, 2021

# Implementation in Software Projects



how the customer  
explained it

how the project  
leader understood it

how the analyst  
designed it

how the programmer  
implemented it

# Lecture Overview

1. Programming Languages
2. Coding Conventions
3. Tools and Environments

# Lecture Contents

## 1. Programming Languages

Questionnaire Results

Analysis and Design

History of Programming Languages

Programming Languages Today

Choice of Programming Languages

Popularity of Programming Languages

Excursion: Windows Calc

Lessons Learned

## 2. Coding Conventions

## 3. Tools and Environments

# Questionnaire Results

1

Warst du jemals betroffen von einer Softwarepanne?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	82%	59
Nein	18%	13

2

Hast du dich im letzten Semester über Softwarefehler geärgert?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	88%	61
Nein	12%	8

3

Hast du selbst Programmiererfahrung?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	93%	65
Nein	7%	5

4

Hast du bereits Software getestet?

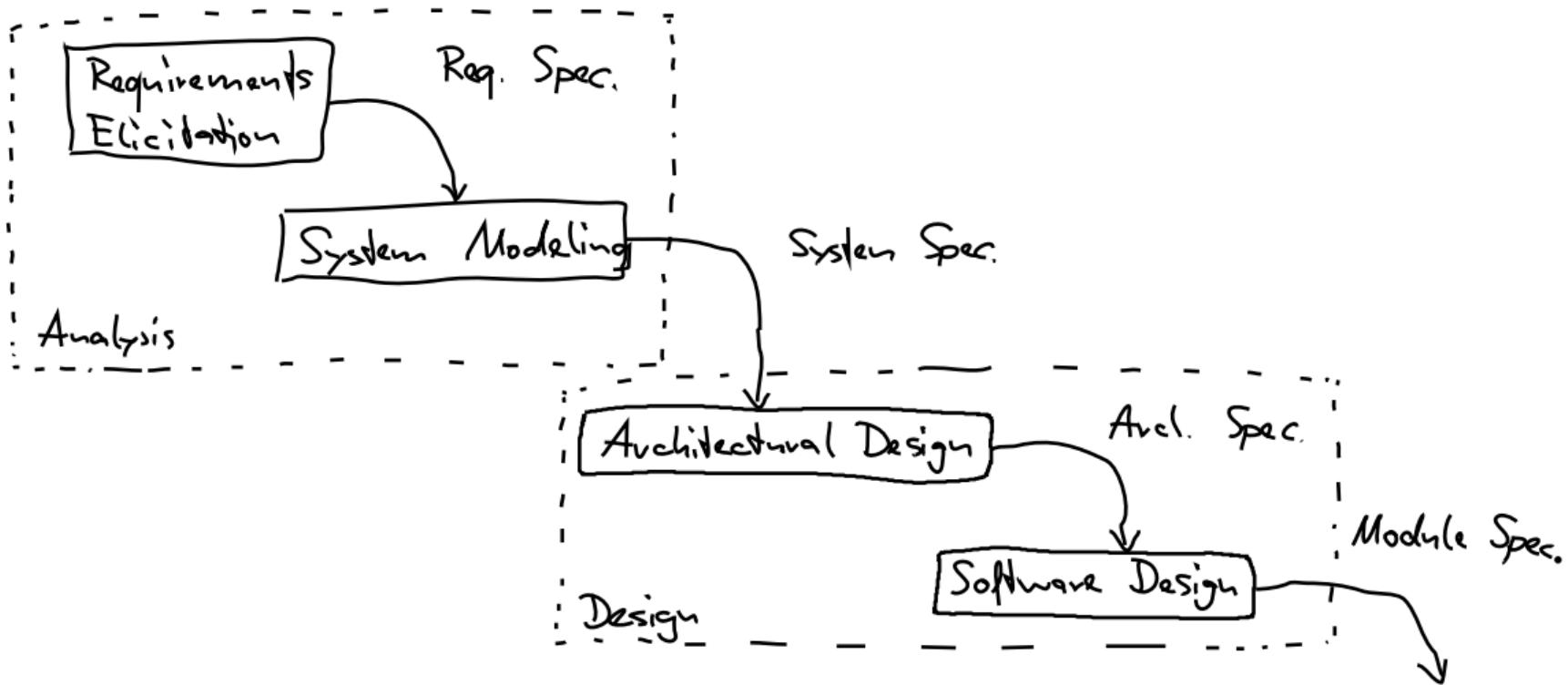
Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	53%	38
Nein	47%	34

5

Hast du schon mal Programmierfehler verursacht?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	87%	60
Nein	13%	9

# Analysis and Design



# History of Programming Languages

## Languages

[Jones + Krypczyk/Bochkor]

- 1945: first high-level language Plankalkül by Konrad Zuse (compiler written in 1998)
- 1954: first professional high-level language FORTRAN (Formula Translator) by IBM
- 1963: Basic as general-purpose language
- 1959: functional language Lisp
- 1970: first object-oriented lang. Smalltalk-80
- 1970: declarative language SQL
- 1971: Pascal by Niklaus Wirth for teaching
- 1974: very common procedural language C
- 1977: logical language Prolog
- 1980: C++ as object-oriented extension of C
- 1990: object-oriented language Java
- 1990: functional language Haskell
- 1991: multi-paradigm language Python
- 1995: scripting language JavaScript

## Milestones

[Jones]

- controlling behavior of mechanical devices by wiring or with punchcards ([Lochkarten](#))
- machine languages used during World War II
- assembly languages: distinction between human-readable instructions (source code) and executable instructions (object code)
- birth of compilers and interpreters having a one-to-many mapping between source and object code (opposed to one-to-one mapping in assemblers)
- structured programming pioneered by David Parnas and Edsger Dijkstra
- high-level programming languages: high number of executable for each human-readable instruction
- domain-specific languages, later general-purpose programming languages

# Programming Languages Today

## Today

[Jones + Krypczyk/Bochkor]

- 2002: C# by Microsoft
- 2009: Go by Google
- 2010: Rust by Mozilla Research
- 2014: Swift by Apple
- thousands of programming languages
- very few programming languages used for more than 10 years
- languages used for more than 25 years: Ada, C, C++, COBOL, Java, Objective C, PL/I, SQL, Visual Basic, ...

## Many Languages

[Jones]

- good: fit for every use case
- bad: developer training for new and dead languages, expensive tool support

# Choice of Programming Languages

## Desired Properties

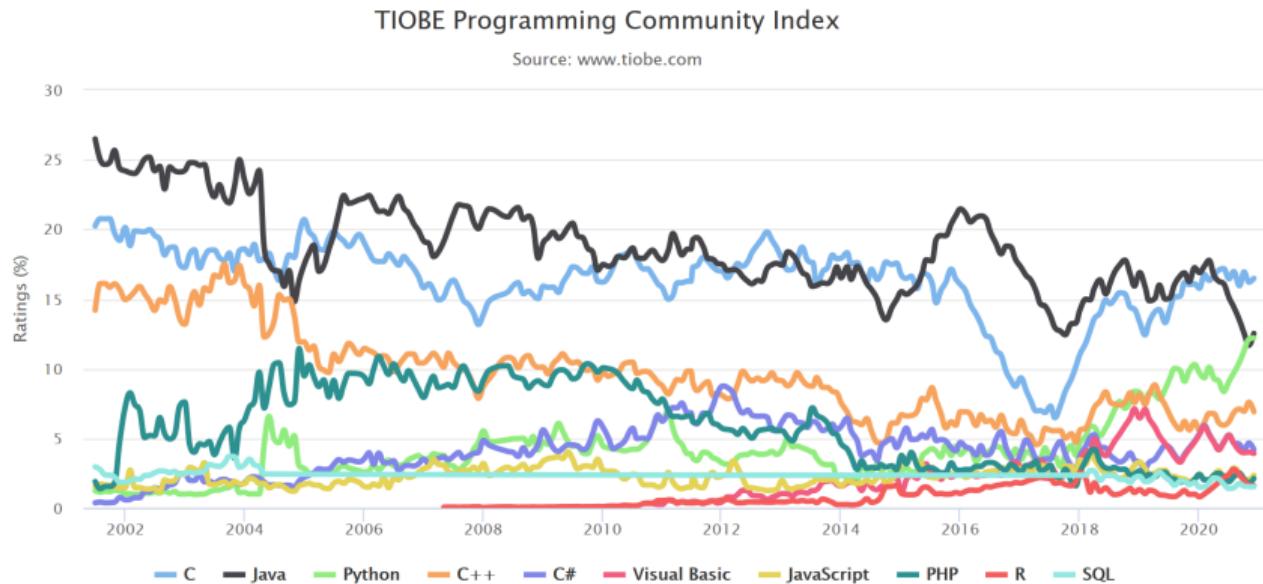
[Ludewig and Licher]

- modular implementation
- separation of interfaces and implementations
- type system: strongly/weakly typed languages
- readable syntax (FORTRAN vs ALGOL60)
- automatic pointer management (C vs Java)
- exception handling

## Criteria in Practice

- language required by the company or customer?
- existing infrastructure?
- domain-specific languages available?
- language known/liked by developers?
- available libraries?
- available tool support?
- language popularity?
- what may change in the future?

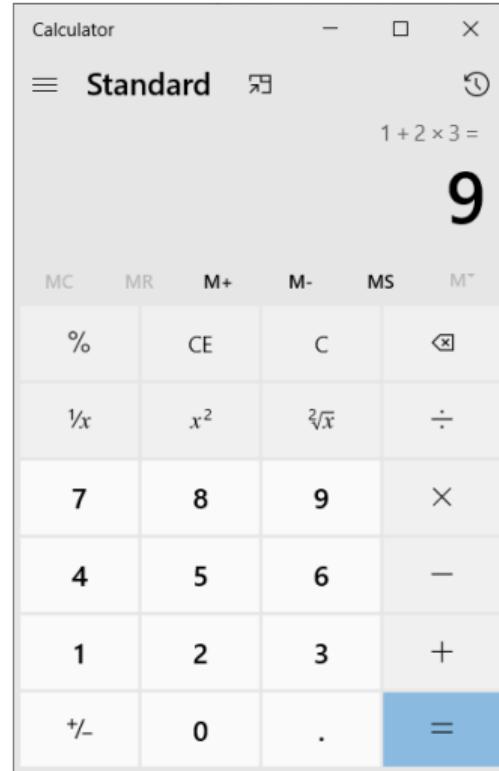
# Popularity of Programming Languages



# Popularity of Programming Languages

Programming Language	2020	2015	2010	2005	2000	1995	1990	1985
C	1	2	2	1	1	1	1	1
Java	2	1	1	2	3	29	-	-
Python	3	5	6	7	22	13	-	-
C++	4	3	3	3	2	2	2	8
C#	5	4	5	6	10	-	-	-
JavaScript	6	8	10	10	7	-	-	-
PHP	7	6	4	4	19	-	-	-
SQL	8	-	-	-	-	-	-	-
R	9	14	46	-	-	-	-	-
Swift	10	15	-	-	-	-	-	-
Lisp	29	26	14	13	9	6	4	2
Fortran	31	21	24	14	13	14	3	5
Ada	34	23	21	16	17	3	9	3

# Excursion: Windows Calc





**Patrick McKenzie**

[[twitter.com](#)]

“Every great developer you know got there by solving problems they were unqualified to solve until they actually did it.”



**Bill Gates**

[[code.org](#)]

“Learning to write programs stretches your mind, and helps you think better, creates a way of thinking about things that I think is helpful in all domains.”

# Programming Languages

## Lessons Learned

- Historical perspective on programming
- Criteria for choosing languages
- Popularity of programming languages
- Further Reading on Programming Languages: [Jones](#), Chapter 8 Programming and Code Development +  
[Krypczyk/Bochkor](#), Chapter 2.4 Programming Languages

## Practice

- See [Moodle](#)
- Look at the [code of my calculator](#) and think about possible improvements to the code quality
- Share your thoughts with your colleagues in Moodle (before watching Part 2)

# Lecture Contents

## 1. Programming Languages

## 2. Coding Conventions

Understanding the Corona-Warn-App

Code Formatting

Rules on Naming

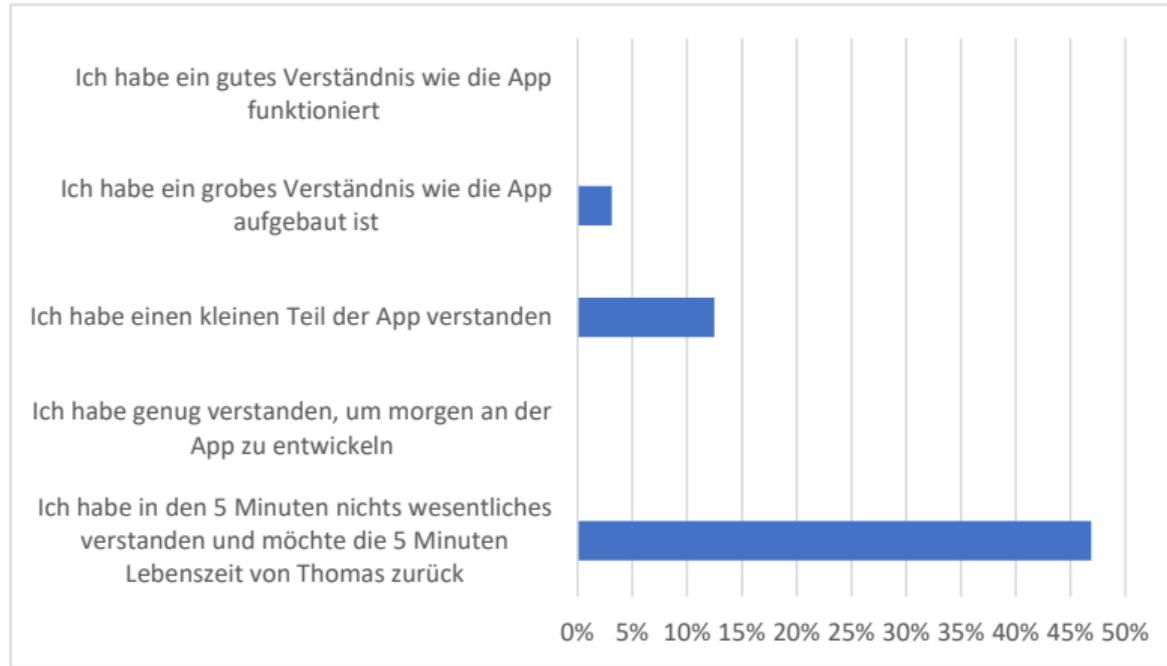
Code Documentation

Excursion: Revising Thomas' Calculator

Lessons Learned

## 3. Tools and Environments

# Understanding the Corona-Warn-App





**Douglas Crockford**

[Crockford 2008]

"It turns out that style matters in programming for the same reason that it matters in writing. It makes for better reading."



**François Chollet**

[twitter.com]

"In software, naming matters, because names reflect how you think about a problem. Code is also communication, and naming is a big part of making it work."

# Code Formatting

## Code Formatting

- motivation: code is read much more often and by more developers than written
- avoid differences by each programmer
- indentation: typically 4 characters per level
- length of a line: often 80 or 100 characters
- extra indentation: typically 8 characters when breaking extra long lines
- empty lines between methods and attributes
- automated code formatters available (on demand or when saving the editor)
- typical formatting rules for each language
- automated code formatters are configurable (handle with care)

# Rules on Naming

## Unwanted Names

- single character as a name
- very long names
- names consisting only of special chars
- synonyms: delete, remove, clear
- abbreviations (unless very common)

## Wanted Names

- nouns for class names: Calculator
- nouns for attribute names: calculateButton
- verbs for method names: getCalculator(), evaluate(), isZero(), hasChildren(), setValue()
- CamelCaseNotation for classes, attributes, methods, local variables, parameters
- UPPER\_CASE\_NOTATION for constants
- lowercasenotation for package names



**Martin Fowler (1999)**

[Fowler's Refactoring]

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”



**Cory House**

[twitter.com]

“Code is like humor. When you have to explain it, it’s bad.”

# Code Documentation

## Comments ...

- in source code are easier to maintain (than in external documents)
- should be written while editing the code
- can be used to generate documentation (e.g., JavaDoc, Doxygen)
- are used to specify classes and public methods (e.g., parameters, exceptions, dependencies)
- document hacks, side effects and unfinished parts (e.g., TODO)
- should not paraphrase the code



Ryan Campbell

[problemsolving.io]

“Commenting your code is like cleaning your bathroom - you never want to do it, but it really does create a more pleasant experience for you and your guests.”

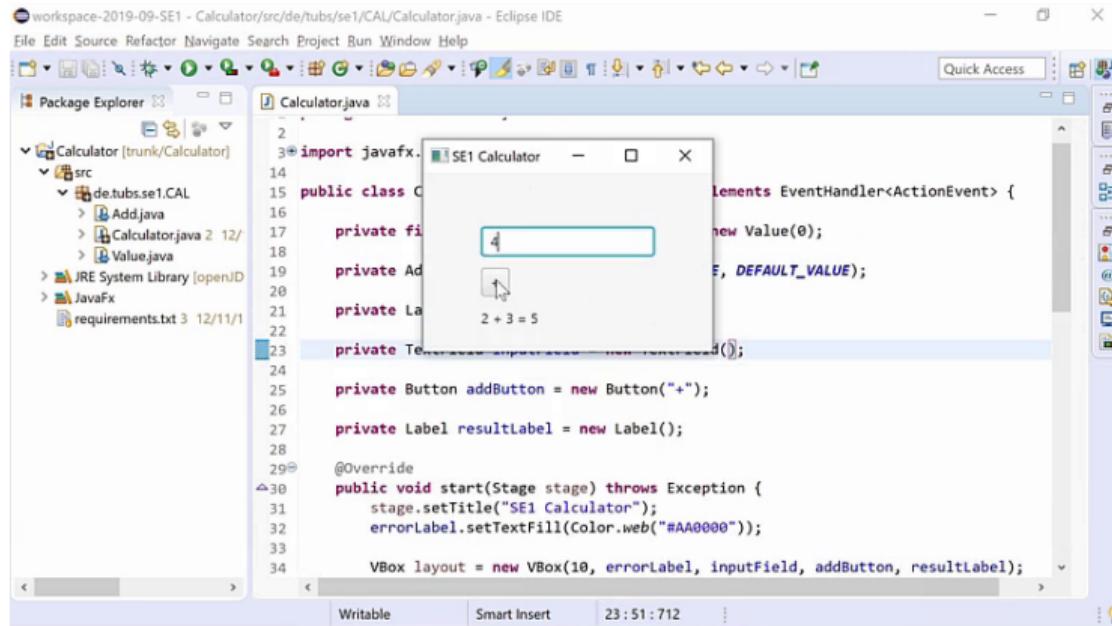


Steve McConnell (2004)

[Code Complete]

“Good code is its own best documentation. As you’re about to add a comment, ask yourself, “How can I improve the code so that this comment isn’t needed?” Improve the code and then document it to make it even clearer.”

# Excursion: Revising Thomas' Calculator



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

- Title Bar:** workspace-2019-09-SE1 - Calculator/src/de/tubs/se1/CAL/Calculator.java - Eclipse IDE
- Menu Bar:** File Edit Source Refactor Navigate Search Project Run Window Help
- Toolbar:** Standard Eclipse toolbar icons.
- Package Explorer:** Shows the project structure:
  - Calculator [trunk/Calculator]
  - src
    - de.tubs.se1.CAL
      - Add.java
      - Calculator.java 2 12/
      - Value.java
  - JRE System Library [openJD]
  - JavaFx
  - requirements.txt 3 12/11/
- Code Editor:** The file `Calculator.java` is open, showing Java code for a calculator application. The code includes imports for `javafx`, a class definition with fields for an input field and result label, and a start method that creates a stage and sets up a UI layout using a VBox.
- JavaFX Preview:** A separate window titled "SE1 Calculator" displays the running application. It shows a text input field containing "4" and a result label below it showing "2 + 3 = 5".
- Status Bar:** Writable, Smart Insert, 23:51:712, ...

# Coding Conventions

## Lessons Learned

- Coding conventions ([Programmierrichtlinien](#))
- Formatting, naming, comments, and documentation
- Further Reading: [Google's Java Style Guide](#)

## Practice

- See [Moodle](#)
- Optional: inspect [changes of the live coding](#)
- The [resulting code](#) does not contain any comments. Give an example comment.
- Argue for one other comment whether it is useful or not?

# Lecture Contents

1. Programming Languages
2. Coding Conventions
3. Tools and Environments
  - Computer-Aided Software Engineering
  - Overview on Development Tools
  - Excursion: Extending Thomas' Calculator
  - Lessons Learned

# Computer-Aided Software Engineering

## Terms

[adapted from Ghezzi/Jazayeri/Mandrioli]

A **tool** is an application that supports a particular activity. An **environment** is a collection of related tools. Tools and environments aim at automating some of the activities that are involved in software engineering. The generic term for this field of study is **computer-aided software engineering**.

nano? REAL  
PROGRAMMERS  
USE emacs



HEY. REAL  
PROGRAMMERS  
USE vim.



WELL, REAL  
PROGRAMMERS  
USE ed.



NO, REAL  
PROGRAMMERS  
USE cat.



REAL PROGRAMMERS  
USE A MAGNETIZED  
NEEDLE AND A  
STEADY HAND.



EXCUSE ME, BUT  
REAL PROGRAMMERS  
USE BUTTERFLIES.



THEY OPEN THEIR  
HANDS AND LET THE  
DELICATE WINGS FLAP ONCE.

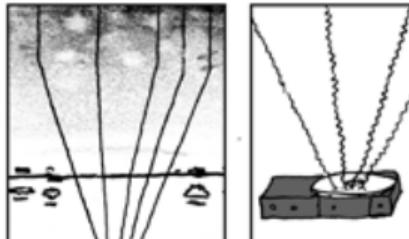


THE DISTURBANCE RIPPLES  
OUTWARD, CHANGING THE FLOW  
OF THE EDDY CURRENTS  
IN THE UPPER ATMOSPHERE.



THESE CAUSE MOMENTARY POCKETS  
OF HIGHER-PRESSURE AIR TO FORM,

WHICH ACT AS LENSES THAT  
DEFLECT INCOMING COSMIC  
RAYS, FOCUSING THEM TO  
STRIKE THE DRIVE PLATTER  
AND FLIP THE DESIRED BIT.



NICE.  
'COURSE, THERE'S AN EMACS  
COMMAND TO DO THAT.  
OH YEAH! GOOD OL'  
C-x M-c M-butterfly...



DAMMIT, EMACS.

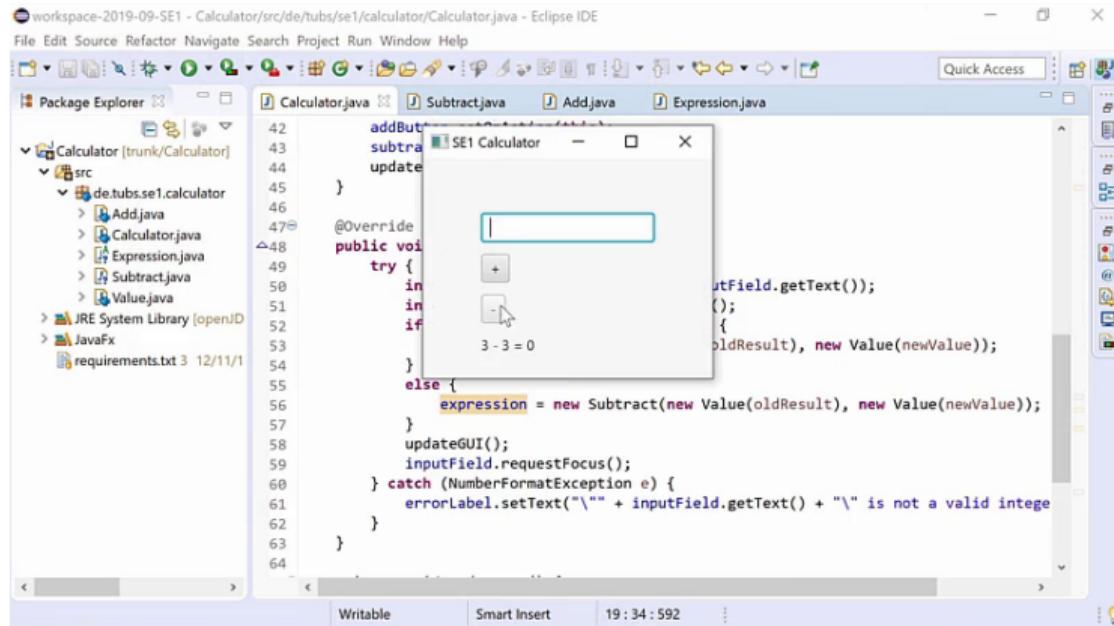
# Overview on Development Tools

## Variety of Tools

[Ghezzi/Jazayeri/Mandrioli]

- text(ual) editors: emacs, vim, ed, Word, ...
- graphical editors: UML editors, Powerpoint, ...
- assembler, compiler, interpreter
- configuration management tools: git, SVN, CVS, ...
- tracking tools (issue trackers): Github, Gitlab, ...
- tools for code navigation and refactoring
- tools for test specification, generation, execution, reporting
- tools for static and dynamic code analysis (e.g., debugger), reverse/reengineering, project management
- integrated development environments (IDEs): Eclipse, IntelliJ, Android Studio, Visual Studio

# Excursion: Extending Thomas' Calculator



# Tools and Environments

## Lessons Learned

- Tool supports by means of tools, environments, and IDEs
- Further Reading: [Ghezzi/Jazayeri/Mandrioli](#), Chapter 9 Software Engineering Tools and Environments

## Practice

- See [Moodle](#)
- Choose a tool or an IDE.
- Tryout tooling that you have learned about in this video but never used before (e.g., automated code formatting or code navigation).
- Post a screenshot and brief description in Moodle.



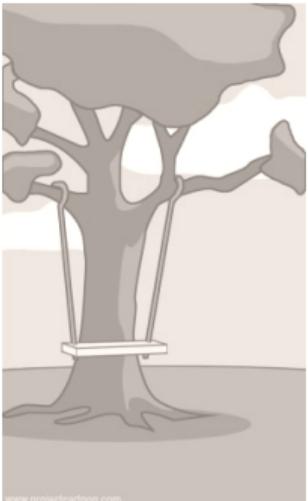
# Software Engineering

7. Design Patterns | Thomas Thüm | December 13, 2021

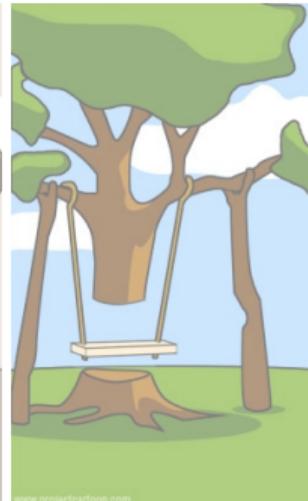
# Design Patterns (Entwurfsmuster)



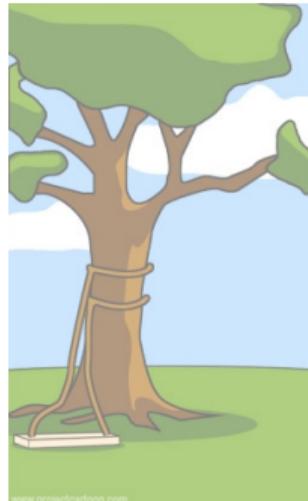
how the customer  
explained it



how the project  
leader understood it

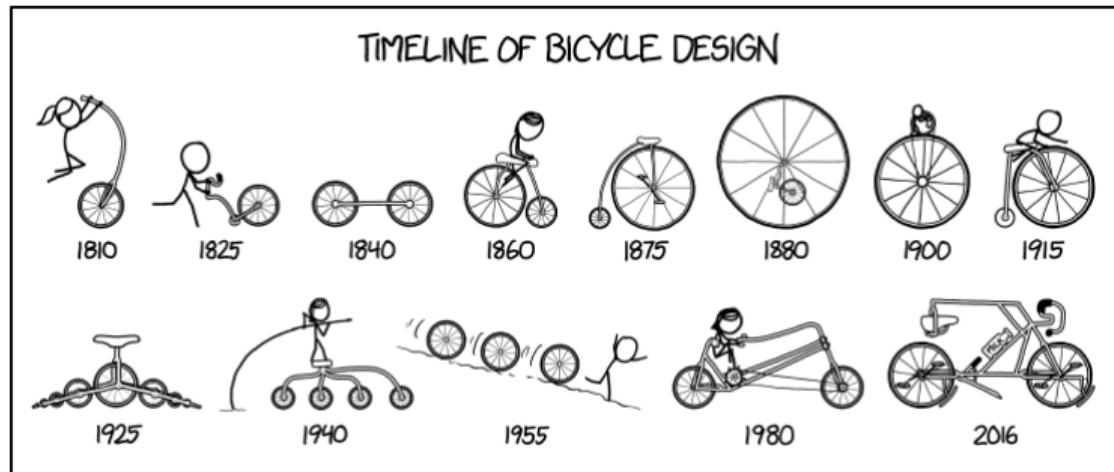


how the analyst  
designed it



how the programmer  
implemented it

# Tired of Swings? What about Bikes?



# Lecture Overview

1. Structural Patterns
2. Creational Patterns
3. Behavioral Patterns

# Lecture Contents

## 1. Structural Patterns

Gang of Four

Design Patterns

Overview on Design Patterns

Object Adapter Pattern

Composite Pattern

Excursion: Show Complete Formula

Decorator Pattern

Example of the Decorator Pattern

Excursion: Introduce Necessary Brackets

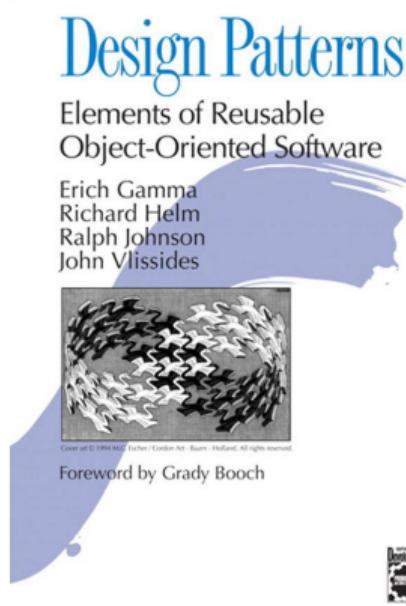
Overview on Design Patterns

Lessons Learned

## 2. Creational Patterns

## 3. Behavioral Patterns

# Gang of Four [Gang of Four (GoF)]



# Design Patterns

[Gang of Four]

## Motivation

"Designing object-oriented software is hard, and designing **reusable** object-oriented software is even harder. [...] It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it?"

## Design Patterns (Entwurfsmuster)

pattern name for communication and high-level abstraction

problem when to apply the pattern

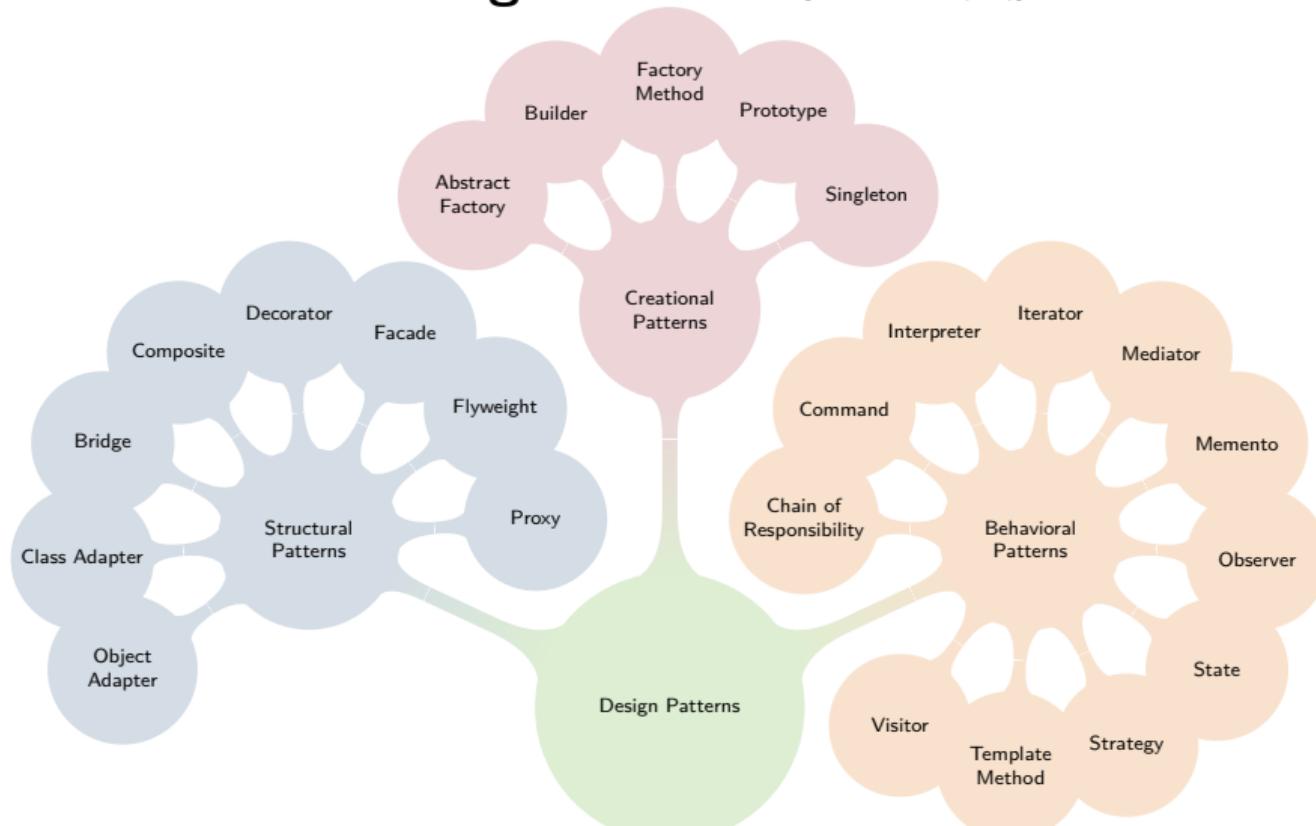
solution template on how to arrange classes and objects

consequences trade-offs of applying the pattern

## Kinds of Patterns

"Creational patterns (**Erzeugungsmuster**) concern the process of object creation. Structural patterns (**Strukturmuster**) deal with the composition of classes or objects. Behavioral patterns (**Verhaltensmuster**) characterize the ways in which classes or objects interact and distribute responsibility."

# Overview on Design Patterns [Gang of Four (GoF)]



# Object Adapter Pattern

## Object Adapter

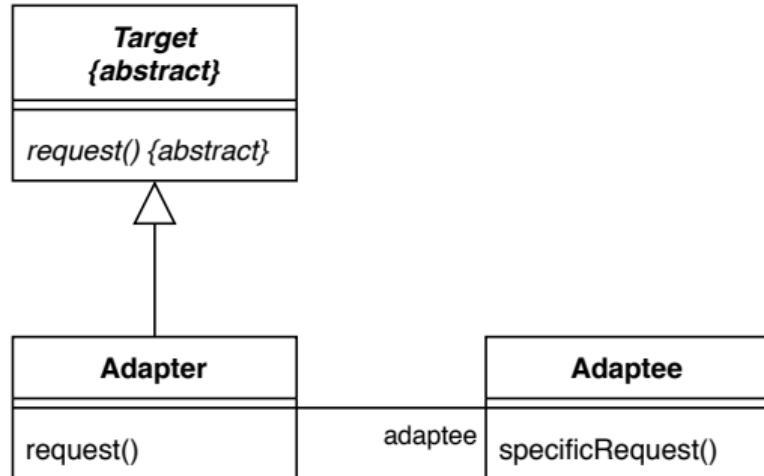
[Gang of Four]

intent “Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.”

aka. wrapper

motivation enable reuse of classes even though incompatible interfaces cannot be made compatible

idea create a new class with a compatible interface that forwards all requests



# Composite Pattern

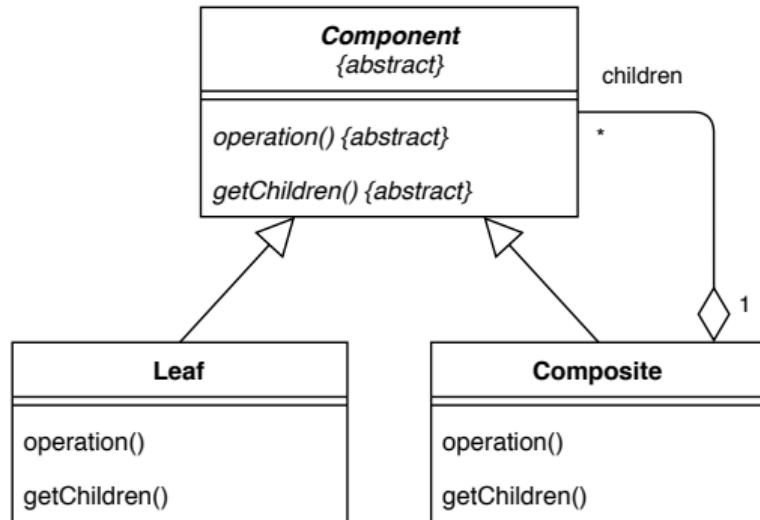
## Composite

[Gang of Four]

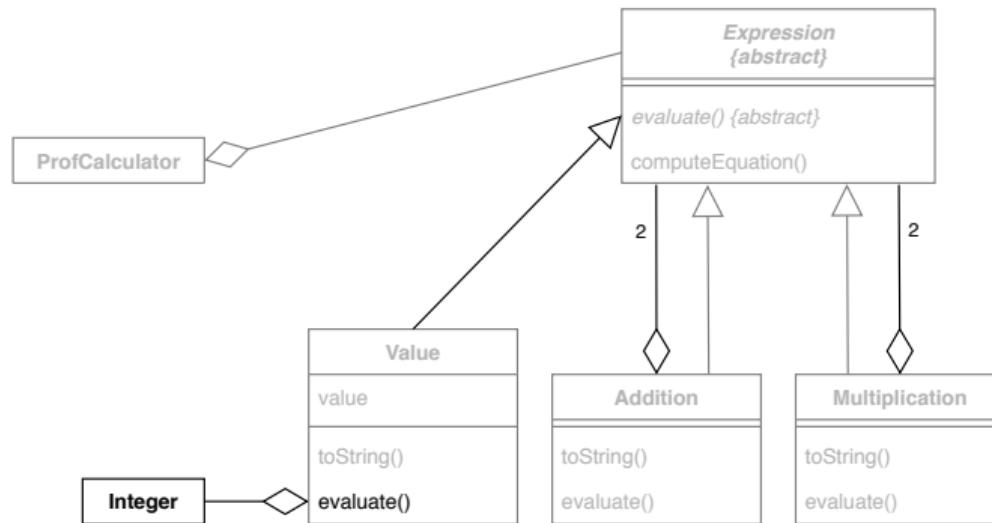
intent “Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.”

motivation avoid case distinctions for primitive and compound objects

idea create a common abstract super class enabling a unified access



# Excursion: Show Complete Formula



# Decorator Pattern

## Decorator

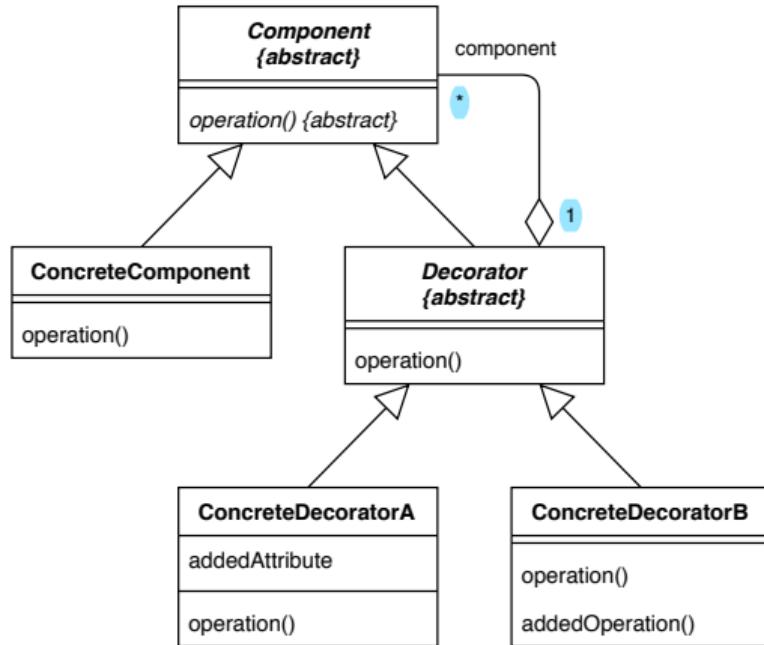
[Gang of Four]

intent “Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.”

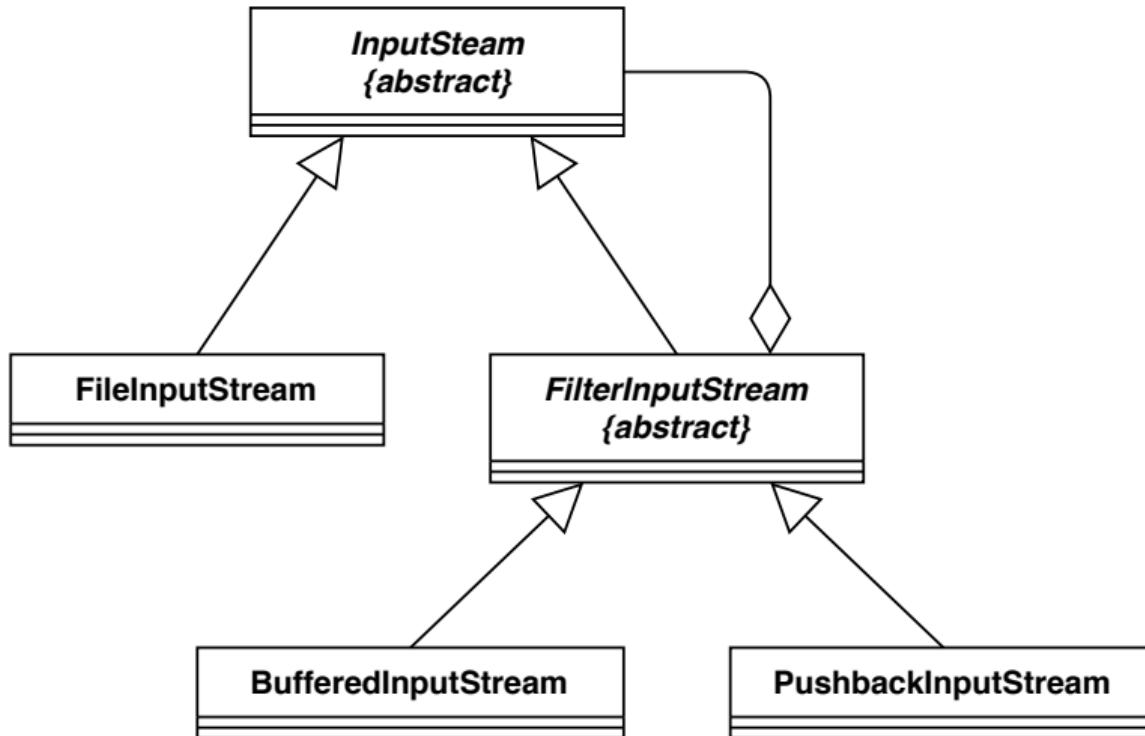
aka. wrapper (again)

motivation avoid explosion of static classes when combining all additional behaviors with all applicable classes

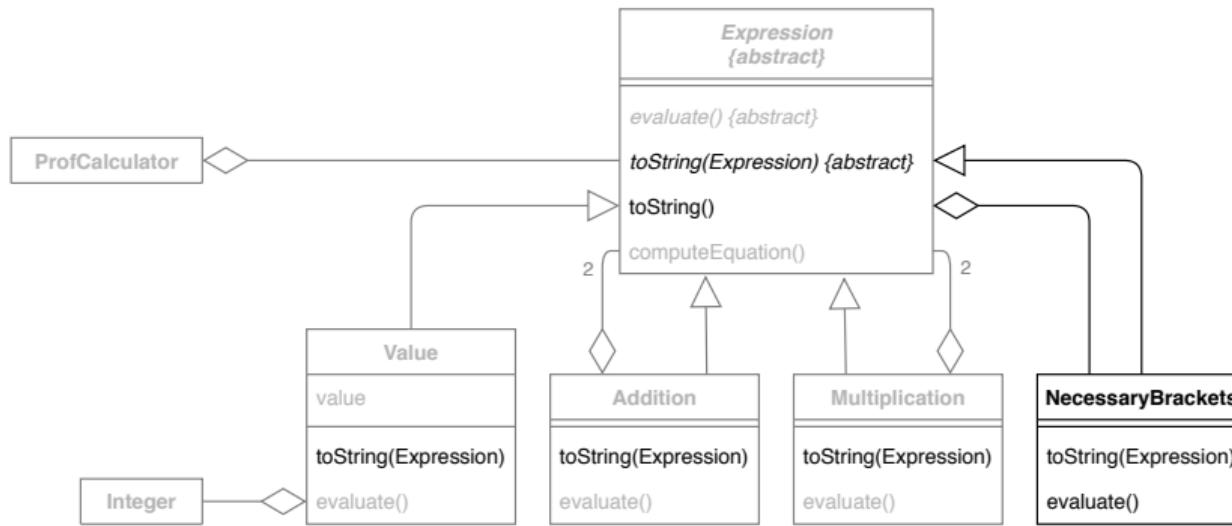
idea create decorators and components with the same interface, whereas decorators forward behavior whenever feasible



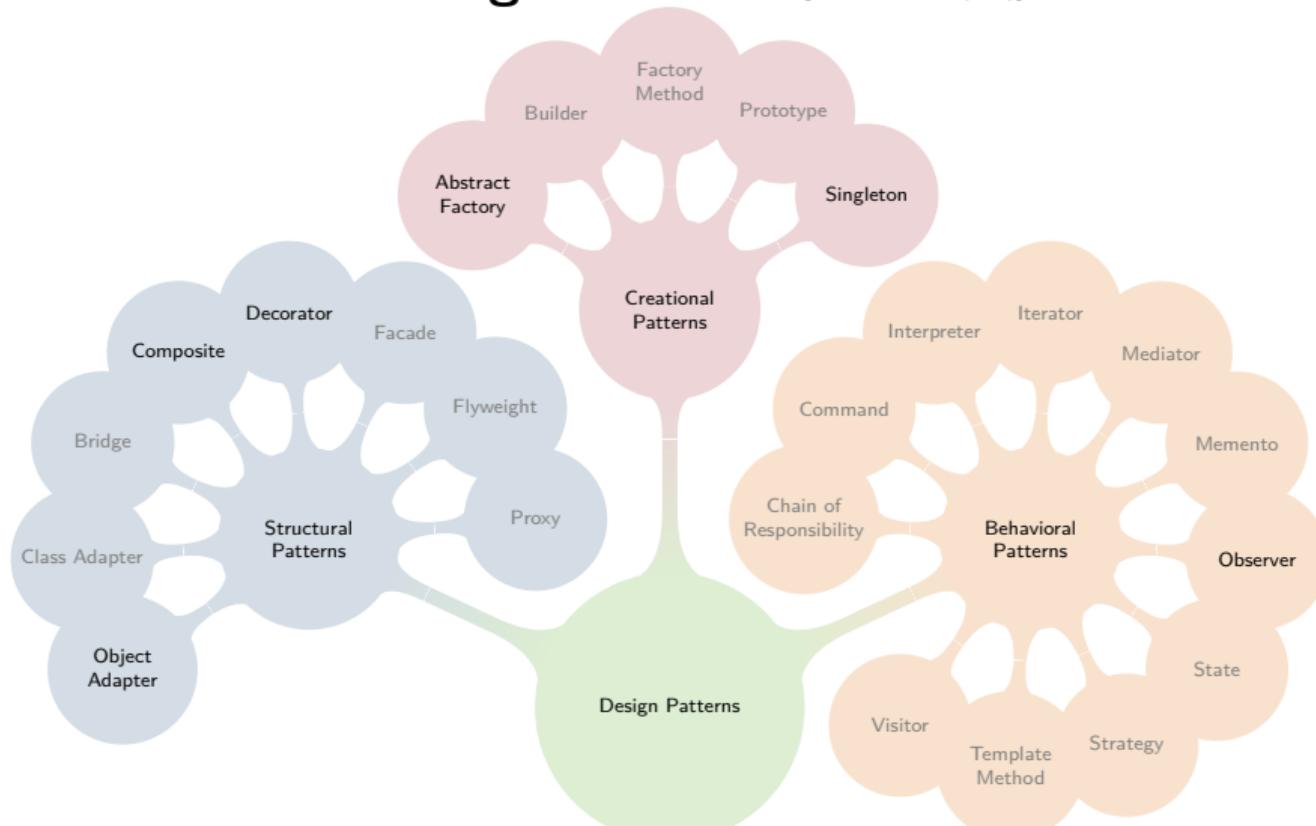
# Example of the Decorator Pattern



# Excursion: Introduce Necessary Brackets



# Overview on Design Patterns [Gang of Four (GoF)]



# Structural Patterns

## Lessons Learned

- Design patterns by the Gang of Four
- Object adapter, composite pattern, decorator pattern
- Further Reading: [Gang of Four \(GoF\)](#), Chapter 4

## Practice

- See [Moodle](#)
- Inspect code on Github: <https://github.com/tthuem/2020WS-SWT-Calculator/tree/xmaslecturev1>
- Implement further operations or the ability to show formulas without brackets and by using evaluation if needed (and checkbox to switch between brackets and evaluation)
- Fork the project on Github, upload your changes, and send pull request to get feedback

# Lecture Contents

## 1. Structural Patterns

## 2. Creational Patterns

Singleton Pattern

Excursion: Add Logging for Debugging

Abstract Factory Pattern

Excursion: Create Styles for GUI

Overview on Design Patterns

Lessons Learned

## 3. Behavioral Patterns

# Singleton Pattern

## Singleton

[Gang of Four]

intent "Ensure a class [has only] one instance, and provide a global point of access to it."

motivation avoid the uncontrolled creation of multiple instances

idea a private constructor is called on class initialization and a public static method is used for access to that single instance

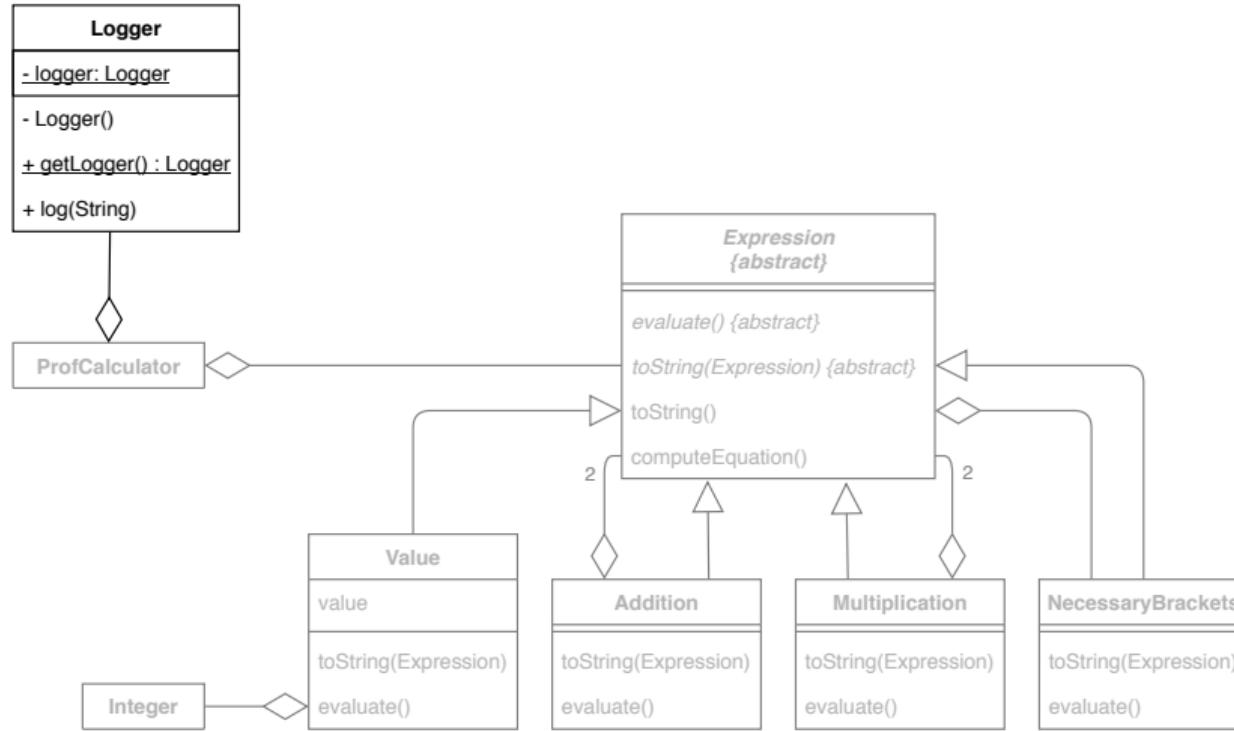
## Singleton

- instance: Singleton

- Singleton()

+ getInstance(): Singleton

# Excursion: Add Logging for Debugging



# Abstract Factory Pattern

## Abstract Factory

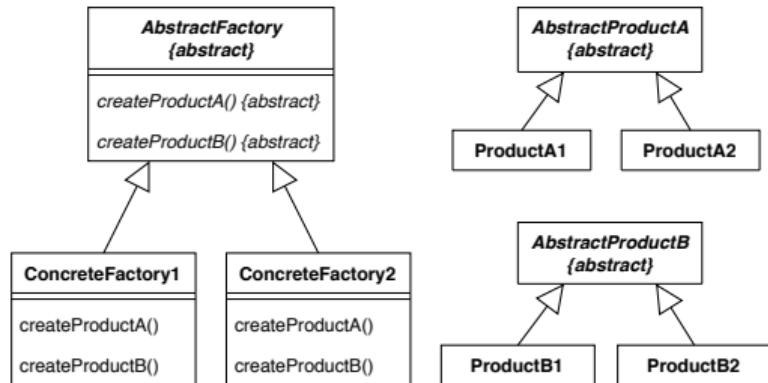
[Gang of Four]

intent “Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”

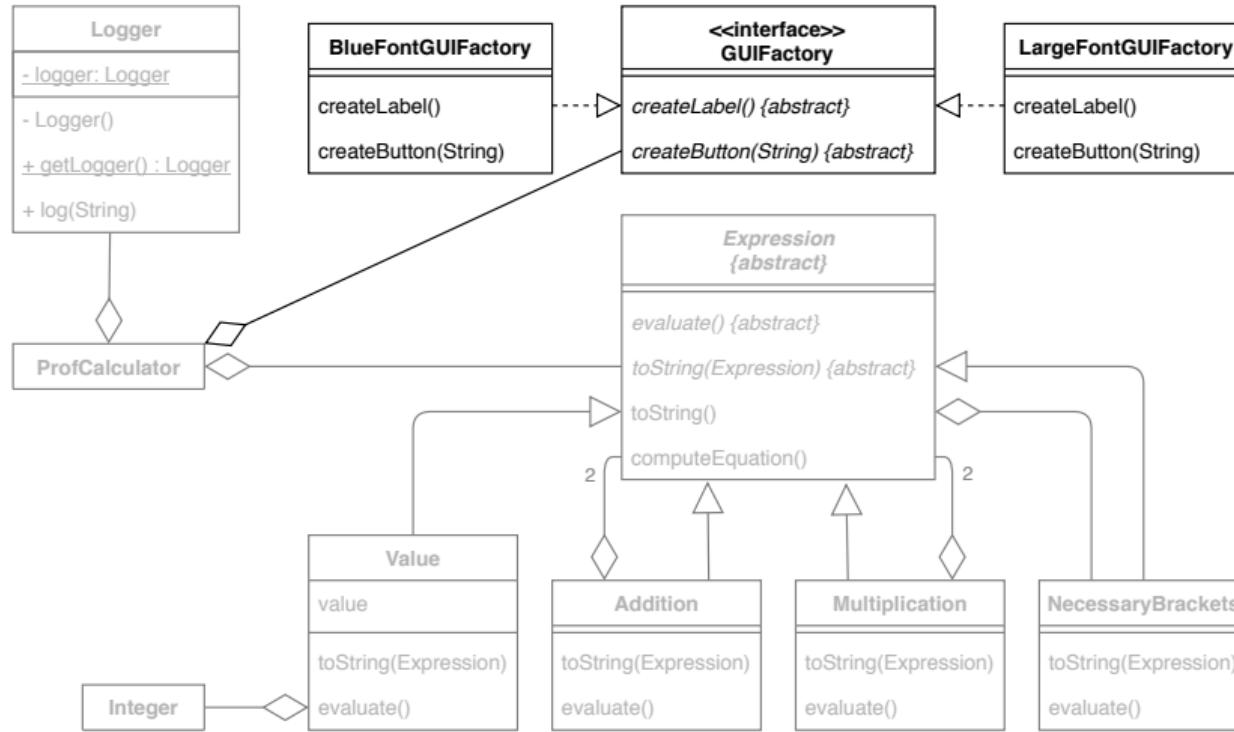
aka. kit

motivation avoid case distinctions when creating objects of certain kind, consistently create objects of a particular kind

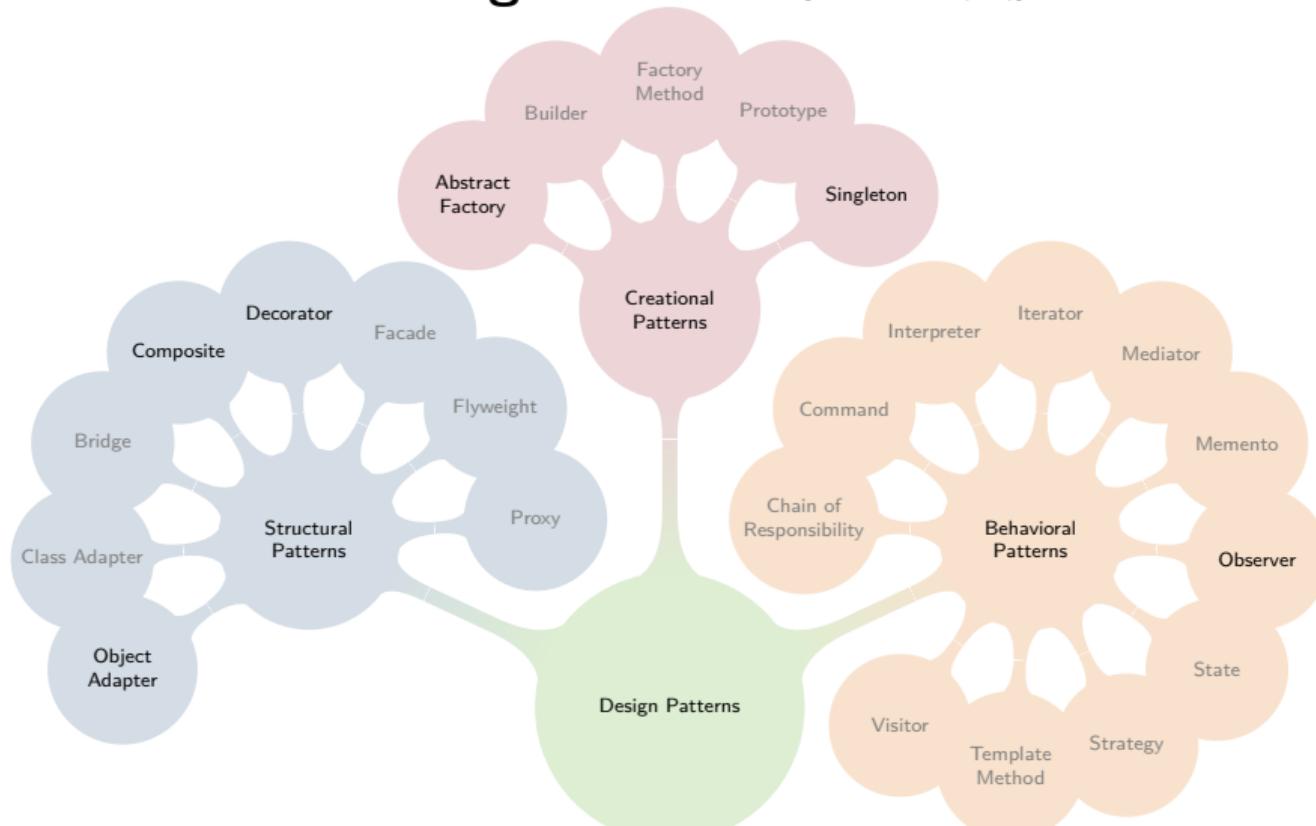
idea create classes for the consistent creation of objects



# Excursion: Create Styles for GUI



# Overview on Design Patterns [Gang of Four (GoF)]



# Creational Patterns

## Lessons Learned

- Singleton pattern, abstract factory pattern
- Further Reading: [Gang of Four \(GoF\)](#), Chapter 3

## Practice

- See [Moodle](#)
- Inspect code on Github: <https://github.com/tthuem/2020WS-SWT-Calculator/tree/xmaslecturev2>
- Implement your own GUI factory **or** add the textbox to all factories
- Fork the project on Github, upload your changes, and send pull request to get feedback

# Lecture Contents

1. Structural Patterns
2. Creational Patterns
3. Behavioral Patterns
  - Observer Pattern
  - Excursion: Counter from 1 to 10
  - Overview on Design Patterns
  - Lessons Learned

# Observer Pattern

## Observer

[Gang of Four]

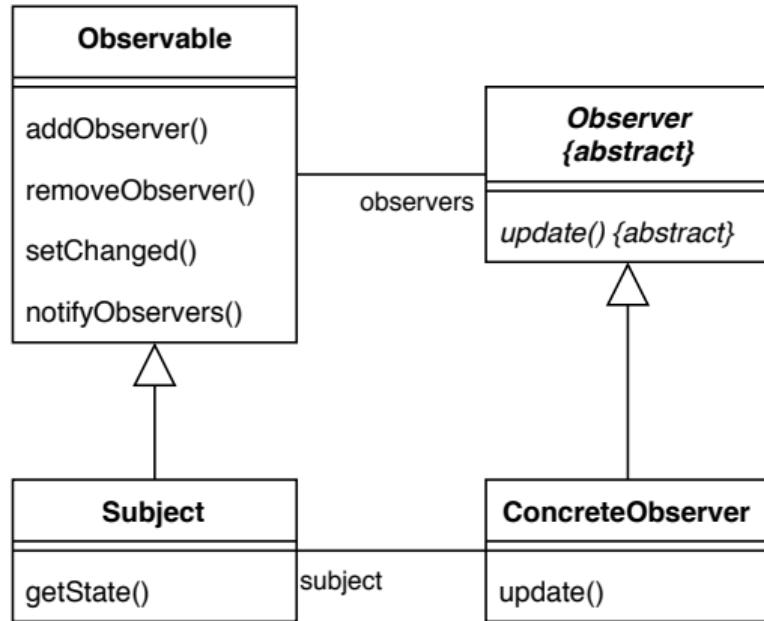
intent “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

aka. dependents, publish-subscribe

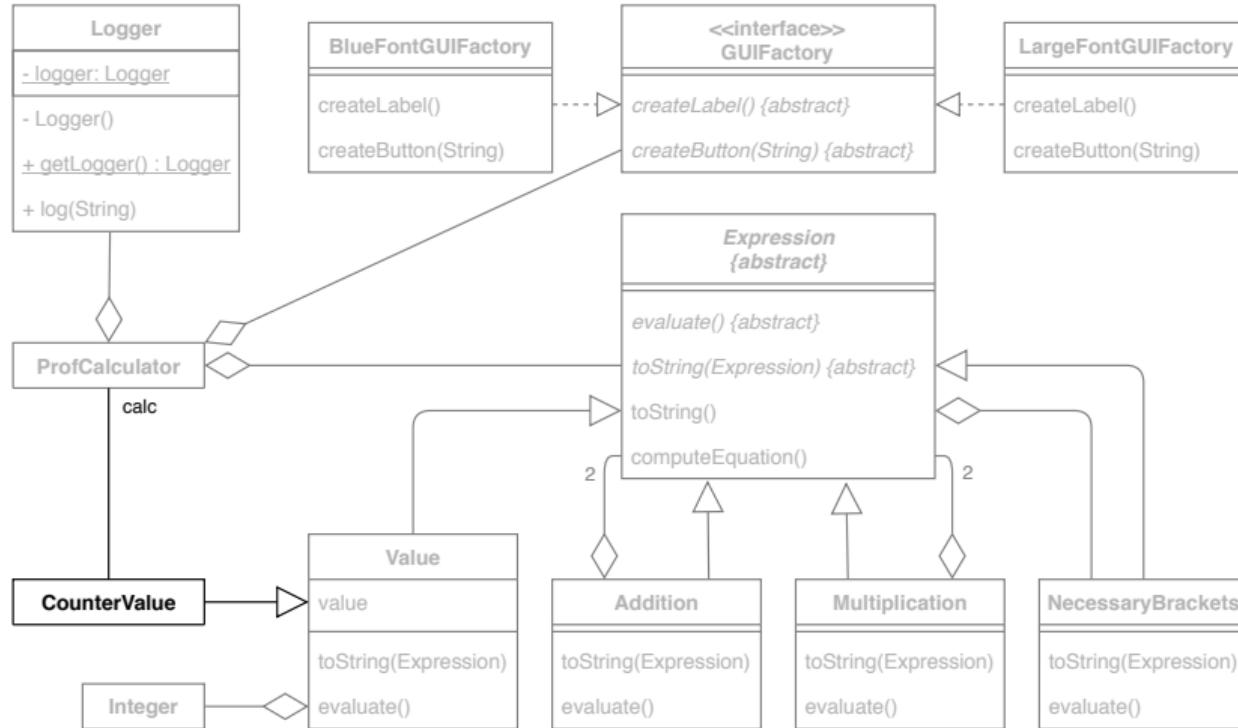
motivation avoid coupling of classes (i.e., explicit references such as imports)

example no coupling between model and view in model-view-controller architectures

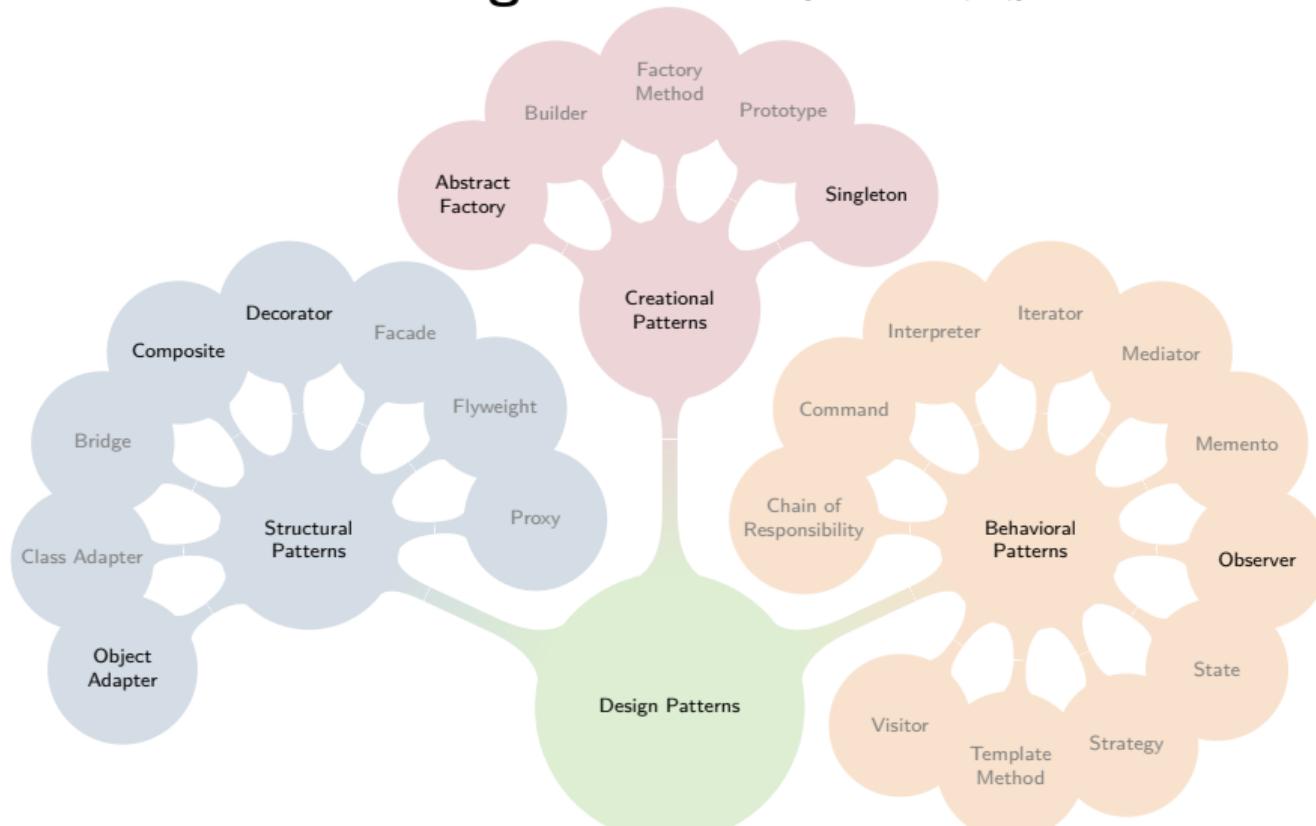
idea data object (model) has a list of observers (views) that are notified about changes



# Excursion: Counter from 1 to 10



# Overview on Design Patterns [Gang of Four (GoF)]



# Behavioral Patterns

## Lessons Learned

- Observer pattern
- Further Reading: Gang of Four (GoF), Chapter 5

## Practice

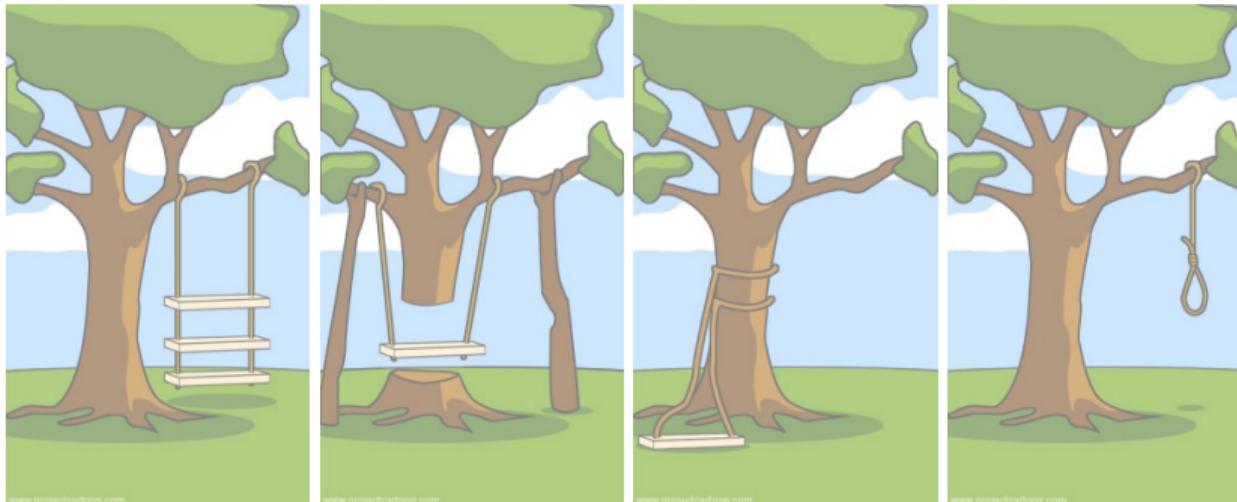
- See Moodle
- Inspect code on Github: <https://github.com/tthuem/2020WS-SWT-Calculator/tree/xmaslecturev3>
- Implement the observer pattern (to avoid explicit reference of the calculator in CounterValue) or one other behavioral pattern not discussed in the lecture
- Fork the project on Github, upload your changes, and send pull request to get feedback



# Software Engineering

8. Software Testing | Thomas Thüm | January 12, 2022

# Software Testing



how the customer  
explained it

how the analyst  
designed it

how the programmer what the beta testers  
implemented it received

# Lecture Overview

1. Quality Assurance
2. White-Box Testing
3. Black-Box Testing

# Lecture Contents

## 1. Quality Assurance

Software Quality

Product Quality

Quality in Use

Software Testing

Quality Assurance

Code Reviews

Code Reviews on Github

Modulo in Different Programming Languages

Lessons Learned

## 2. White-Box Testing

## 3. Black-Box Testing



**Andy Hunt**

[The Pragmatic Programmer]

"No one in the brief history of computing has ever written a piece of perfect software. It's unlikely that you'll be the first."



**Donald Trump (May 2020)**

[huffpost.com]

"If we didn't do any testing, we would have very few cases."

# Software Quality

## Quality

[Ludewig and Licher]

Quality is the entirety of properties and characteristics of a product or process that indicate adequacy with respect to given requirements.

## Quality Assurance

[Ludewig and Licher]

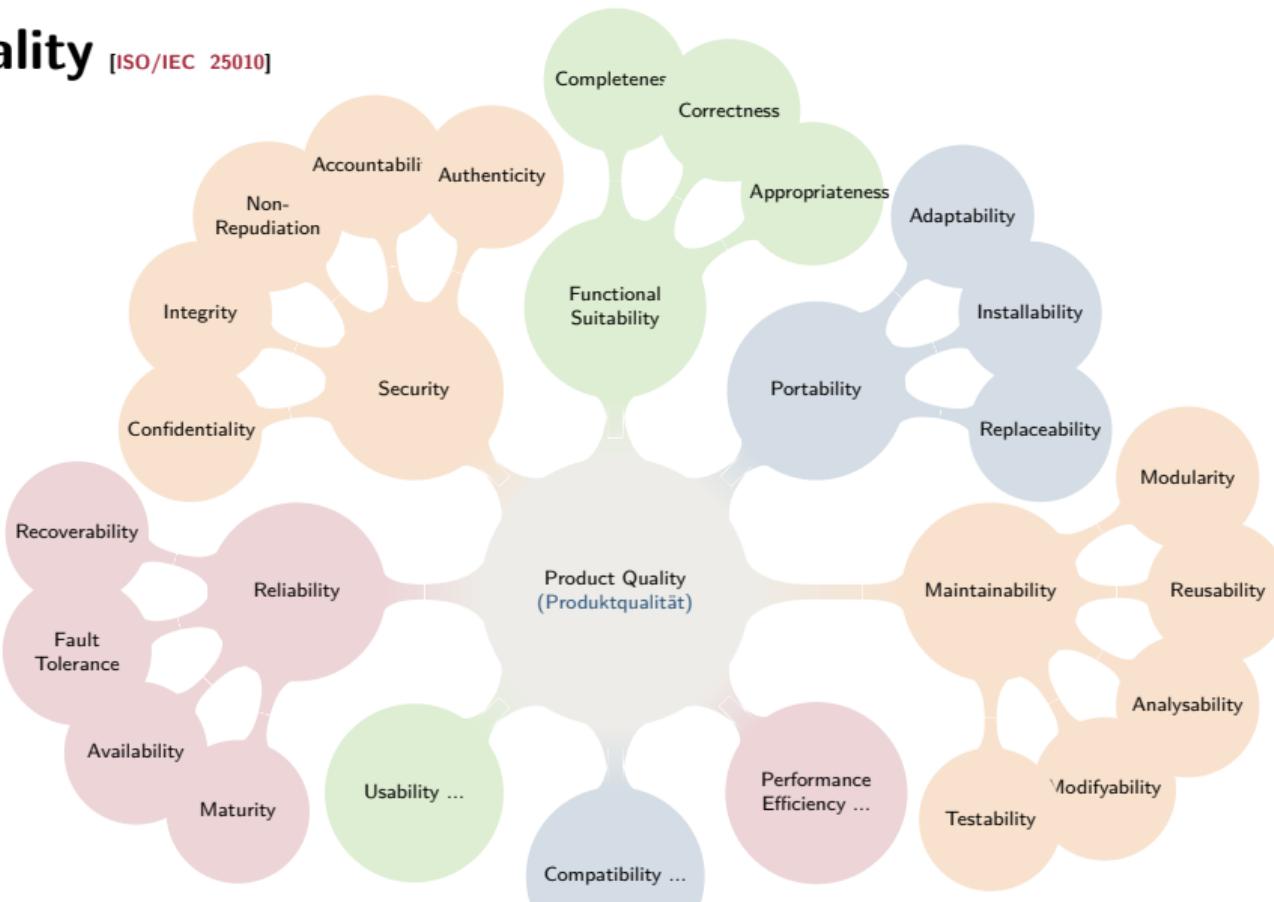
Quality assurance ([Qualitätssicherung](#)) are all activities with the goal to improve the quality.

## Expectations on Quality

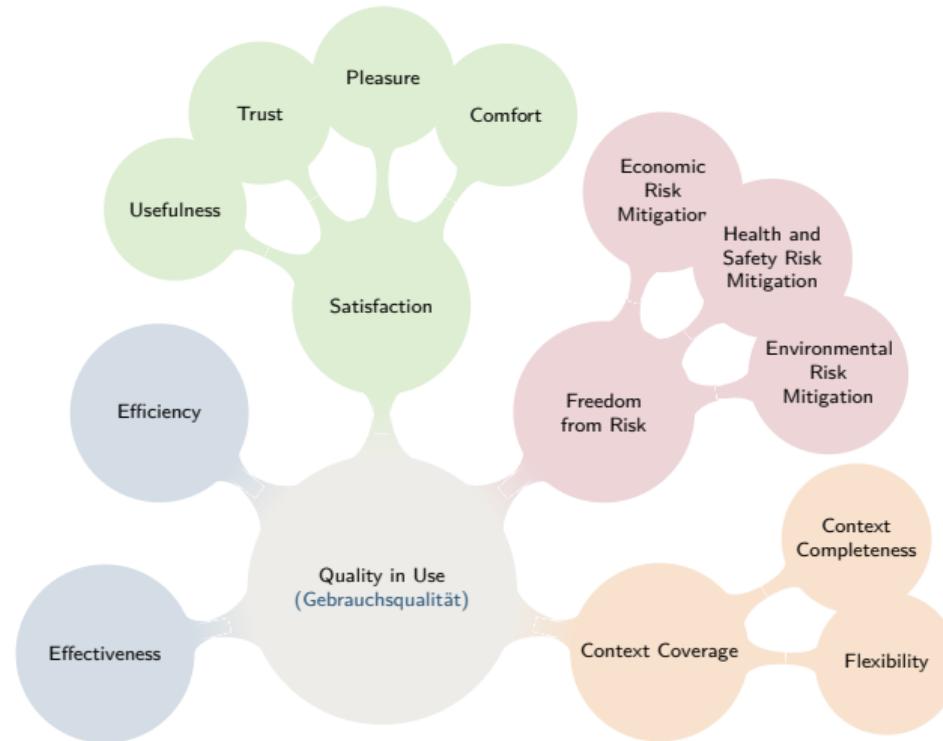
[Sommerville]

“Because of their previous experiences with buggy, unreliable software, users sometimes have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery. However, as a software product becomes more established, users expect it to become more reliable. [...] If a software product or app is very cheap, users may be willing to tolerate a lower level of reliability. [...] Customers may be willing to accept the software, irrespective of problems, because the costs of not using the software are greater than the costs of working around the problems.”

# Product Quality [ISO/IEC 25010]



# Quality in Use [ISO/IEC 25010]



## ⚠ ERROR

IF YOU'RE SEEING THIS, THE CODE IS IN WHAT  
I THOUGHT WAS AN UNREACHABLE STATE.

I COULD GIVE YOU ADVICE FOR WHAT TO DO.  
BUT HONESTLY, WHY SHOULD YOU TRUST ME?  
I CLEARLY SCREWED THIS UP. I'M WRITING A  
MESSAGE THAT SHOULD NEVER APPEAR, YET  
I KNOW IT WILL PROBABLY APPEAR SOMEDAY.

ON A DEEP LEVEL, I KNOW I'M NOT  
UP TO THIS TASK. I'M SO SORRY.



NEVER WRITE ERROR MESSAGES TIRED.

# Software Testing

V&V

[SE Economics]

“Validation: Are we building the right product?  
Verification: Are we building the product right?”

## Software Testing

[Sommerville]

“Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.”

## Validation Testing

[Sommerville]

“Demonstrate to the developer and the customer that the software meets its requirements.”

## Defect Testing

[Sommerville]

“Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification.”

## Stages of Testing

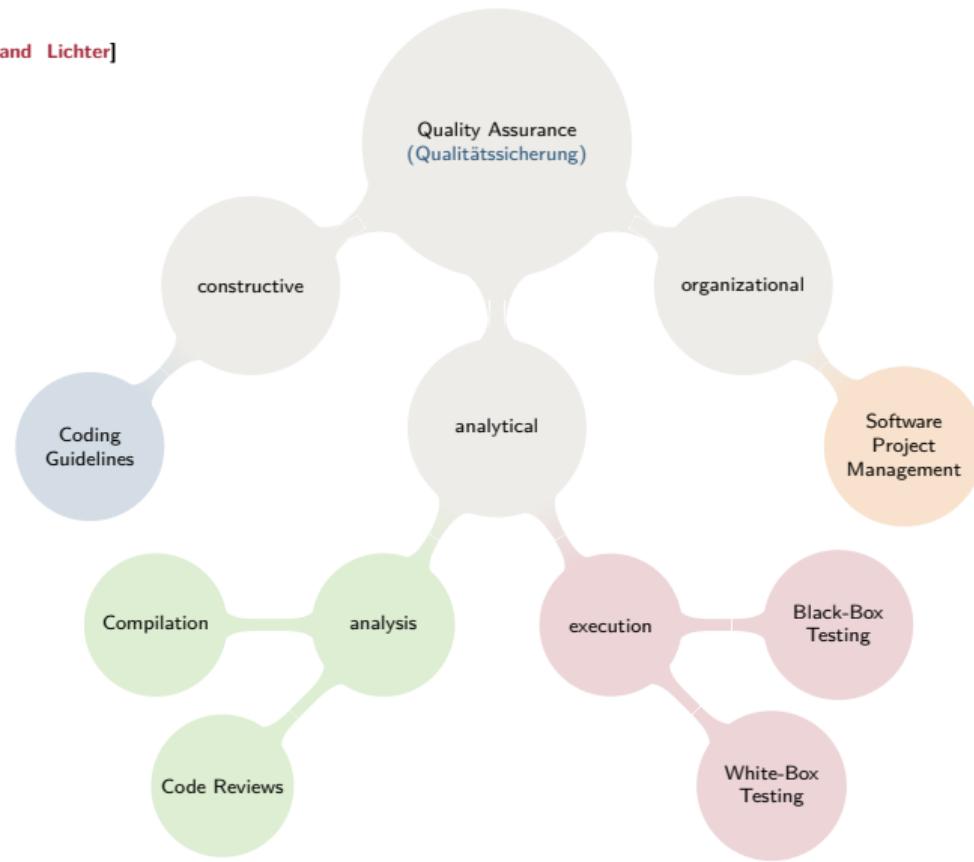
[Sommerville]

1. “**Development testing**, where the system is tested during development to discover bugs and defects”
2. “**Release testing**, where a separate testing team tests a complete version of the system before it is released to users”
3. “**User testing**, where users or potential users of a system test the system in their own environment”

“In **manual testing**, a tester runs the program with some test data and compares the results to their expectations. [...] In **automated testing**, the tests are encoded in a program that is run each time the system under development is to be tested.”

[Sommerville]

# Quality Assurance [Ludewig and Licher]



# Code Reviews

- Idea: improve quality by asking other programmers for feedback
- Typically applied with quality checklist
- Quality criteria: functionality, comprehensibility, maintainability, coding guidelines, design patterns, ...
- Reviewer selection: based on familiarity with code, availability, expertise

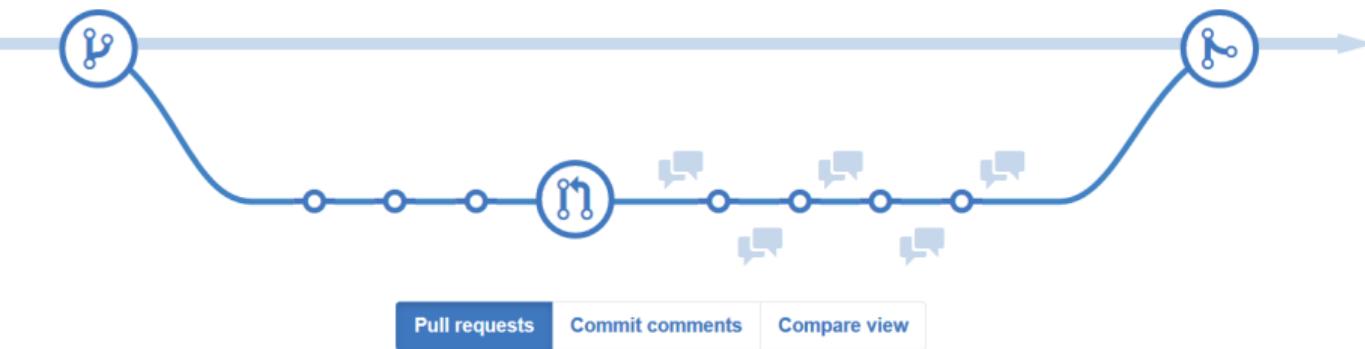
- Cannot be done by yourself
- Reviewers need programming experience and knowledge of the code (mutual feedback)
- Feedback should be timely and constructive
- Only changes reviewed, not too many



# Code Reviews on Github

## Collaborative code review.

Code review is an essential part of the GitHub workflow. After creating a branch and making one or more commits, a Pull Request starts the conversation around the proposed changes. Additional commits are commonly added based on feedback before merging the branch.



# Modulo in Different Programming Languages

## Overview by Torsten Curdt:

Language	13 mod 3	-13 mod 3	13 mod -3	-13 mod -3
C	1	-1	1	-1
Go	1	-1	1	-1
PHP	1	-1	1	-1
Rust	1	-1	1	-1
Scala	1	-1	1	-1
Java	1	-1	1	-1
Javascript	1	-1	1	-1
Ruby	1	2	-2	-1
Python	1	2	-2	-1

## Perform Example Code Review

```
/**  
 * Computes the remainder of the  
 * Euclidean devision of a by b. In  
 * contrast to the Java version a % b,  
 * the output will always be positive.  
 * Throws ArithmeticException when b is  
 * equal to 0.  
 *  
 * @param a dividend  
 * @param b divisor != 0  
 * @return remainder r with 0 <= r < b  
 */  
public static int modulo(int a, int b) {  
    if (b < 0) { // 1  
        b *= -1; // 2  
    }  
    int m = a; // 3  
    while (m < 0 || m > b) { // 4  
        m += m < 0 ? b : -b; // 5  
    }  
    return m; // 6  
}
```

# Quality Assurance

## Lessons Learned

- What is quality, quality assurance, product quality, quality in use?
- Terms: software testing, validation/defect testing, validation/verification, development/release/user testing, manual/automated testing, constructive/analytical/organizational quality assurance
- Code reviews
- Further Reading: [Sommerville](#), Chapter 8 Software Testing  
[Ludewig and Licher](#), Chapter 5 (Software-Qualität) and Chapter 13 (Software-Qualitätssicherung und -Prüfung)

## Practice

- See [Moodle](#)
- Think of a change to the modulo method (cf. previous slide) that would be classified as error or warning by the Java compiler
- Do a code review of the [modulo method](#) (i.e., comment on possible improvements)
- Submit your results in Moodle and inspect other submissions

# Lecture Contents

## 1. Quality Assurance

## 2. White-Box Testing

Test Cases

Test-Case Design

White-Box Testing

Coverage Criteria

Statement Coverage

Branching Coverage

Term Coverage

Lessons Learned

## 3. Black-Box Testing



**Edsger W. Dijkstra (1972)**

[The Humble Programmer]

“Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.”

**Burt Rutan**

[King et al. 2018]

“Testing leads to failure, and failure leads to understanding.”

# Test Cases

(Testfälle)

## Systematic Test

[Ludewig and Licher]

A systematic test is a test, in which

1. the setup is defined,
2. the inputs are chosen systematically,
3. the results are documented and evaluated by criteria being defined prior to the test.

## Test Case

[Ludewig and Licher]

In a test, a number of test cases are executed, whereas each test case consists **input values** for a single execution and **expected outputs**. An

**exhaustive test** refers a test in which the test cases exercise all the possible inputs.

## Exhaustive Testing in Practice?

```
boolean a, b, c;  
int i, j;
```

`bla(a,b,c)` has  $2^3 = 8$  possible inputs  
`blub(i,j)` has  $(2^{32})^2 = 2^{64} \approx 10^{19}$  inputs

- assuming  $10^9$  test cases can be executed in 1 second (cf. CPU with more than 1 GHz)
- exhaustive test of `blub` takes  $\approx 585$  years
- testing for a day would cover less than 0.0005 % of the inputs

How to test thousands of such methods several times a day?

# Test-Case Design

(Testfallentwurf)

## Goal

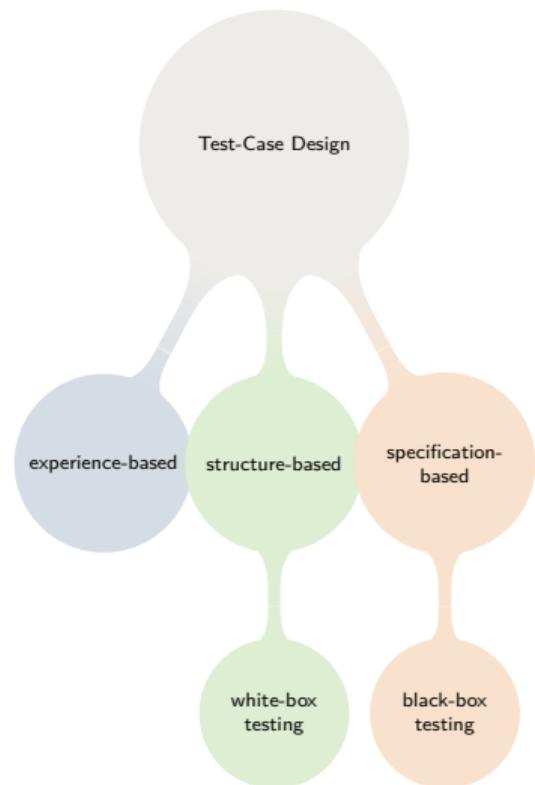
[Ludewig and Licher]

Detect a large number of failures with a low number of test cases. A test case (execution) is **positive**, if it detects a failure, and **successful** if it detects an unknown failure.

## An ideal test case is ...

[Ludewig and Licher]

- representative: represents a large number of feasible test cases
- failure sensitive: has a high probability to detect a failure
- non-redundant: does not check what other test cases already check



# White-Box Testing

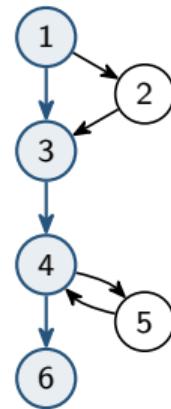
(Strukturtest)

## White-Box Testing

[Ludewig and Licher]

- inner structure of test object is used
- idea: coverage of structural elements
- code translated into control flow graph
- specific test case (concrete inputs)
  - derived from logical test case (conditions)
  - derived from path in control flow graph

```
public static int modulo(int a, int b) {  
    if (b < 0) { // 1  
        b *= -1; // 2  
    }  
    int m = a; // 3  
    while (m < 0 || m > b) { // 4  
        m += m < 0 ? b : -b; // 5  
    }  
    return m; // 6  
}
```



# Coverage Criteria

(Überdeckungskriterien)

## Coverage Criteria

[Ludewig and Licher]

1. statement coverage (**Anweisungsüberdeck.**):  
all statements are executed for at least one test case
2. branching coverage (**Zweigüberdeckung**):  
statement coverage and for each branching statement all branches have been exercised
3. term coverage (**Termüberdeckung**):  
branching coverage and terms ( $n$ ) used in a branching statement are combined exhaustively ( $2^n$ ) (simplified)

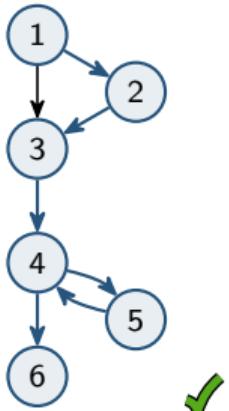
## In Practice

100% statement coverage not feasible in presence of dead code or some unreachable error handling

100% term coverage not feasible for certain dependencies between choices: `even() || odd()`

## Statement Coverage for Modulo Example

```
public static int modulo(int a, int b) {  
    if (b < 0) { // 1  
        b *= -1; // 2  
    }  
    int m = a; // 3  
    while (m < 0 || m > b) { // 4  
        m += m < 0 ? b : -b; // 5  
    }  
    return m; // 6  
}
```



### First Test Case

path: 1, 2, 3, 4, 5, 6

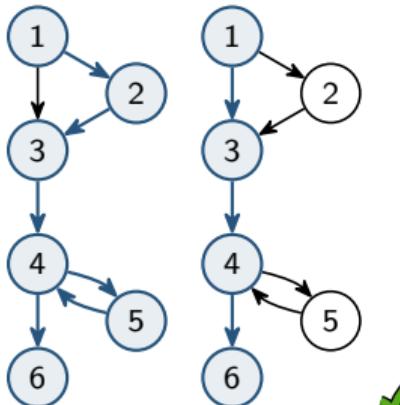
logical test case:  $b < 0 \wedge a > -b \wedge a + b \leq -b$

specific test case:  $a = 5, b = -3$

expected result:  $m = 2$

## Branching Coverage for Modulo Example

```
public static int modulo(int a, int b) {  
    if (b < 0) { // 1  
        b *= -1; // 2  
    }  
    int m = a; // 3  
    while (m < 0 || m > b) { // 4  
        m += m < 0 ? b : -b; // 5  
    }  
    return m; // 6  
}
```



### First Test Case

path: 1, 2, 3, 4, 5, 4, 6

logical test case:  $b < 0 \wedge a > -b \wedge a + b \leq -b$

specific test case:  $a = 5, b = -3$  and  $m = 2$

### Second Test Case

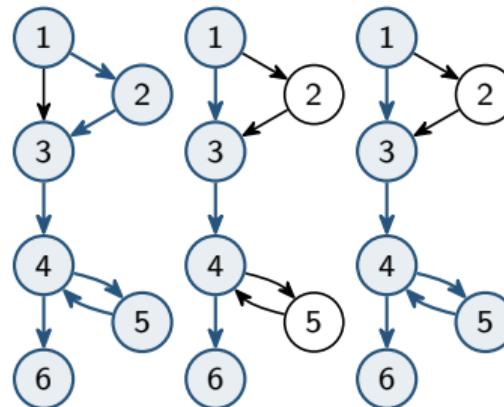
path: 1, 3, 4, 6

logical test case:  $b \geq 0 \wedge 0 \leq a \leq b$

specific test case:  $a = 0, b = 5$  and  $m = 0$

## Term Coverage for Modulo Example

```
public static int modulo(int a, int b) {  
    if (b < 0) { // 1  
        b *= -1; // 2  
    }  
    int m = a; // 3  
    while (m < 0 || m > b) { // 4  
        m += m < 0 ? b : -b; // 5  
    }  
    return m; // 6
```



logical test case:  $b < 0 \wedge a > -b \wedge a + b \leq -b$

specific test case:  $a = 5, b = -3$  and  $m = 2$  ✓

$\neg(m < 0)$  and  $m > b$  ✓

logical test case:  $b \geq 0 \wedge 0 \leq a \leq b$

specific test case:  $a = 0, b = 5$  and  $m = 0$  ✓

$\neg(m < 0)$  and  $\neg(m > b)$  ✓

$m < 0$  and  $m > b$  impossible\* ✗

path: 1, 3, 4, 5, 4, 6

logical test case:  $b \geq 0 \wedge a < 0 \wedge 0 \leq a + b \leq b$

specific test case:  $a = -2, b = 5$  and  $m = 3$  ✓

$m < 0$  and  $\neg(m > b)$  ✓

\* see third part of the lecture

# White-Box Testing

## Lessons Learned

- Systematic test, exhaustive testing
- Test case: representative, failure sensitive, non-redundant
- Test-case design: experience-/structure-/specification-based
- Coverage in white-box testing: statement/branching/term coverage
- Further Reading: [Ludewig and Licher](#), Chapter 19 ([Programmtest](#))

## Practice

- See [Moodle](#)
- Watch screencast on how to do white-box testing with JUnit
- Implement a new operation in the calculator and update the white-box tests accordingly:  
<https://github.com/tthuem/2020WS-SWT-Calculator/tree/testinglecturev1>
- Post your code changes and tests in Moodle

# Lecture Contents

1. Quality Assurance
2. White-Box Testing
3. Black-Box Testing
  - Black-Box Testing
  - Equivalence Class Testing
  - Boundary Testing
  - Detected Faults in Modulo Example
  - Reasons for Positive Test Cases
  - Lessons Learned

# Black-Box Testing

[Ludewig and Licher]

## Motivation

- source code not always available (e.g., outsourced components, obfuscated code)
- specific test cases derived from logical ones using arbitrary values
- specification not incorporated so far (only for expected results)
- invalid inputs not tested
- errors are not equally distributed

## Black-Box Testing (Funktionstest)

- test-case design based on specification
- source code and its inner structure is ignored (assumed as a black-box)

## 1. Equivalence Class Testing

- idea: classify inputs and outputs into equivalence classes
- assumption: equivalent test cases detect the same faults, one test case is sufficient

## 2. Boundary Testing

- extension of equivalence class testing
- goal: use experience (e.g., off-by-one errors)
- for every equivalence class: consider smallest, typical, and largest value

## In Practice

- often combinations of white-box and black-box testing
- more techniques with requirements or design

# Equivalence Class Testing

## JavaDoc Specification for Modulo Example

```
/**  
 * Computes the remainder of the  
 * Euclidean devision of a by b. In  
 * contrast to the Java version  $a \% b$ ,  
 * the output will always be positive.  
 * Throws ArithmeticException when b is  
 * equal to 0.  
 *  
 * @param a dividend  
 * @param b divisor != 0  
 * @return remainder r with  $0 \leq r < b$   
 */  
public static int modulo(int a, int b) {
```

## Equivalence Classes

- input a:  $a < 0$ ,  $a \geq 0$
- input b:  $b < 0$ ,  $b \geq 0$
- output:  $m = 0$ ,  $m > 0$ , exception

## Test Cases

	TC1	TC2	TC3
$a < 0$	X		
$a \geq 0$		X	X
$b < 0$	X		
$b > 0$		X	
$b = 0$			X
$m = 0$	X		
$m > 0$		X	
exception			X
input a	-3	1	2
input b	-3	2	0
expected output	0	1	exception
result	0 ✓	1 ✓	timeout ✘

# Boundary Testing

## Test Cases

	TC1	TC2	TC3	TC4	TC5	TC6	TC7
$a < 0$	X			min	max		
$a \geq 0$		X	X			min	max
$b < 0$	X			max		min	
$b > 0$		X			max		
$b = 0$			X				
$m = 0$	X			X		X	X
$m > 0$		min			max		
exception			X				
input a	-3	1	2	minInt	-1	0	maxInt
input b	-3	2	0	-1	maxInt	minInt	1
expected output	0	1	exception	0	maxInt-1	0	0
result	0 ✓	1 ✓	timeout ✗	0 ✓	maxInt-1 ✓	timeout ✗	1 ✗

## Detected Faults in Modulo Example

```
public static int modulo(int a, int b) {  
    if (b < 0) { // 1  
        b *= -1; // 2  
    }  
    int m = a; // 3  
    while (m < 0 || m > b) { // 4  
        m += m < 0 ? b : -b; // 5  
    }  
    return m; // 6  
}
```

✖ TC3: infinite loop for  $b = 0$ , missing exception compared to JavaDoc

✖ TC6:  $b$  remains negative as  
`-Integer.MIN_VALUE == Integer.MIN_VALUE`  
and the loop condition is fulfilled for any integer

✖ TC7: indicates that  $m > b$  in the loop condition should be fixed to  $m \geq b$

## Improved Modulo Example

```
/**  
 * Computes the remainder of the Euclidean division of a by b. In contrast to the Java version a % b, the output will always be positive.  
 * Throws ArithmeticException when b is equal to 0.  
 *  
 * @param a dividend  
 * @param b divisor != 0  
 * @return remainder r with 0 <= r < b  
 */  
public static int modulo(int a, int b) {  
    int m = a % b;  
    return m < 0 ? m + b : m;  
}
```

Passes All Test Cases



# Reasons for Positive Test Cases

## Reasons for Positive Test Cases

- actual fault
- wrong test case (input and expected results do not match)
- interaction with other programs/libraries
- fault in the compiler
- fault in the operating system / device drivers
- fault in the hardware or hardware defect
- not enough memory
- does not halt (cf. undecidability of the halting problem)
- bitflip due to cosmic ray
- ...



**Edsger W. Dijkstra**

[Goodliffe 2014]

“If debugging is the process of removing software bugs, then programming must be the process of putting them in.”



**Brian Kernighan (1978)**

[Kernighan and Plauger]

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you’re as clever as you can be when you write it, how will you ever debug it?”

# Black-Box Testing

## Lessons Learned

- Black-box testing: equivalence class testing and boundary testing
- Even systematic testing cannot ensure finding all faults
- Further Reading: [Ludewig and Licher](#), Chapter 19 ([Programmtest](#))

## Practice

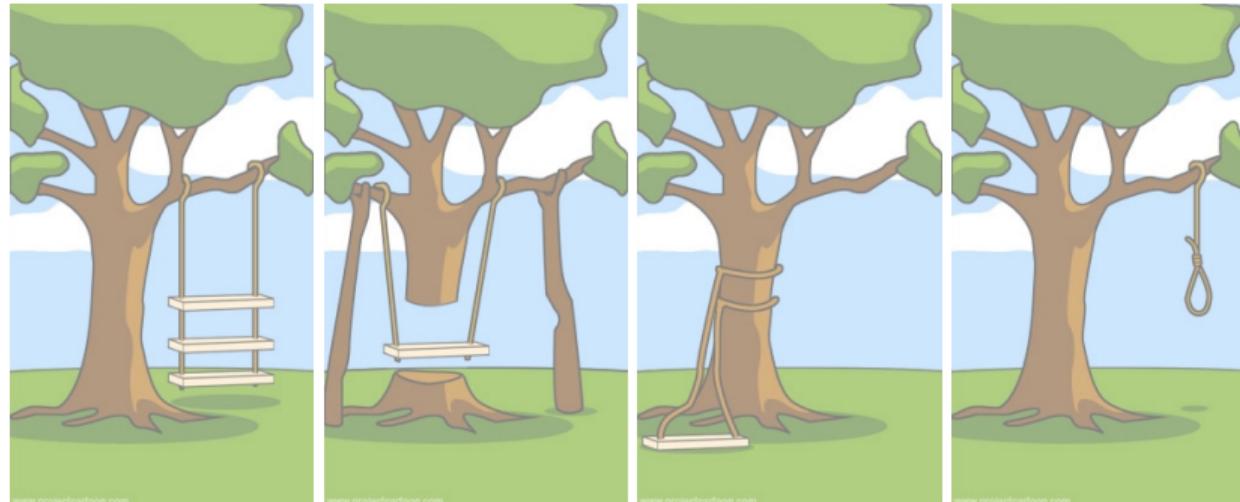
- See [Moodle](#)
- Watch screencast on how to do black-box testing with JUnit
- Extend the black-box tests of the calculator for your new operation: <https://github.com/tthuem/2020WS-SWT-Calculator/tree/testinglecturev2>
- Post your changed code and tests in Moodle



# Software Engineering

9. Development Process | Thomas Thüm | January 26, 2022

# Development Process (Vorgehensmodelle)



how the customer  
explained it

how the analyst  
designed it

how the programmer what the beta testers  
implemented it received

# Lecture Overview

1. Development Processes: The Waterfall Model
2. V-Model for Extensive Testing
3. Scrum for Agile Development

# Lecture Contents

## 1. Development Processes: The Waterfall Model

- The Process of Food Delivery
- Questionnaire Results
- Software Development Process
- Waterfall Model
- Waterfall Model – Discussion
- Lessons Learned

## 2. V-Model for Extensive Testing

## 3. Scrum for Agile Development

# The Process of Food Delivery



eat, deliver, order, pay, choose, wait  
– not necessarily in this order

# Questionnaire Results

1

Wirst du in 10 Jahren Software entwickeln?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	 77%	34
Nein	 23%	10

2

Wirst du in 10 Jahren Software testen?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	 82%	36
Nein	 18%	8

3

Wirst du in 10 Jahren Software beauftragen?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	 69%	27
Nein	 31%	12

4

Wirst du in 10 Jahren Softwareanforderungen ermitteln?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	 93%	40
Nein	 7%	3

5

Wirst du in 10 Jahren die Entwicklung von Software leiten?

Antworten	relative Häufigkeit	absolute Häufigkeit
Ja	 57%	24
Nein	 43%	18

# Software Development Process

## Motivation

- how to **structure** the project?
- what are **activities** and phases?  
analysis (requirements elicitation + system modeling), design (architectural + software design), implementation, test, deployment
- how to organize **communication**?
- who has which **responsibilities**?
- did we **forget** anything?
- can we **predict** the project result?
- how to **manage and control** progress?
- how to share and elicit **experience**?
- how to synchronize **hardware** and software development?

## Software Process

[Sommerville]

"A **software process** is a set of related activities that leads to the production of a software system. [...] **Products** or deliverables are the outcomes of a process activity. [...] **Roles** reflect the responsibilities of the people involved in the process. [...] Pre- and postconditions are **conditions** that must hold before and after a process activity."

## Why Different Processes?

[Sommerville]

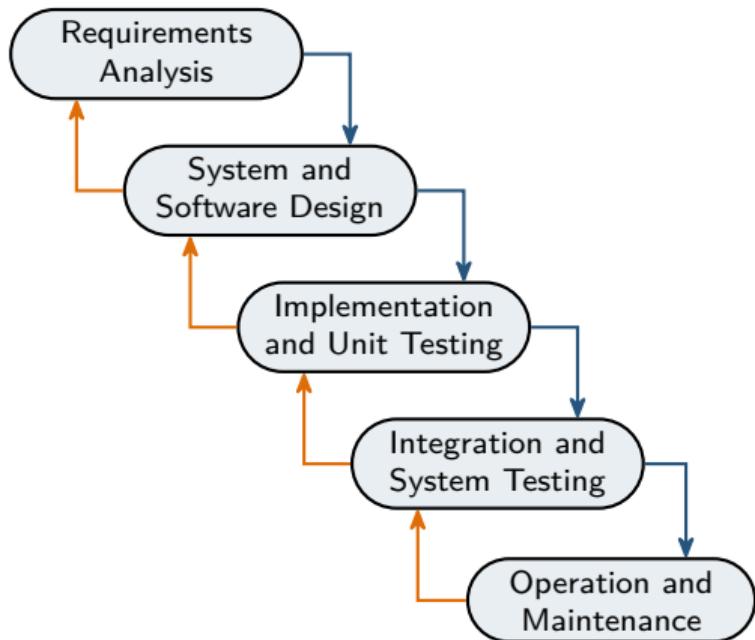
"The process used in different companies depends on the type of software being developed, the requirements of the software customer, and the skills of the people writing the software."

# Waterfall Model (Wasserfallmodell)

[Sommerville]

## Waterfall Model

- first process model, motivated by practice
- by Winston W. Royce 1970
- each development phase ends by the approval of one or more documents (document-driven process model)
- phases do not overlap
- numerous variants with varying number of phases: 5–7
- here: simplified variant by Sommerville



# Waterfall Model

[Sommerville]

## 1. Requirements Analysis

"The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a **system specification**."

## 2. System and Software Design

"The systems design process allocates the requirements to either hardware or software systems. It establishes an overall **system architecture**. Software design involves identifying and describing the fundamental software system abstractions and their relationships."

## 3. Implementation and Unit Testing

"During this stage, the software design is realized as a set of **programs or program units**. Unit testing involves verifying that each unit meets its specification."

## 4. Integration and System Testing

"The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the **software system is delivered** to the customer."

## 5. Operation and Maintenance

"Normally, this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves **correcting errors** that were not discovered in earlier stages of the life cycle [...]."

# Waterfall Model – Discussion

[Sommerville]

## Advantages

- easy to understand, manage, and control
- good for systems development (i.e., with high manufacturing costs for hardware)
- easier to use the same model as for hardware
- combination with formal system development feasible (e.g., B method)

## Example Domains

- **embedded systems** where software has to interface with hardware systems
- **critical systems** with extensive safety and security analysis of specification and design
- **large software systems** that are typically developed by several companies

## Disadvantages

- for software development: stages should feed information to each other
- changes in previous stages are hard to achieve
- problems from previous stages left for later resolution
- freezing of requirements may lead to software not wanted by the user
- freezing of design may lead to bad structure and implementation tricks
- requires clear and stable requirements and good design upfront

# Development Processes: The Waterfall Model

## Lessons Learned

- Software development process: motivation and goal
- Waterfall model: phases, results, example domains, advantages and disadvantages
- Further Reading: [Sommerville](#), Chapter 2 Software Processes

## Practice

- See [Moodle](#)
- Answer the quiz in Moodle to track your learning progress

# Lecture Contents

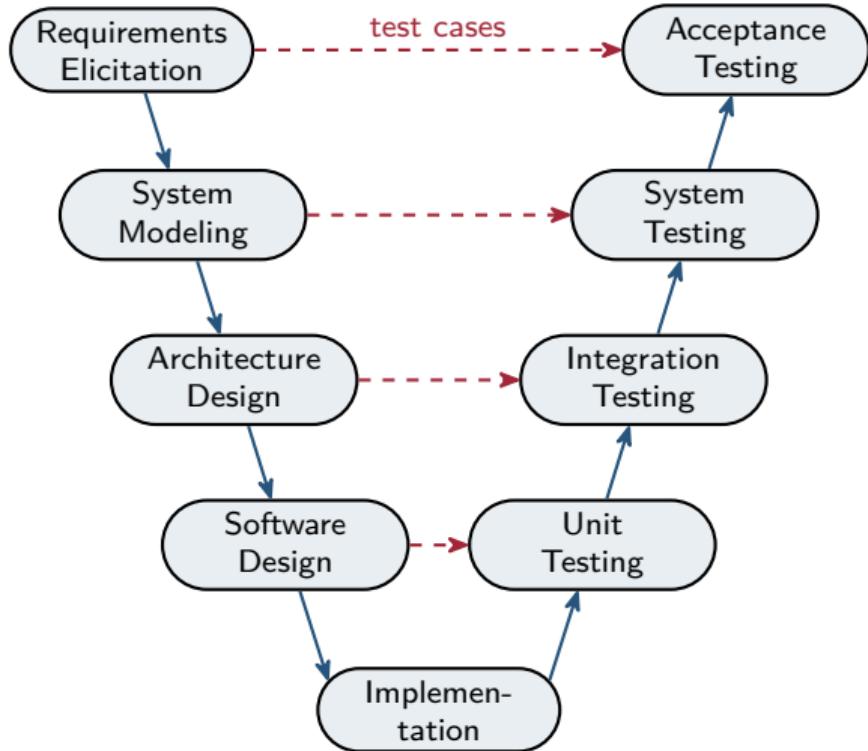
1. Development Processes: The Waterfall Model
2. V-Model for Extensive Testing
  - V-Model
  - Stages of Testing
  - V-Model – Discussion
  - Lessons Learned
3. Scrum for Agile Development

# V-Model

## V-Model

[Ludewig and Licher]

- developed by the German Ministry of Defense ([Verteidigungsministerium](#)) and required since 1992
- extension of the waterfall model: project-aligned activities such as quality assurance, configuration management, project management
- 1997 V-model 97: incremental development, inclusion of hardware, object-oriented development
- 2004 V-model XT (for extreme tailoring): adaptability, application beyond software
- integration of four testing stages



# Stages of Testing (Teststufen)

[Ludewig and Licher, Sommerville]

## 1. Unit Testing (Komponententest)

- each component is tested independently
- unit may stand for a component or smaller entities (package, class, method)
- tests created by the developers
- automation is common (e.g., JUnit)

## 2. Integration Testing (Integrationstest)

- some components are integrated (e.g., into subsystems) and tested together
- detects inconsistencies in interfaces and communication between components
- top-down vs bottom-up integration

## 3. System Testing (Systemtest)

- all components are integrated to the complete system
- detects further inconsistencies and unanticipated interactions
- system is tested against system requirements

## 4. Acceptance Testing (Abnahmetest)

- final stage in the testing process before accepted for operational use
- system is tested against user requirements and with real data
- performed by (potential) customer

# V-Model – Discussion

[Ludewig and Licher]

## Advantages

- quality assurance in several testing stages
- completeness helps to not miss activities
- V-model 97/XT are widely applicable (e.g., hardware, incremental)

## Example Domains

- since 1992 V-model required by German government (e.g., Bundeswehr), since 2004 V-model XT
- embedded, critical, and large software systems as for the waterfall model

## Disadvantages

- complex and extensive process
- adaptations often required (cf. XT for extreme tailoring)
- overhead useful only for large software systems
- changes in requirements are problematic

# V-Model for Extensive Testing

## Lessons Learned

- V-model
- Testing stages: unit testing, integration testing, system testing, acceptance testing
- Further Reading: [Ludewig and Licher](#), Chapter 10.3 ([Das V-Modell](#))  
[Sommerville](#), Chapter 2.2.3 Software Validation and Chapter 8.1 Development Testing

## Practice

- See [Moodle](#)
- Choose two of the four testing phases.
- Give an example of a fault that can be found in one of selected phases.
- Explain whether it could and should have been found in the other selected phase.
- Upload example and explanation to Moodle

# Lecture Contents

1. Development Processes: The Waterfall Model
2. V-Model for Extensive Testing
3. Scrum for Agile Development
  - Motivation for Agile Development
  - Manifesto of Agile Software Development
  - Principles behind the Agile Manifesto
  - Scrum
  - Burndown Chart
  - Scrum: Roles and Terms
  - Scrum – Discussion
  - Lessons Learned

# Motivation for Agile Development

## Motivation

[Sommerville]

- businesses operate globally and in a rapidly changing environment
- software is part of almost all business operations
- new software has to be developed quickly
- often infeasible to derive a complete set of stable requirements
- plan-driven process models (e.g., waterfall) deliver software long after originally specified

## Agile (Development) Methods

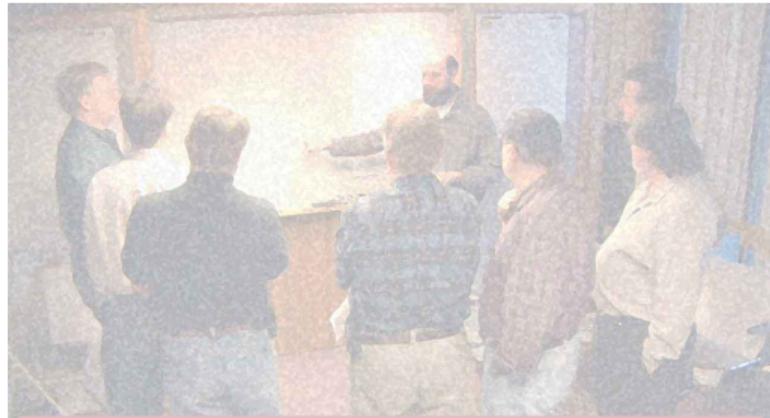
[Sommerville]

Development of agile methods since late 1990s:

1. specification, design, implementation are interleaved
2. each increment is specified and evaluated by stakeholders (e.g., end-users)
3. extensive tool support is used

# Manifesto of Agile Software Development

[[agilemanifesto.org](http://agilemanifesto.org)]



## Ski Resort in Utah (February 2001)

17 experts on software development:

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

## Manifesto

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- individuals and interactions  
over processes and tools
- working software  
over comprehensive documentation
- customer collaboration  
over contract negotiation
- responding to change  
over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

# Principles behind the Agile Manifesto

[[agilemanifesto.org](http://agilemanifesto.org)]

## Principles 1–6

1. “Our highest priority is to **satisfy the customer** through early and continuous delivery of valuable software.
2. **Welcome changing requirements**, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must **work together daily** throughout the project.
5. Build projects around **motivated individuals**. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.”

## Principles 7–12

7. “**Working software** is the primary measure of progress.
8. Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to **technical excellence and good design** enhances agility.
10. **Simplicity**—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from **self-organizing teams**.
12. At regular intervals, the **team reflects** on how to become more effective, then tunes and adjusts its behavior accordingly.”

# Scrum

## User Story

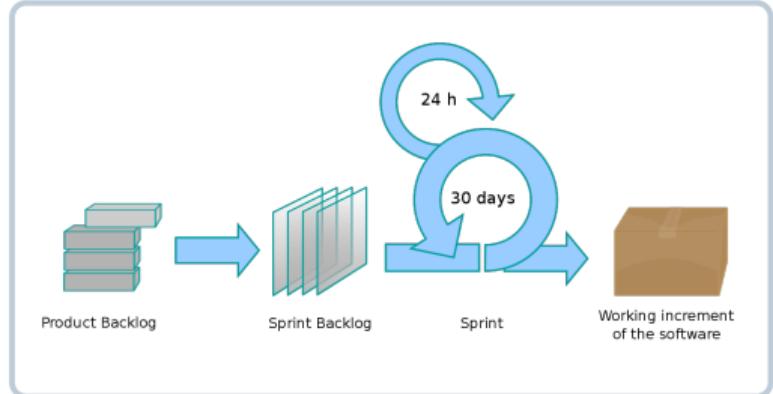
[Sommerville]

- a scenario of use that might be experienced by a system user
- aka. **story card** as user stories are sometimes written on physical cards
- user stories are prioritized by the customer
- subset of all user stories is chosen for the next iteration
- used in many agile methods

## Scrum

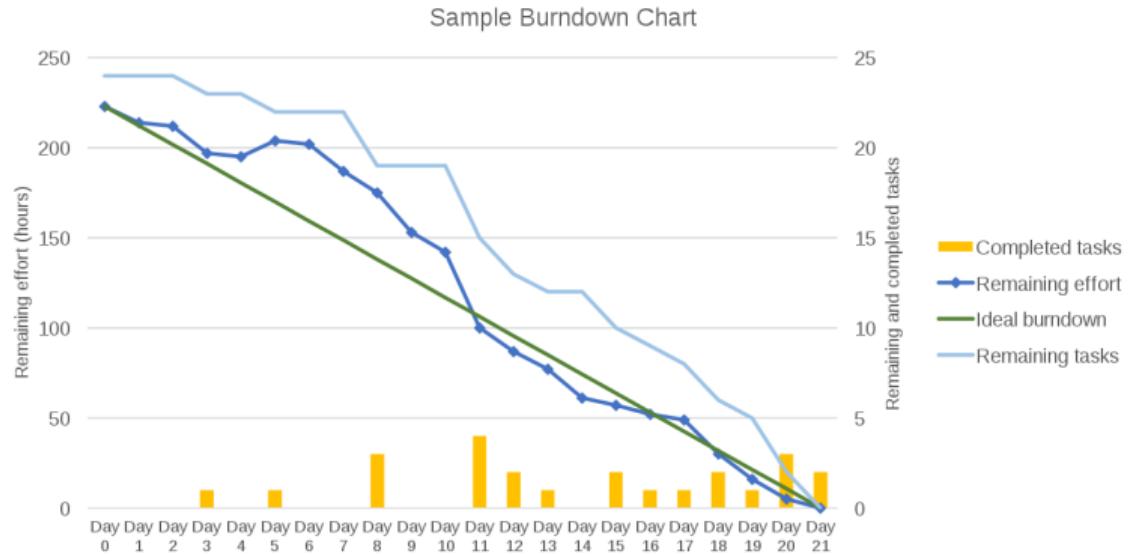
[Sommerville]

- an agile method, most-widely used method
- no special development techniques (like pair programming, test-driven development)
- **product backlog**: list of user stories, collected and prioritized by the **product owner**
- **sprint backlog**: user stories selected by the scrum team for the next **sprint**



# Burndown Chart

Burndown Chart: used by scrum master to track progress



# Scrum: Roles and Terms

## Scrum Roles

[Sommerville]

**Development team:** A self-organizing group of software developers, which should be **no more than seven people**. They are responsible for developing the software and other essential project documents.

**Product owner:** An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development, and **continuously review the product backlog** to ensure that the project continues to meet critical business needs. The product owner can be a customer but might also be a product manager in a software company or other stakeholder representative.

**Scrum master:** The scrum master is responsible for ensuring that the scrum process is followed and **guides the team** in the effective use of scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the scrum team is not diverted by outside interference."

## Scrum Terms

[Sommerville]

**"Potentially shippable product increment:** The software increment that is delivered from a sprint. The idea is that this should be potentially shippable, which means that it is in a **finished state** and no further work, such as testing, is needed to incorporate it into the final product.

**Product backlog:** This is a list of **to-do items** that the scrum team must tackle. They may be feature definitions for the software, software requirements, user stories, or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.

**Daily scrum:** A daily meeting (cf. stand-up meeting) of the scrum team that **reviews progress and prioritizes work** to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.

**Sprint:** A development iteration. Sprints are usually **2 to 4 weeks** long.

**Velocity:** An estimate of how much product backlog effort a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for **measuring and improving performance**.

# Scrum – Discussion

[Sommerville]

## Advantages

- product is broken down into manageable and understandable chunks
- **unstable requirements** can be easily incorporated
- good **team communication** and transparency
- **customers can inspect increments** and understand how the product works
- establishes **trust** between customers and developers

## Disadvantages

- unclear how scale to **larger teams**
- problematic when **contract negotiation** is required (as customer pays for development time rather than set of requirements)
- **documentation and testing** not explicitly covered (requires extra story cards)
- requires **continuous customer input**
- **tacit knowledge** not available during maintenance (of long-life systems)
- detailed documentation required for **external regulation** and **outsourcing**

# Scrum for Agile Development

## Lessons Learned

- Motivation for agile development
- Agile Manifesto (four values, twelve principles)
- User stories
- Scrum: roles, terms, discussion
- Further Reading: [Sommerville](#), Chapter 3 Agile Software Development

## Practice

- See [Moodle](#)
- What did you learn? Formulate 1–2 questions and post them in Moodle.
- Check your learning progress! Answer 1–2 questions of your colleagues.



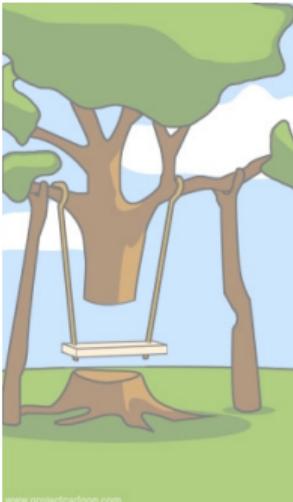
# Software Engineering

10. Project Management | Thomas Thüm | February 9, 2022

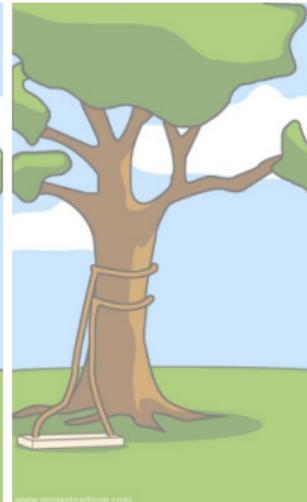
# Project Management



how the customer  
explained it



how the analyst  
designed it



how the programmer  
implemented it



how the customer  
was billed

# Lecture Overview

1. Introduction to Project Management
2. Project Planning and Scheduling
3. Summary on Software Engineering I

# Lecture Contents

## 1. Introduction to Project Management

Software Development Project

Project Management

Activities in Project Management

Risk Management

People Management

Lessons Learned

## 2. Project Planning and Scheduling

## 3. Summary on Software Engineering I

# Software Development Project

[Ludewig and Licher]

## Software Development Project

- aka. software engineering project
- temporary activity with start and end date
- has goals
  - ▶ creation / modification of a software product
  - ▶ creation / modification of components for future projects
  - ▶ gain experience / knowledge
  - ▶ capacity utilization  
*(Mitarbeiterauslastung)*
  - ▶ ...
- is successful if goals are largely fulfilled

# Project Management

[Sommerville]

## Motivation

"Good management cannot guarantee project success. However, bad management usually results in project failure: The software may be delivered late, cost more than originally estimated, or fail to meet the expectations of customers."

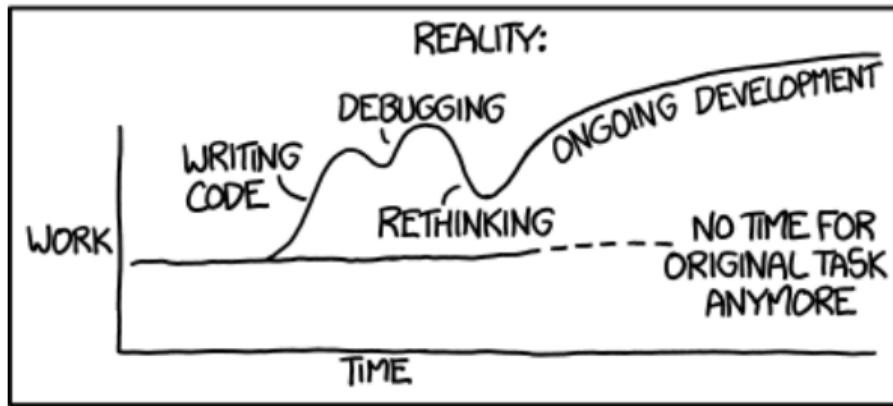
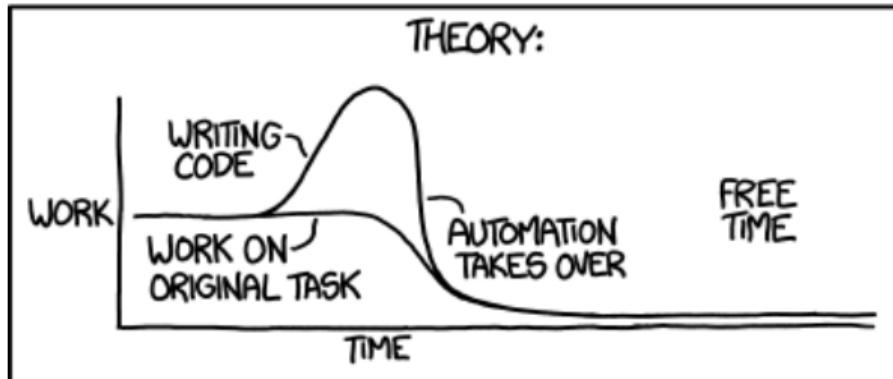
## Goals of Project Management

- "deliver the software to the customer at the agreed **time**
- keep overall **costs** within budget
- deliver software that meets the customer's **expectations**
- maintain a coherent and well-functioning development **team**"

## Project Management Depends on ...

- company size** large companies have management hierarchies and reporting / budgeting / approval processes
- customers** external customers (i.e., government agencies) usually have policies
- software size** large systems require multiple development teams in different companies / locations
- software type** safety-critical systems require all design decisions to be documented
- dev. process** project management heavily depends on process model

"I SPEND A LOT OF TIME ON THIS TASK.  
I SHOULD WRITE A PROGRAM AUTOMATING IT!"



# Activities in Project Management

[Somerville]

## Project Planning

"Project managers are responsible for **planning, estimating, and scheduling** project development and assigning people to tasks. They supervise the work to ensure that it is carried out to the required standards, and they **monitor progress** to check that the development is on time and within budget."

## Risk Management

"Project managers have to **assess the risks** that may affect a project, monitor these risks, and take action when problems arise."

## People Management

"Project managers are responsible for **managing a team** of people. They have to choose people for their team and establish ways of working that lead to effective team performance."

## Reporting

"Project managers are usually responsible for **reporting on the progress** of a project to customers and to the managers of the company developing the software. They have to be able to communicate at a range of levels, from detailed technical information to management summaries."

## Proposal Writing

"The first stage in a software project may involve writing a proposal to **win a contract** to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out. It usually includes **cost and schedule estimates** and justifies why the project contract should be awarded to a particular organization or team. Proposal writing is a critical task as the survival of many software companies depends on having enough proposals accepted and contracts awarded."

# Risk Management

[Sommerville]

## Risk

**Probability** insignificant, low, moderate, high, very high

**Severity** insignificant, tolerable, serious, catastrophic

## Classification of Risks

**Project Risks** affect project schedule or resources: loss of an experienced system architect may result in longer development time

**Product Risks** affect software quality: purchased component may not scale

**Business Risks** affect organization / company: product of a competitor may reduce number of sales

## Stages in Risks Management

- 1. Risk Identification** identify possible project, product, and business risks
- 2. Risk Analysis** assess likelihood and consequences
- 3. Risk Planning** plan how to address risks: avoidance or minimization of effects
- 4. Risk Monitoring** regularly assess risks and revise plans if needed

## Risks in Agile Development

reduced risks for requirements changes, increased risks for loss of stuff due to fewer documentation

# People Management

[Somerville]

## Motivation

"The people working in a software organization are its **greatest assets**. It is expensive to recruit and retain good people, and it is up to software managers to ensure that the engineers working on a project are as **productive** as possible. In successful companies and economies, this productivity is achieved when people are respected by the organization and are assigned responsibilities that reflect their skills and experience."

## In Practice

"Software engineers often have strong **technical skills** but may lack the softer skills that enable them to **motivate and lead a project development team**."

## Critical Factors

- Consistency** treat people comparably with similar rewards
- Respect** let all people contribute and respect their differences in skills
- Inclusion** consider views of least experienced peoples
- Honesty** manager is honest about own skills and team performance

## Teamwork

"Most professional software is developed by project teams that range in size from two to several hundred people. However, as it is impossible for everyone in a large group to work together on a single problem, **large teams are usually split** into a number of smaller groups. Each group is responsible for developing part of the overall system."

# Introduction to Project Management

## Lessons Learned

- Software development projects
- Project management: goals, influences, activities
- Risk and people management
- Further Reading: [Sommerville](#), Chapter 22 Project Management and [Ludewig and Licher](#), Chapter 7.2 ([Software-Projekte](#))

## Practice

- See [Moodle](#)
- Risk identification and analysis: Give an example for a risk of a messenger app in Moodle (2–3 sentences) and specify probability, severity, and classification
- Risk planning and monitoring: How to address the risk mentioned by one of your colleagues? What could change during the project?

# Lecture Contents

1. Introduction to Project Management
2. Project Planning and Scheduling
  - Project Planning
  - Gantt Chart
  - Network Diagram
  - Gantt Charts vs Network Diagrams
  - Lessons Learned
3. Summary on Software Engineering I

# Project Planning

**Anonymous**

[[wikiquote.org](#)]

“A goal without a plan is just a wish.”

# Project Planning

[Sommerville]

## At the Proposal Stage

- when bidding for a contract
- enough resources?
- price for the bidding?
- not all requirements known (i.e., system requirements)  $\Rightarrow$  inevitable speculative

## Software Pricing

- effort costs (software engineers / managers)
- hardware and software costs (incl. hardware maintenance and software support)
- travel and training costs
- price = estimated costs + profit + contingency (extra effort, 30–50%)

## On Project Startup

- who will work on the project?
- how to split into increments?
- refine initial estimates

## Throughout the Project

- update plan based on new insights
- learn about the software and team capabilities
- estimates get more accurate

WHEN A USER TAKES A PHOTO,  
THE APP SHOULD CHECK WHETHER  
THEY'RE IN A NATIONAL PARK...

SURE, EASY GIS LOOKUP.  
GIMME A FEW HOURS.

... AND CHECK WHETHER  
THE PHOTO IS OF A BIRD.

I'LL NEED A RESEARCH  
TEAM AND FIVE YEARS.



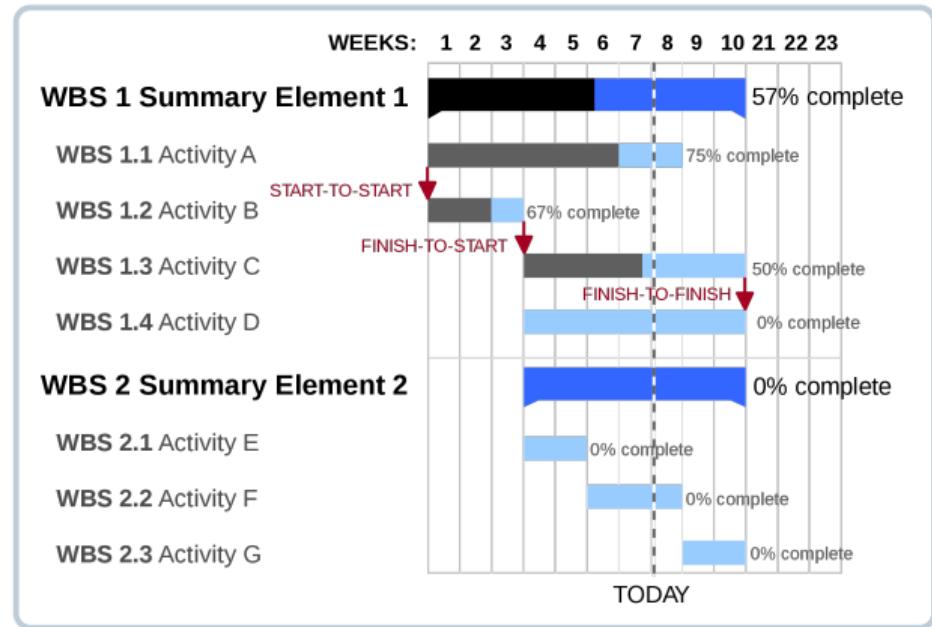
IN CS, IT CAN BE HARD TO EXPLAIN  
THE DIFFERENCE BETWEEN THE EASY  
AND THE VIRTUALLY IMPOSSIBLE.

# Gantt Chart

[Ludewig and Licher, Sommerville]

## Gantt Chart

- named after Henry L. Gantt (1861–1919)
- bar chart with timeline on x axis and activities on the y axis
- optional: progress bars and marker for observation date
- optional: dependencies between tasks
- optional, not shown: highlight dependencies on the critical path
- **critical path:** tasks whose delay also delays the project



# Network Diagram

[Ludewig and Lichter, Sommerville]

## Network Diagram (Netzplan)

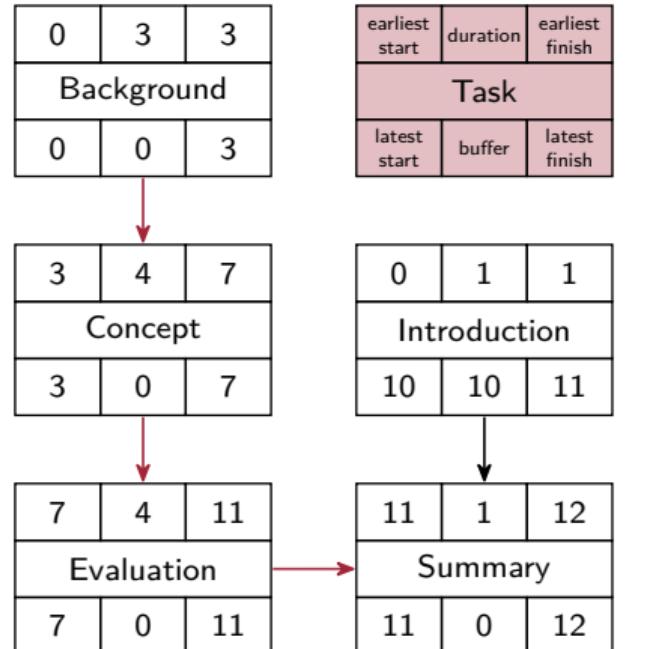
- aka. PERT charts
- directed, acyclic graph
- nodes represent tasks
- edges represent dependencies

## Metra Potential Method

Given project start date and **duration** of each activity we can compute:

- **earliest start** and **earliest finish** time with **forward pass**
- **latest start** and **latest finish** time with **backwards pass**
- **buffer** (time span between earliest and latest start/finish)

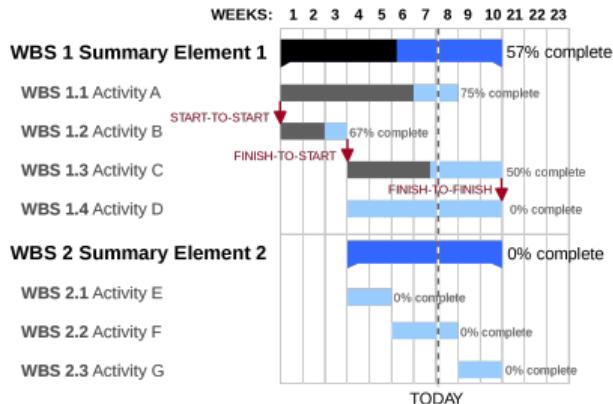
## Example Network for a Bachelor's Thesis



# Gantt Charts vs Network Diagrams [Sommerville]

## Gantt Chart

- very common technique
- many tools available
- great visualization of timing and progress



## Network Diagram (Netzplan)

- clear visualization of dependencies
- explicitly includes buffer times  
(cf. metra potential method)

0	3	3
Background		
0	0	3

earliest start	duration	earliest finish
Task		
latest start	buffer	latest finish

3	4	7
Concept		
3	0	7

0	1	1
Introduction		
10	10	11

# Project Planning and Scheduling

## Lessons Learned

- Project planning (incl. software pricing)
- Project scheduling with Gantt charts and network diagrams
- Further Reading: [Sommerville](#), Chapter 23 Project Planning and [Ludewig and Licher](#), Chapter 8.3.2 ([Projektphasen](#))

## Practice

- See [Moodle](#)
- Search for a tool to create Gantt charts and use it to schedule the writing of a term paper or bachelor thesis
- Upload your schedule to Moodle and report about your experiences with the tool

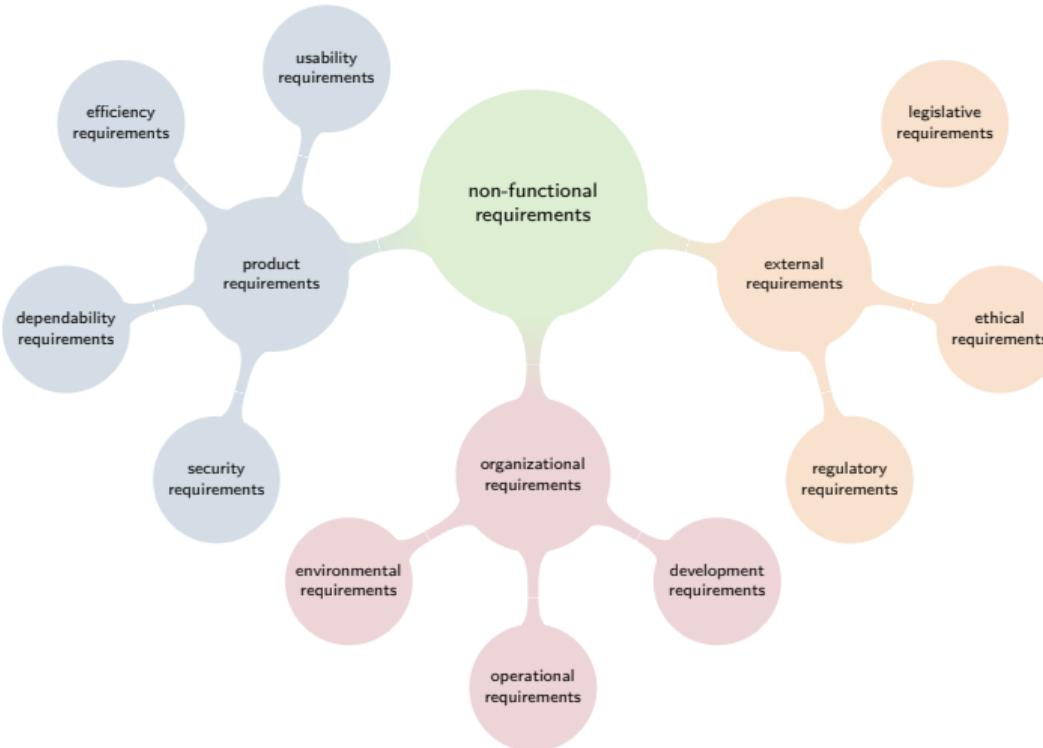
# Lecture Contents

1. Introduction to Project Management
2. Project Planning and Scheduling
3. Summary on Software Engineering I
  - Recap: Software Engineering vs Programming
  - Recap: Requirements
  - Recap: Modeling with UML Diagrams
  - Recap: Architecture
  - Recap: Implementation
  - Recap: Design Patterns
  - Recap: Quality Assurance
  - Recap: Process Models
  - Recap: Project Management
  - Lessons Learned

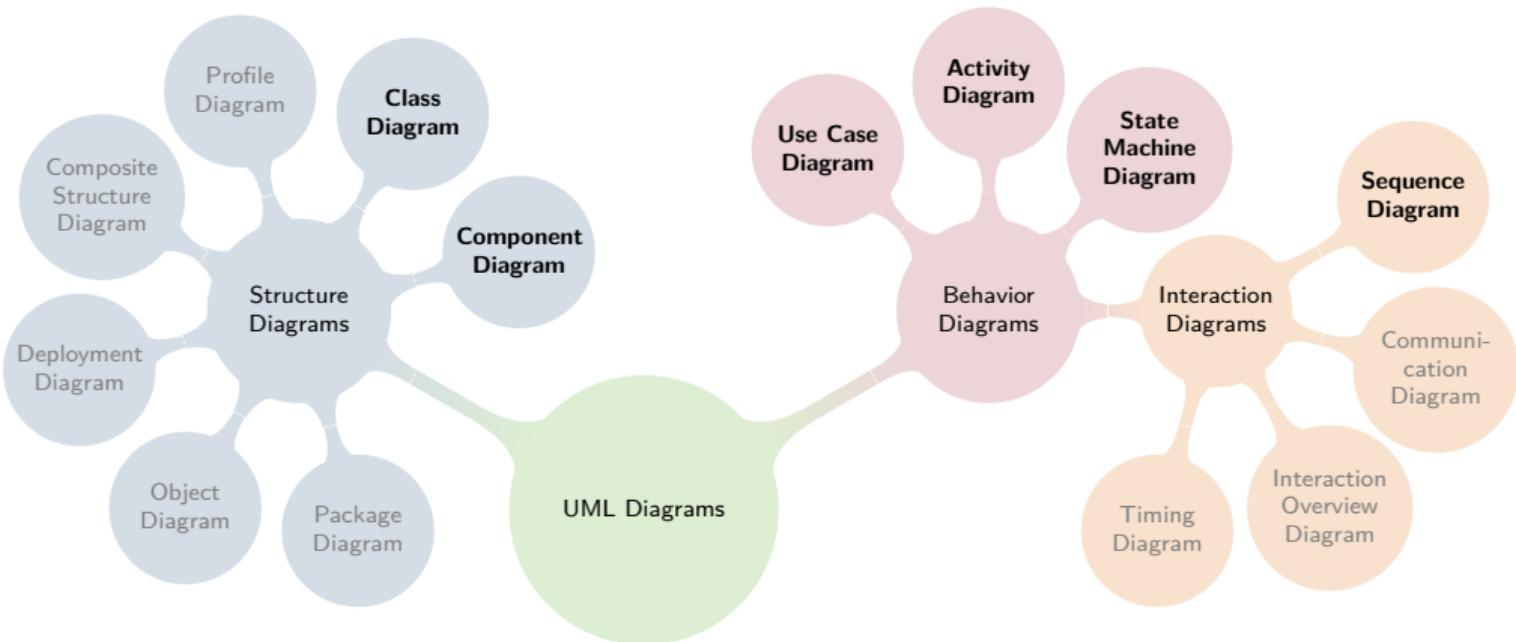
# Recap: Software Engineering vs Programming



# Recap: Requirements



# Recap: Modeling with UML Diagrams [UML 2.5.1]



# Recap: Architecture

## Architectural Pattern (Architekturmuster)

“Architectural patterns capture the essence of an architecture that has been used in different software systems. [...] Architectural patterns are a means of reusing knowledge about generic system architectures.”

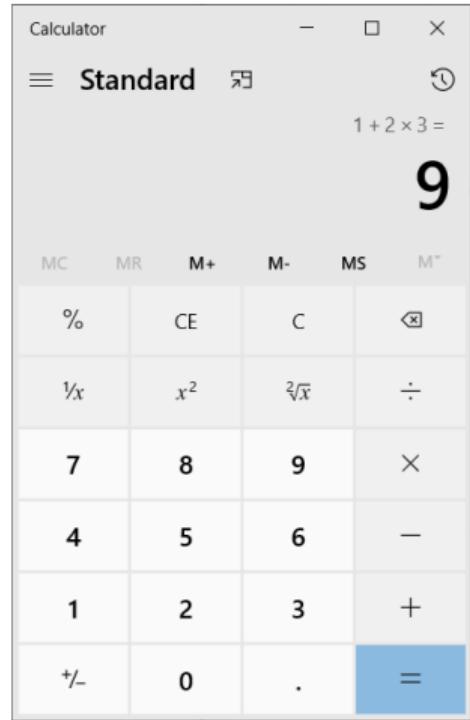
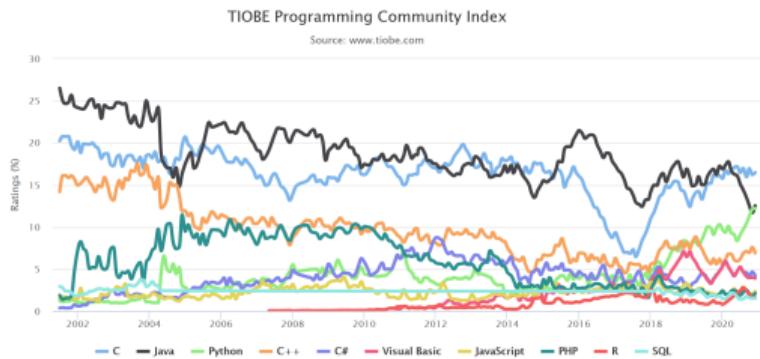
[Sommerville]

## Goals

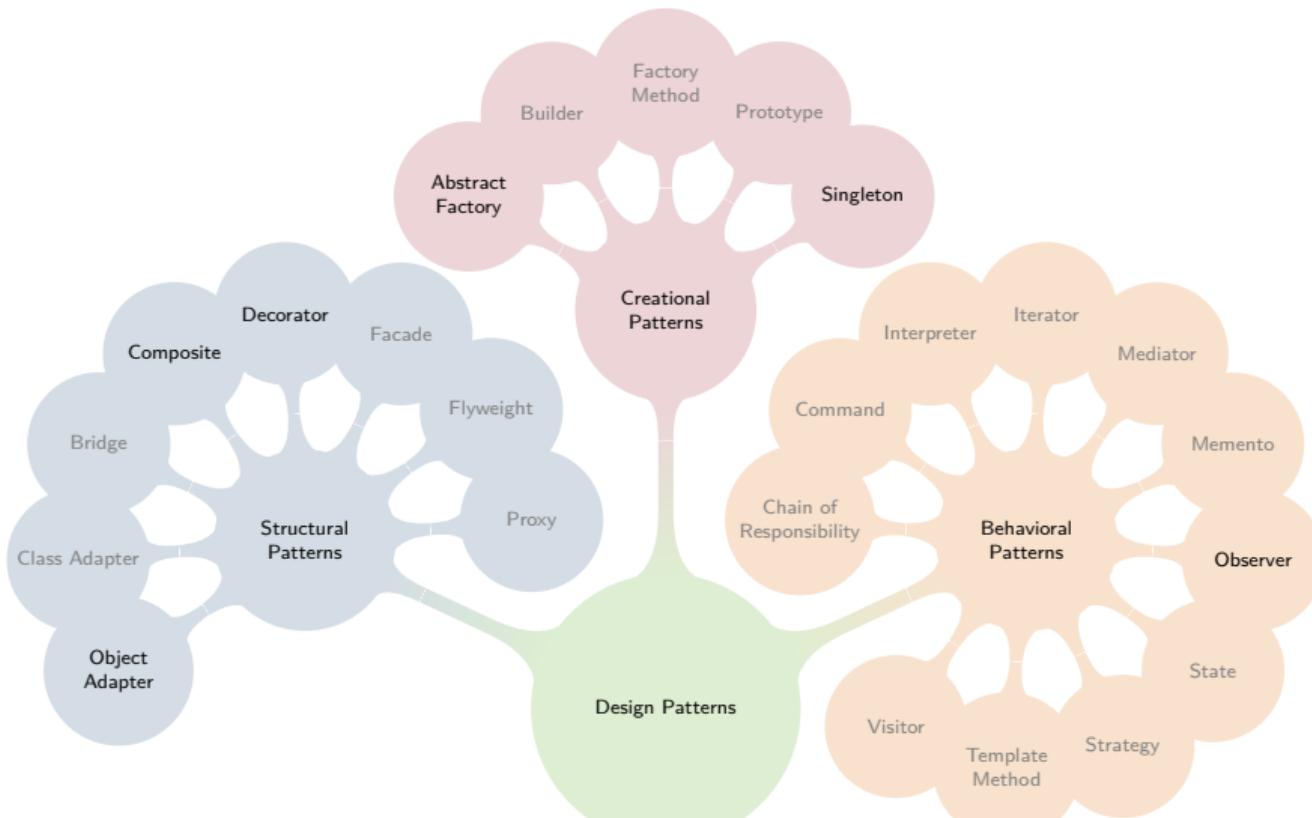


- preserve knowledge of software architects
- reuse of established architectures
- enable efficient communication

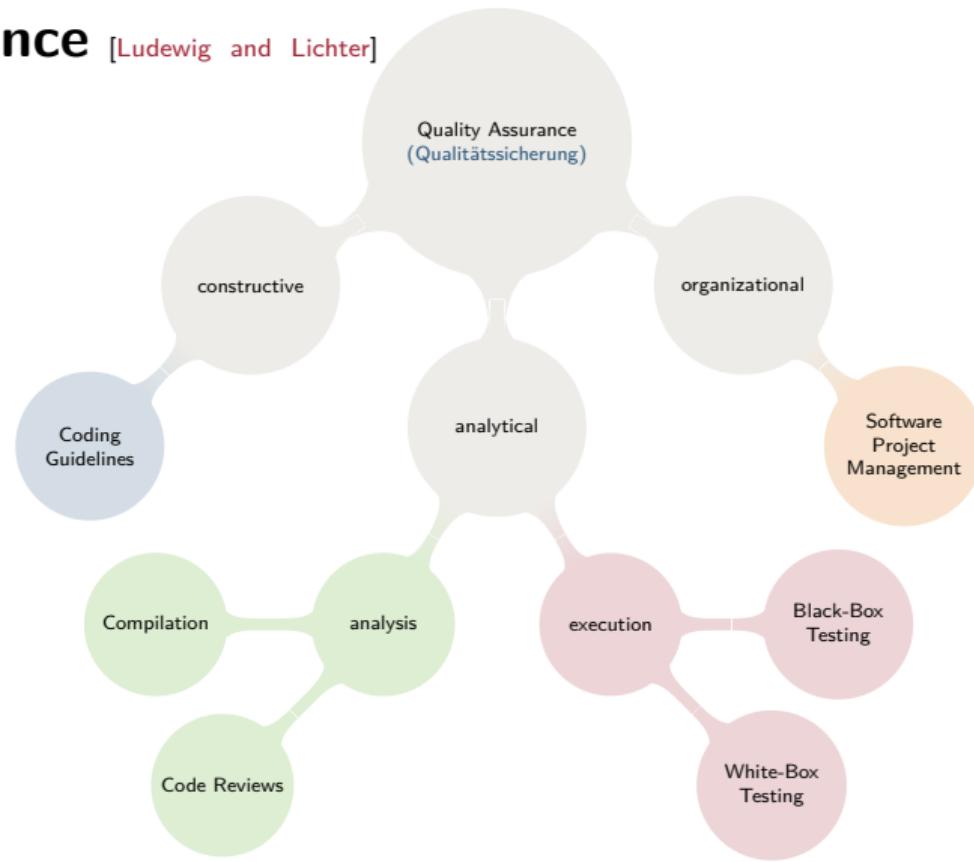
# Recap: Implementation



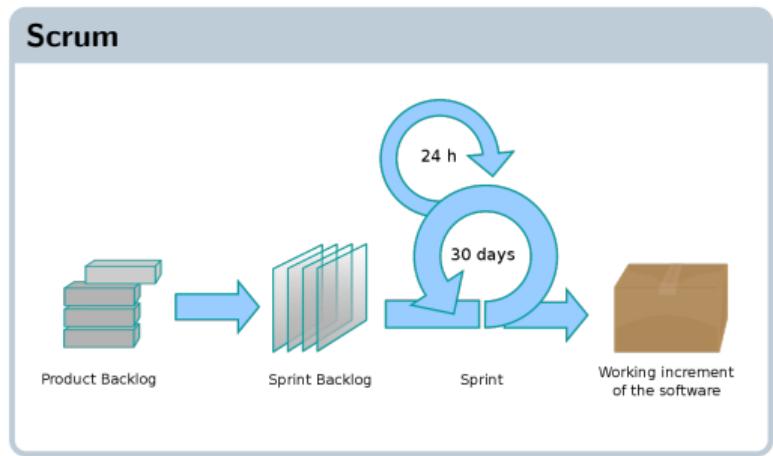
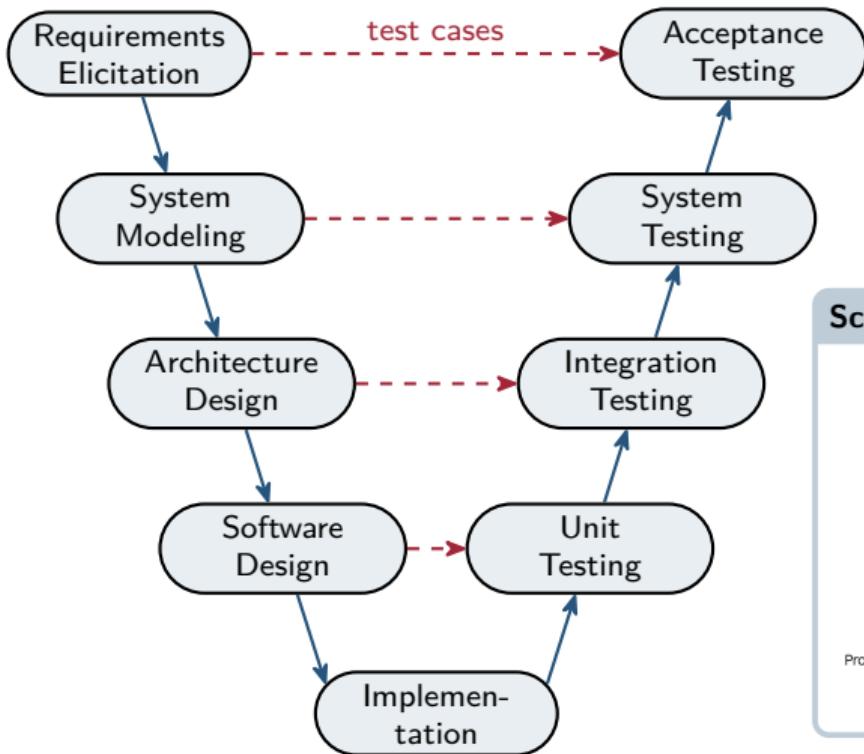
# Recap: Design Patterns [Gang of Four (GoF)]



# Recap: Quality Assurance [Ludewig and Lichter]

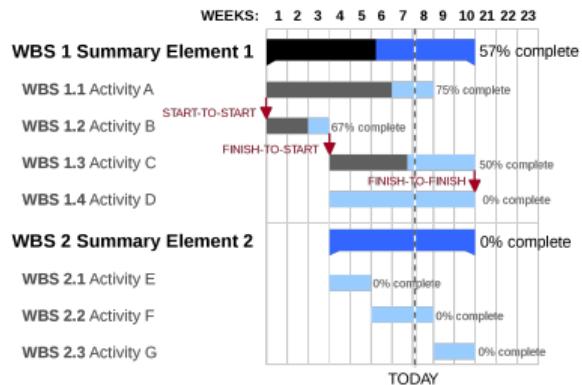


# Recap: Process Models



# Recap: Project Management

## Gantt Charts

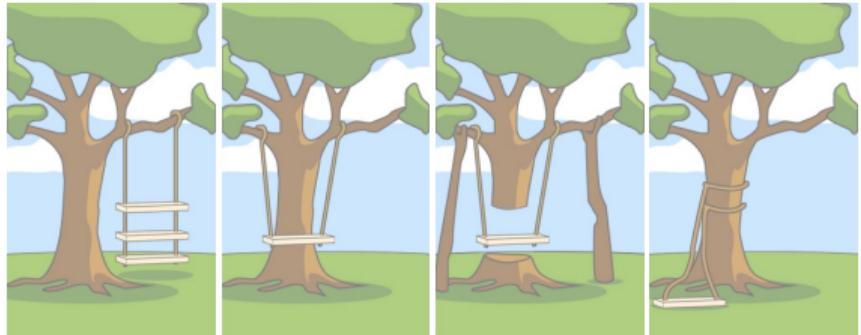


## Network Diagrams

0	3	3
Background		
0	0	3

earliest start	duration	earliest finish
Task		
latest start	buffer	latest finish

# Software Engineering I



requirements

modeling

architecture and  
design

implementation



testing

process

management  
and pricing

Software  
Engineering II

# Summary on Software Engineering I

## Lessons Learned

1. Software, its impacts, its engineering
2. Requirements, different kinds, its engineering, use case diagrams
3. System modeling, UML, activity/state machine diagrams
4. Software architecture, component diagrams, architectural patterns
5. Software design, class/sequence diagrams
6. Implementation, programming languages, coding conventions, tooling
7. Design patterns, structural (object adapter, composite, decorator), creational (singleton, abstract factory), behavioral (observer)
8. Quality assurance, compilation, code reviews, white-box/black-box testing
9. Process models, Waterfall, V-Model, Scrum
10. Project management, planning, scheduling

## Practice

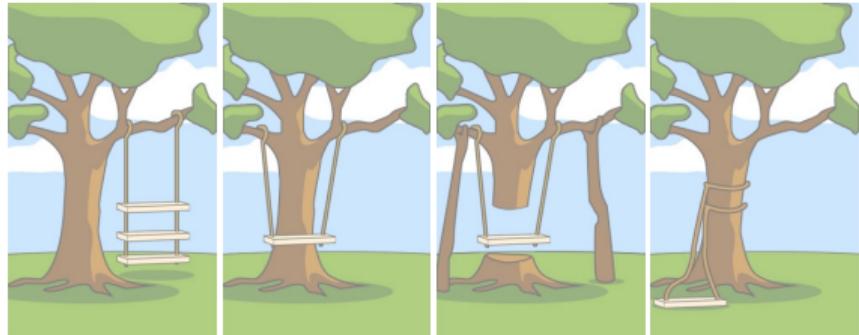
- See [Moodle](#)
- Answer the quiz in Moodle to track your learning progress



# Software Engineering

11. Software Evolution | Thomas Thüm | April 26, 2022

# Software Engineering I

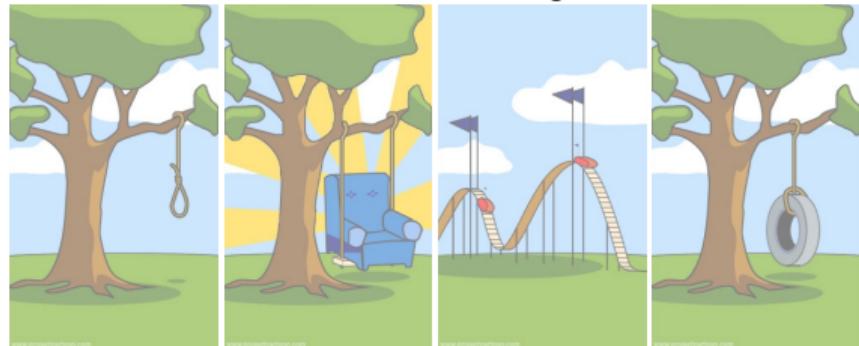


requirements

modeling

architecture and  
design

implementation



testing

process

management  
and pricing

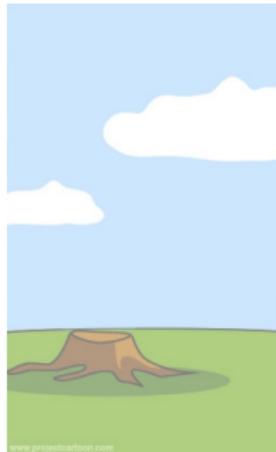
Software  
Engineering II

# Software Engineering II



how patches were applied

**software evolution**



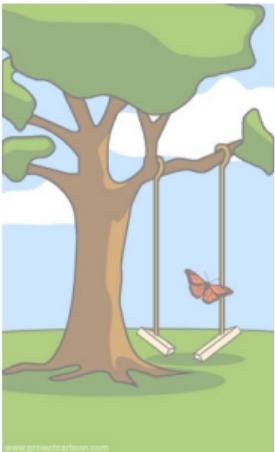
how it was supported

**software maintenance**



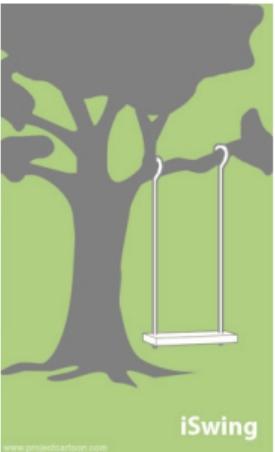
when it was delivered

**configuration management**



how it performed under load

**compilation and static analyses**



what marketing advertised

**software product lines**



the open source version

**open-source software**

# Lecture Overview

1. Software Evolution vs Software Engineering I
2. Programming Wisdom on Software Evolution
3. Smells and Refactorings

## **Software Evolution vs Software Engineering I**

# Change is Inevitable

## Sommerville:

"Change is inevitable in all large software projects. The **system requirements change** as businesses respond to external pressures, competition, and changed management priorities. As **new technologies become available**, new approaches to design and implementation become possible. Therefore whatever software process model is used, it is essential that it can **accommodate changes** to the software being developed. [...] It may then be necessary to **redesign** the system to deliver the new requirements, **change any programs** that have been developed, and **retest** the system."





**Edward V. Berard (1993)**

[[wikiquote.org](#)]

Recap: “Walking on water and developing software from a specification are easy if both are frozen.”

# Changes to Requirements

## Motivation

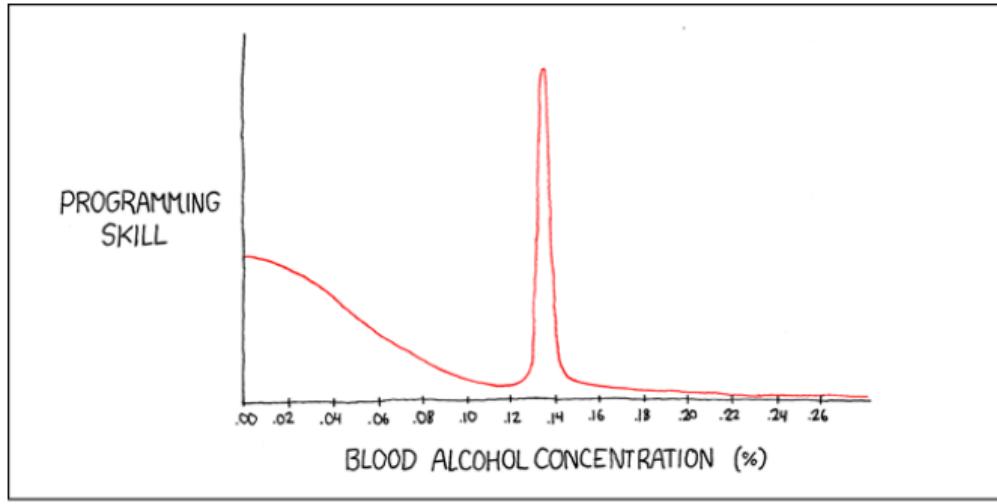
[Sommerville]

- changed problem understanding for all stakeholders
- new or updated hardware
- interfacing to other systems
- new legislation and regulations
- new compromise on conflicting requirements of different stakeholders

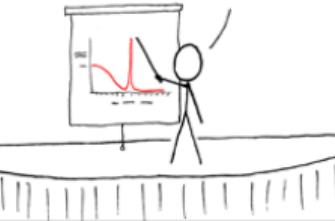
# Recap: Extending Thomas' Calculator

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

- Title Bar:** workspace-2019-09-SE1 - Calculator/src/de/tubs/se1/calculator/Calculator.java - Eclipse IDE
- Menu Bar:** File Edit Source Refactor Navigate Search Project Run Window Help
- Toolbar:** Standard Eclipse toolbar icons.
- Package Explorer:** Shows the project structure:
  - Calculator [trunk/Calculator]
  - src
    - de.tubs.se1.calculator
      - Add.java
      - Calculator.java
      - Expression.java
      - Subtract.java
      - Value.java
    - JRE System Library [openJDK]
    - JavaFx
    - requirements.txt 3 12/11/1
- Code Editor:** Displays the `Calculator.java` file content. The cursor is at line 48, where the `update` method is being edited. A tooltip for `update` shows the signature `void update()`. The code includes logic for addition and subtraction operations.
- JavaFX Application Window:** A modal dialog titled "SE1 Calculator" is displayed, showing a simple GUI with a text input field and two buttons labeled "+" and "-".
- Bottom Status Bar:** Writable, Smart Insert, 19 : 34 : 592



CALLED THE BALLMER PEAK, IT WAS DISCOVERED BY MICROSOFT IN THE LATE 80's. THE CAUSE IS UNKNOWN, BUT SOMEHOW A BAC BETWEEN 0.129% AND 0.138% CONFFERS SUPERHUMAN PROGRAMMING ABILITY.



HOWEVER, IT'S A DELICATE EFFECT REQUIRING CAREFUL CALIBRATION - YOU CAN'T JUST GIVE A TEAM OF CODERS A YEAR'S SUPPLY OF WHISKEY AND TELL THEM TO GET CRACKING.



...HAS THAT EVER HAPPENED?

REMEMBER  
WINDOWS ME?  
I KNEW IT!

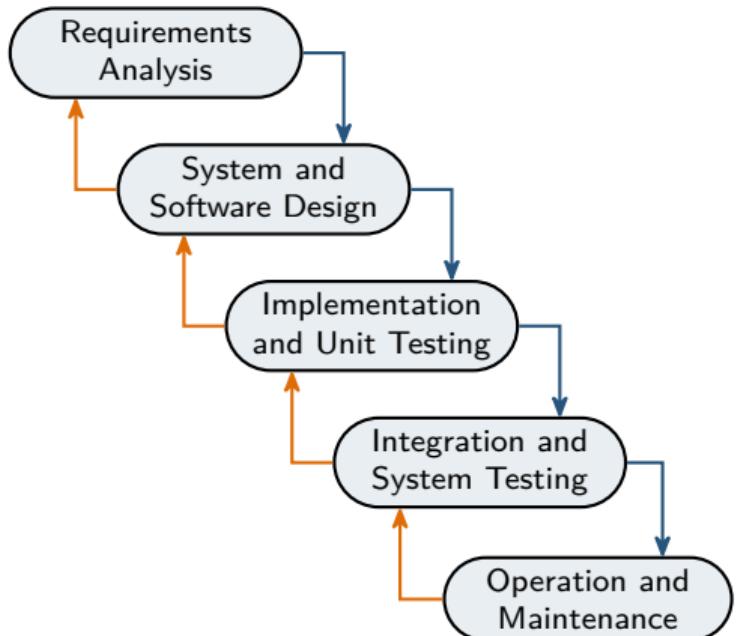


# Recap: Design Patterns [Gang of Four (GoF)]

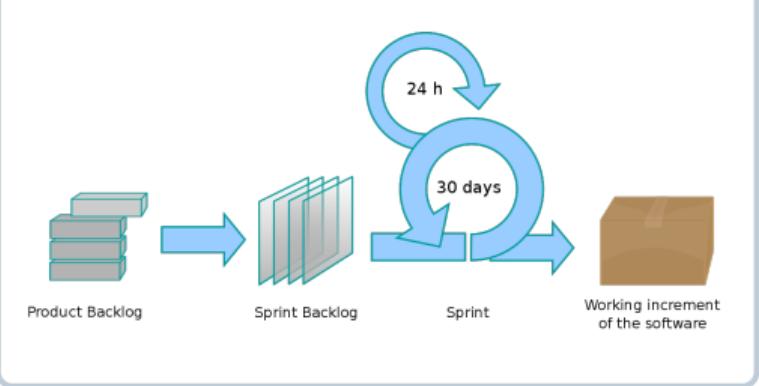


# Recap: Process Models

## Waterfall Model

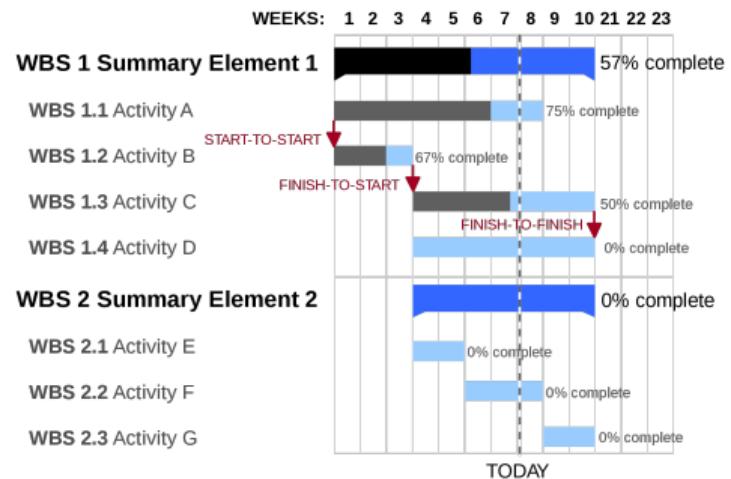


## Scrum



# Recap: Project Management

Gantt Charts



# Regression Testing

[Ludewig and Licher]

## Regression Testing (Regressionstesten)

"Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements."

## Goal

- every change has desired effects
- regression testing aims to detect undesired side-effects

## Assumptions

- minor change to the program behavior
- program was largely correct before
- results only differ in old and new version where desired

## Characteristics

- requires to document test results before changes
- actual results are not only compared to specification, but also to previous results for the same inputs

# Software Evolution vs Software Engineering I

## Lessons Learned

- Incremental development and changed requirements cause evolution
- Design patterns ease evolution
- Regression testing incorporates results of new **and old** version
- Further Reading: **Sommerville**, 2.3 Coping with Change and 4.6 Requirements Change
- Further Reading: **Ludewig and Licher**, 19.1.3 (**Der Regressionstest**)

## Practice

- Form groups of 2–3 students
- Think: what are further examples for **software evolution**?
- Pair: which of those examples are amendable to **regression testing**?
- Share: briefly describe one example of each kind in Moodle



# **Programming Wisdom on Software Evolution**

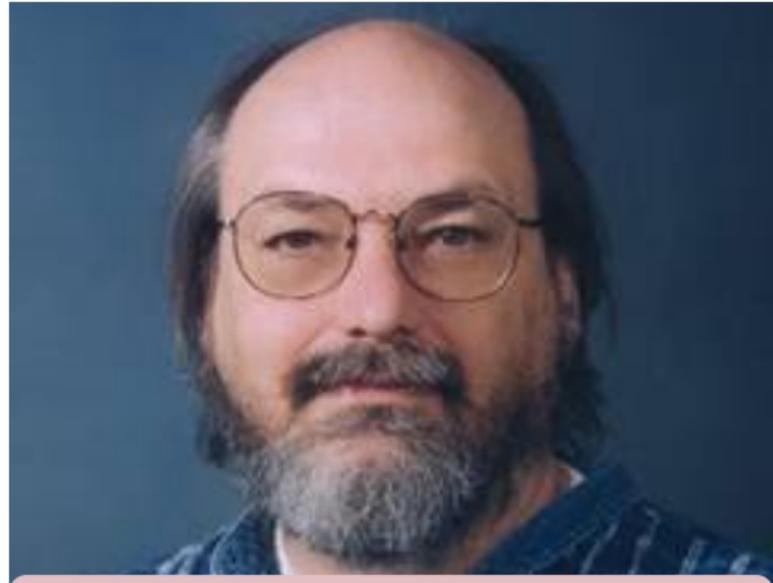
# Avoiding Complexity



**Richard E. Pattis**

[cmu.edu]

“When debugging, novices insert corrective code; experts remove defective code.”



**Ken Thompson (born 1943)**

[twitter.com]

“One of my most productive days was throwing away 1,000 lines of code.”

# ALGORITHMS BY COMPLEXITY

MORE COMPLEX →

LEFTPAD

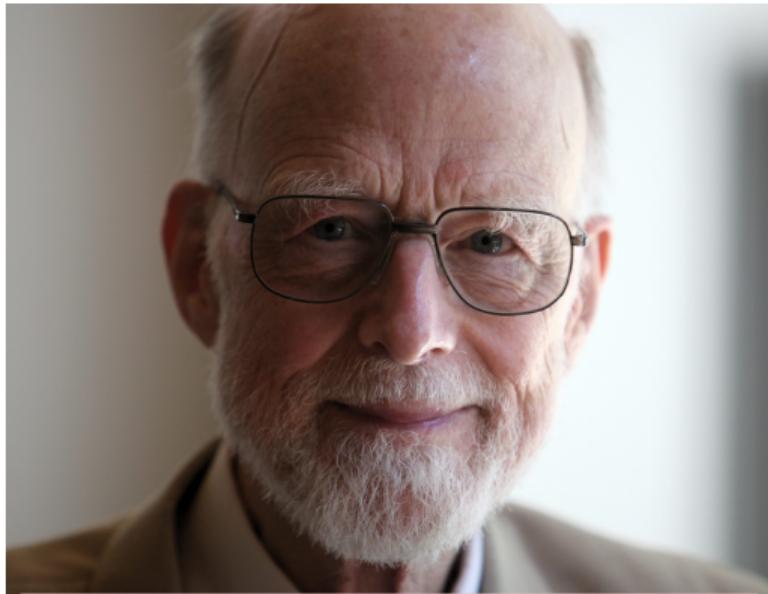
QUICKSORT  
MERGE

GIT  
SELF-  
DRIVING  
CAR

GOOGLE  
SEARCH  
BACKEND

SPRAWLING EXCEL SPREADSHEET  
BUILT UP OVER 20 YEARS BY A  
CHURCH GROUP IN NEBRASKA TO  
COORDINATE THEIR SCHEDULING

# Increasing Complexity



**Tony Hoare (born 1934)**

[[twitter.com](#)]

“Inside every large program, there is a small program trying to get out.”



**Meir “Manny” Lehman (1925–2010)**

[[twitter.com](#)]

“An evolving system increases its complexity unless work is done to reduce it.”

# Lehman's Laws of Software Evolution

[Lehman et al. 1997]

## Lehman's Laws (excerpt)

- Continuing Change: systems must be continually adapted to stay satisfactory
- Increasing Complexity: complexity increases during evolution unless work is done to maintain or reduce it
- Conservation of Familiarity: satisfactory evolution excludes excessive growth
- Continuing Growth: functionality must be continually increased to maintain user satisfaction
- Declining Quality: quality will decline unless rigorously maintained and adapted to operational environment changes

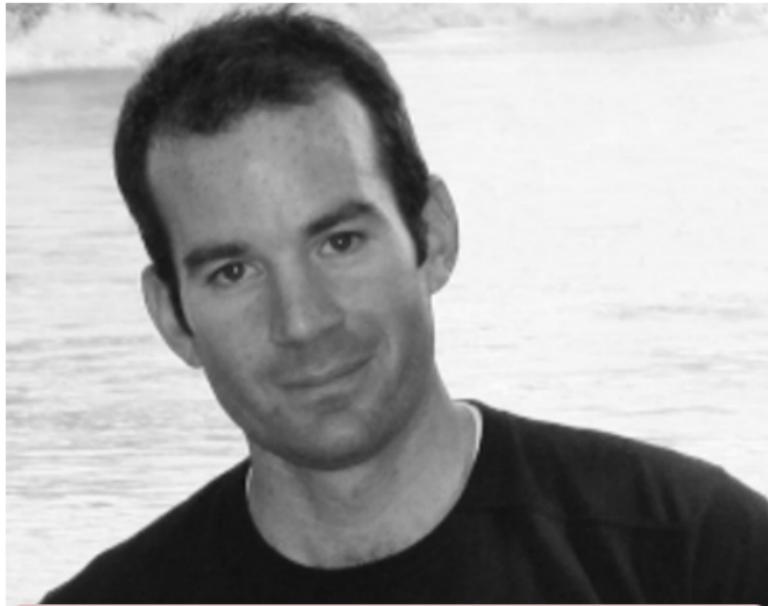
## Essence of the Laws

- software that is used will be modified
- when modified, its complexity will increase (unless one does actively work against it)

## Consequences

- functional changes are inevitable
- changes are not necessarily a consequence of errors (e.g., in requirements engineering or programming)
- there are limits to what a development team can achieve (cf. Continuing Growth)

# Simplicity over Performance



**Wes Dyer**

[[twitter.com](#)]

“Make it correct, make it clear, make it concise, make it fast. In that order.”



**Joshua J. Bloch (born 1961)**

[[twitter.com](#)]

“The cleaner and nicer the program, the faster it's going to run. And if it doesn't, it'll be easy to make it fast.”

# Programming Wisdom on Software Evolution

## Lessons Learned

- Sources for complexity: workarounds, unnecessary code, performance tweaks
- Lehman's laws: software is modified and gets more complex
- Further Reading: [Ludewig and Licher](#), 23.1 ([Software-Evolution](#))

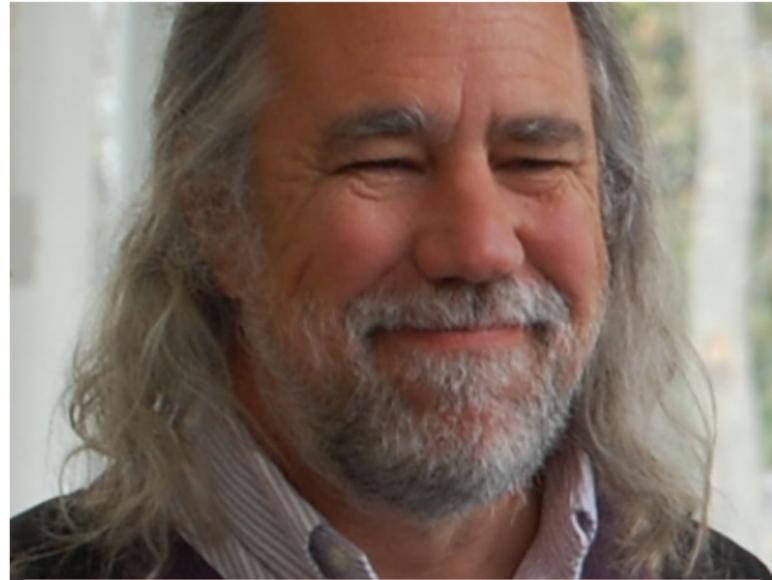
## Practice

- Form groups of 2–3 students
- Think: find a (counter) example for one of Lehman's laws
- Pair: discuss the example with your colleagues – does it fulfill further laws?
- Share: share one example and the according law in Moodle



## **Smells and Refactorings**

# Simple and Clean



**Grady Booch (born 1955)**

[[twitter.com](#)]

“The function of good software is to make the complex appear to be simple.”



**Robert C. Martin (Uncle Bob, born 1952)**

Boy Scouts Rule: “Leave the campground cleaner than the way you found it.”

[[oreilly.com](#)]

# Code Refactoring

[Sommerville]

## Motivation

- anticipating changes is typically infeasible
- anticipated changes may not materialize and unanticipated changes are required
- make changes easier by constantly refactoring the code

## Refactoring

“Refactoring means that the programming team looks for possible improvements to the software and implements them immediately. When team members see code that can be improved, they make these improvements even in situations where there is no immediate need for them.”

## Structure of the Software

“A fundamental problem of incremental development is that local changes tend to **degrade the software structure**. Consequently, further changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system.

**Refactoring improves the software structure and readability and so avoids the structural deterioration that naturally occurs when software is changed.”**

# Refactoring in Practice

## Theory vs Practice

[Sommerville]

"In principle, when refactoring is part of the development process, the software should always be easy to understand and change as new requirements are proposed. In practice, this is not always the case. Sometimes **development pressure means that refactoring is delayed** because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and **require that the architecture of the system be modified.**"

## Grandma Beck's Child-Rearing Philosophy

"If it stinks, change it."

[oreilly.com]

# Smells

[Fowler's Refactoring]

## Mysterious Name

"Puzzling over some text to understand what's going on is a great thing if you're reading a detective novel, but not when you're reading code. [...] One of the most important parts of clear code is good names, so we put a lot of thought into naming functions, modules, variables, classes, so they clearly communicate what they do and how to use them."

— Rename Method/Field/Variable Refactoring

## Duplicated Code

"If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them. Duplication means that every time you read these copies, you need to read them carefully to see if there's any difference. If you need to change the duplicated code, you have to find and catch each duplication."

— Extract/Pull-Up Method Refactoring

# Smells

[Fowler's Refactoring]

## Long Method

"Since the early days of programming, people have realized that the longer a function is, the more difficult it is to understand. Older languages carried an overhead in subroutine calls, which deterred people from small functions. [...] [T]he real key to making it easy to understand small functions is good naming. If you have a good name for a function, you mostly don't need to look at its body. [...] A heuristic we follow is that whenever we feel the need to comment something, we write a function instead."

— Extract Method Refactoring



Robert C. Martin (Uncle Bob, born 1952)

"Functions should do one thing. They should do it well. They should do it only."

[\[oreilly.com\]](http://oreilly.com)

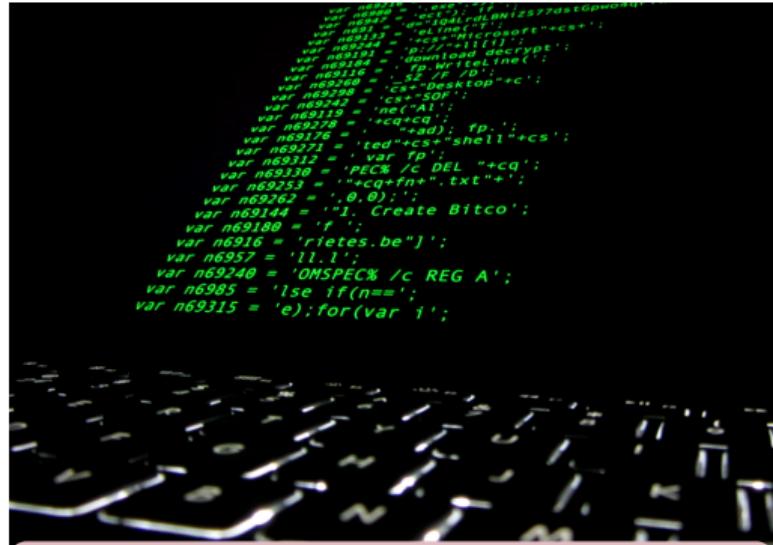
# Writing Requires Reading



**Robert C. Martin (Uncle Bob, born 1952)**

“So if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, make it easy to read.”

[\[oreilly.com\]](http://oreilly.com)



**Eagleson's Law**

[\[nus.edu.sg\]](http://nus.edu.sg)

“Any code of your own that you haven't looked at for six or more months might as well have been written by someone else.”

I COULD RESTRUCTURE  
THE PROGRAM'S FLOW

OR USE ONE LITTLE  
'GOTO' INSTEAD.



EH, SCREW GOOD PRACTICE.  
HOW BAD CAN IT BE?

goto main\_sub3;

N

\*COMPILE\*



# Spaghetti or Lasagne?



**Edsger W. Dijkstra (1968)**

[acm.org]

"Go-To Statement Considered Harmful"  
(today known as spaghetti code)



**Roberto Waltman**

[twitter.com]

"In the one and only true way. The object-oriented version of 'Spaghetti code' is, of course, 'Lasagna code'. (Too many layers)."

# Smells and Refactorings

## Lessons Learned

- Smells (anti patterns): structures to avoid
- Examples: mysterious name, duplicated code, long method
- Refactorings: clean-up of structures before and after every change
- Examples: rename X refactoring, extract/pull-up method refactoring
- Further Reading: [Sommerville](#), 3.2.2 Refactoring
- Further Reading: [Fowler's Refactoring](#)
- Further Reading: [Uncle Bob's Clean Code](#)

## Practice

- Research another smell and report it (incl. source) to Moodle
- Find a suitable refactoring for an anti-pattern of your colleagues (answer to their post)
- Will be discussed in next lecture
- Deadline: May 2 at 12 noon





# Software Engineering

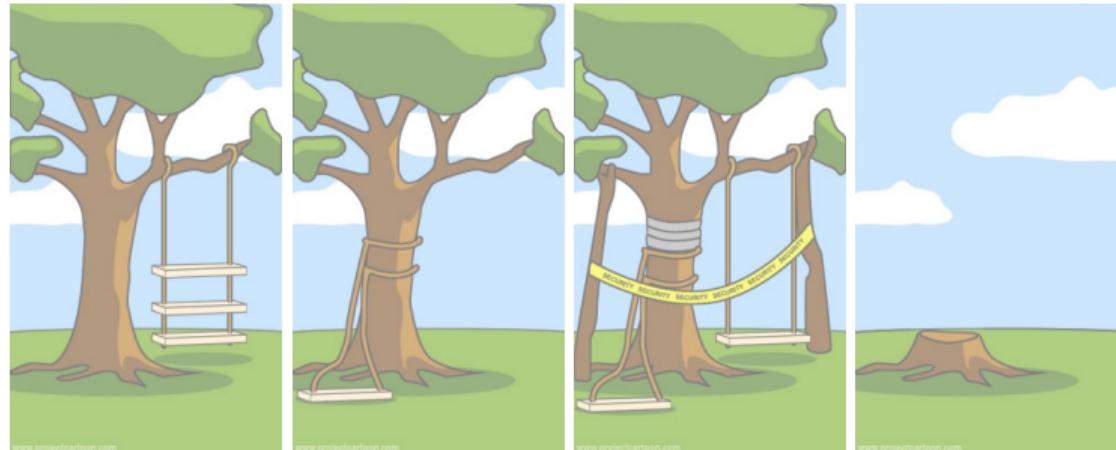
11. Software Evolution | Thomas Thüm | April 26, 2022



# Software Engineering

12. Software Maintenance | Thomas Thüm | May 3, 2022

# Software Maintenance (Software-Wartung)



how the customer  
explained it

how the  
programmer  
implemented it

how patches were  
applied

how it was  
supported

software  
evolution

software  
maintenance

## Lessons Learned

- Smells (anti patterns): structures to avoid
- Examples: mysterious name, duplicated code, long method
- Refactorings: clean-up of structures before and after every change
- Examples: rename X refactoring, extract/pull-up method refactoring
- Further Reading: [Sommerville](#), 3.2.2 Refactoring
- Further Reading: [Fowler's Refactoring](#)
- Further Reading: [Uncle Bob's Clean Code](#)

## Practice

- Research another smell and report it (incl. source) to Moodle
- Find a suitable refactoring for an anti-pattern of your colleagues (answer to their post)
- Will be discussed in next lecture
- Deadline: May 2 at 12 noon



# Lecture Overview

1. Software Maintenance in Action
2. Introduction to Software Maintenance
3. Migration of Legacy Software

# Lecture Contents

## 1. Software Maintenance in Action

New Year's Eve: Party Time?!?

New Year's Eve for Maintainers

Maintenance at Work

Wisdom on Debugging

The One-Character-Fix

Leap Seconds

Wisdom on Maintenance

Lessons Learned

## 2. Introduction to Software Maintenance

## 3. Migration of Legacy Software

# New Year's Eve: Party Time?!?



London Eye on January 1st, 2008 2017

# New Year's Eve for Maintainers

## Meanwhile in the Internet



**Oops! Website currently not available.**

*Nice of you to come by, but currently this web page is feeling a bit under the weather.*

*Why not check back later?*

## Your DNS Zone File

Export your DNS zone file - Append a zone file

Type	Name	Value	TTL	Active
A	theburritobot.com	points to 205.178.189.129	Automatic	<input checked="" type="checkbox"/>
A	direct	points to 205.178.189.129	Automatic	<input checked="" type="checkbox"/>
A	*	points to 205.178.189.129	Automatic	<input checked="" type="checkbox"/>
CNAME	www	is an alias of theburritobot.herokuapp.com	Automatic	<input checked="" type="checkbox"/>
MX	theburritobot.com	mail handled by theburritobot.com with priority 0	Automatic	<input checked="" type="checkbox"/>

## Meanwhile in San Francisco

[cloudflare.com]

**"At midnight UTC on New Year's Day, deep inside Cloudflare's custom RRDNS software, a number went negative when it should always have been, at worst, zero. [...] At peak approximately 0.2% of DNS queries to Cloudflare were affected and less than 1% of all HTTP requests to Cloudflare encountered an error."**

# Maintenance at Work

[cloudflare.com]

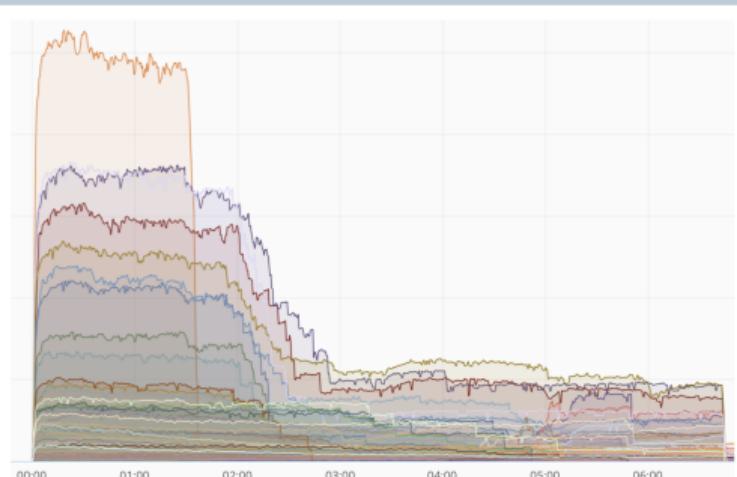
## Meanwhile in San Francisco

"This problem was quickly identified. The most affected machines were **patched in 90 minutes** and the fix was **rolled out worldwide by 06:45 UTC**. We are sorry that our customers were affected, but we thought it was worth writing up the root cause for others to understand."

## The Timeline

- 2017-01-01 00:00 UTC Impact starts
- 2017-01-01 00:10 UTC Escalated to engineers
- 2017-01-01 00:34 UTC Issue confirmed
- 2017-01-01 00:55 UTC Mitigation deployed to one canary node and confirmed
- 2017-01-01 01:03 UTC Mitigation deployed to canary data center and confirmed
- 2017-01-01 01:23 UTC Fix deployed in most impacted data center
- 2017-01-01 01:45 UTC Fix being deployed to major data centers
- 2017-01-01 01:48 UTC Fix being deployed everywhere
- 2017-01-01 02:50 UTC Fix rolled out to most of the affected data centers
- 2017-01-01 06:45 UTC Impact ends

## Monitoring



## Fun Fact

San Francisco's standard time zone is UTC-8

# Wisdom on Debugging



Gordon Glegg

[[twitter.com](#)]

"Sometimes the problem is to discover what the problem is."



Jessica Joy Kerr (2020)

[[jessitron.com](#)]

"In programming, it's dangerous to work near your working memory threshold. You get more mistakes and more complicated code."

# The One-Character-Fix

[cloudflare.com]

## The Patch

```
121 121      rttMax := servers[server_count-1].GetRTT()  
122 -       if rttMax == 0 {  
122 +     if rttMax <= 0 {  
123 123         rttMax = DefaultTimeout  
124 124     }
```

## A Falsehood Programmers Believe About Time

“The root cause of the bug that affected our DNS service was the belief that **time cannot go backwards**. In our case, some code assumed that the difference between two times would always be, at worst, zero.

RRDNS is written in Go and uses Go’s `time.Now()` function to get the time. Unfortunately, this function does not guarantee monotonicity.”

# Leap Seconds (Schaltsekunden)

23:59:60

## Leap Second 37

[spinellis.gr]

	Linux NTP	FreeBSD NTP	Linux NTP Google	Linux TAI
SI	64.677	64.648	64.641	64.652
Unix	1483228803.685	1483228804.649	1483228803.663	1483228830.882
Human	2017-01-01 00:00:03	2017-01-01 00:00:04	2017-01-01 00:00:03	2017-01-01 00:00:03
SI				
58	1483228798	1483228798	1483228798	1483228824
	2016-12-31 23:59:58	2016-12-31 23:59:58	2016-12-31 23:59:58	2016-12-31 23:59:58
59	1483228799	1483228799	1483228799	1483228825
	2016-12-31 23:59:59	2016-12-31 23:59:59	2016-12-31 23:59:59	2016-12-31 23:59:59
60	1483228799	1483228800	1483228799	1483228826
	2016-12-31 23:59:59	2017-01-01 00:00:00	2016-12-31 23:59:59	2016-12-31 23:59:60
61	1483228800	1483228801	1483228800	1483228827
	2017-01-01 00:00:00	2017-01-01 00:00:01	2017-01-01 00:00:00	2017-01-01 00:00:00
62	1483228801	1483228802	1483228801	1483228828
	2017-01-01 00:00:01	2017-01-01 00:00:02	2017-01-01 00:00:01	2017-01-01 00:00:01
63	1483228802	1483228803	1483228802	1483228829
	2017-01-01 00:00:02	2017-01-01 00:00:03	2017-01-01 00:00:02	2017-01-01 00:00:02

## 27 Leap Seconds since 1972

[wikipedia.org]

Year	Jun 30	Dec 31	
1991	0	0	
1992	+1	0	
1993	+1	0	
1994	+1	0	
1995	0	+1	
1996	0	0	
1997	+1	0	
1998	0	+1	
1999	0	0	
2000	0	0	
2001	+1	0	
2002	+1	0	
2003	0	0	
2004	0	0	
2005	+1	0	
2006	0	0	
2007	0	+1	
2008	0	0	
2009	+1	0	
2010	0	0	
2011	0	0	
2012	+1	0	
2013	0	0	
2014	0	0	
2015	+1	0	
2016	0	+1	
2017	0	0	
2018	0	0	
2019	0	0	
2020	0	0	
2021	0	TBA	
Year	Jun 30	Dec 31	
Total	11	16	
		27	
		Current TAI – UTC	
		37	

WHY DO THE CLOCKS SAY IT'S 3AM?

ADDING AN EXTRA DAY CREATES  
TOO MANY GLITCHES. INSTEAD,  
WE'RE JUST RUNNING OUR CLOCKS  
3.4% SLOWER DURING FEBRUARY  
TO AVOID THE IRREGULARITY.



THIS YEAR, GOOGLE HAS EXPANDED  
THEIR LEAP SECOND "SMEARING" TO  
COVER LEAP DAYS AS WELL.

# Wisdom on Maintenance



**Karolina Szczur**

[[twitter.com](#)]

“Writing software as if we are the only person that ever has to comprehend it is one of the biggest mistakes and false assumptions that can be made.”



**John F. Woods**

[[google.com](#)]

“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”

# Software Maintenance in Action

## Lessons Learned

- Maintenance problems caused by leap seconds
- Solutions: consider negative intervals, leap second smearing
- Time pressure for maintenance
- Develop software for maintenance
- Further Reading: see references above

## Practice

- Form groups of 2–3 students
- Think (2 min): which software maintenance has affected you? how long did it to fix it?
- Pair (5 min): discuss the example with your colleagues
- Share (3 min): share one example in Moodle and vote for other examples



# Lecture Contents

1. Software Maintenance in Action
2. Introduction to Software Maintenance
  - Software Maintenance
  - Kinds of Maintenance
  - Maintenance and Evolution
  - Reengineering Tasks
  - Wisdom on Reengineering
  - Lessons Learned
3. Migration of Legacy Software

# Software Maintenance

[Ludewig and Licher]

## Motivation

- for software: no compensation of deterioration, repair, spare parts
- corrections (especially shortly after first delivery)
- modification and reconstruction

## Operation and Maintenance Phase

[IEEE Std 610.12]

"The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements."

## Maintenance

[IEEE Std 610.12]

"The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment."

# Kinds of Maintenance

[Ludewig and Licher]

## Adaptive Maintenance

[Lientz and Swanson 1980]

“Software maintenance performed to make a computer program usable in a changed environment.”

desktop application for a new version of an operating system (e.g., from Windows 8.1 to 10)

## Corrective Maintenance

[Lientz and Swanson 1980]

“Maintenance performed to correct faults in software.”

Windows calculator showing wrong formulas

## Perfective Maintenance

[Lientz and Swanson 1980]

“Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.”

better handling of very large files in a text editor

## Preventive Maintenance

[Lientz and Swanson 1980]

“Maintenance performed for the purpose of preventing problems before they occur.”

2,000 year problem, leap seconds/years

LATEST: 10.17

UPDATE

## CHANGES IN VERSION 10.17: THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.

### COMMENTS:

LONGTIMEUSER4 WRITES:

THIS UPDATE BROKE MY WORKFLOW!  
MY CONTROL KEY IS HARD TO REACH,  
SO I HOLD SPACEBAR INSTEAD, AND I  
CONFIGURED EMACS TO INTERPRET A  
RAPID TEMPERATURE RISE AS "CONTROL".

ADMIN WRITES:

THAT'S HORRIFYING.

LONGTIMEUSER4 WRITES:

LOOK, MY SETUP WORKS FOR ME.  
JUST ADD AN OPTION TO REENABLE  
SPACEBAR HEATING.

EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

# Maintenance and Evolution

[Ludewig and Licher]

## Maintenance

- mostly corrections
- small changes
- often unforeseen changes
- results in patches and hot fixes (minor updates)
- minor release, new minor version: 2.3.0 ⇒ 2.3.1

## Evolution

- new or removed functionality
- large changes
- often foreseen changes
- results in upgrades or service packs (major updates, cumulative updates)
- major release, new major version: 2.3.2 ⇒ 2.4.0

# Reengineering Tasks

[Ludewig and Licher]

## Reverse Engineering

[Chikofsky und Cross]

“Reverse engineering is the process of analyzing a system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.”

updating UML diagrams from source code

## Forward Engineering

[Chikofsky und Cross]

“Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.”

see Software Engineering I

## Restructuring

[Chikofsky und Cross]

“Restructuring is a transformation from one form of representation to another at the same relative level of abstraction. The new representation is meant to preserve the semantics and external behavior of the original.”

refactorings as introduced in lecture on evolution

## Reengineering

[Chikofsky und Cross]

“Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.”

combination of reverse engineering, refactoring, and forward engineering

# Wisdom on Reengineering



**Kyle Simpson**

[[twitter.com](#)]

“There’s nothing more permanent than a temporary hack.” (Sprichwort: Nichts ist so beständig wie das Provisorium.)



**Jeff Sickel**

[[twitter.com](#)]

“Deleted code is debugged code.”

# Introduction to Software Maintenance

## Lessons Learned

- Maintenance (Phase)
- Kinds: adaptive, corrective, perfective, preventive
- Maintenance vs evolution
- Reengineering = reverse engineering + restructuring + forward engineering
- Further Reading: [Ludewig and Lichter](#), Chapter 22 ([Software-Wartung](#)) and 23 ([Reengineering](#))

## Practice

- Quiz'n'Disquiz
- Quiz: fill out the Moodle quiz on your own
- Disquiz: compare and discuss the results with your colleagues



# Lecture Contents

1. Software Maintenance in Action
2. Introduction to Software Maintenance
3. Migration of Legacy Software
  - Legacy Software
  - Flash Player
  - Migration
  - Lessons Learned

# Legacy Software (Altsysteme)

## Legacy Software

[Ludewig and Licher]

“A large software system that we don't know how to cope with but that is vital to our organization.”

## Winamp 5.623 on 4k Display in Windows 10



## Typical Properties

[Ludewig and Licher]

- larger than 100k LOC
- older than 10 years
- original developers and architects not available anymore
- outdated programming languages and development concepts
- business-critical
- outdated or missing documentation
- based on outdated hardware and system software
- subject to numerous iterations of corrective and adaptive maintenance
- high costs for maintenance

LOADING... PLEASE  
INSERT DISK INTO DRIVE A:  
↳ \*CLICK\* THERE YOU GO.  
THANK YOU. WOW, THIS  
DISK IS INCREDIBLY FAST!  
↳ YEAH, UH, IT'S THE NEW  
MODEL FROM MEMOREX.  
AMAZING. AND HOW IS  
PRESIDENT REAGAN?  
↳ HE'S...HE'S FINE.



I FEEL WEIRD USING OLD  
SOFTWARE THAT DOESN'T  
KNOW IT'S BEING EMULATED.

# Flash Player

End of Life: December 31, 2020



The screenshot shows the official Adobe Flash Player End of Life page. At the top, there's a navigation bar with a menu icon, the Adobe logo, and a "Sign In" link. Below the navigation is a search bar with a magnifying glass icon and the text "ADOLE FLASH PLAYER". The main content area features a large heading "Adobe Flash Player" and "EOL General Information Page". A paragraph explains that Adobe no longer supports Flash Player after December 31, 2020, and blocked Flash content from running in Flash Player beginning January 12, 2021. It advises users to immediately uninstall Flash Player to help protect their systems. Another paragraph notes that some users may continue to see reminders from Adobe to uninstall Flash Player.

Online Game Fuchstreff Discontinued



The screenshot shows the homepage of the online game Fuchstreff. The header includes the Fuchstreff logo, a user profile icon, and links for "wleefuchs", "SPIELEN!", "COMMUNITY", "LERNEN", "LIGA", "FUCHSSCHAU", "FORUM", "KALENDER", and "HILFE". A prominent yellow banner in the center states: "Leider findet der Fuchstreff ohne den Flash Player ein Ende." and "Wir möchten uns bei Euch allen für die schönen Jahre bedanken und wünschen Euch altes Gute." Below the banner, there's a link to "Der Fuchstreff-Blog • RSS-Feed". A post by Sarah dated December 17, 2020, is shown, titled "Danke für alles", with 214 comments. The post text expresses gratitude to the community for the good years and expresses sadness over the discontinuation of the game due to the lack of Flash support.

# Migration

[Ludewig and Licher]

## Software Migration

- opposed to data and hardware migration
- renewing or replacing legacy software
- new software needs to be downward compatible
- new software meets current functional and non-functional requirements
- data often needs to be migrated
- outage as short as possible

## Migration Strategies

- wrapping: building a new software around a stable version of the legacy software
- redevelopment: functionality implemented again and replaces legacy software
- incremental migration: stepwise renewal and replacement
- big bang migration: whole legacy software is migrated in a single step
- combinations feasible (except for incremental and big bang)

# Migration of Legacy Software

## Lessons Learned

- Legacy software: typical properties, examples
- Software migration
- Migration strategies: wrapping, redevelopment, incremental migration, big bang migration
- Further Reading: [Ludewig and Lichter](#), Chapter 23 ([Reengineering](#))

## Practice

- 1. Post an example of software in Moodle that reached its end of life
- 2. Interpret the reasons for an example by a colleague
- Will be discussed in next lecture
- Deadline: May 9 at 12 noon

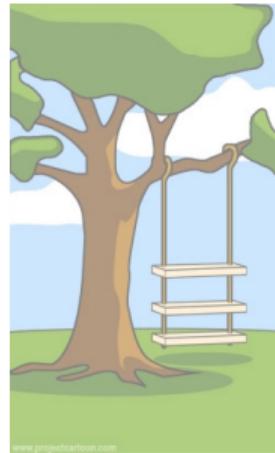




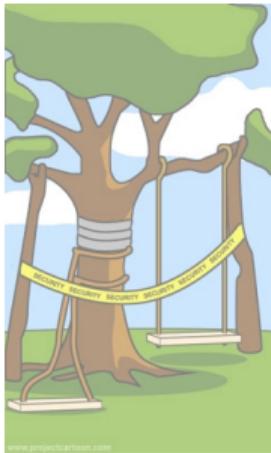
# Software Engineering

13. Configuration Management | Thomas Thüm | May 10, 2022

# Configuration Management



how the customer  
explained it



how patches were  
applied



how it was  
supported



when it was  
delivered

**software  
evolution**

**software  
maintenance**

**configuration  
management**

## Lessons Learned

- Legacy software: typical properties, examples
- Software migration
- Migration strategies: wrapping, redevelopment, incremental migration, big bang migration
- Further Reading: [Ludewig and Lichter](#), Chapter 23 ([Reengineering](#))

## Practice

- 1. Post an example of software in Moodle that reached its end of life
- 2. Interpret the reasons for an example by a colleague
- Will be discussed in next lecture
- Deadline: May 9 at 12 noon



# Lecture Overview

1. Configuration Management and Version Control
2. Operations of Version Control Systems
3. Continuous Integration and Deployment

# Lecture Contents

## 1. Configuration Management and Version Control

Which Products are Affected?

Configuration Management

Version Control

Git vs Mercurial

Centralized Version Control

Distributed Version Control

Lessons Learned

## 2. Operations of Version Control Systems

## 3. Continuous Integration and Deployment

# Which Products are Affected?



# Configuration Management

[Sommerville]

## Configuration Management

"**Configuration management** is concerned with the policies, processes, and tools for managing changing software systems."

## Four Activities in Configuration Management

- version control / management: keeping track of multiple versions, enabling simultaneous changes
- system building: collecting, compiling, and linking components into executable systems
- change management: tracking change requests and planning if/when realized
- release management: preparing new and managing old releases

## Development Stages (Entwicklungsphasen)

- **development phase**: adding new functionality
- **system testing phase**: internal release, bug fixes, performance improvements, security fixes, but no new functionality
- **release phase**: bug reports and feature requests by users or customers
- often several versions co-exist at different stages

# Version Control

## Goals of Version Control

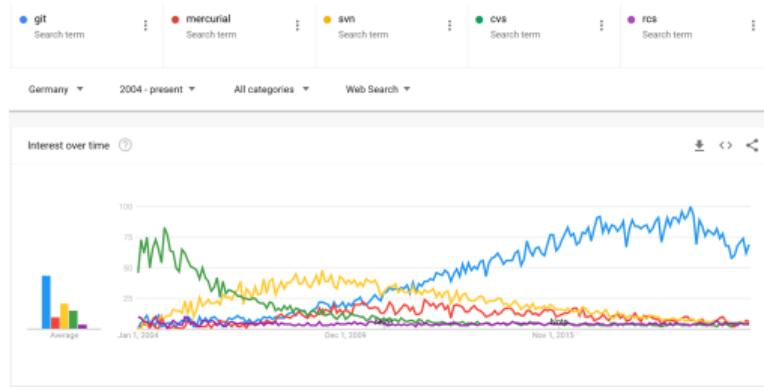
- collaborative work: synchronize files and folders with other users
- compare own version with other versions
- merge files edited by several users
- history: access old versions, log changes

## Version Control Systems

- local only: SCCS (1972), RCS (1982), ...
- centralized: CVS (1986), SVN (2000), ...
- distributed: [git](#) (2005), mercurial (2005), ...

invented for software, useful for most files

Alternatives? Why not use locks (cf. databases)?



## Personal Experience

- 2004: started working with CVS in industry
- 2007: initiated research prototype FeatureIDE with SVN
- 2009: published FeatureIDE open source (history neglected)
- 2014: migrated FeatureIDE's code to git

# Git vs Mercurial

## Sunsetting Mercurial support in Bitbucket

April 21, 2020 | 3 min read



Denise Chan

[Update Aug 26, 2020] All hg repos have now been disabled and cannot be accessed.

[Update July 1, 2020] Today, mercurial repositories, snippets, and wikis will turn to read-only mode. After July 8th, 2020 they will no longer be accessible.

The version control software market has evolved a lot since Bitbucket began in 2008. When we launched, centralized version control was the norm and we only supported Mercurial repos.

But Git adoption has grown over the years to become the default system, helping teams of all sizes work faster as they become more distributed.

As we surpass [10 million registered users](#) on the platform, we're at a point in our growth where we are conducting a deeper evaluation of the market and how we can best support our users going forward.

After much consideration, we've decided to remove Mercurial support from Bitbucket Cloud and its API. **Mercurial features and repositories will be officially deprecated on July 1, 2020.**

Read on to learn more about this decision, the important timelines, and get migration resources and support.

### The timeline and how this may affect your team

Here are the key dates as we sunset Mercurial functionality:

- **February 1, 2020:** users will no longer be able to create [new](#) Mercurial repositories
- **[Extended] July 1, 2020:** users will not be able to use Mercurial features. All hg repos, wikis, and snippets will be in read-only mode.

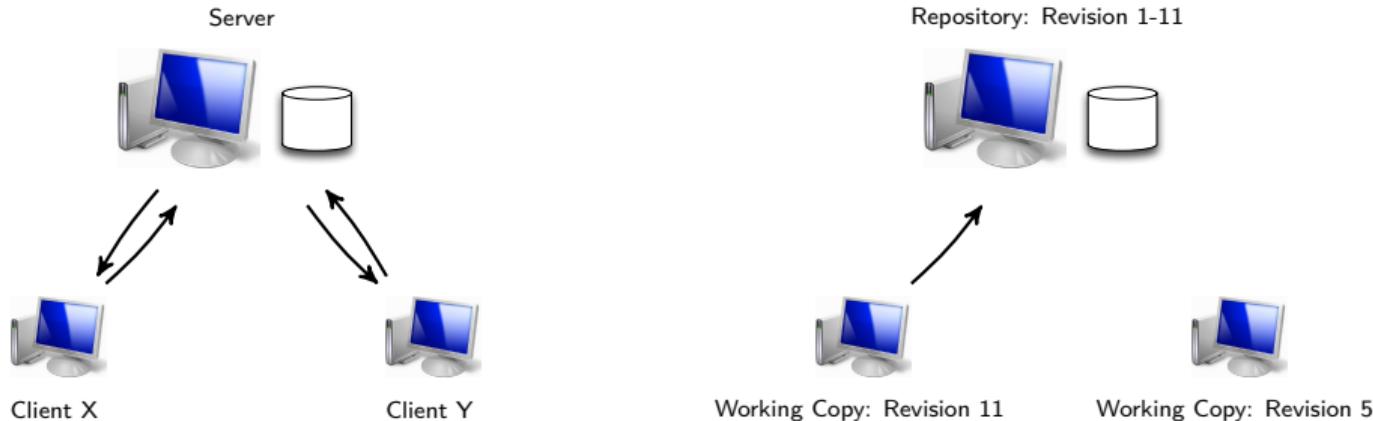
THIS IS GIT. IT TRACKS COLLABORATIVE WORK  
ON PROJECTS THROUGH A BEAUTIFUL  
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZIZE THESE SHELL  
COMMANDS AND TYPE THEM TO SYNC UP.  
IF YOU GET ERRORS, SAVE YOUR WORK  
ELSEWHERE, DELETE THE PROJECT,  
AND DOWNLOAD A FRESH COPY.



# Centralized Version Control



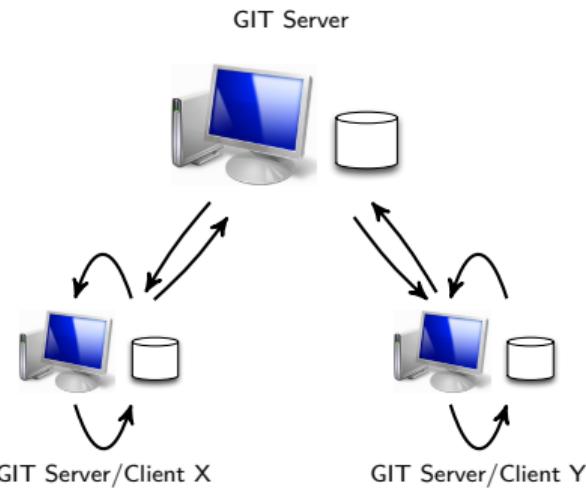
## Centralized Version Control Systems

- client-server architecture
- server manages the main copy
- clients use server to synchronize files/folders

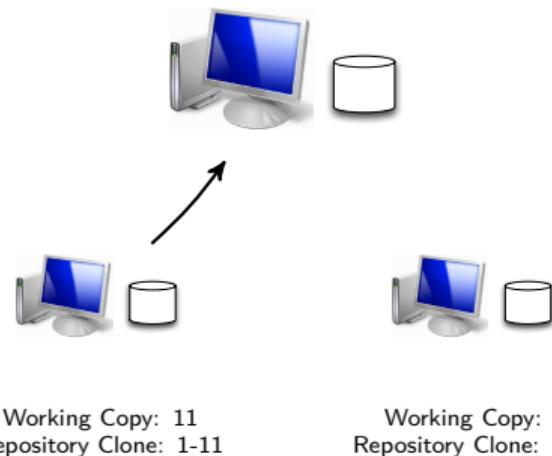
## Centralized Version Control Systems

- repository on server
- working copy on each client
- new revision for every change on the server
- revisions used to undo changes, merge files

# Distributed Version Control



Remote Repository: Revision 1-11



## Distributed Version Control Systems

- peer-to-peer architecture
- clients use repository clone to synchronize
- version history available on each client

## Distributed Version Control Systems

- main repository on one or several servers
- clone of the repository on each client
- new revision on every change on the clients

# Configuration Management and Version Control

## Lessons Learned

- Configuration management: 4 activities and 3 development stages
- Version control: goals, centralized and distributed
- Further Reading: [Sommerville](#), Chapter 25 Configuration Management

## Practice

- Github live demo
- Explore another project on Github
- Share your insights in Moodle



# Lecture Contents

## 1. Configuration Management and Version Control

## 2. Operations of Version Control Systems

Clone and Fetch

Commit and Push

How to Write Good Commit Messages?

Example: Thomas' Calculator

Merge and Pull

Ignore

Branching & Merging

Automatic Merge

Git Merge in Pictures

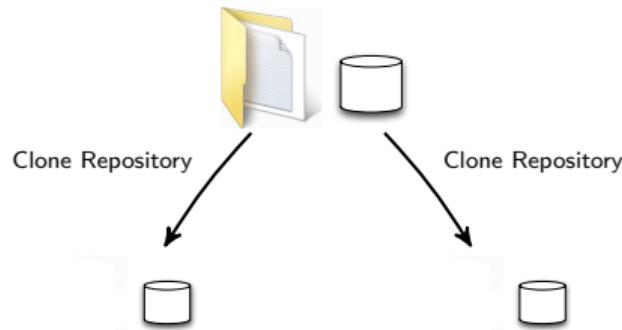
Merge Conflicts

Lessons Learned

## 3. Continuous Integration and Deployment

# Clone and Fetch

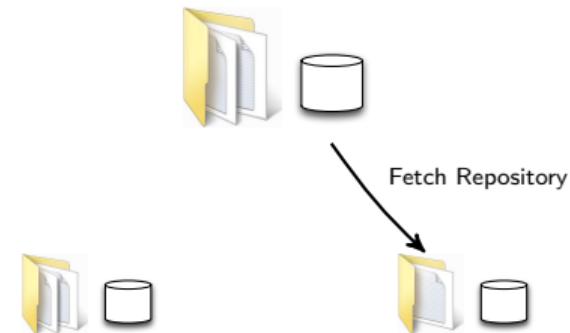
Repository: Revision 1-10



Repository Clone: 1-10

Repository Clone: 1-10

Repository: Revision 1-11



Working Copy: 11  
Repository Clone: 1-11

Working Copy: 5  
Repository Clone: 1-10

## Clone

- create a local repository clone
- use local repository to create a working copy
- specify folders for the clone
- specify revision or head (latest revision)

## Fetch

- download the remote repository
- note: working copy remains unchanged

# Commit and Push

Repository: Revision 1-10



Commit "Change message"



Working Copy: 10 (changed)  
Repository Clone: 1-10



Working Copy: 5  
Repository Clone: 1-10

Repository: Revision 1-10



Push Repository



Repository Clone: 1-11

Working Copy: 5

## Commit

- commit changes to local repository
- commits are atomic (all or nothing)
- specify changed folders and files
- mandatory message describing your change

## Push

- push local repository to server
- condition: local repository is up to date (might require previous pull)

	COMMENT	DATE
O	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
O	ENABLED CONFIG FILE PARSING	9 HOURS AGO
O	MISC BUGFIXES	5 HOURS AGO
O	CODE ADDITIONS/EDITS	4 HOURS AGO
O	MORE CODE	4 HOURS AGO
O	HERE HAVE CODE	4 HOURS AGO
O	AAAAAAA	3 HOURS AGO
O	ADKFJSLKDFJSDFKJ	3 HOURS AGO
O	MY HANDS ARE TYPING WORDS	2 HOURS AGO
O	HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

# How to Write Good Commit Messages?

## Structure of Commit Messages

[github.com]

### Subject Line (required)

- short summary (72 chars or less)
- should complete the following sentence:  
“If applied, this commit will ...”

### Message Body (optional)

- blank line followed by message body
- explain what has changed and why

## Integration with Issues

[github.com, gitlab.com]

- write **#42** to refer to issue – Github/Gitlab will create links in both ways
- **close/fix/resolve/... #42** – issue automatically closed when commit is pushed to default branch

## Conventional Commits

[conventionalcommits.org]

- Machine readable subject line
- **fix:** patches a bug (**patch**)
- **feat:** introduces a new feature (**minor** change)
- **BREAKING CHANGE:** introduces a breaking API change (**major** change)
- **refactor:** applies a refactoring
- ...
- semantic versioning: major.minor.patch (e.g., v2.6.0)

# Example: Thomas' Calculator

- o Commits on Dec 17, 2020

Merge branch 'main' into xmaslecturev3

tthuem committed on Dec 17, 2020

33e8618

- o Commits on Dec 16, 2020

- feat: Increase the default value by one every second
- feat: Allow to change the visual style of the calculator
- feat: Enable logging with the singleton pattern
- feat: Add brackets to expressions when needed
- refactor: Move classes into a new subpackage for expressions
- feat: Show complete computation as formula
- refactor: Push down fields from class Expression

tthuem committed on Dec 16, 2020

8c0298a

6164eb5

1bf0a96

37d0dac

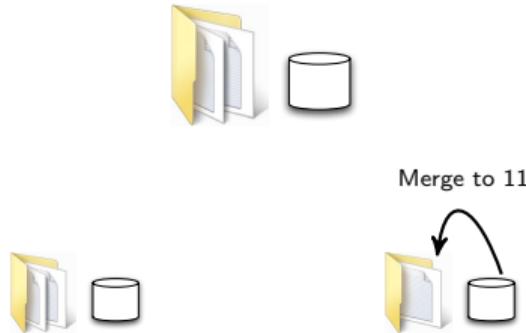
5cf7472

63d059b

b3dd5b0

# Merge and Pull

Remote Repository: Revision 1-11



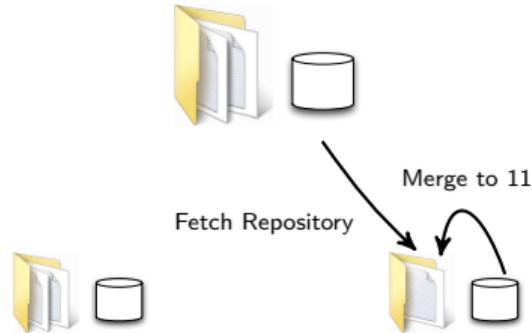
Working Copy: 11  
Repository Clone: 1-11

Working Copy: 5  
Repository Clone: 1-11

## Merge

- update working copy with local repository
- integrate new commits from other branch
- fast forward / automatic merge / manual merge

Remote Repository: Revision 1-11



Working Copy: 11  
Repository Clone: 1-11

Working Copy: 11  
Repository Clone: 1-11

## Pull

- pull = fetch + merge
- download and merge in one step!

# Ignore

Remote Repository: Revision 1-10



Ignore Foo.class + Commit



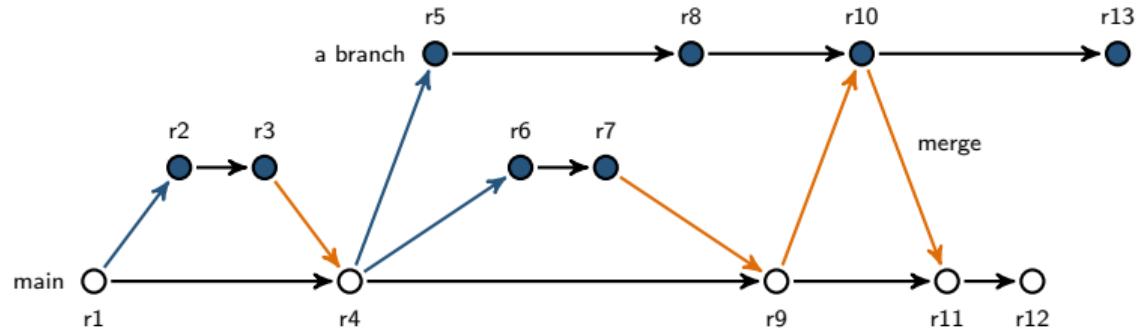
Working Copy: 10 (changed)  
Repository Clone: 1-10

Working Copy: 5  
Repository Clone: 1-10

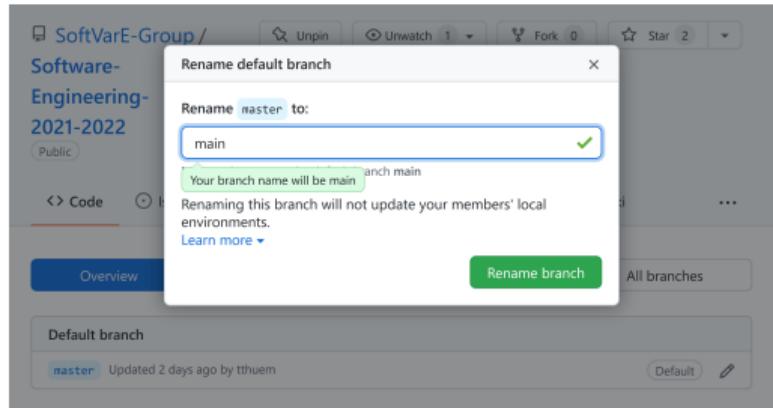
## .gitignore

- resources will be ignored on commit
- user-specific files or derived/generated resources
- specify files and folders (e.g., \*.log files)

# Branching & Merging



- simultaneous, independent development
- option to merge in the future
- main/ – main development
- branch/\*/ – parallel developments



# Automatic Merge

Working Copy: Revision 11\*

```
class Foo {  
    void bar() {  
        print("Foo");  
        print("Bar");  
        print("\n");  
    }  
}
```

Repository: Revision 11

```
class Foo {  
    void bar() {  
        print("Foo");  
        print("Bar");  
    }  
}
```

- Checkout of or update to revision 10
- Changing the working copy (10\*)
- In the meantime: New commit to repository (11)
- Update to head revision, automatically merged

# Git Merge in Pictures



# Merge Conflicts

Working Copy: Revision 11\*

```
class Foo {  
    void bar() {  
        <<<<< .mine  
        print("Bar\n");  
        ======  
        print("FooBar");  
        >>>>> .r11  
    }  
}
```

Repository: Revision 11

```
class Foo {  
    void bar() {  
        print("FooBar");  
    }  
}
```

- Checkout of or update to revision 10
- Changing the working copy (10\*)
- In the meantime: New commit to repository (11)
- Update to head revision results in conflict
- Automatic merge fails: user has to provide a fix

# Operations of Version Control Systems

## Lessons Learned

- Version control with clone, fetch, commit, push, merge, pull, ignore
- Commit messages
- Branching, merging, merge conflicts

## Practice

- Quiz'n'Disquiz
- Quiz: fill out the Moodle quiz on your own
- Disquiz: compare and discuss the results with your colleagues



# Lecture Contents

1. Configuration Management and Version Control
2. Operations of Version Control Systems
3. Continuous Integration and Deployment
  - System Building
  - Continuous Integration
  - DevOps
  - Lessons Learned

# System Building

[Sommerville]

## System Building

**“System building** is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, and other information.”

## Building Involves Three Platforms

- **development system:** compilers and editors used on the developer's system to test prior to commit
- **build server:** server to build and distribute executable versions, triggered by commits or schedule (i.e., nightly builds)
- **target environment:** intended platform for executable system (e.g., ECU in a car)

## Tooling for Building and System Integration

- build script generation: identify dependent components, automated generation or tool support for creation and editing
- version control system integration: checkout required versions of components
- minimal recompilation: determine which parts need to be recompiled
- executable system creation: compilation and linking
- test automation: run automated tests (e.g., unit tests)
- reporting: reports about success or failure of builds and tests
- documentation generation: release notes, help pages

# Continuous Integration

[Sommerville]

## Continuous Integration

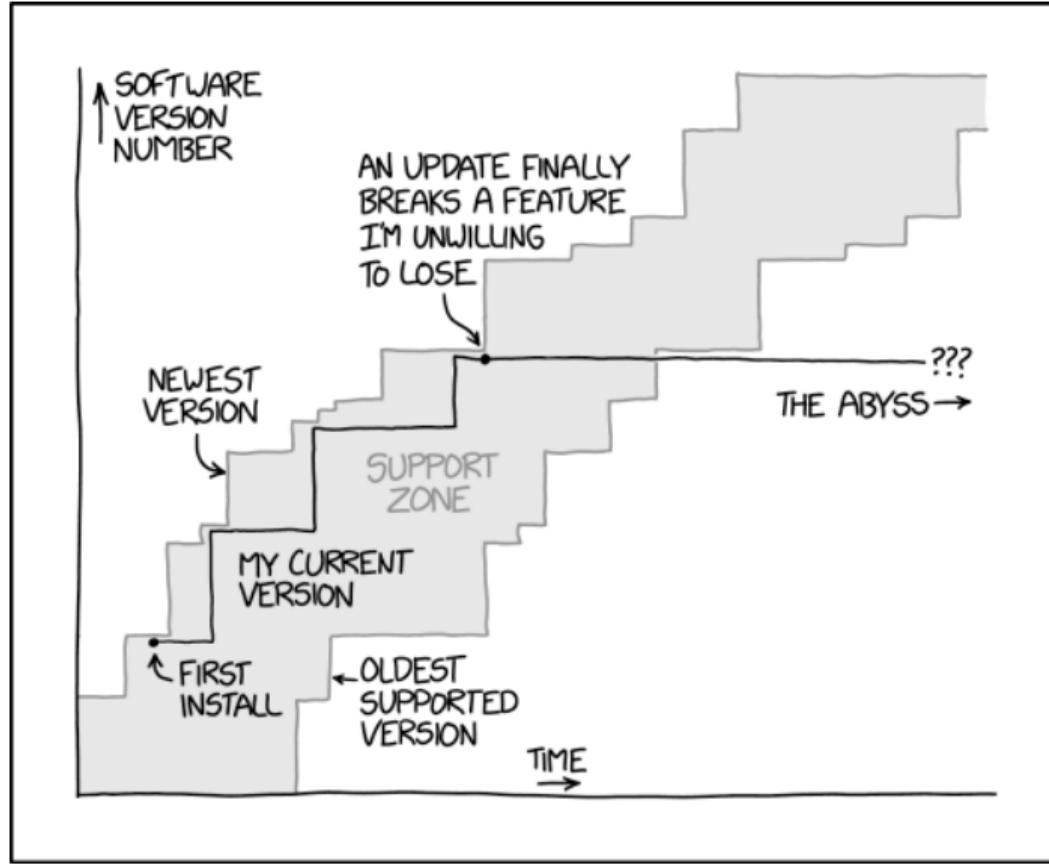
“Agile methods recommend that very frequent system builds should be carried out, with automated testing used to discover software problems. Frequent builds are part of a process of continuous integration [...].”

## Continuous Integration Tools

- **CruiseControl** (2001–2010) — first open-source tool
- **TeamCity** (2006–)
- **Hudson** (2008–2017)
- **Jenkins** (2011–) — fork of Hudson
- **Travis CI** (2011–) — made in Germany
- **GitLab** (2014–)

## Steps in Continuous Integration

- clone/fetch from version control
- if feasible: build and run automated tests, if it fails others are responsible
- apply changes
- build and run automated tests locally, if it fails continue editing
- if local tests pass, commit to feature branch in version control
- commit triggers build server, if it fails continue editing
- if tests pass (and code review approves changes), merge branch into main development branch



ALL SOFTWARE IS SOFTWARE AS A SERVICE.

# DevOps

[Krypczyk/Bochkor]

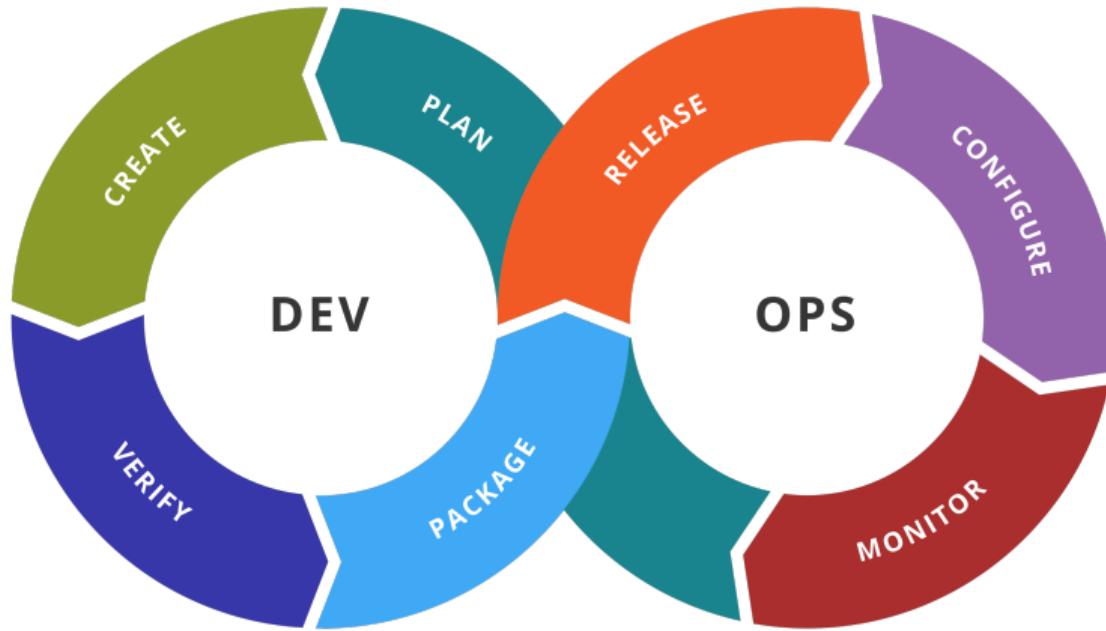
## Motivation

- if software fails:
  - ▶ programmers blame administrators for misconfiguration
  - ▶ administrators blame programmers for erroneous software
- programmers want frequent updates
- administrators follow the slogan: “never change a running system”
- customers and users want a single responsibility
- shorter and shorter update cycles

## DevOps

- promoted in agile development
- **Dev**: development by programmers
- **Ops**: operation (**Betrieb**) by administrators
- **DevOps**: teams that are responsible for both, development and operations
- goal: avoid blaming each other by shared responsibility

# DevOps



# Continuous Integration and Deployment

## Lessons Learned

- System building: 3 platforms, requirements on tooling
- Continuous integration: tools, steps
- DevOps
- Further Reading: [Sommerville](#), Chapter 25 Configuration Management and [Krypczyk/Bochkor](#), Chapter 9.3 DevOps

## Practice

- Form groups of 2–3 students
- 1. Discuss flavors of continuous integration
- 2. Report one flavor in Moodle

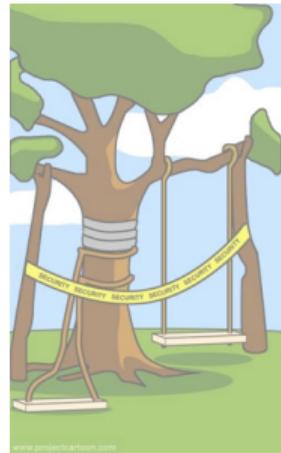




# Software Engineering

14. Compilation and Static Analysis | Thomas Thüm | May 17, 2022

# Compilation and Static Analyses



how patches were applied

**software evolution**



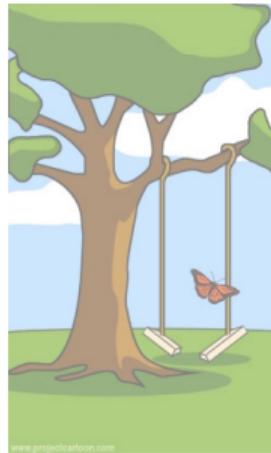
how it was supported

**software maintenance**



when it was delivered

**configuration management**



how it performed under load

**compilation and static analyses**

## Lessons Learned

- System building: 3 platforms, requirements on tooling
- Continuous integration: tools, steps
- DevOps
- Further Reading: [Sommerville](#), Chapter 25 Configuration Management and [Krypczyk/Bochkor](#), Chapter 9.3 DevOps

## Practice

- Form groups of 2–3 students
- 1. Discuss flavors of continuous integration
- 2. Report one flavor in Moodle



# Lecture Overview

1. Fundamentals on Compilation
2. Misunderstandings on Performance
3. Test-Driven Development & Design by Contract

# Lecture Contents

## 1. Fundamentals on Compilation

And the 2020 Turing Award Goes To

Recap: History of Programming Languages

Compilation vs Interpretation

The Power of Intermediate Languages

Compilation and Performance

What Happens for Incorrect Programs?

Recap: Pipe-and-Filter Architecture

Compiler Architecture

Type Safety and Type Correctness

Lessons Learned

## 2. Misunderstandings on Performance

## 3. Test-Driven Development & Design by Contract

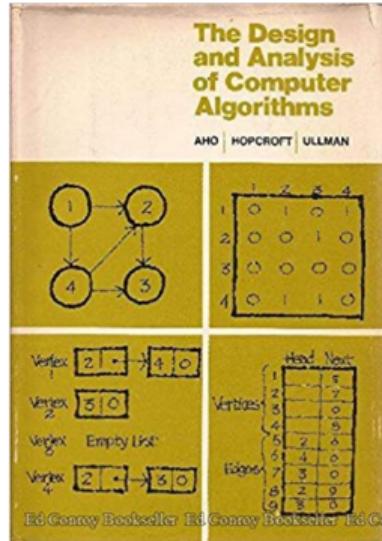
# And the 2020 Turing Award Goes To [awards.acm.org]



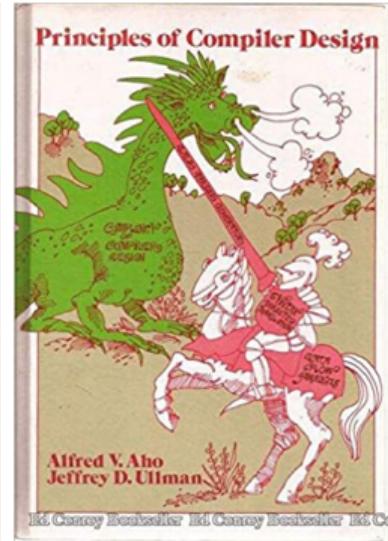
Alfred V. Aho



Jeffrey D. Ullman



1974



"Dragon Book", 1977

# Recap: History of Programming Languages

## Languages

[Jones + Krypczyk/Bochkor]

- 1945: first high-level language Plankalkül by Konrad Zuse (compiler written in 1998)
- 1954: first professional high-level language FORTRAN (Formula Translator) by IBM
- 1963: Basic as general-purpose language
- 1959: functional language Lisp
- 1970: first object-oriented lang. Smalltalk-80
- 1970: declarative language SQL
- 1971: Pascal by Niklaus Wirth for teaching
- 1974: very common procedural language C
- 1977: logical language Prolog
- 1980: C++ as object-oriented extension of C
- 1990: object-oriented language Java
- 1990: functional language Haskell
- 1991: multi-paradigm language Python
- 1995: scripting language JavaScript

## Milestones

[Jones]

- controlling behavior of mechanical devices by wiring or with punchcards ([Lochkarten](#))
- machine languages used during World War II
- assembly languages: distinction between human-readable instructions (source code) and executable instructions (object code)
- birth of compilers and interpreters having a one-to-many mapping between source and object code (opposed to one-to-one mapping in assemblers)
- structured programming pioneered by David Parnas and Edsger Dijkstra
- high-level programming languages: high number of executable for each human-readable instruction
- domain-specific languages, later general-purpose programming languages

# Compilation vs Interpretation

## Compilation

- C/C++/Go/Rust/Swift to machine code
- Java/Groovy/Kotlin/Scala/Clojure to Java bytecode

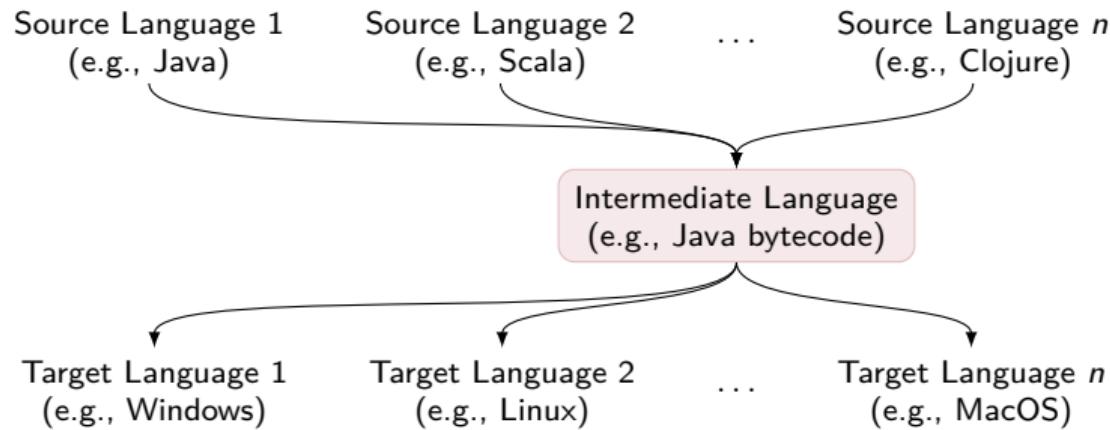
Source Code → Compiler → Target Code

## Interpretation

- of source code:  
Ruby/Python/Perl/PHP/Matlab
- of bytecode: Java Virtual Machine (JVM)



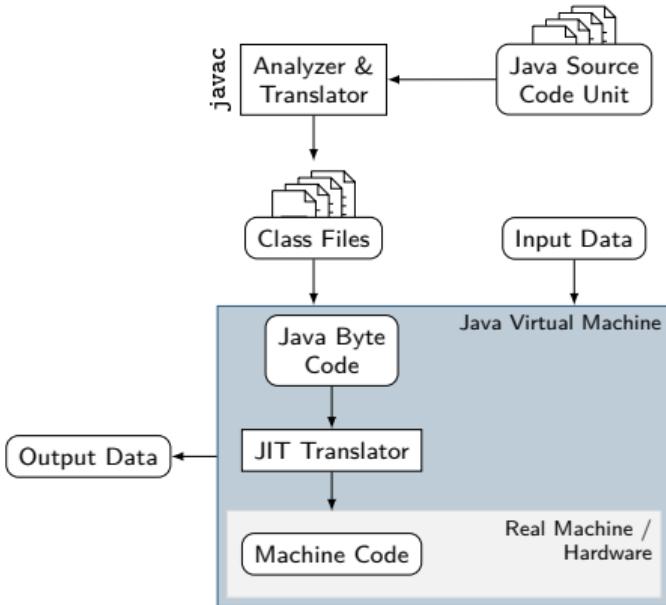
# The Power of Intermediate Languages



# Compilation and Performance

## Goals of Compiler Optimizations

- fast execution
- low memory / energy consumption
- small binaries (fast start/download/updates)
- desired for both compiler (compile time) and compiled program (run time)



## Compile Time vs Run Time

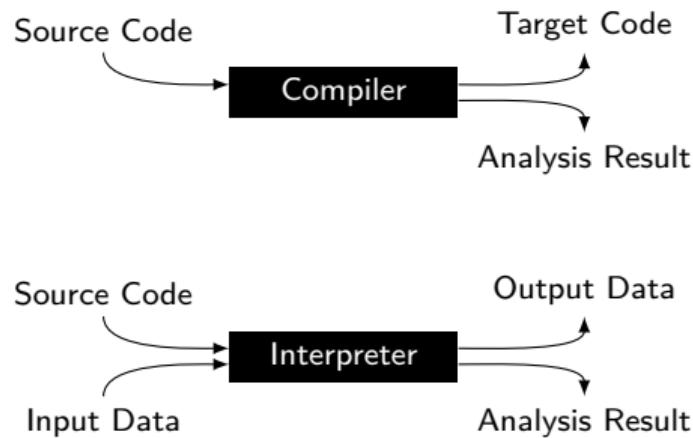
run time: when program or software is executed

compile time: during (ahead-of-time) compilation

## Just-in-Time Compilation

- often executed code is compiled at run time
- warm-up time: execution is slower when new code is executed

# What Happens for Incorrect Programs?



## Warnings and Errors

- errors indicate that compilation failed (target code is incomplete)
- warnings indicate potential problems or that compilation may fail in the future (cf. deprecated methods)

## Common Types of Errors

lexical errors, parse error, type error, runtime errors, logical errors

# Recap: Pipe-and-Filter Architecture

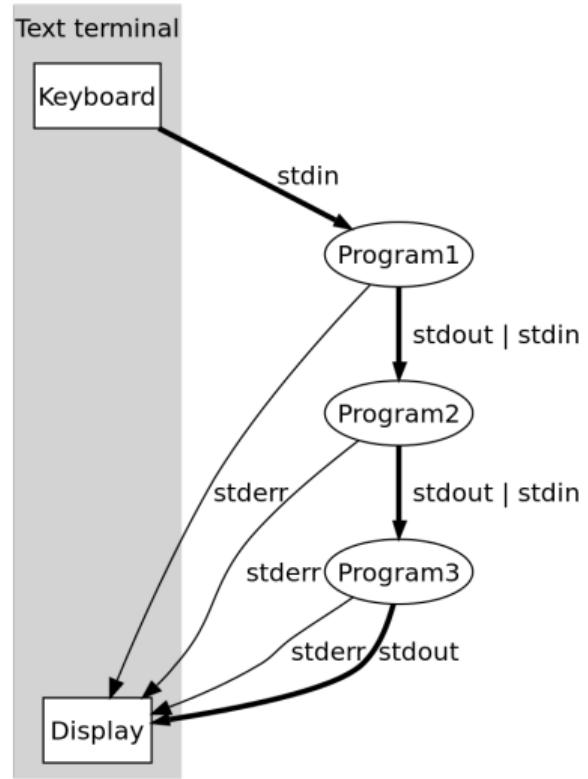
## Pipe-and-Filter Architecture

[Sommerville]

- **Problem:** data is processed in numerous processing steps, which are prone to change
- **Idea:** modularization of each processing step into a component
- filter components process a stream of data continuously
- pipes transfer data unchanged from filter output to filter input

## Pipe Operator in UNIX

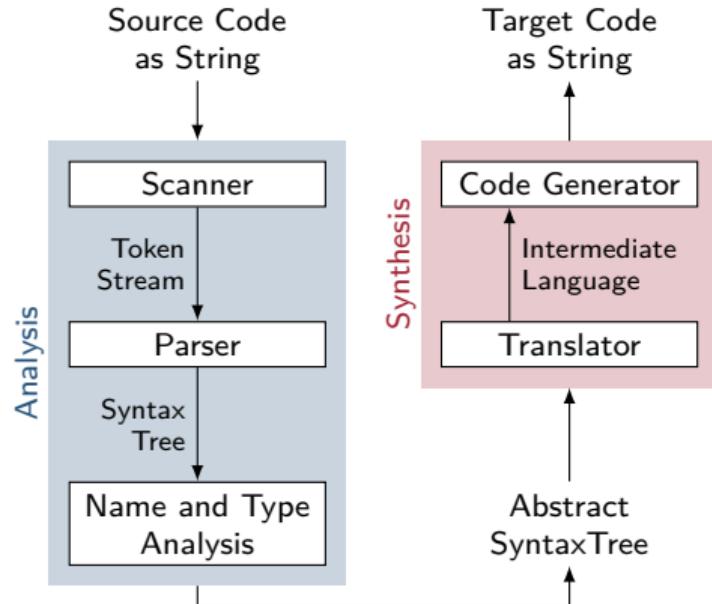
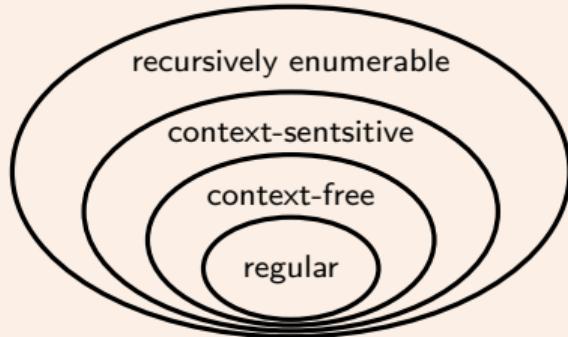
"ls -al | grep '2020' | grep -v 'Nov' | more" searches files in a folder from the year 2020 except those from November and delivers the results in pages.



# Compiler Architecture

## Application of the Chomsky Hierarchy

- scanning/lexing: regular expression for each token class
- parsing: context-free grammars (e.g., a class has fields, methods, and inner classes)
- name and type analysis: context-sensitive analysis (e.g., lookup type for identifier)



# Type Safety and Type Correctness

## Type Safety

A **type** characterizes properties of program elements, for example:

- a variable can only store particular values
- an expression returns particular values
- an object has a method with a certain signature (name and parameter types)

A **type error** occurs if properties are not met. A program is **type safe** if its execution cannot lead to type errors.

## Type Errors

undefined identifier, assignment of incompatible type, method call with incompatible parameter

## Type Correctness

- The language specification defines type rules checked by the compiler (**statically typed language**) or by the interpreter (**dynamically typed language**).
- All type rules together constitute the **type system**.
- A program is **type correct** if it satisfies the type rules.
- A programming language is **strongly typed** if every type correct program is type safe (**weakly typed** otherwise).

## In Practice

continuum between strongly (e.g., Java) and weakly typed (e.g., JavaScript) languages

# Fundamentals on Compilation

## Lessons Learned

- 2020 Turing Award for high-level programming languages
- Compilation vs interpretation vs just-in-time compilation
- Compile time vs run time: goals of optimizations
- Compiler architecture and intermediate languages
- Type safety and correctness

## Practice

- Form groups of 2–3 students
- Discussion: What are chances and risks of high-level programming languages (i.e., an increasing gap between high-level instructions and low-level machine code)?
- Add one argument to Moodle



# Lecture Contents

1. Fundamentals on Compilation
2. Misunderstandings on Performance
  - Common Misunderstandings
  - Compiler Optimizations
  - No Array Access?
  - No Loops?
  - No Method Calls?
  - No Objects?
  - No Garbage Collection?
  - Speed = Instructions per Minute?
  - Recap: Simplicity over Performance
  - Lessons Learned
3. Test-Driven Development & Design by Contract

# Common Misunderstandings

## Misunderstanding #1

"I've heard array access is slow.  
Now I try to avoid them."

## Misunderstanding #2

"I've heard loops are slow.  
Now I try to avoid them."

## Misunderstanding #3

"I've heard method calls are slow.  
Now I try to avoid them."

## Misunderstanding #4

"I've heard objects are slow.  
Now I try to avoid them."

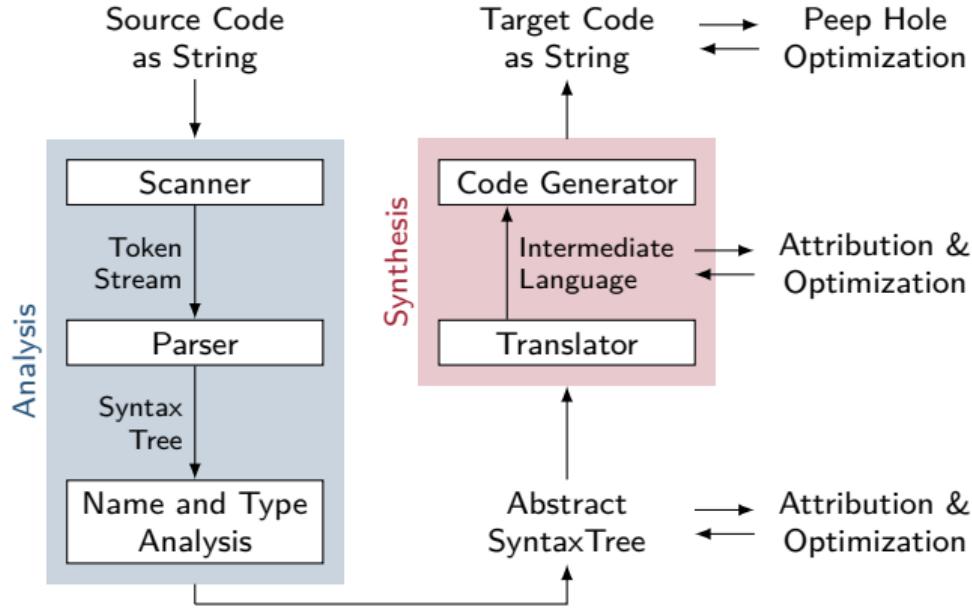
## Misunderstanding #5

"I've heard garbage collection is slow.  
Now I try to avoid it."

## Misunderstanding #6

"I'm new to programming and think that every instruction takes equally long."

# Compiler Optimizations



## Kinds of Optimizations

- machine-dependent optimizations: exploit properties of a particular machine
- machine-independent optimizations: applicable to several machines
- local optimizations: e.g., switch order of two statements
- intra-procedural optimizations: affect only one method
- inter-procedural optimizations: affect several methods or require global knowledge

# No Array Access?

## Misunderstanding #1

"I've heard array access is slow.  
Now I try to avoid them."

## Tasks for Array Access $a[b]$

- evaluate the expression  $b$  (could be arbitrarily complex)
- compute offset =  $b * \text{size of each field}$
- compute memory location = position of  $a$  + offset
- access memory location
- only for objects: use value as memory location

## Example

```
a[b.n()] = a[b.n()-1] * a[b.n()-1];
```

## Simplified Example

Assumption 1: method  $n$  has no side-effects

Assumption 2: method  $n$  cannot be overridden  
(e.g., a private method)

```
int n = b.n();  
a[n] = a[n-1] * a[n-1];
```

## Note

arrays are very common, compilers and hardware have plenty of optimizations

# No Loops?

## Misunderstanding #2

"I've heard loops are slow.  
Now I try to avoid them."

## Tasks for Loops

- create and initialized loop variable
- check loop condition
- **run loop body**
- increment loop variable
- repeat from check loop condition

## Example Loop

```
for (int i = 0; i++; i < 3) { a[i] = i; }
```

## Loop Unrolling

```
a[0] = 0; a[1] = 1; a[2] = 2;
```

## Do Not Avoid Loops!

**smells:** duplicated code, long method

**compiler optimization:** loop unrolling (if number of runs is statically known and small enough)

# No Method Calls?

## Misunderstanding #3

"I've heard method calls are slow.  
Now I try to avoid them."

## Tasks for Each Method Call

- pass parameters and return address
- save registers
- run method body, pass return value
- restore registers, return to caller

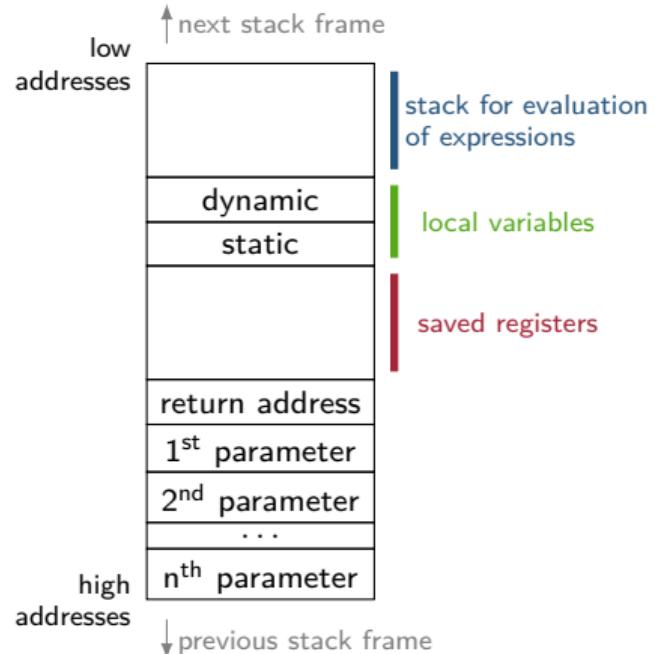
## Do Not Avoid Method Calls!

**smells:** duplicated code, long method

**refactoring:** extract method

**compiler optimization:** method inlining

## Memory Layout: The Method Stack



# Method Inlining in Practice

What is the output of this program?

```
class A {  
    public static void main(String[] args) {  
        A x = new B();  
        Object y = new String();  
        System.out.print(x.m(y));  
    }  
    int m(Object o) { return 1; }  
    int m(String s) { return 2; }  
}  
class B extends A {  
    int m(Object o) { return 3; }  
    int m(String s) { return 4; }  
}
```

Hint: Java uses static dispatch for overloading (y) and  
dynamic dispatch for overriding (x – required for inheritance)

```
class A {  
    public static void main(String[] args) {  
        System.out.print(3);  
    }  
}
```

# No Objects?

## Misunderstanding #4

"I've heard objects are slow.  
Now I try to avoid them."

## Memory Layout for Objects

- if life time not bound to a method, cannot be stored on method stack
- stored in free position on the heap
- pointer and dereference required
- need to store class information
- pointer to virtual method table for dynamic dispatch
- all fields, even inherited fields

## Do Not Avoid Objects!

unless performance or memory consumption is a problem, then identify which objects (a) consume most memory and (b) have the shortest life time

## Thomas' Experience with LinkedList in Java

LinkedList can be problematic, as there is a list object for every entry in the list and list manipulations lead to new list objects

solution: use arrays or ArrayList instead

large speed-ups in FeatureIDE: [github.com](https://github.com)

also reported by others: [stackoverflow.com](https://stackoverflow.com)

# No Garbage Collection?

## Misunderstanding #5

"I've heard garbage collection is slow.  
Now I try to avoid it."

## Garbage Collection

- find objects not referenced anymore
- free memory space assigned by them
- algorithms: reference counting, mark-and-sweep, copying collection
- causes random delays

## Reasons for Automatic Memory Management

- simplified programming, programs
- fewer memory leaks
- improved safety, security, reliability



## Avoiding Garbage Collection

- switch to a language with manual memory allocation (e.g., C/C++)
- deactivate garbage collection completely (only for programs with short runtime and low memory consumption)
- web services: do garbage collection when service is idle

# Speed = Instructions per Minute?

## Misunderstanding #6

"I'm new to programming and think that every instruction takes equally long."

## Misunderstanding #7

"I'm new to computers / smartphones and think that every instruction takes equally long."

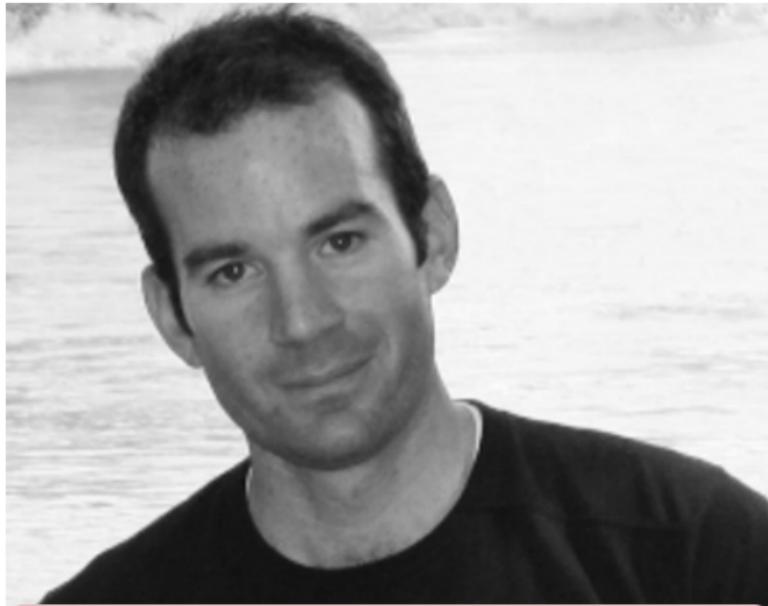
## Hint

Use all language constructs and let compilers do their job.

If performance is not sufficient, inspect bottleneck.

Only reduce code quality in favor of performance where necessary.

# Recap: Simplicity over Performance



**Wes Dyer**

[[twitter.com](#)]

“Make it correct, make it clear, make it concise, make it fast. In that order.”



**Joshua J. Bloch (born 1961)**

[[twitter.com](#)]

“The cleaner and nicer the program, the faster it's going to run. And if it doesn't, it'll be easy to make it fast.”

# Misunderstandings on Performance

## Lessons Learned

- Compiler optimizations
- What are common misunderstandings about performance?
- Why are array, loops, method calls, objects, and garbage collection slow?
- What is the connection between compiler optimizations, smells, and refactorings?

## Practice

- Thomas decides about your opinion
- What is better large or small classes?
- Long or short methods?
- One- or multidimensional arrays?
- Clean or fast programs?



# Lecture Contents

1. Fundamentals on Compilation
2. Misunderstandings on Performance
3. Test-Driven Development & Design by Contract
  - Type Error
  - Runtime Error
  - Unit Testing with JUnit
  - Logical Error
  - Runtime Assertions
  - Design by Contract
  - Generated Assertions and Blame Assignment
  - Behavioral Subtyping
  - Lessons Learned

# Type Error

Is the program type correct?

```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first;  
        this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
    public static void main(String[] args) {  
        Node a = new Node();  
        Node b = new Node();  
        System.out.println(new Edge(a, b).equals());  
    }  
}
```

Error message by the Java compiler

“The method equals(Edge) in the type Edge is not applicable for the arguments ()”

No – type error for a method call

```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first;  
        this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
    public static void main(String[] args) {  
        Node a = new Node();  
        Node b = new Node();  
        System.out.println(new Edge(a, b).equals());  
    }  
}
```

# Runtime Error

Is the revised program type correct?

```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first;  
        this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
    public static void main(String[] args) {  
        Node a = new Node();  
        Node b = new Node();  
        System.out.println(new Edge(a, b).equals(null));  
    }  
}
```

## NullPointerException

not detected by the Java compiler, but when executing the program

Yes, it is type correct

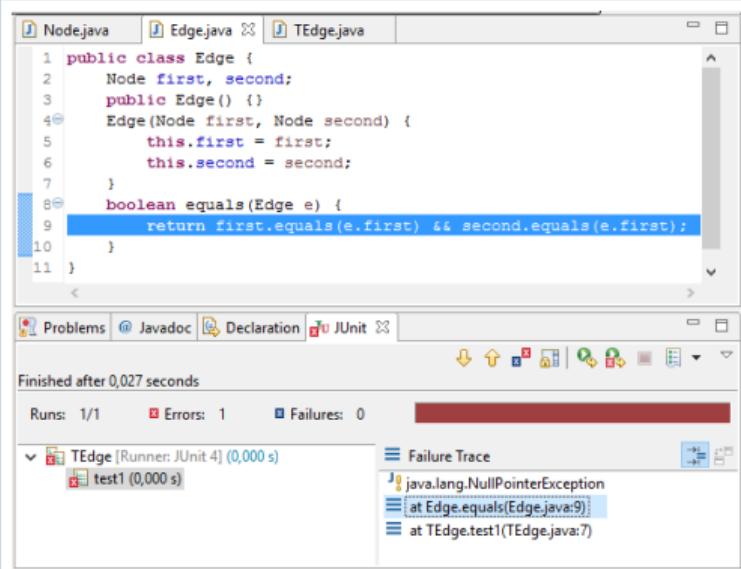
```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first;  
        this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
    public static void main(String[] args) {  
        Node a = new Node();  
        Node b = new Node();  
        System.out.println(new Edge(a, b).equals(null));  
    }  
}
```

# Unit Testing with JUnit

Test-driven development: write a test first

```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first; this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
}  
  
import org.junit.Test;  
public class TEdge {  
    @Test  
    public void test1() {  
        Node a = new Node();  
        Node b = new Node();  
        System.out.println(new Edge(a, b).equals(null));  
    }  
}
```

JUnit in Eclipse: test is supposed to fail

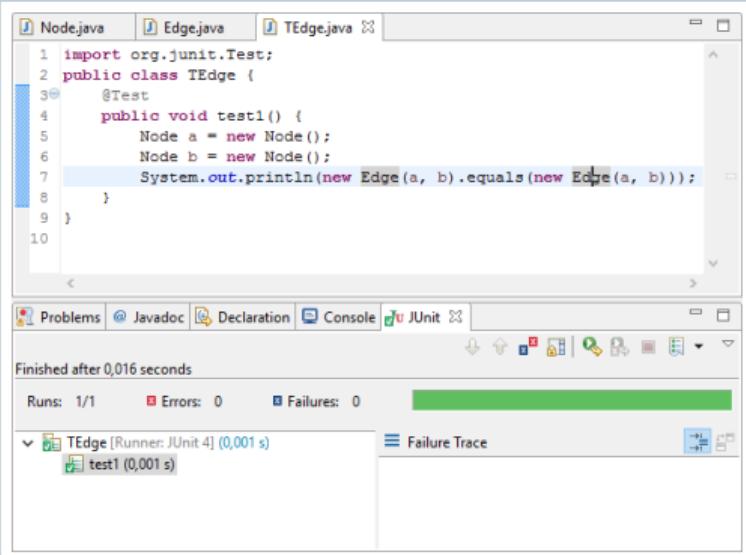


# Unit Testing with JUnit

Test-driven development: fix code afterwards

```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first; this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
}  
  
import org.junit.Test;  
public class TEdge {  
    @Test  
    public void test1() {  
        Node a = new Node();  
        Node b = new Node();  
        System.out.println(new Edge(a, b).equals(new Edge(a, b)));  
    }  
}
```

JUnit in Eclipse: check whether test still fails



# Logical Error

## Ops: unexpected output **false**

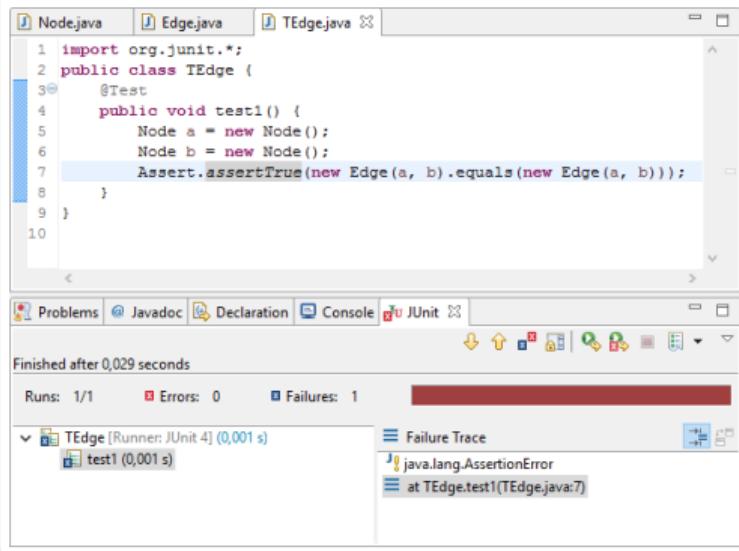
```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first; this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
}  
import org.junit.Test;  
public class TEdge {  
    @Test  
    public void test1() {  
        Node a = new Node();  
        Node b = new Node();  
        System.out.println(new Edge(a, b).equals(new Edge(a, b)));  
    }  
}
```

## Using assertions in JUnit tests

```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first; this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
}  
import org.junit.*;  
public class TEdge {  
    @Test  
    public void test1() {  
        Node a = new Node();  
        Node b = new Node();  
        Assert.assertTrue(new Edge(a, b).equals(new Edge(a, b)));  
    }  
}
```

# Runtime Assertions

## JUnit failure in Eclipse



## Violated assertion

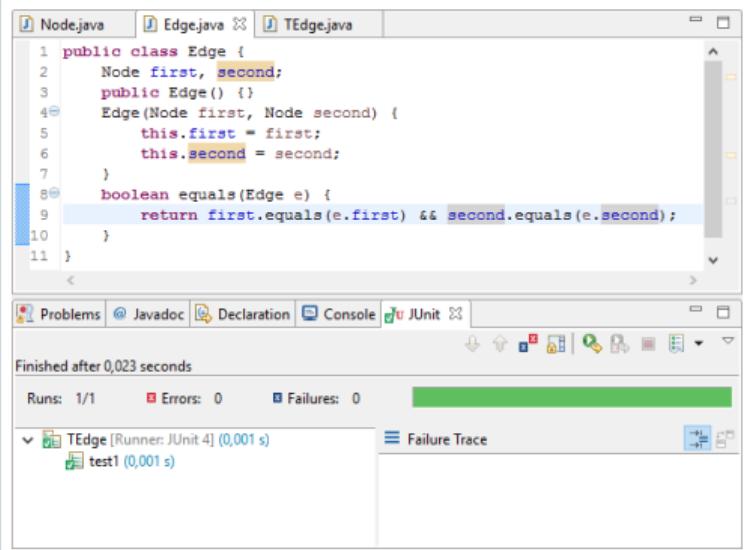
```
class Node {}
class Edge {
    Node first, second;
    Edge(Node first, Node second) {
        this.first = first; this.second = second;
    }
    boolean equals(Edge e) {
        return first.equals(e.first) && second.equals(e.second);
    }
}
import org.junit.*;
public class TEdge {
    @Test
    public void test1() {
        Node a = new Node();
        Node b = new Node();
        Assert.assertTrue(new Edge(a, b).equals(new Edge(a, b)));
    }
}
```

# Runtime Assertions

All type and runtime errors fixed?

```
class Node {}  
class Edge {  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first; this.second = second;  
    }  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.second);  
    }  
}  
  
import org.junit.*;  
public class TEdge {  
    @Test  
    public void test1() {  
        Node a = new Node();  
        Node b = new Node();  
        Assert.assertTrue(new Edge(a, b).equals(new Edge(a, b)));  
    }  
}
```

JUnit is happy again. What does it mean?



# Design by Contract

## Motivation

- code pollution by defensive programming
- specification far away from code: in tests or separate documents
- informal specification (e.g., JavaDoc) often outdated, only limited automated checks

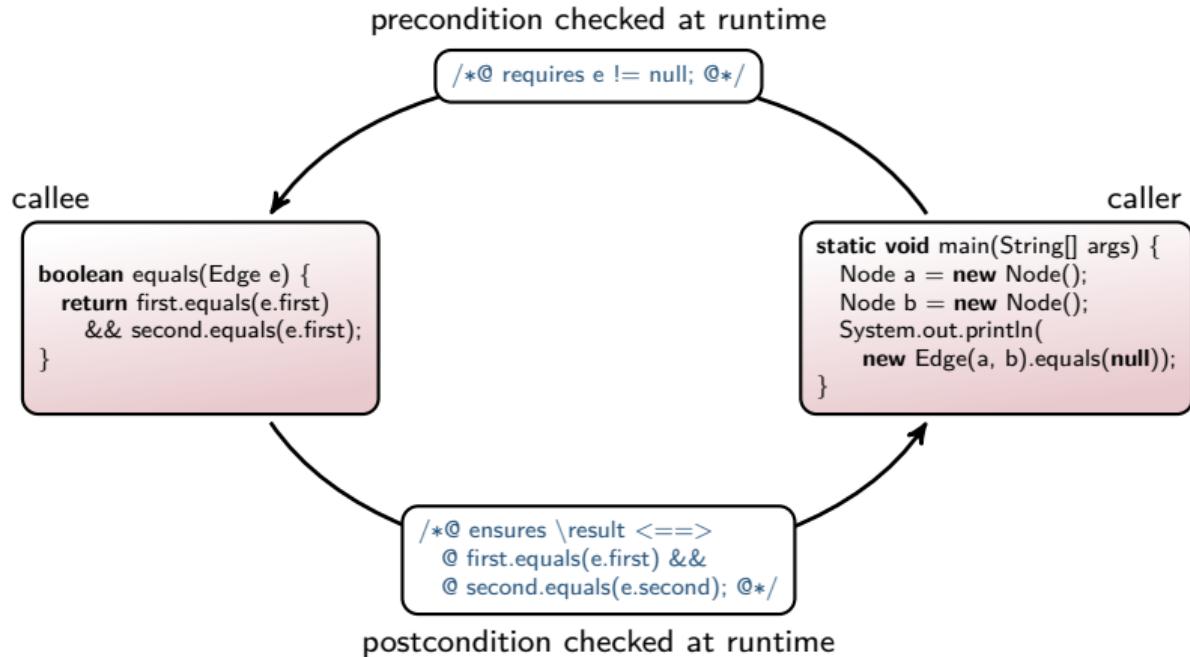
## Design by Contract

- code-level specification, formal specification
- assertions on methods: preconditions, postconditions, assignable locations
- assertions on classes: class invariants
- for example: Java Modeling Language (JML)
- generation of documentation (jmldoc), runtime assertions (jmlc), unit tests (jmlunit), for formal verification (KeY), ...

## Class invariants, preconditions, and postconditions

```
class Node {}  
class Edge {  
    /*@ invariant first != null && second != null;  
    Node first, second;  
    Edge(Node first, Node second) {  
        this.first = first;  
        this.second = second;  
    }  
    /*@ requires e != null;  
     * @ ensures \result <==> first.equals(e.first) &&  
     * @ second.equals(e.second); @*/  
    boolean equals(Edge e) {  
        return first.equals(e.first) && second.equals(e.first);  
    }  
    public static void main(String[] args) {  
        Node a = new Node();  
        Node b = new Node();  
        System.out.println(new Edge(a, b).equals(null));  
    }  
}
```

# Generated Assertions and Blame Assignment



# Behavioral Subtyping

```
class Edge {  
    // @ invariant first != null && second != null;  
    Node first, second;  
    /*@ requires e != null;  
     * @ ensures \result ==> first.equals(e.first) &&  
     *   second.equals(e.second); @*/  
    boolean equals(Edge e) {  
        return first.equals(e.first) &&  
               second.equals(e.second);  
    }  
    [...]  
}
```

```
new Edge(...).equals(...);
```

```
Edge e = new WeightedEdge(...);  
e.equals(...);
```

```
class WeightedEdge extends Edge {  
    Integer weight = 0;  
    /*@ also  
     * @ requires e != null && weight != null;  
     * @ ensures \result ==> weight == e.weight; @*/  
    boolean equals(WeightedEdge e) {  
        return super(e) && weight.equals(e.weight);  
    }  
    [...]  
}
```

```
new WeightedEdge(...).equals(...);
```

# Test-Driven Development & Design by Contract

## Lessons Learned

- Test-driven development
- Examples for type, runtime, and logical errors
- Examples for JUnit and assertions
- Design by contract: class invariants, preconditions, postconditions
- Blame assignment, behavioral subtyping
- Java Modeling Language

## Practice

- Quiz with examples for parse error, type errors, runtime errors, logical errors
- Post questions to Moodle, if any

