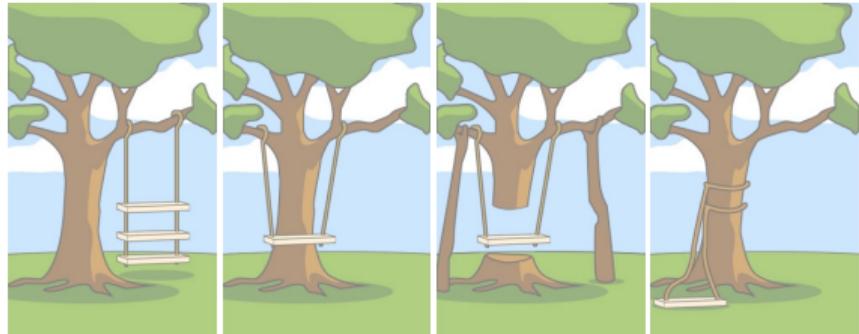




# Software Engineering

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**



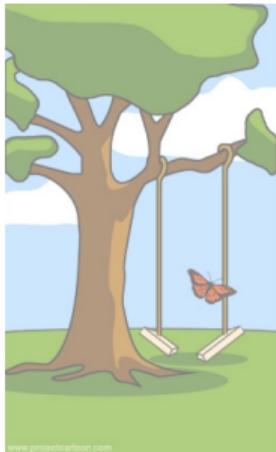
how it was supported

**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

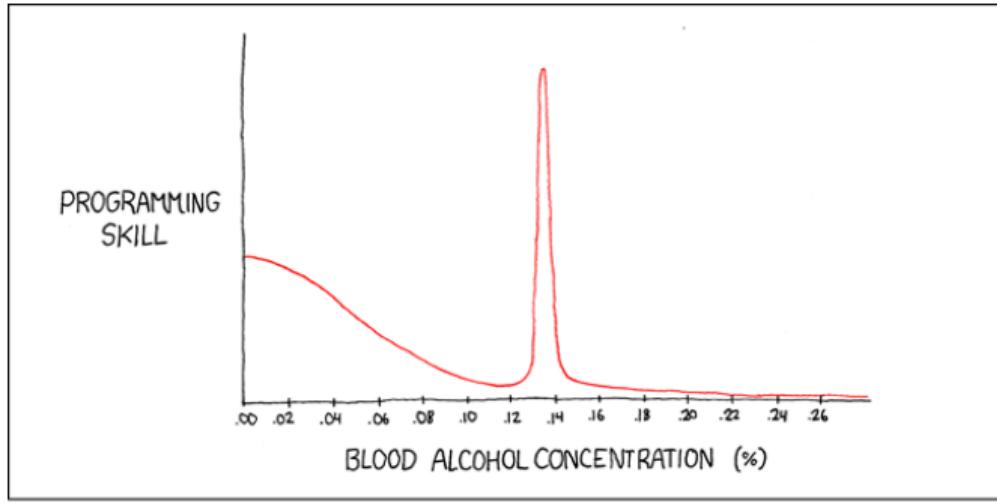
[Sommerville]

- 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

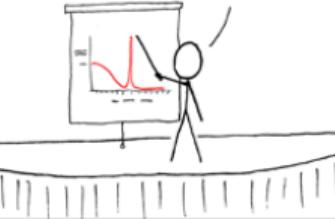
# Recap: Extending Thomas' Calculator

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% CONFERS 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!*

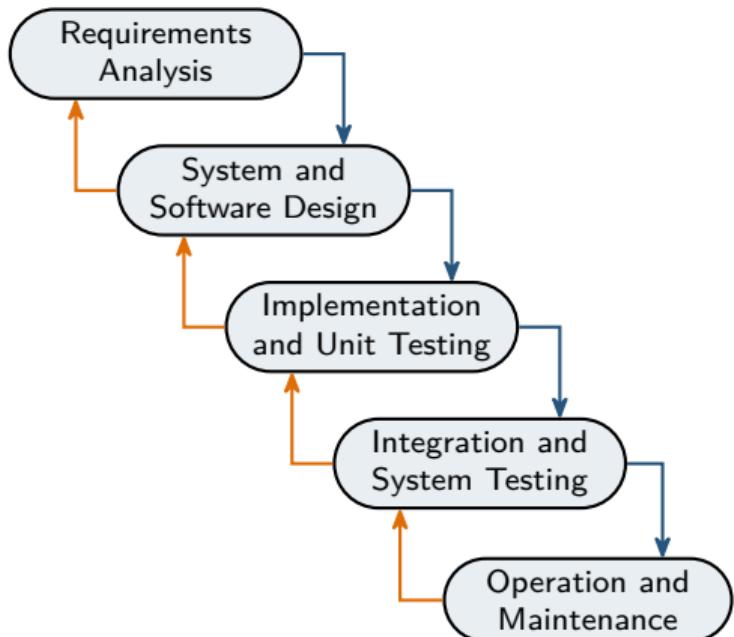


# Recap: Design Patterns [Gang of Four (GoF)]

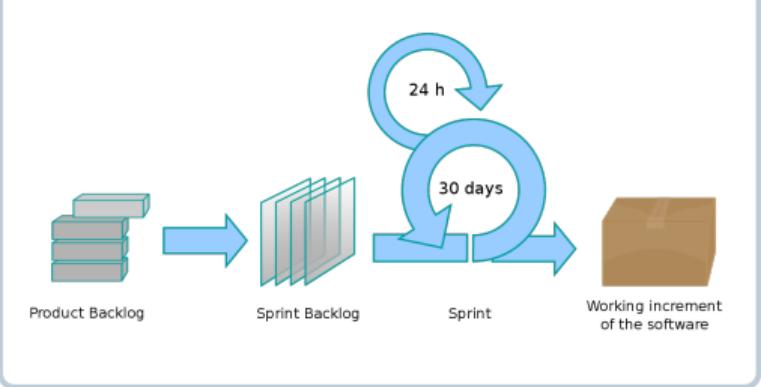


# Recap: Process Models

## Waterfall Model

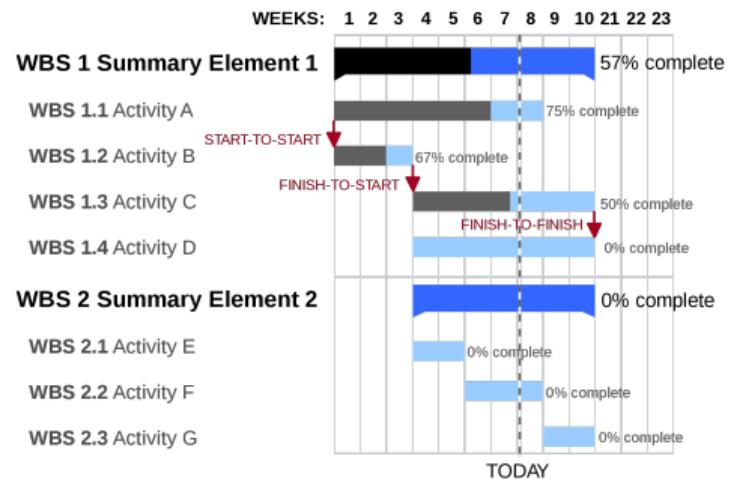


## Scrum



# 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**

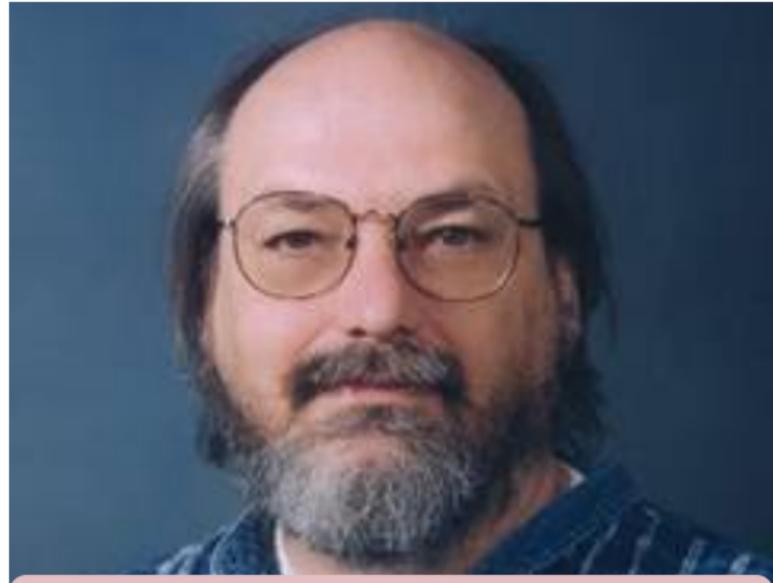
# Avoiding Complexity



**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

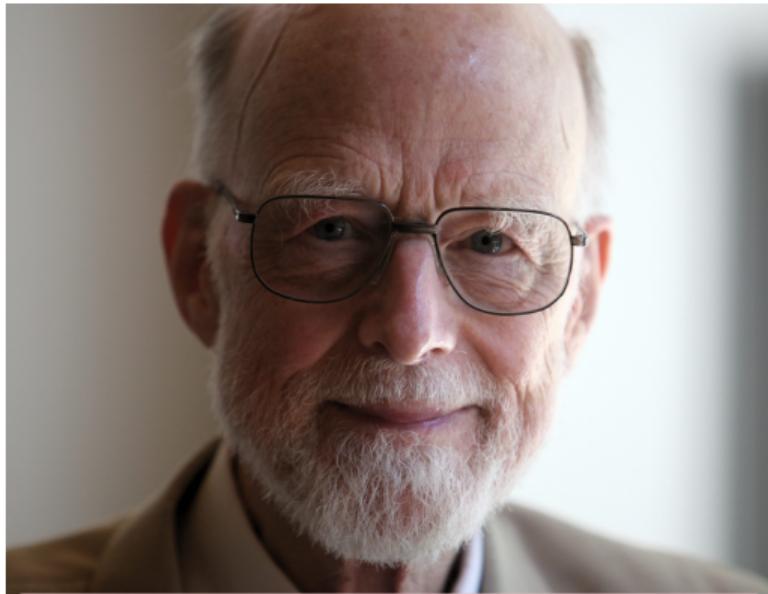
QUICKSORT  
MERGE

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

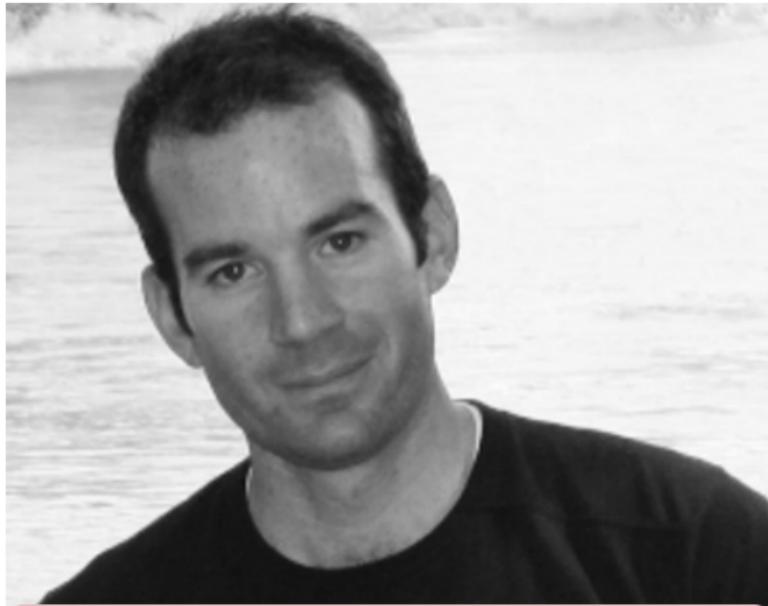
## Essence of the Laws

- 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](#))

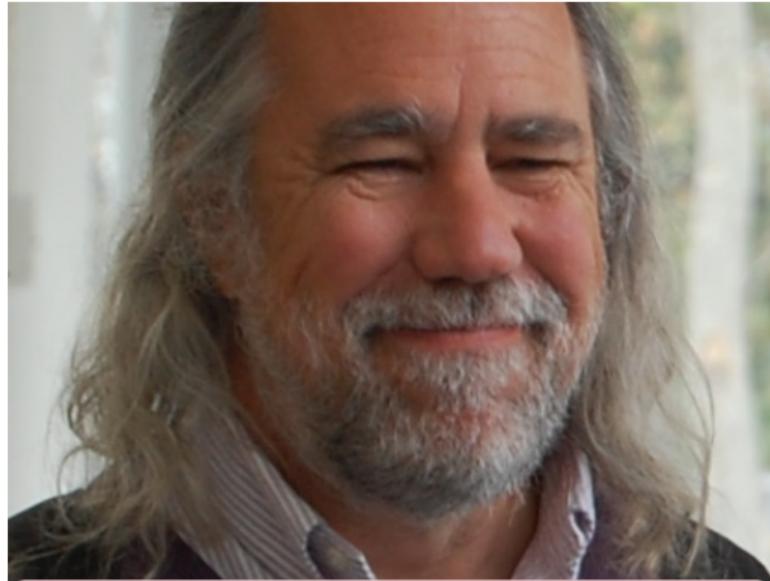
## Practice

- 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