Universität zu Köln

Software Engineering

Evolution and Maintenance

Software & Systems Engineering | Prof. Dr. Andreas Vogelsang | 17.01.2024

@andivogelsang
vogelsang@cs.uni-koeln.de

Learning Goals for Today

- Know why software is subject to constant evolution and change
- Know what software maintenance is and what types of maintenance work exist
- Know the technical debt metaphor

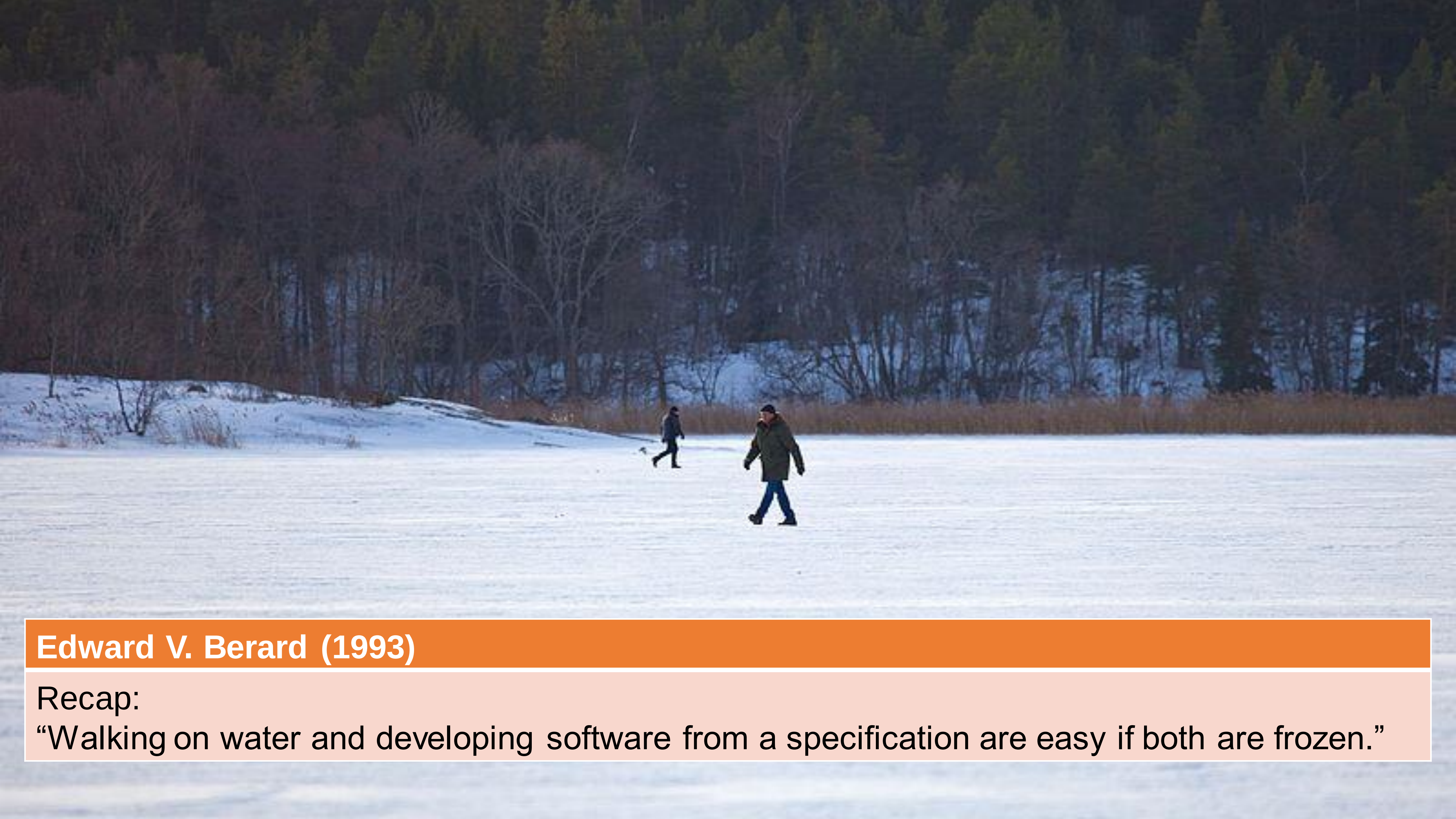
Change is Inevitable

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.”

What might change?

- 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



Edward V. Berard (1993)

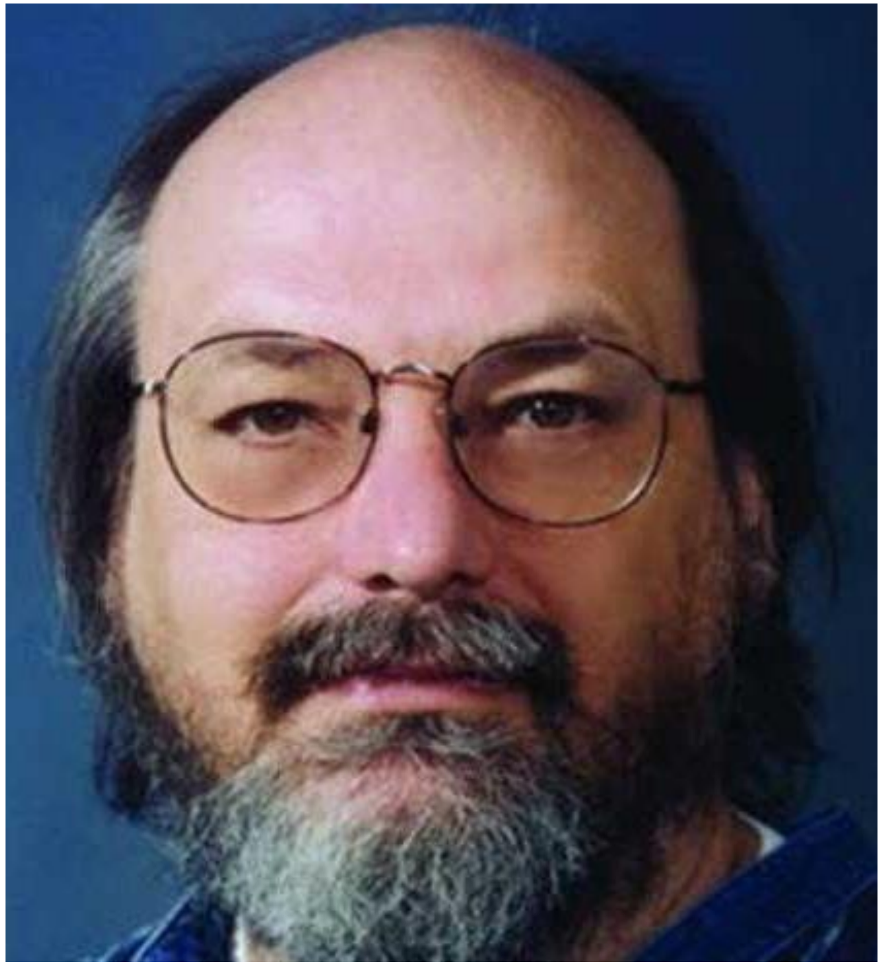
Recap:
“Walking on water and developing software from a specification are easy if both are frozen.”

Software Evolution

Avoid Complexity

Richard E. Pattis

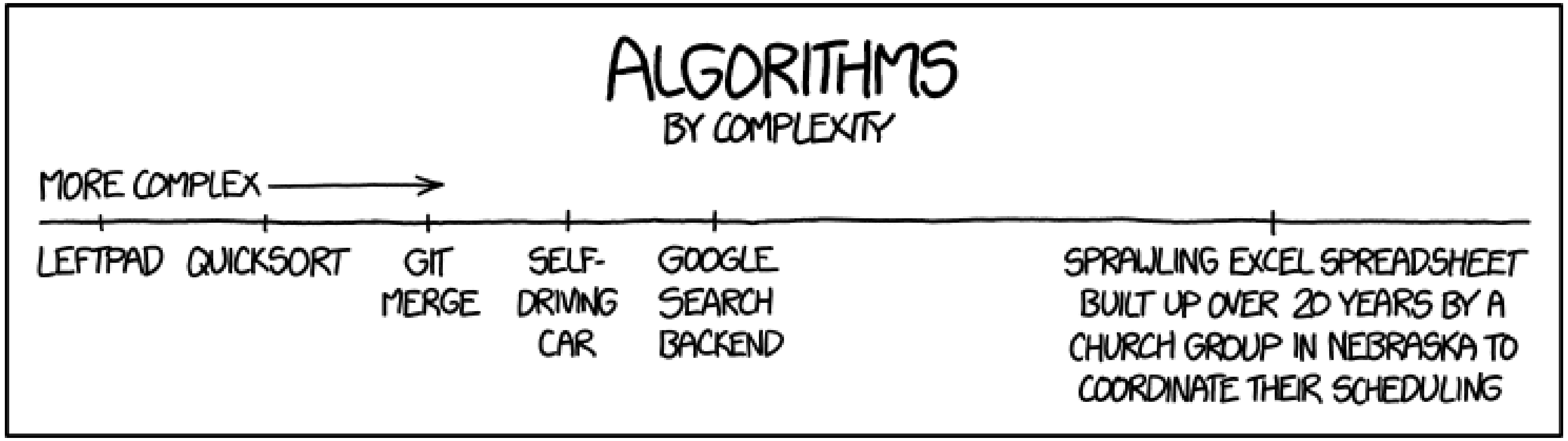
“When debugging, novices insert corrective code; experts remove defective code.”



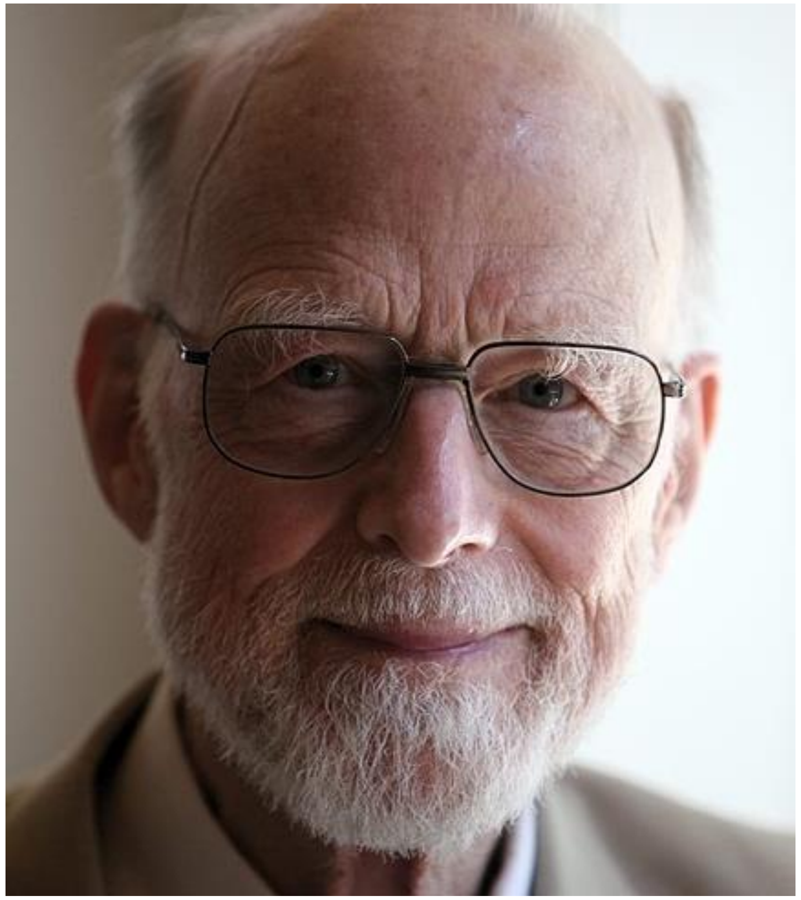
Ken Thompson

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

Essential vs. Accidental Complexity


ALGORITHMS
BY COMPLEXITY
MORE COMPLEX
LEFTPAD
QUICKSORT
GIT MERGE
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

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

Meir "Manny" Lehman

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

Lehman's Laws of Evolution [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

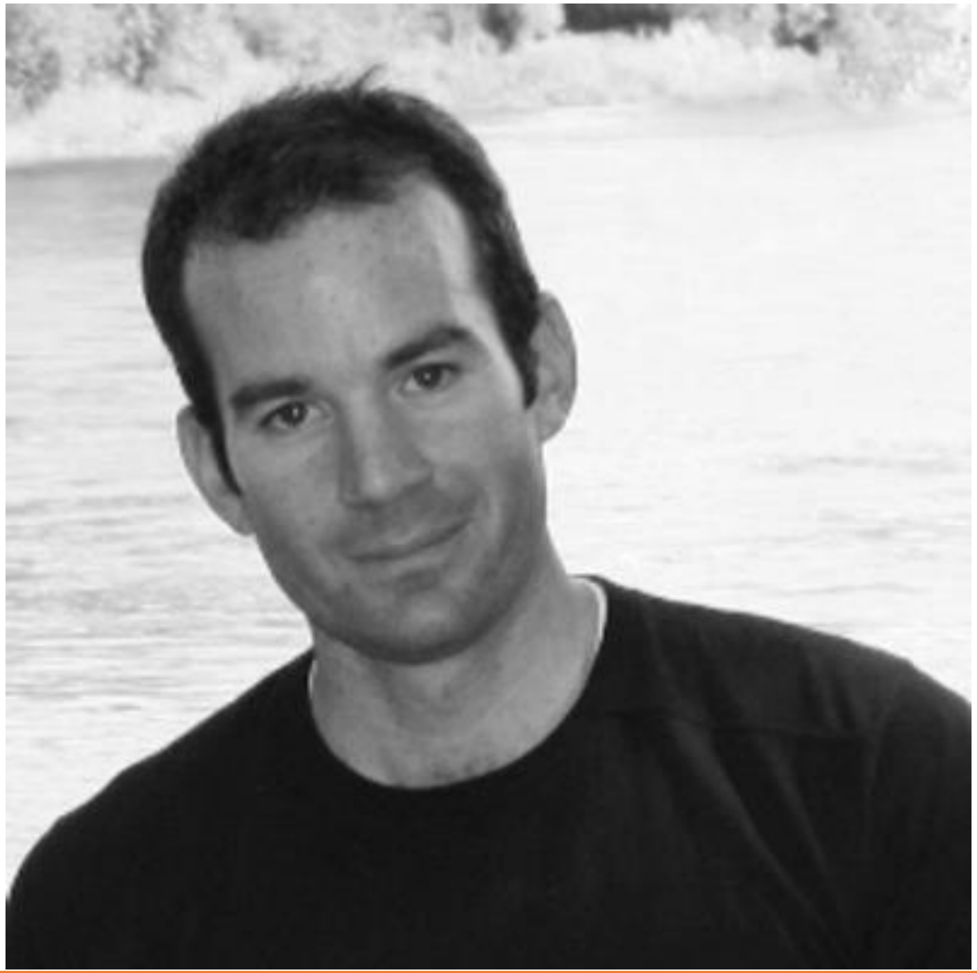
Essence of the Laws

- 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

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

Joshua J. Bloch

“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.”

Software Maintenance

Software Maintenance

Motivation

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

Operations 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

Adaptive Maintenance

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 Win10 to 11)

Perfective Maintenance

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

Corrective Maintenance

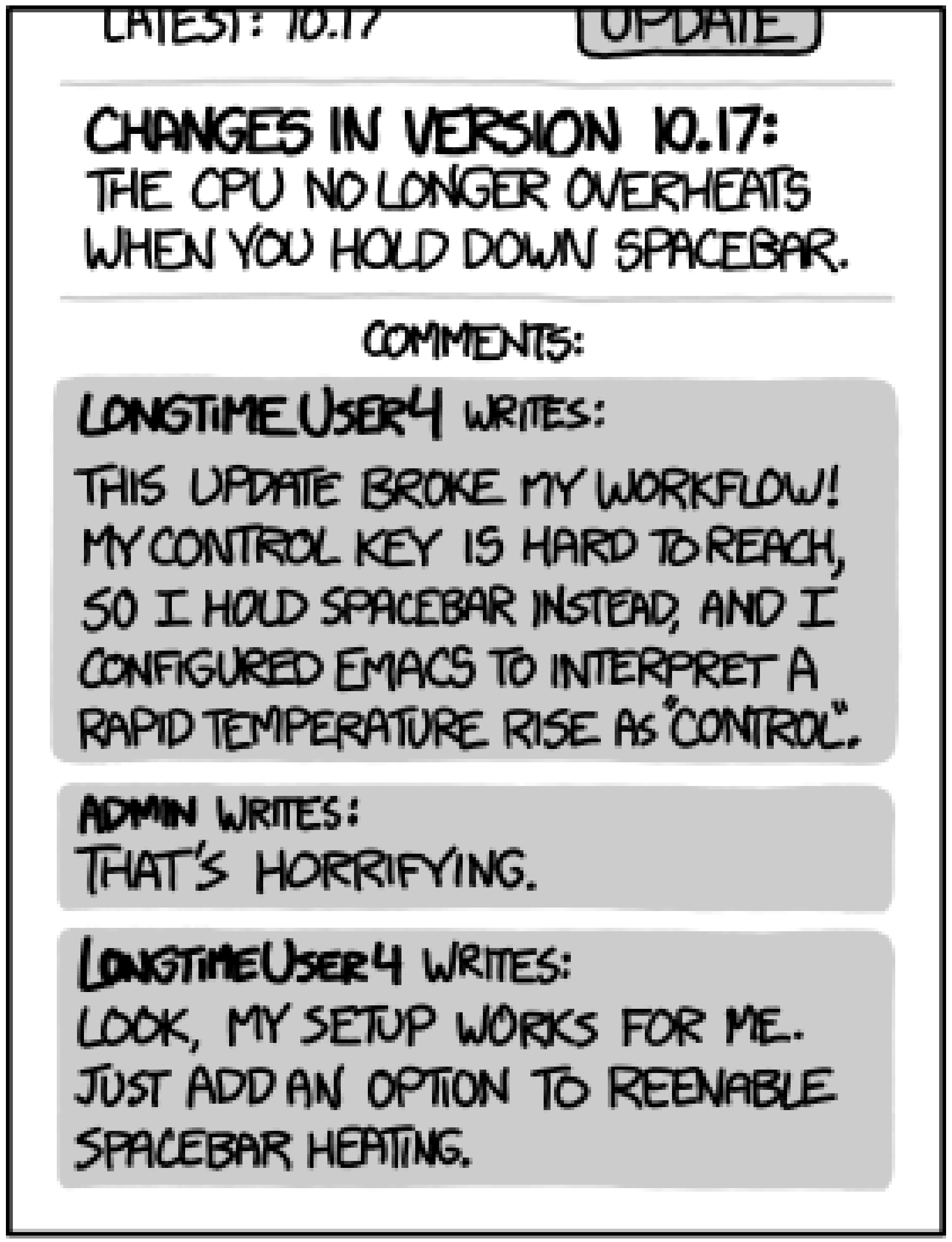
Maintenance performed to correct faults in software

→ Windows calculator showing wrong formulas

Preventive Maintenance

Maintenance performed for the purpose of preventing problems before they occur

→ Y2K bug, leap seconds/years


LATEST: 10.17
UPDATE
CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.
COMMENTS:
LONGTIME_USER4 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

Maintenance	Evolution
• mostly corrections	• new or removed functionality
• small changes	• large changes
• often unforeseen changes	• often foreseen changes
• results in patches and hot fixes (minor updates)	• results in upgrades or service packs (major updates, cumulative updates)
• minor release, new minor version: 2.3.0 → 2.3.1	• major release, new major version: 2.3.2 → 2.4.0

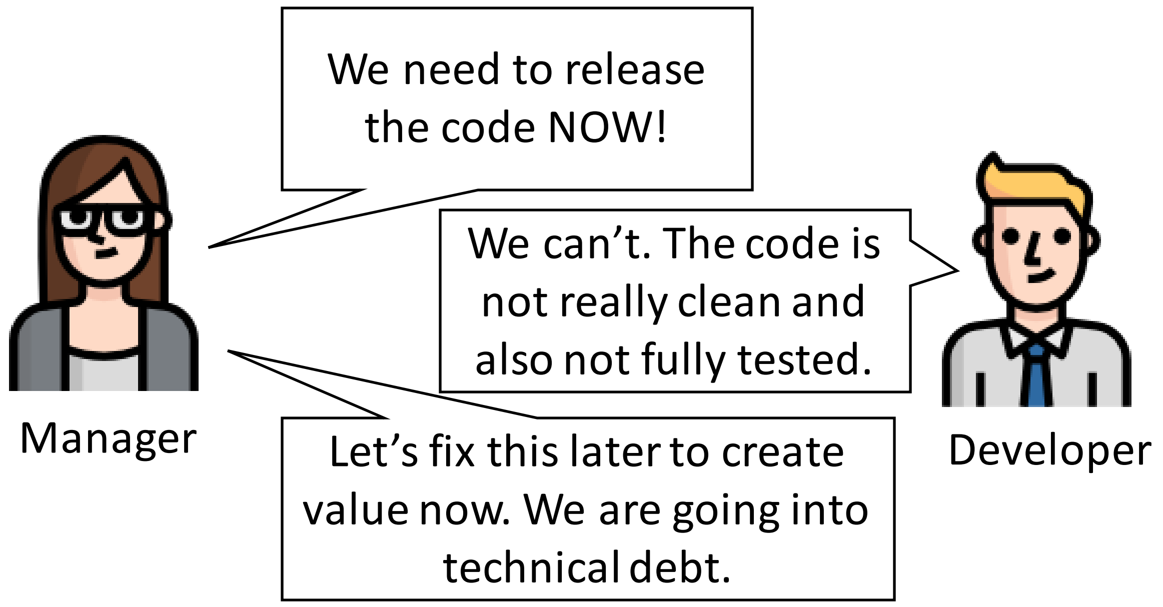
Technical Debt

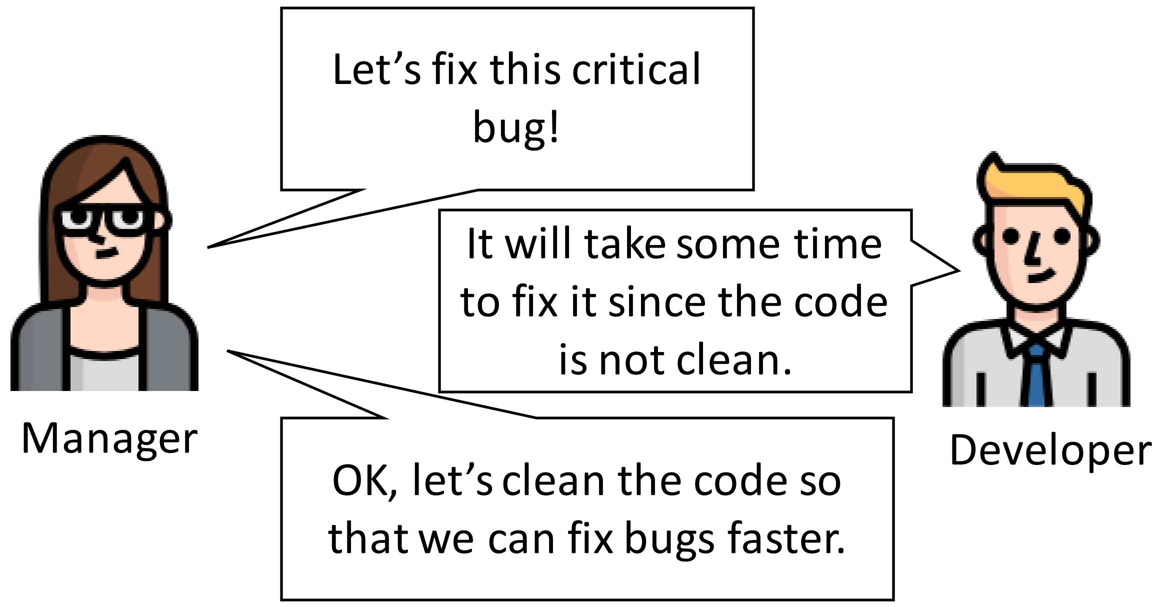
TECHNICAL DEBT
I DON'T UNDERSTAND WHY IT TAKES SO LONG TO ADD A NEW WINDOW.
@VINCENTDNL

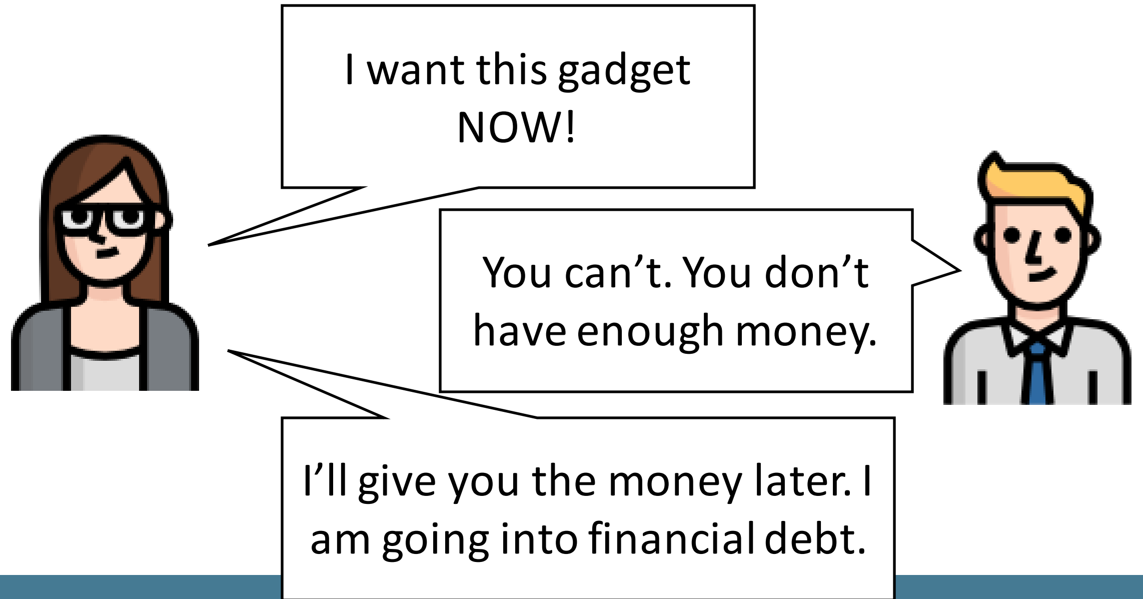
Universität zu Köln

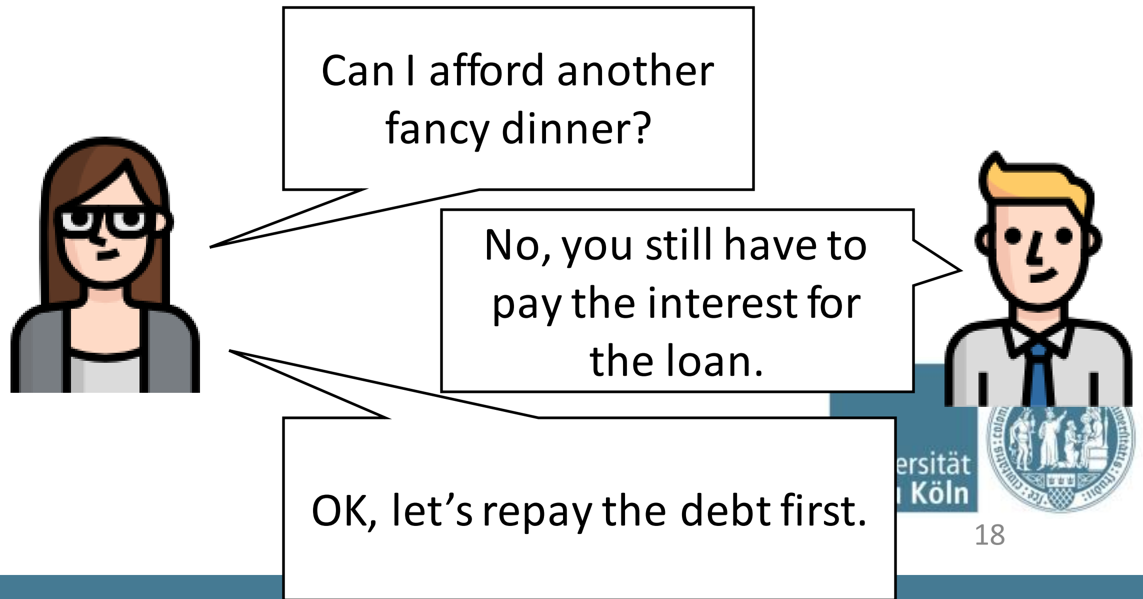
The Analogy

Later….


We need to release the code NOW!
We can't. The code is not really clean and also not fully tested.
Let's fix this later to create value now. We are going into technical debt.
Manager
Developer


Let's fix this critical bug!
It will take some time to fix it since the code is not clean.
OK, let's clean the code so that we can fix bugs faster.
Manager
Developer


I want this gadget NOW!
You can't. You don't have enough money.
I'll give you the money later. I am going into financial debt.


Can I afford another fancy dinner?
No, you still have to pay the interest for the loan.
OK, let's repay the debt first.

Technical Debt

Technical Debt

A metaphor to think about trading off **short-term benefits** with **later or long-term costs**.

Financial debt: borrow money against a future date (payback with interest)

Technical debt: borrow time against a future date (rework with additional effort)

TD is the gap between making a change **perfect** and making a change **work.**

Not all debt is bad

Technical debt isn't inherently bad. But, like financial debt, it can cause serious problems if you don't pay it back.

Elements of the Metaphor

Debt: A suboptimal solution being deployed (e.g., undocumented SW, untested SW, "hacks", hard to understand code)

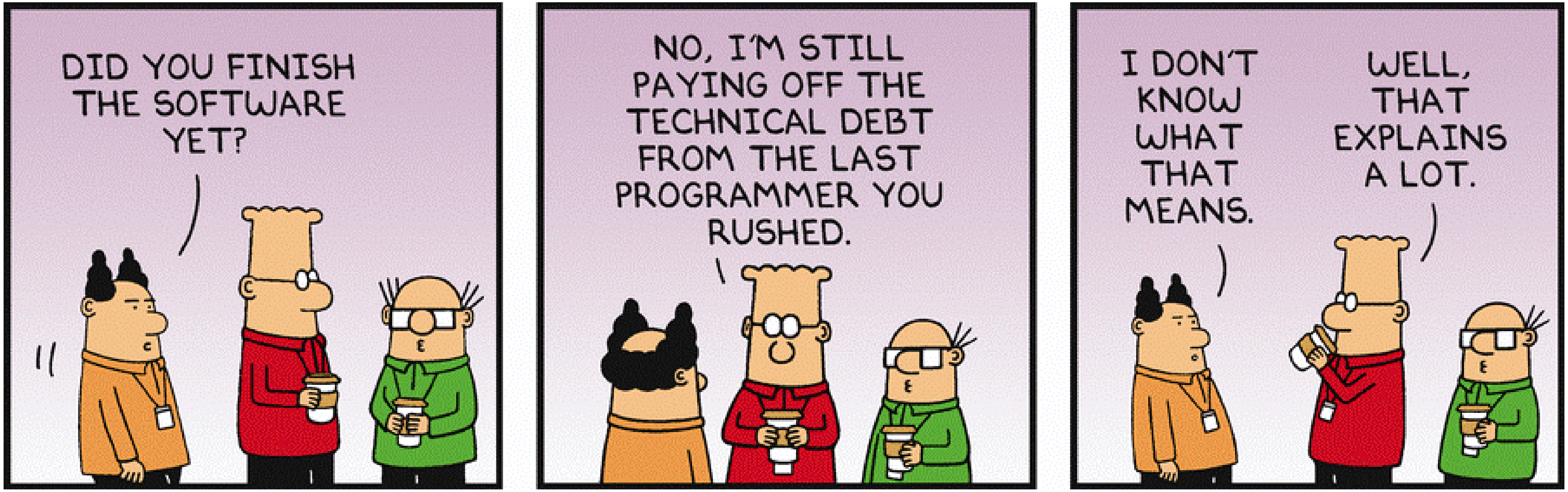
Interest rate: The amount of additional effort needed to perform tasks in the future (e.g., fix a bug, add a new feature)

Principal: The effort necessary to pay off the debt (e.g., refactoring the code, writing additional tests)

Value vs. debt over time

In general, the value of a feature **decreases** while the interest for a debt **increases** over time

Universität zu Köln


DID YOU FINISH THE SOFTWARE YET?
NO, I'M STILL PAYING OFF THE TECHNICAL DEBT FROM THE LAST PROGRAMMER YOU RUSHED.
I DON'T KNOW WHAT THAT MEANS.
WELL, THAT EXPLAINS A LOT.

Universität zu Köln

Reasons for Technical Debt

Reasons

Technology
- Technology limitations
- Legacy code
- Commercial off-the-shelf (COTS) components
- Changes in technology

Process
- Little consideration of code maintenance
- Unclear requirements
- Not knowing or adopting best practices

People
- Postpone work until needed
- Making bad assumptions
- Inexperience
- Poor leadership/team dynamics

Product
- Schedule and budget constraints
- Poor communication between developers and management
- Changing priorities (market information)

	reckless	**prudent**
deliberate	*“We don’t have time for design”*	*“We must ship now and deal with consequences”*
inadvertent	*“What’s Layering?”*	*“Now we know how we should have done it”*

Typical indicators

- “Don’t worry about the documentation for now.”
- “The only one who can change this code is Susan.”
- “It’s OK for now but we’ll refactor it later”
- “Let’s just copy and paste this part”
- “Does anybody know where we store the database access password?”
- “I know if I touch that code, everything else breaks”
- “Let’s finish the testing in the next release”

Managing Technical Debt

How to manage technical debt

- Defining and tracking technical debt
- Make technical debt visible by adding it to and prioritizing it in the backlog (see table on the right)
- For all deliberate and prudent technical debt, define measures to resolve the technical debt in the short or long-term
- Improve the general development process (automation, quality checks, quality awareness)

Types of tasks in a backlog

	visible	**invisible**
positive value	*New features, added functionality*	*Architectural, structural features*
negative value	*Defects*	*Technical debt*

Summary

SE Evolution and Maintenance

- For most SW systems, change is inevitable and leads to SW evolution.
- An evolving system increases its complexity unless work is done to reduce it.
- Although software does not wear out, it needs to be maintained to remain functional.

Case Studies from Student Projects

The scenarios are generalized and paraphrased

Scenario 1: “The Expert-Teams“

Planung:

- „Wir haben uns dazu entschieden **agil** vorzugehen. Daher haben wir folgendes geplant […]“
- „Aufgrund der Vorerfahrungen haben wir Teams gebildet. 2 Leute kümmern sich ums Frontend, 2 ums Backend und 2 um die Doku.“

Reflexion:

- „Wir mussten am Ende feststellen, dass im Frontend Dinge umgesetzt waren, die im Backend fehlten, außerdem passte die Doku nicht zur Implementierung“

What is (not) agile here?
Which (agile) practice would have helped?

Scenario 2: “The Big-Bang“

Planung:

- „Wir haben uns dazu entschieden **agil** vorzugehen. Daher haben wir folgendes geplant […]“
- „Aufgrund der knappen Zeit haben wir die Implementierung komplett in einem Sprint geplant“

Reflexion:

- „Am Ende haben wir es nicht mehr geschafft die Einzelteile des Systems (Frontend, Backend) zusammenzubauen. Wir haben unterschätzt wie viel Zeit die Integration braucht.“

What is (not) agile here?
Which (agile) practice would have helped?

Scenario 3: Der „WaterScrumFall“

Planung:

- „Wir haben uns dazu entschieden **agil** vorzugehen. Daher haben wir folgendes geplant […]“

Sprint 1: Planung	🟦				
Sprint 2: Implementierung		🟦	🟦	🟦	
- Backend		🟦	🟦	🟦	
- Frontend		🟦	🟦	🟦	
- Integration				🟦	
Sprint 3: Reflexion					🟦

Reflexion:

- „Wir haben den Aufwand für manche Aktivitäten massiv unterschätzt und konnten am Ende manche Dinge nicht mehr umsetzen.“

What is (not) agile here?
Which (agile) practice would have helped?

Scenario 4: “The perfect solution”

Planung:

- „Wir haben uns dazu entschieden **agil** vorzugehen. Daher haben wir folgendes geplant […]“
- “Wir haben im Team alle Anforderungen gesammelt, konkretisiert und in einem Backlog abgelegt und dann (für den Sprint) geplant“

Reflexion:

- „Wir mussten am Ende feststellen, dass der geplante Umfang völlig überdimensioniert war.“

What is (not) agile here?
Which (agile) practice would have helped?

Scenario 5: “Test-last”

Planung:

- „Wir haben uns dazu entschieden **agil** vorzugehen. Daher haben wir folgendes geplant […]“
- “Alle Features werden nach der Fertigstellung getestet“

Reflexion:

- „Am Ende hatten wir leider keine Zeit mehr zu testen. Teilweise konnten wir auch erst am Ende testen wie noch nicht alles fertig war“

What is (not) agile here?
Which (agile) practice would have helped?

Scenario 6: “Pseudo-Planning”

Planung:

- „Wir haben uns dazu entschieden **agil** vorzugehen. Daher haben wir folgendes geplant […]“

Reflexion:

- „Wir mussten feststellen, dass sich eigentlich keiner mehr die ursprüngliche Planung angeschaut hat. Wir haben einfach entwickelt. Teilweise fehlte daher der Überblick was schon fertig ist und was noch nicht.“

What is (not) agile here?
Which (agile) practice would have helped?

Scenario 7: “The Meeting Hell”

Planung:

- „Wir haben uns dazu entschieden **agil** vorzugehen. Daher haben wir folgendes geplant […]“
- „Wir machen täglich ein Daily Stand-up wie es SCRUM vorsieht“

Reflexion:

- „Tägliche Scrum Meetings waren zu eng getaktet. Daher alle 2 Tage. Teilweise dauerten die Meetings aber 2-3 Stunden. Die Meetings waren gut, es wurde alles besprochen. Manchmal wurden allerdings auch Details besprochen, die nicht für alle relevant waren. Manche Team Mitglieder waren sehr passiv.“

What is (not) agile here?
Which (agile) practice would have helped?