Empirical Software Engineering

What is it and why do we need it?

Daniel Méndez

🏛 Blekinge Institute of Technology, Sweden

🏛 fortiss GmbH, Germany

🌐 www.mendezfe.org

🐦 mendezfe

Ground rules

(1) Whenever you have questions / remarks,
please don’t ask Google, but
share them with the whole group.

(2) As we have to use zoom,
please feel free to interrupt me any time as I
might not see you raising your hand.

What could be wrong with such a study?

Research Question: Which car has the best driving performance?

H_0: There is no difference.

20 people without a driving licence participate.
They are taught to drive in a lecture of 2 hours.

Results: The BMW is significantly better than the Audi ($p<0.01$)

Empirical research is more than simply applying statistical equations

Adopted from: Dag I.K. Sjøberg, Keynote at the International Conference on Product-Focused SW Process Improvement 2016, Trondheim, Norway.
Image sources: Manufacturer websites

Goal of the lecture

What is this little thing called "Empirical Software Engineering"?

What we will discuss

Focus:
What & Why

- In a nutshell:
  - Broader perspective on Software Engineering (SE) as a scientific discipline
  - A few principles, concepts, and terms in Empirical SE
- Why we need empiricism in Software Engineering research
- What the perspectives are for the research community and for you

Basis for…

Focus:
How

- Course on research methods
- Master Thesis projects
- The time afterwards

Outline

- What is Empirical Software Engineering?

- Why do we need Empirical Software Engineering?

- What are the perspectives in Empirical Software Engineering?

Outline

- **What is Empirical Software Engineering?**

- Why do we need Empirical Software Engineering?

- What are the perspectives in Empirical Software Engineering?

Let’s start step by step….

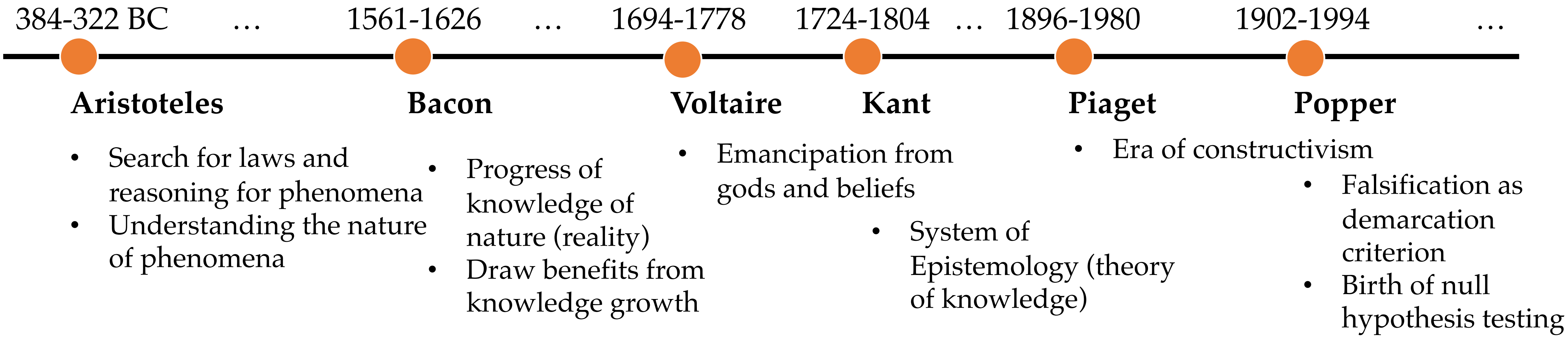
What is science?

-- What do you think? --

“Science” wasn’t built in a day…

> Science is understood as the human undertaking for the search of knowledge (through systematic application of scientific methods)

→ Needs to be considered in a historical context
→ Increased understanding of scientific practice (and what science eventually is)



384-322 BC … 1561-1626 … 1694-1778 1724-1804 … 1896-1980 1902-1994 …

Aristoteles

- Search for laws and reasoning for phenomena
- Understanding the nature of phenomena

Bacon

- Progress of knowledge of nature (reality)
- Draw benefits from knowledge growth

Voltaire

- Emancipation from gods and beliefs

Kant

- System of Epistemology (theory of knowledge)

Piaget

- Era of constructivism

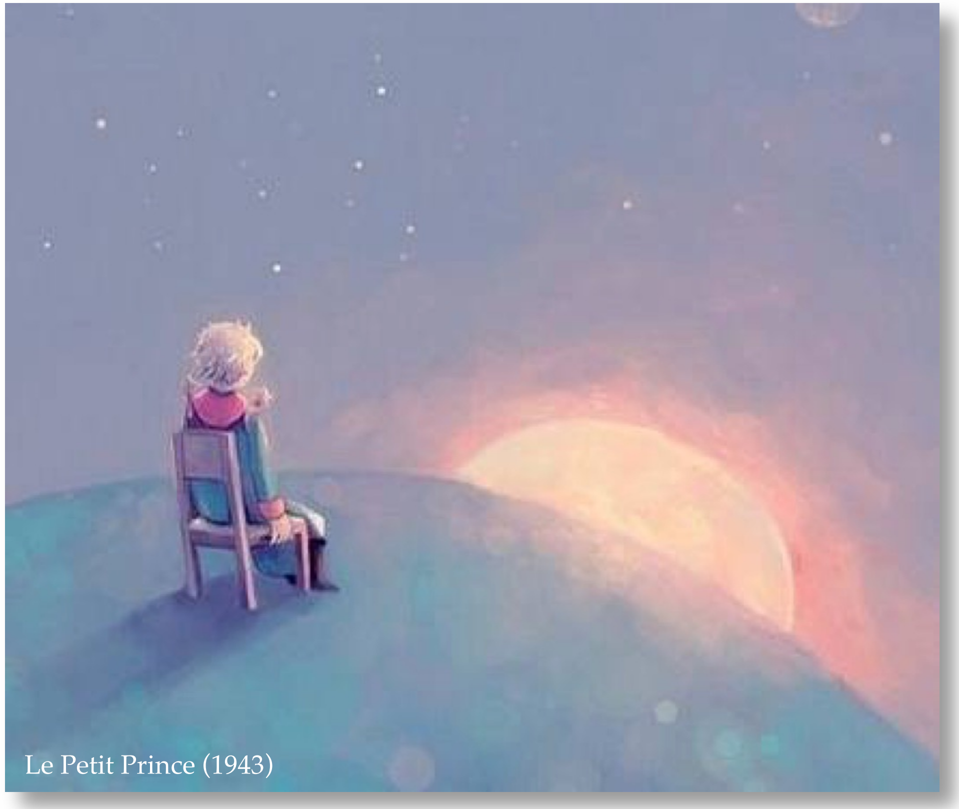
Popper

- Falsification as demarcation criterion
- Birth of null hypothesis testing

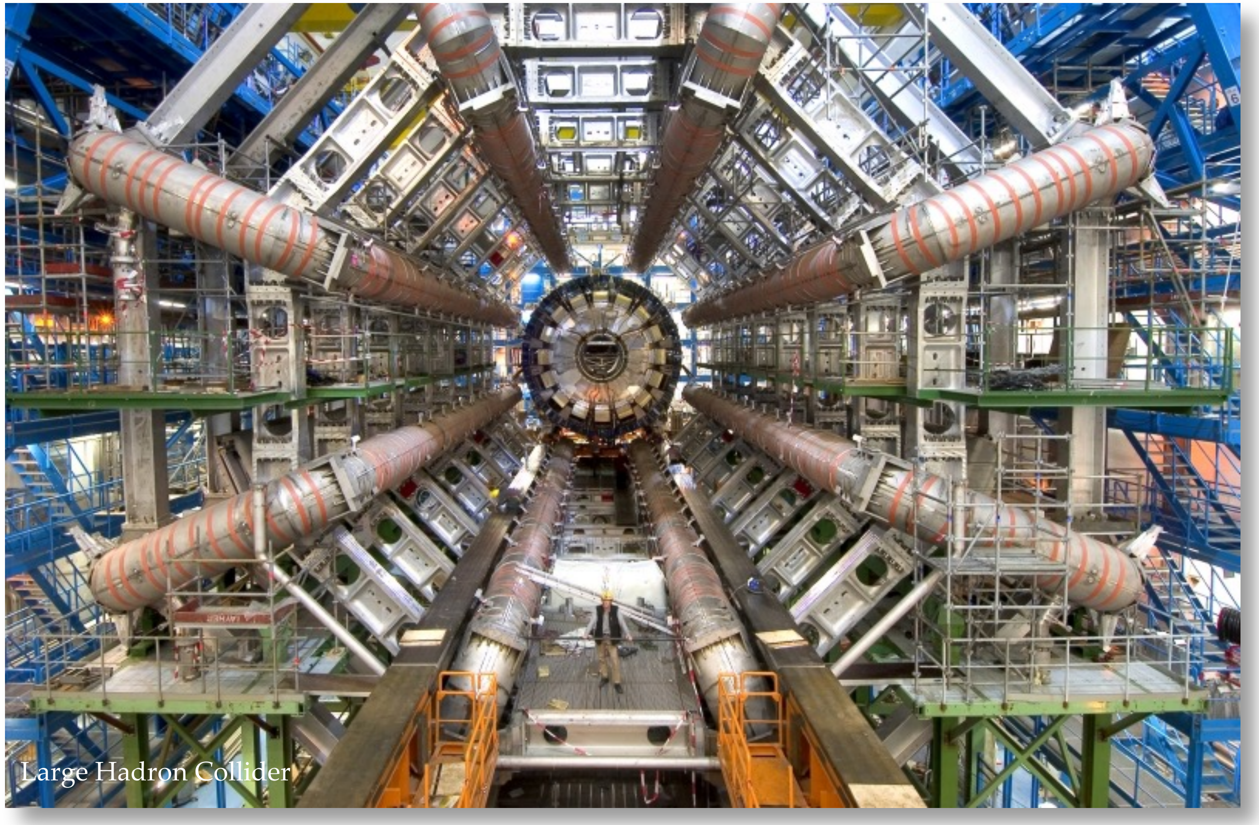
Scientific practices and research methods have changed over time, the role of empiricism* not

384-322 BC

Today



Le Petit Prince (1943)



Large Hadron Collider

* Gaining knowledge through sensory experiences

Scientific knowledge and practice

> Scientific knowledge is the portrait of
> our understanding of reality (via scientific theories).

Necessary postulates for scientific practice (selected):

- There are certain rules, principles, and norms for scientific practices
  - Rationalism: Reasoning by argument / logical inference / mathematical proof
  - Empiricism: Reasoning by sensory experiences (case studies, experiments,…)
- There is nothing absolute about truth
- There is a scientific community to judge about the quality of empirical studies

What is Software Engineering research?
Is it scientific?

-- What do you think? --

Different purposes in science

Basic Science

- Gaining and validating new insights
- Often theoretical character
- Typically addressed by natural and social sciences

Applied Science

- Applying scientific methods to practical ends
- Often practical (& pragmatic) character
- Typically addressed by engineering disciplines

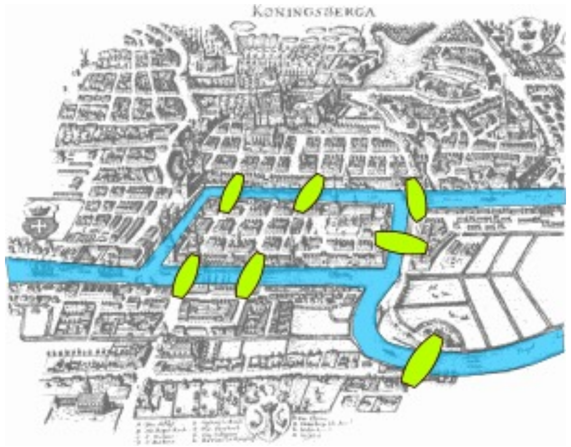
In software engineering research, we

- apply scientific methods to practical ends (treating design science problems)
- treat insight-oriented questions, thus, we are an insight-oriented science, too

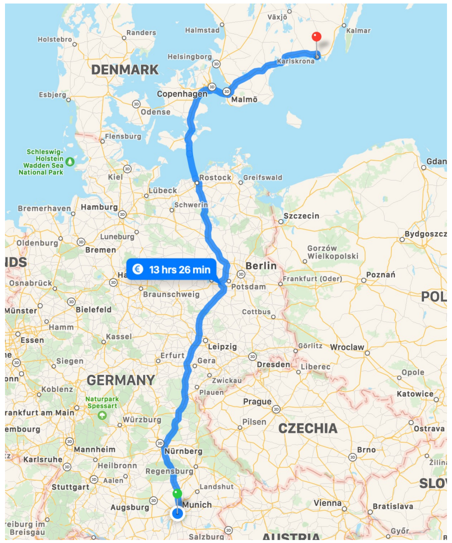
Different purposes in science

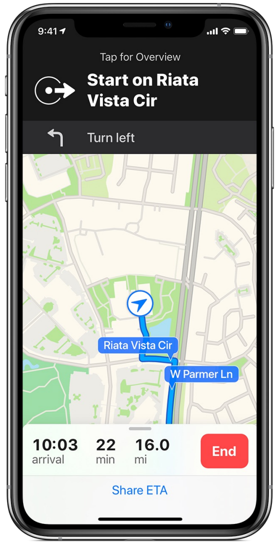
Basic Science

Applied Science



* Graph theory
(Königsberg Bridge problem)





Fundamental / basic research

Applied research

Image Sources: Wikipedia (l), Apple (r)

What is Software Engineering research?
Is it scientific?

-- Yes, Software Engineering research is scientific! --

Empirical Software Engineering

> The ultimate goal of Empirical Software Engineering is to advance our body of knowledge by building and evaluating theories.

Relevance from a theoretical and practical perspective:

- Reason about the discipline and (e.g. social) phenomena involved
- Recognise and understand limitations and effects of artefacts (e.g. by evaluating technologies, techniques, processes, models, etc.) in their practical contexts

But what is a Scientific Theory?

-- What do you think? --

Theories (generally speaking)

> **A theory** is a belief that there is a pattern in phenomena.

Examples (following this general notion of theory):

- “Earth is flat”
- “Vaccinations lead to autism”
- “Wearing face masks does more harm than the effects of COVID-19”
- …

Are these theories scientific?

No: Speculations based on narrow views, imagination, and hopes and fears – often resulting in opinions that cannot be refuted (i.e. logical fallacies)

Scientific Theories

A scientific theory is a belief that there is a pattern in phenomena while having survived

1. tests against sensory experiences
2. criticism by critical peers

Note: Addresses so-called **Demarcation Problem** to distinguish science from non-science (as per introduction by K. Popper)

1. Tests

- Experiments, simulations, …
- Replications

In scope of empirical research methods (but out of scope for today)

2. Criticism

- Peer reviews / acceptance in the community
- Corroborations / extensions with further theories

Scientific Theories have…

… a purpose:

	Analytical	Explanatory	Predictive	Explanatory & Predictive
Scope	Descriptions and conceptualisation, including taxonomies, classifications, and ontologies - What is?	Identification of phenomena by identifying causes, mechanisms or reasons - Why is?	Prediction of what will happen in the future - What will happen?	Prediction of what will happen in the future and explanation - What will happen and why?

… quality criteria:

- Testability
- Level of confidence („relation to existing evidence“
- Usefulness to researchers and practitioners („impact and implications“)
- …

Note: Law “versus” Theory
A law is a descriptive theory without explanations (i.e. an analytical theory)

Adapted from: Sjøberg, D., Dybå, T., Anda, B., Hannay, J. Building Theories in Software Engineering, 2010.
Further information on: https://goo.gl/SQQwxt

Exemplary framework for describing theories in Software Engineering

- **Constructs:** What are the basic elements?
  (Actors, technologies, activities, system entities, context factors)
- **Propositions:** How do the constructs interact?
- **Explanations:** Why are the propositions as specified?
- **Scope:** What is the universe of discourse in which the theory is applicable?

Source (framework): Sjøberg, D., Dybå, T., Anda, B., Hannay, J. Building Theories in Software Engineering, 2010.

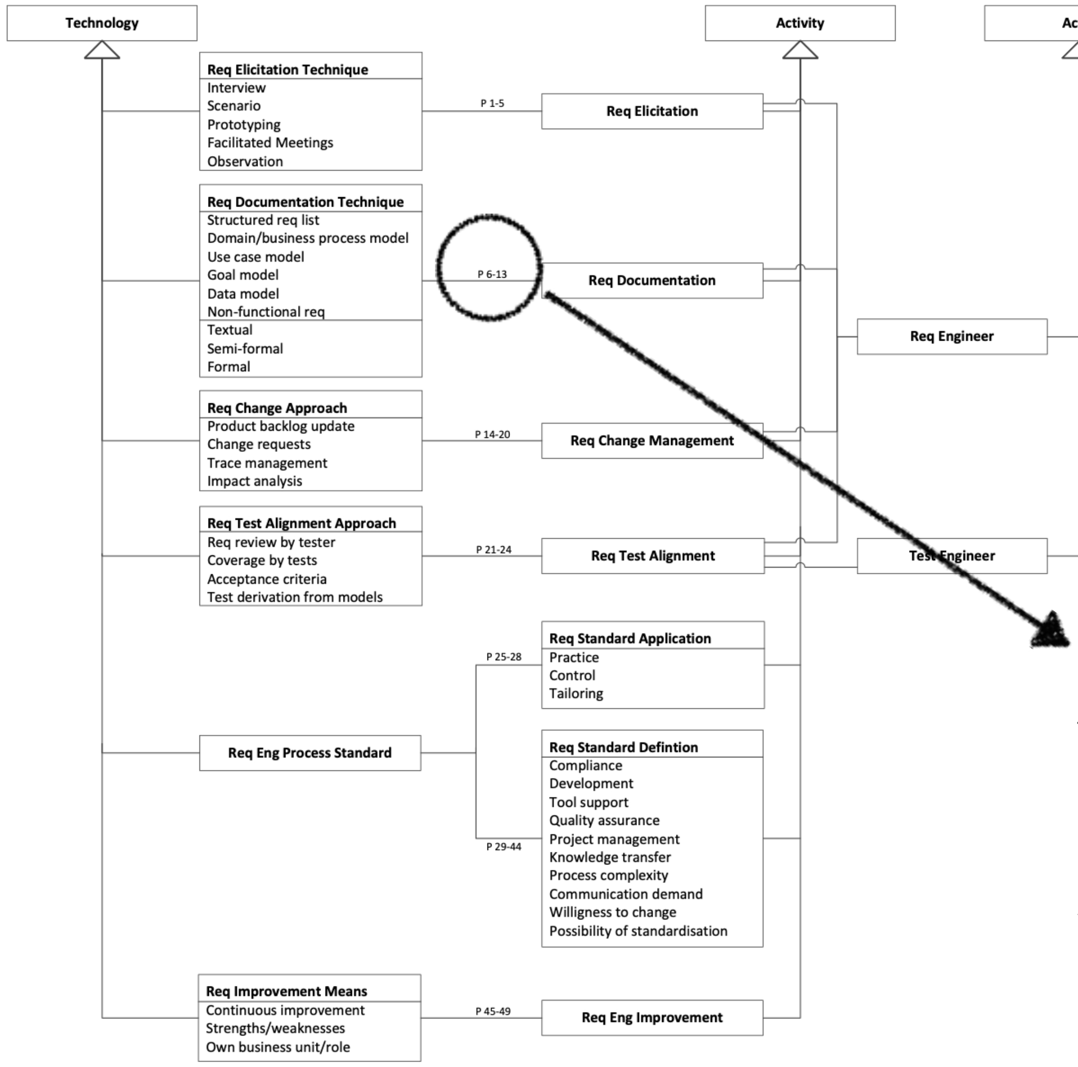
Source (example): Wagner, Mendez et al. Status Quo in Requirements Engineering: A Theory and a Global Family of Surveys, TOSEM 2018.

Exemplary framework for describing theories

xt factors)

theory is

Example



Proposition:
"Structured requirements lists are documented textually in free form or textually with constraints."

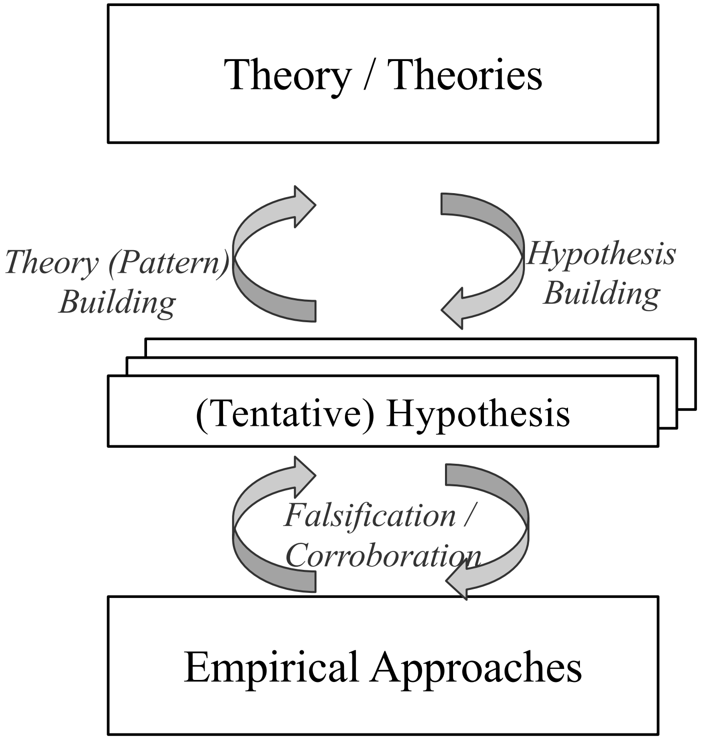
Explanation and Scope:
"Free-form and constraint textual requirements are sufficient for many contexts such as in agile projects where they only act as reminders for further conversations."

Source (framework): Sjøberg, D., Dybå, T., Anda, B., Hannay, J. Building Theories in Software Engineering, 2010.

Source (example): Wagner, Mendez et al. Status Quo in Requirements Engineering: A Theory and a Global Family of Surveys, TOSEM 2018.

Theories and hypotheses

Note: We don't "test theories", but their consequences via hypotheses (i.e. testable propositions)



Scientific theory

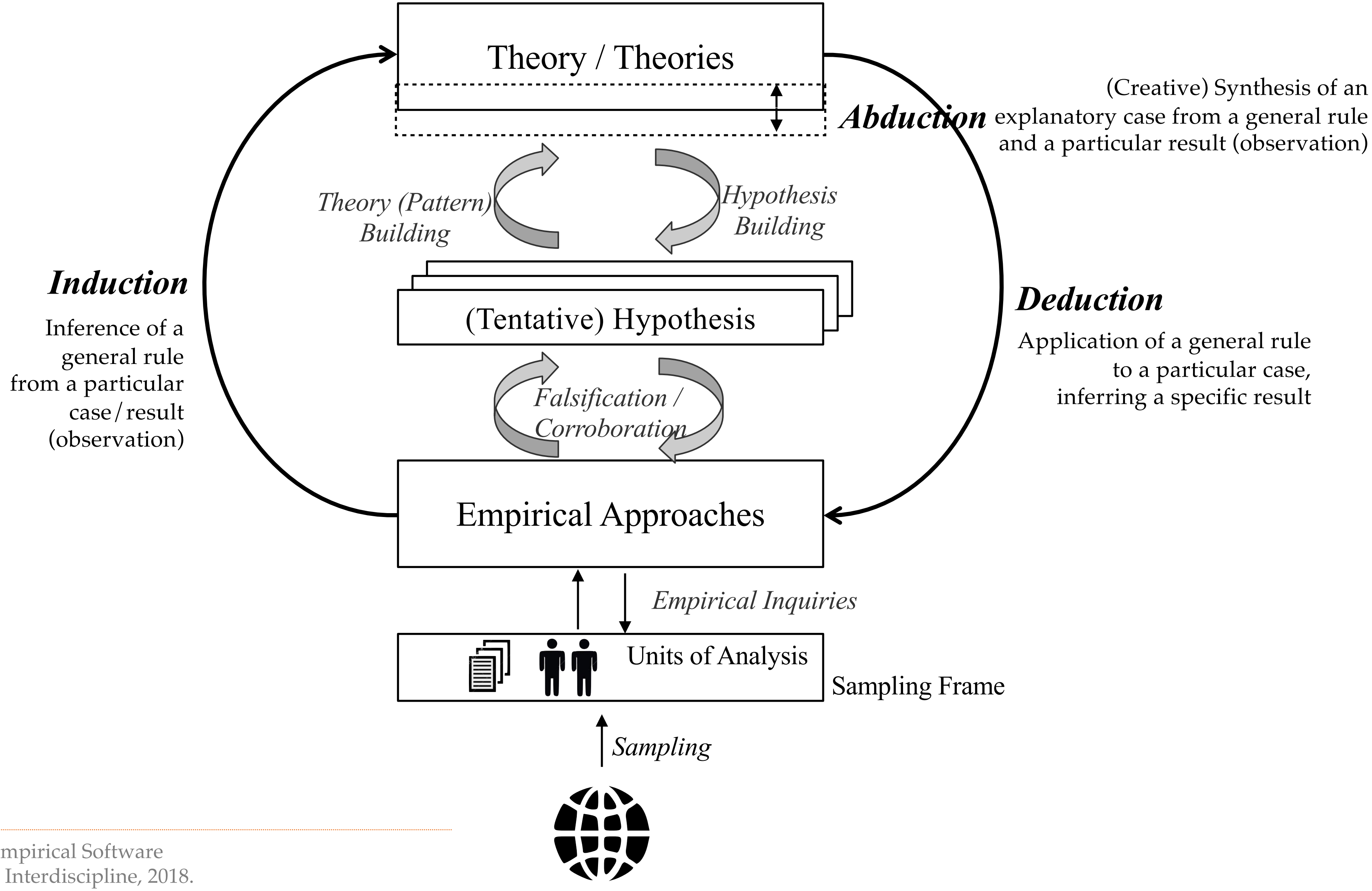
- "[…] based on hypotheses tested and verified multiple times by detached researchers" (J. Bortz and N. Döring, 2003)

Hypothesis

- "[…] a statement that proposes a possible explanation to some phenomenon or event" (L. Given, 2008)
- Grounded in theory, testable and falsifiable
- Often quantified and written as a conditional statement

> **If cause/assumption** (independent variables)
> *then* (=>) **consequence** (dependent variables)

From real world phenomena to theories and back: The empirical life cycle

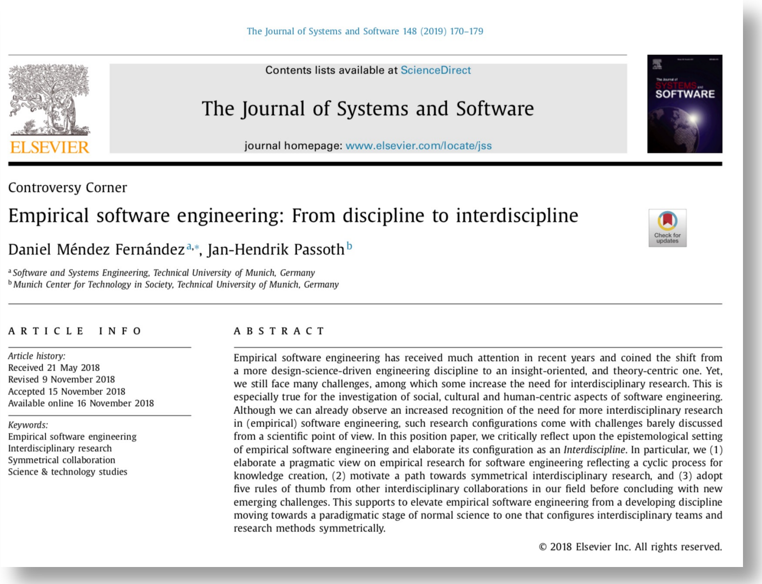


Theory / Theories
(Creative) Synthesis of an explanatory case from a general rule and a particular result (observation)
Abduction
Theory (Pattern) Building
Hypothesis Building
Induction
Inference of a general rule from a particular case/result (observation)
(Tentative) Hypothesis
Deduction
Application of a general rule to a particular case, inferring a specific result
Falsification / Corroboration
Empirical Approaches
Empirical Inquiries
Units of Analysis
Sampling Frame
Sampling

Source: Mendez and Passoth. Empirical Software Engineering: from Discipline to Interdiscipline, 2018.

From real world phenomena to theories and back: The empirical life cycle

Further reading and outlook



The Journal of Systems and Software 148 (2019) 170–179

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Controversy Corner

Empirical software engineering: From discipline to interdiscipline

Daniel Méndez Fernández[a,*], Jan-Hendrik Passoth[b]

[a] Software and Systems Engineering, Technical University of Munich, Germany
[b] Munich Center for Technology in Society, Technical University of Munich, Germany

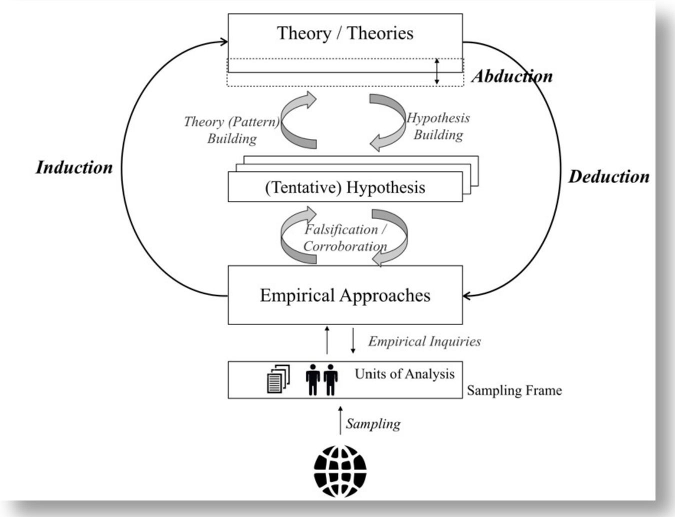
ARTICLE INFO

Article history:
Received 21 May 2018
Revised 9 November 2018
Accepted 15 November 2018
Available online 16 November 2018

Keywords:
Empirical software engineering
Interdisciplinary research
Symmetrical collaboration
Science & technology studies

ABSTRACT

Empirical software engineering has received much attention in recent years and coined the shift from a more design-science-driven engineering discipline to an insight-oriented, and theory-centric one. Yet, we still face many challenges, among which some increase the need for interdisciplinary research. This is especially true for the investigation of social, cultural and human-centric aspects of software engineering. Although we can already observe an increased recognition of the need for more interdisciplinary research in (empirical) software engineering, such research configurations come with challenges barely discussed from a scientific point of view. In this position paper, we critically reflect upon the epistemological setting of empirical software engineering and elaborate its configuration as an *Interdiscipline*. In particular, we (1) elaborate a pragmatic view on empirical research for software engineering reflecting a cyclic process for knowledge creation, (2) motivate a path towards symmetrical interdisciplinary research, and (3) adopt five rules of thumb from other interdisciplinary collaborations in our field before concluding with new emerging challenges. This supports to elevate empirical software engineering from a developing discipline moving towards a paradigmatic stage of normal science to one that configures interdisciplinary teams and research methods symmetrically.



Preprint: https://arxiv.org/abs/1805.08302

- Epistemological setting of Empirical Software Engineering
- Theory building and evaluation
- Challenges in Empirical Software Engineering

duction

(Creative) Synthesis of an explanatory case from a general rule and a particular result (observation)

Deduction

Application of a general rule to a particular case, inferring a specific result

pling Frame

Engineering: from Discipline to Interdiscipline, 2018.

Outline

- What is Empirical Software Engineering?
- **Why do we need Empirical Software Engineering?**
- What are the perspectives in Empirical Software Engineering?

RECAP

Empirical Software Engineering

> The ultimate goal of Empirical Software Engineering is to advance our body of knowledge by building and evaluating theories.

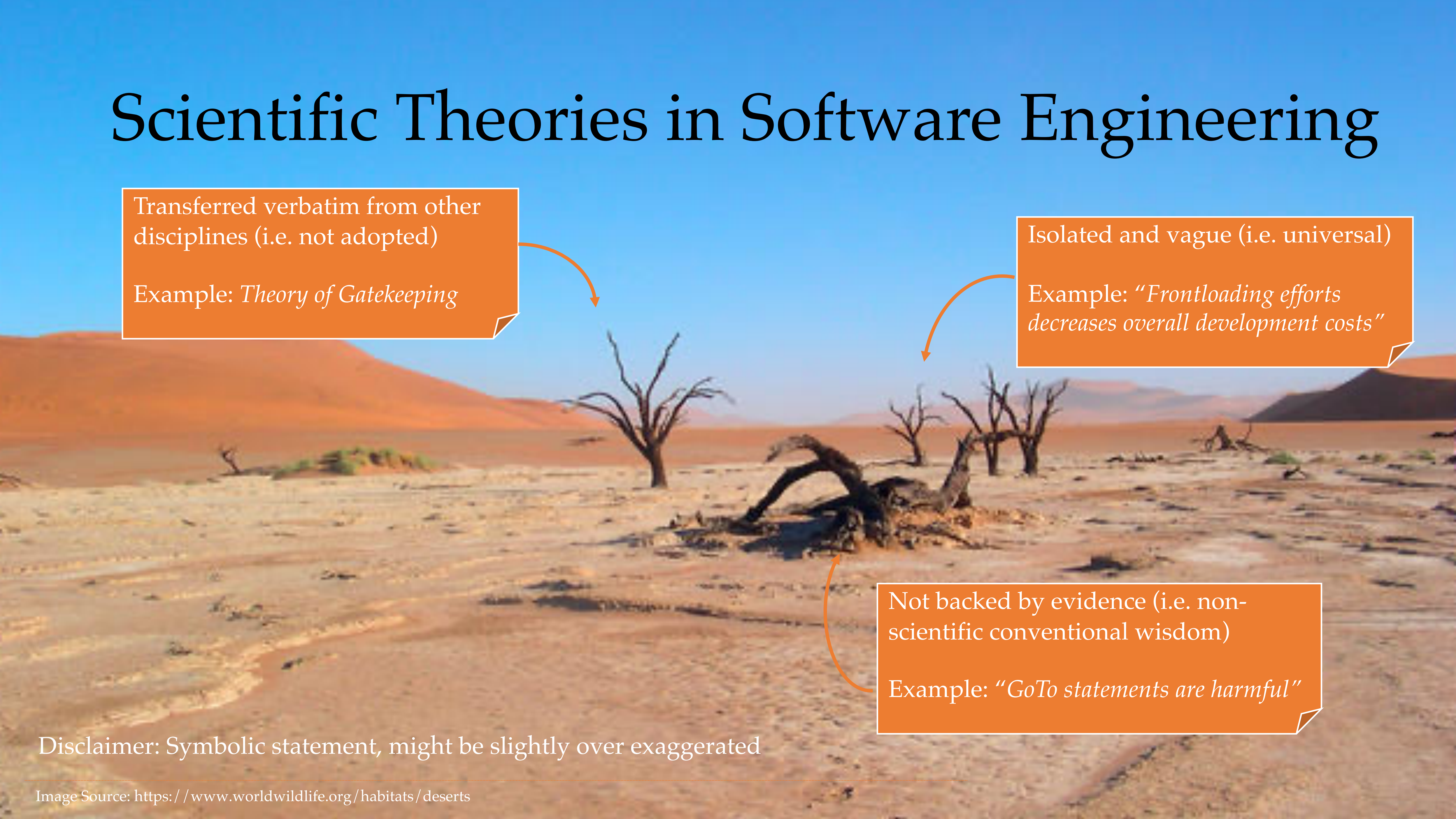
Relevance from a theoretical and practical perspective:

- Reason about the discipline and (e.g. social) phenomena involved
- Recognise and understand limitations and effects of artefacts (e.g. by evaluating technologies, techniques, processes, models, etc.) in their practical contexts

What are exemplary scientific Software Engineering Theories?

-- Which ones do you know? --

Scientific Theories in Software Engineering



Transferred verbatim from other disciplines (i.e. not adopted)

Example: *Theory of Gatekeeping*

Isolated and vague (i.e. universal)

Example: *“Frontloading efforts decreases overall development costs”*

Not backed by evidence (i.e. non-scientific conventional wisdom)

Example: *“GoTo statements are harmful”*

Disclaimer: Symbolic statement, might be slightly over exaggerated

Image Source: https://www.worldwildlife.org/habitats/deserts

Current state of evidence in Software Engineering

"[…] judging a theory by assessing the number, faith, and vocal energy of its supporters […] basic political credo of contemporary religious maniacs"

— Imre Lakatos, 1970

* Addressing the situation in the quantum mechanics research community, an analogy

Example: Goal-oriented RE

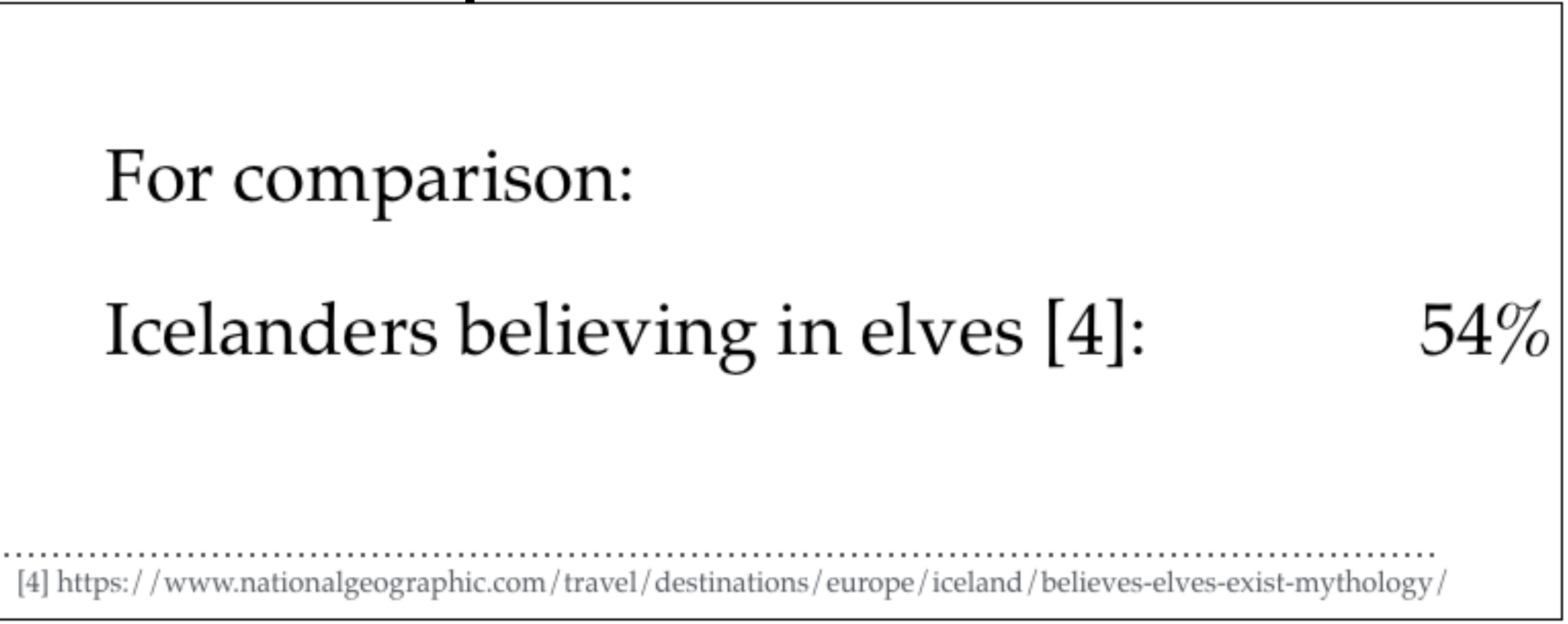
Papers published [1]:	**966**
Papers including a case study [1]:	**131**
Studies involving practitioners [2]:	**20**
Practitioners actually using GORE [3]:	~ **5%**

[1] Horkoff et al. Goal-Oriented Requirements Engineering: A Systematic Literature Map, 2016

[2] Mavin, et al. Does Goal-Oriented Requirements Engineering Achieve its Goal?, 2017

[3] Mendez et al. Naming the Pain in Requirements Engineering Initiative – www.napire.org

Example: Goal-oriented RE



For comparison:

Icelanders believing in elves [4]: 54%

[4] https://www.nationalgeographic.com/travel/destinations/europe/iceland/believes-elves-exist-mythology/

Goal-Oriented Requirements Engineering: A Guided Tour

Axel van Lamsweerde

Practitioners actually using GORE [3]: ~ **5**%

[1] Horkoff et al. Goal-Oriented Requirements Engineering: A Systematic Literature Map, 2016

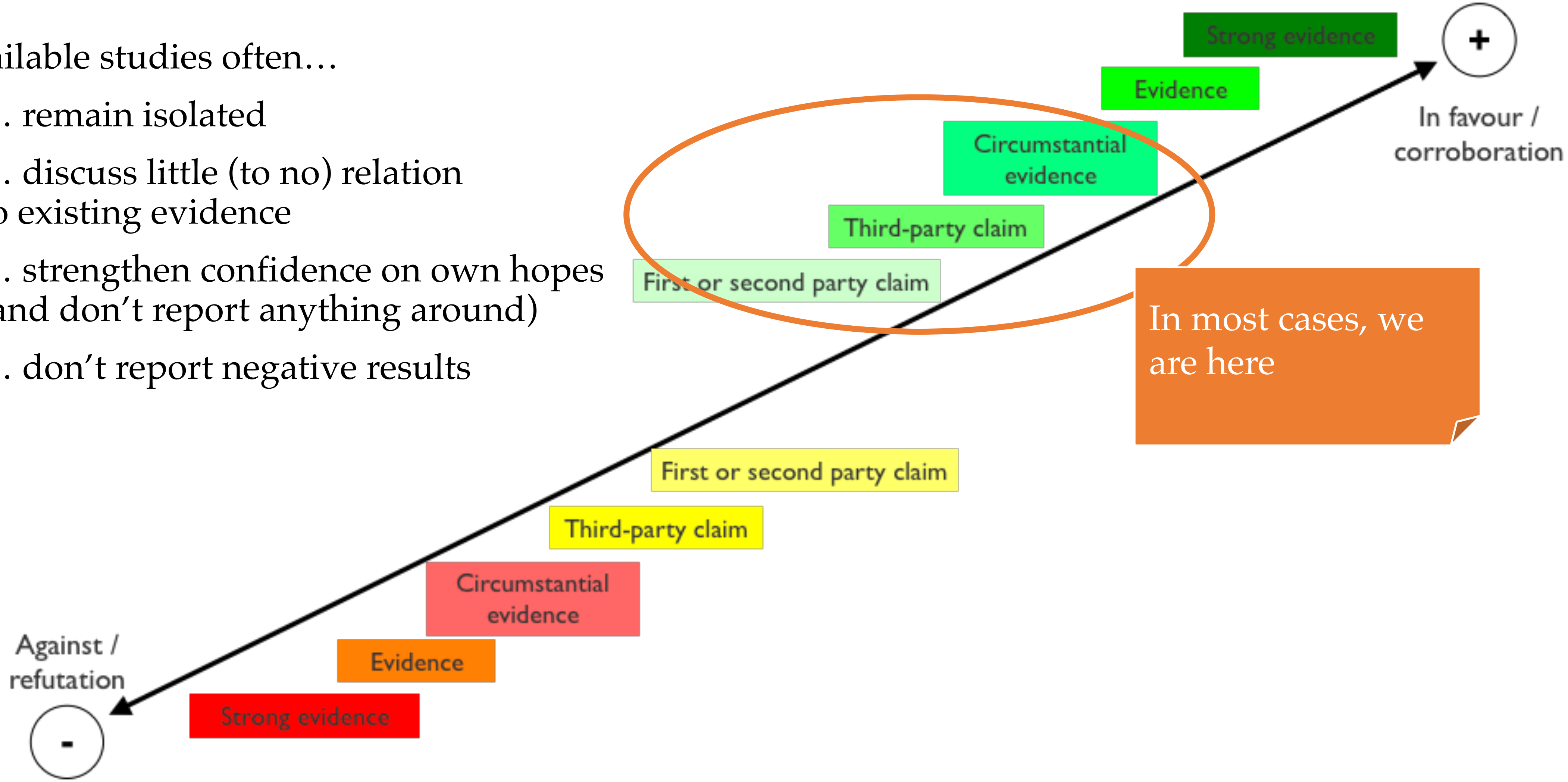
[2] Mavin, et al. Does Goal-Oriented Requirements Engineering Achieve its Goal?, 2017

[3] Mendez et al. Naming the Pain in Requirements Engineering Initiative – www.napire.org

Current state of evidence in Software Engineering

Available studies often…

- … remain isolated
- … discuss little (to no) relation to existing evidence
- … strengthen confidence on own hopes (and don't report anything around)
- … don't report negative results



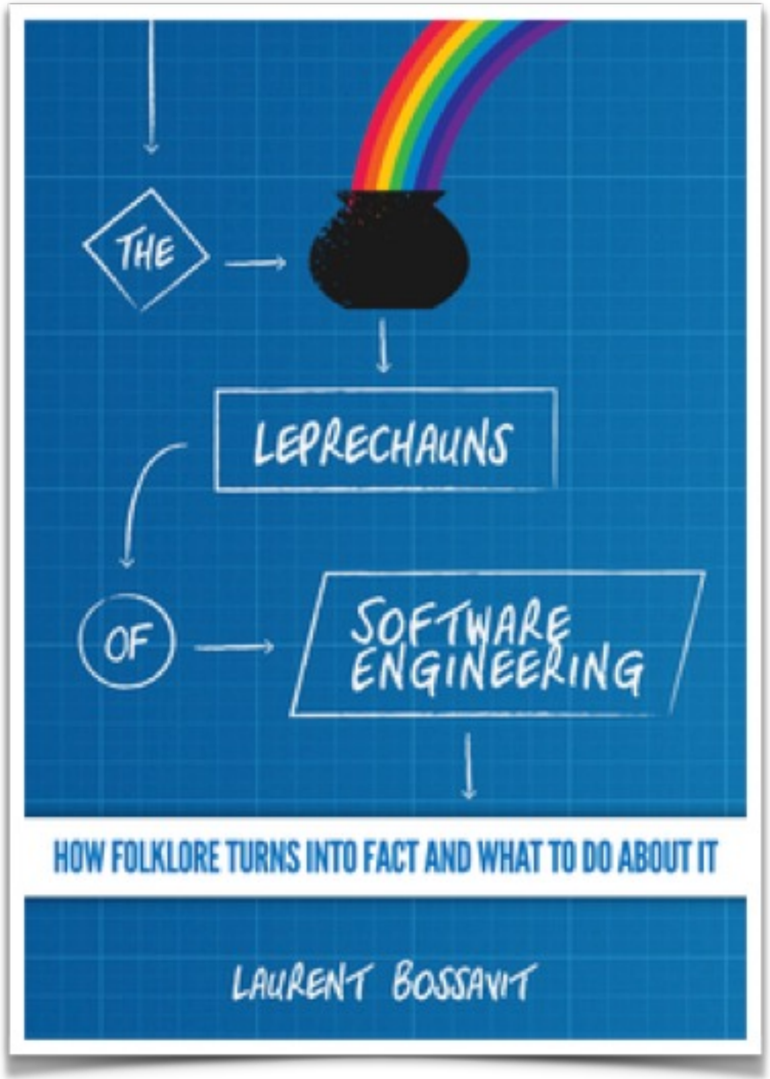
In most cases, we are here

Source (levels of evidence): Wohlin. An Evidence Profile for Software Engineering Research and Practice, 2013.

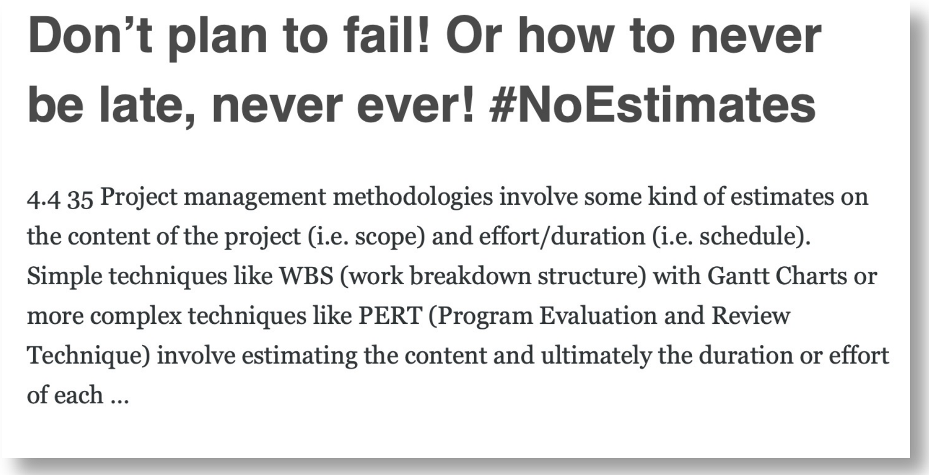
Conventional Wisdom in SE

“Leprechauns”: Folklore turned into facts

- Emerge from times where claims by authorities were treated as “facts”
- Reasons manifold:
  - Lack of empirical awareness
  - Neglecting particularities of practical contexts
  - Neglecting relation to existing evidence
  - No proper citations (one side of the medal, over-conclusions, etc.)
  - Lack of data
  - …



Exemplary symptoms: #*NoEstimates*



Don’t plan to fail! Or how to never be late, never ever! #NoEstimates

4.4 35 Project management methodologies involve some kind of estimates on the content of the project (i.e. scope) and effort/duration (i.e. schedule). Simple techniques like WBS (work breakdown structure) with Gantt Charts or more complex techniques like PERT (Program Evaluation and Review Technique) involve estimating the content and ultimately the duration or effort of each ...

„The key insight is this: spiraling delays are normal in projects. They are entirely predictable (as in we know ***they will happen****), but also entirely unpredictable (as in* ***we don’t know which delays will spiral out of control****). So we must prepare for them. […] core principle of #NoEstimates: Always be ready to stop the project and deliver value, at any time.”*

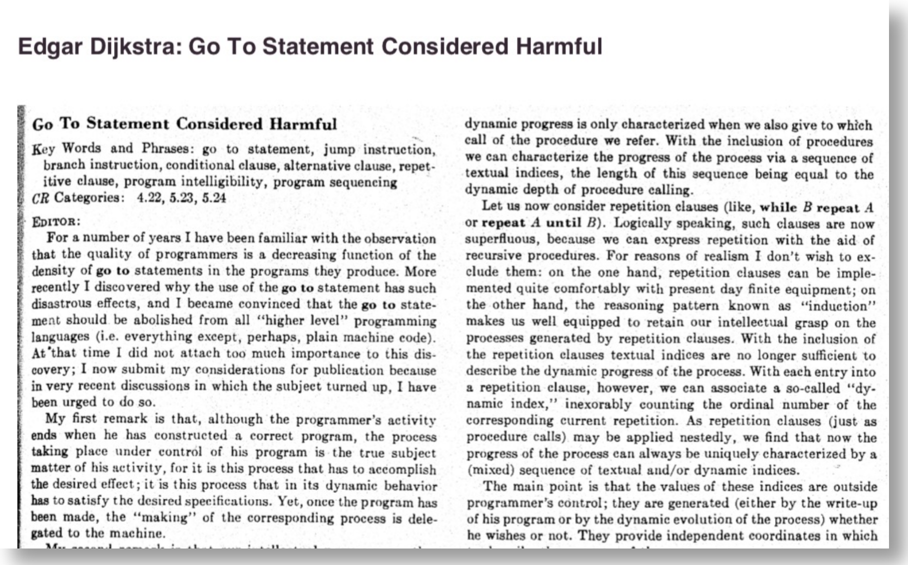
- What are benefits and drawbacks of #noestimates?
- What are project circumstances/ characteristics under which such philosophy applies?

> So far: no evidence, no trade-offs, no balanced discussion, but only rather “religious” discussions.

Source (first post on this topic): http://zuill.us/WoodyZuill/2012/12/10/no-estimate-programming-series-intro-post/

Exemplary “leprechaun”: *Go To statements considered harmful*

1968



Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all “higher level” programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the “making” of the corresponding process is delegated to the machine.

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as “induction” makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called “dynamic index,” inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

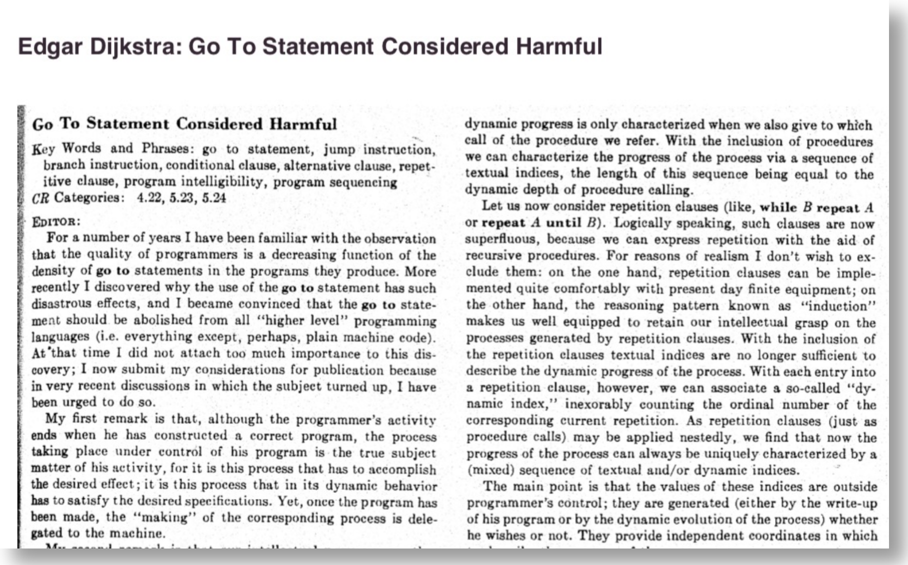
The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which

- Public exchange based on reasoning by argument (rationalist arguments)...
- … finally tackled by one empirical study nearly 50 years later.

[1] Edsger Dijkstra . Go To Statement Considered Harmful. Communications of the ACM, 1968.
[2] Frank Rubin. ”GOTO Considered Harmful" Considered Harmful. Communications of the ACM, 1969.
[3] Donald Moore et al. " 'GOTO Considered Harmful' Considered Harmful" Considered Harmful?" Communications of the ACM, 1987.
[4] Nagappan et al. An empirical study of goto in C code from GitHub repositories, 2015.

Exemplary "leprechaun": *Go To statements considered harmful*

1968



Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

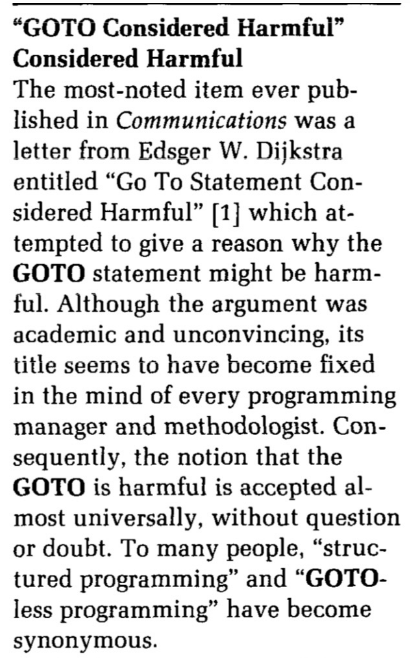
My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which

1969



"GOTO Considered Harmful" Considered Harmful

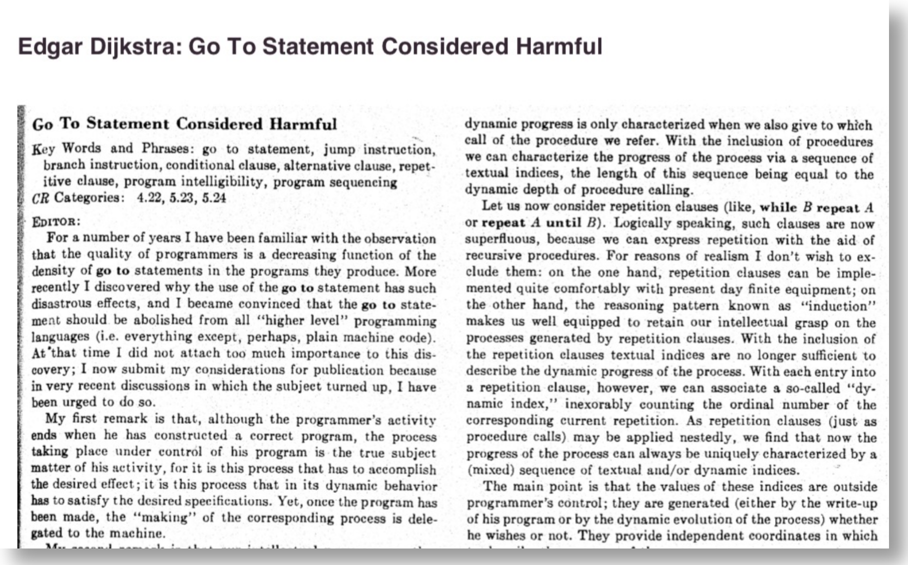
The most-noted item ever published in *Communications* was a letter from Edsger W. Dijkstra entitled "Go To Statement Considered Harmful" [1] which attempted to give a reason why the **GOTO** statement might be harmful. Although the argument was academic and unconvincing, its title seems to have become fixed in the mind of every programming manager and methodologist. Consequently, the notion that the **GOTO** is harmful is accepted almost universally, without question or doubt. To many people, "structured programming" and "**GOTO**-less programming" have become synonymous.

- Public exchange based on reasoning by argument (rationalist arguments)...
- … finally tackled by one empirical study nearly 50 years later.

[1] Edsger Dijkstra . Go To Statement Considered Harmful. Communications of the ACM, 1968.
[2] Frank Rubin. "GOTO Considered Harmful" Considered Harmful. Communications of the ACM, 1969.
[3] Donald Moore et al. " 'GOTO Considered Harmful' Considered Harmful" Considered Harmful?" Communications of the ACM, 1987.
[4] Nagappan et al. An empirical study of goto in C code from GitHub repositories, 2015.

Exemplary “leprechaun”:
Go To statements considered harmful

1968



Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all “higher level” programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

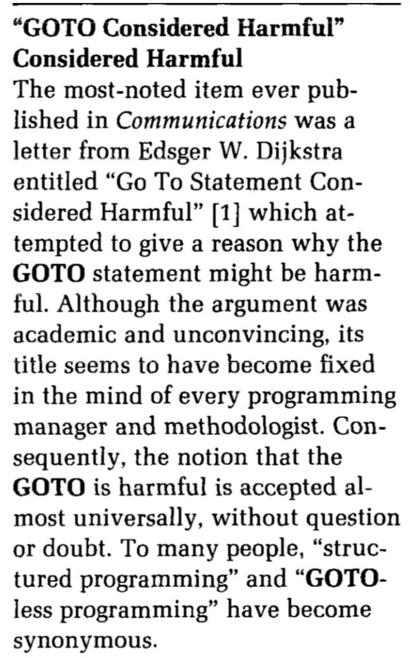
My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the “making” of the corresponding process is delegated to the machine.

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as “induction” makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called “dynamic index,” inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which

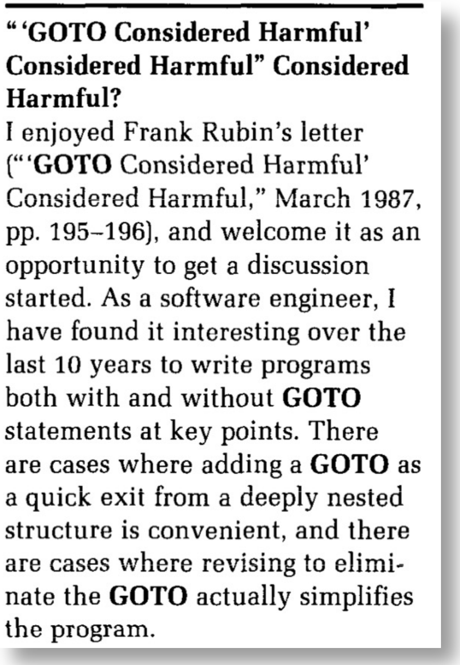
1969



“GOTO Considered Harmful” Considered Harmful

The most-noted item ever published in *Communications* was a letter from Edsger W. Dijkstra entitled “Go To Statement Considered Harmful” [1] which attempted to give a reason why the **GOTO** statement might be harmful. Although the argument was academic and unconvincing, its title seems to have become fixed in the mind of every programming manager and methodologist. Consequently, the notion that the **GOTO** is harmful is accepted almost universally, without question or doubt. To many people, “structured programming” and “**GOTO**-less programming” have become synonymous.

1987



“‘GOTO Considered Harmful’ Considered Harmful” Considered Harmful?

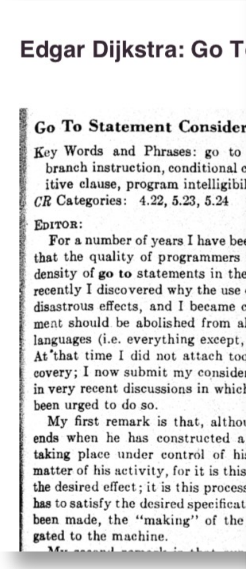
I enjoyed Frank Rubin's letter (“‘**GOTO** Considered Harmful’ Considered Harmful,” March 1987, pp. 195–196), and welcome it as an opportunity to get a discussion started. As a software engineer, I have found it interesting over the last 10 years to write programs both with and without **GOTO** statements at key points. There are cases where adding a **GOTO** as a quick exit from a deeply nested structure is convenient, and there are cases where revising to eliminate the **GOTO** actually simplifies the program.

- Public exchange based on reasoning by argument (rationalist arguments)...
- … finally tackled by one empirical study nearly 50 years later.

[1] Edsger Dijkstra . Go To Statement Considered Harmful. Communications of the ACM, 1968.
[2] Frank Rubin. ”GOTO Considered Harmful" Considered Harmful. Communications of the ACM, 1969.
[3] Donald Moore et al. " 'GOTO Considered Harmful' Considered Harmful" Considered Harmful?" Communications of the ACM, 1987.
[4] Nagappan et al. An empirical study of goto in C code from GitHub repositories, 2015.

Exemplary “leprechaun”: *Go To statements considered harmful*

1968



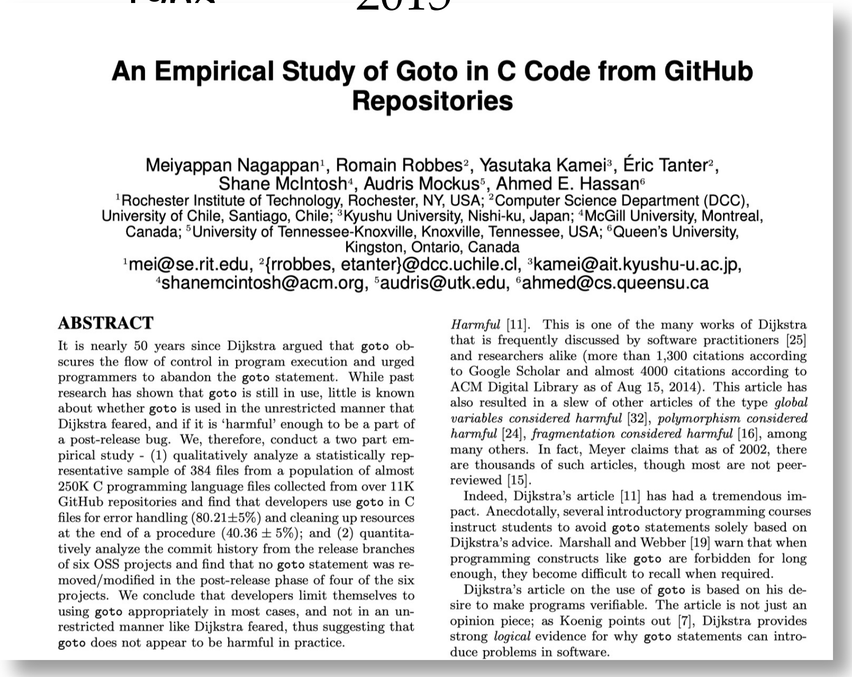
Edgar Dijkstra: Go T

Go To Statement Consider

Key Words and Phrases: go to
branch instruction, conditional c
itive clause, program intelligibil
CR Categories: 4.22, 5.23, 5.24

EDITOR:
For a number of years I have bee
that the quality of programmers
density of **go to** statements in the
recently I discovered why the use
disastrous effects, and I became c
ment should be abolished from al
languages (i.e. everything except,
At that time I did not attach too
covery; I now submit my conside
in very recent discussions in which
been urged to do so.
My first remark is that, althou
ends when he has constructed a
taking place under control of hi
matter of his activity, for it is this
the desired effect; it is this process
has to satisfy the desired specificat
been made, the “making” of the
gated to the machine.

2015



An Empirical Study of Goto in C Code from GitHub Repositories

Meiyappan Nagappan[1], Romain Robbes[2], Yasutaka Kamei[3], Éric Tanter[2],
Shane McIntosh[4], Audris Mockus[5], Ahmed E. Hassan[6]
[1]Rochester Institute of Technology, Rochester, NY, USA; [2]Computer Science Department (DCC),
University of Chile, Santiago, Chile; [3]Kyushu University, Nishi-ku, Japan; [4]McGill University, Montreal,
Canada; [5]University of Tennessee-Knoxville, Knoxville, Tennessee, USA; [6]Queen’s University,
Kingston, Ontario, Canada
[1]mei@se.rit.edu, [2]{rrobbes, etanter}@dcc.uchile.cl, [3]kamei@ait.kyushu-u.ac.jp,
[4]shanemcintosh@acm.org, [5]audris@utk.edu, [6]ahmed@cs.queensu.ca

ABSTRACT

It is nearly 50 years since Dijkstra argued that `goto` obscures the flow of control in program execution and urged programmers to abandon the `goto` statement. While past research has shown that `goto` is still in use, little is known about whether `goto` is used in the unrestricted manner that Dijkstra feared, and if it is ‘harmful’ enough to be a part of a post-release bug. We, therefore, conduct a two part empirical study - (1) qualitatively analyze a statistically representative sample of 384 files from a population of almost 250K C programming language files collected from over 11K GitHub repositories and find that developers use `goto` in C files for error handling (80.21±5%) and cleaning up resources at the end of a procedure (40.36 ± 5%); and (2) quantitatively analyze the commit history from the release branches of six OSS projects and find that no `goto` statement was removed/modified in the post-release phase of four of the six projects. We conclude that developers limit themselves to using `goto` appropriately in most cases, and not in an unrestricted manner like Dijkstra feared, thus suggesting that `goto` does not appear to be harmful in practice.

Harmful [11]. This is one of the many works of Dijkstra that is frequently discussed by software practitioners [25] and researchers alike (more than 1,300 citations according to Google Scholar and almost 4000 citations according to ACM Digital Library as of Aug 15, 2014). This article has also resulted in a slew of other articles of the type *global variables considered harmful* [32], *polymorphism considered harmful* [24], *fragmentation considered harmful* [16], among many others. In fact, Meyer claims that as of 2002, there are thousands of such articles, though most are not peer-reviewed [15].

Indeed, Dijkstra’s article [11] has had a tremendous impact. Anecdotally, several introductory programming courses instruct students to avoid `goto` statements solely based on Dijkstra’s advice. Marshall and Webber [19] warn that when programming constructs like `goto` are forbidden for long enough, they become difficult to recall when required.

Dijkstra’s article on the use of `goto` is based on his desire to make programs verifiable. The article is not just an opinion piece; as Koenig points out [7], Dijkstra provides strong *logical* evidence for why `goto` statements can introduce problems in software.

“We conclude that developers limit themselves to using goto appropriately, [not] like Dijkstra feared, [thus] goto does not appear to be harmful in practice.”

- Public exchange based on reasoning by argument (rationalist arguments)...
- … finally tackled by one empirical study nearly 50 years later.

[1] Edsger Dijkstra . Go To Statement Considered Harmful. Communications of the ACM, 1968.
[2] Frank Rubin. ”GOTO Considered Harmful" Considered Harmful. Communications of the ACM, 1969.
[3] Donald Moore et al. " 'GOTO Considered Harmful' Considered Harmful" Considered Harmful?" Communications of the ACM, 1987.
[4] Nagappan et al. An empirical study of goto in C code from GitHub repositories, 2015.

Key Takeaway

- The current state of evidence in Software Engineering is still weak
  - Practical relevance and impact?
  - Potential for transfer into practice and adoption?

- But there is hope…
  - Importance of empirical research recognised
  - Growth of a strong research community over last two decades

Outline

- What is Empirical Software Engineering?
- Why do we need Empirical Software Engineering?
- **What are the perspectives in Empirical Software Engineering?**

Empirical Software Engineering Community

Empirical Software Engineering research community 2018 (Oulu, Finland)

Goals and perspectives in Empirical Software Engineering

1. Provide tools and methods for empirical research
2. Establish strong Software Engineering theories
3. Eradicate conventional wisdom (“leprechauns”)

Various settings

- Industry settings
- Research initiatives from the community
- Publicly funded research projects

Example 1:
Industry setting
(as part of a Academia-industry collaboration)

SIEMENS

Ingenuity for life

Challenge: Role and Relevance of RE to Business Success unclear

Goal: *“Find the proper Problem before solving it properly”*

Exemplary question: *“How does Customer Satisfaction depend on RE?”*

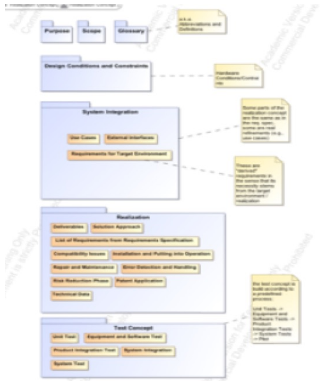
How relevant is RE to business success?

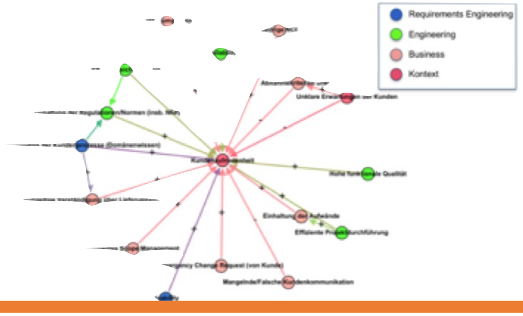
1. How is RE conducted at project level?
2. How does customer satisfaction depend on factors in RE?
3. What factors do influence the way RE is conducted?

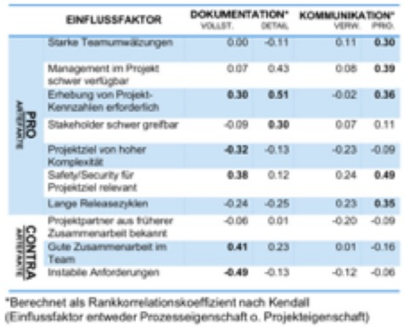
Document Analysis

Interviews

Online Survey




Requirements Engineering
Engineering
Business
Kontext
Unklare Erwartungen der Kunden
Hohe funktionale Qualität
Einhaltung der Aufwände
Effiziente Projektdurchführung
Scope Management
Change Request (von Kunde)
Mangelnde/Falsche Kundenkommunikation
Kundenprozesse (Domänenwissen)
Regulationen/Normen
Kundenzufriedenheit

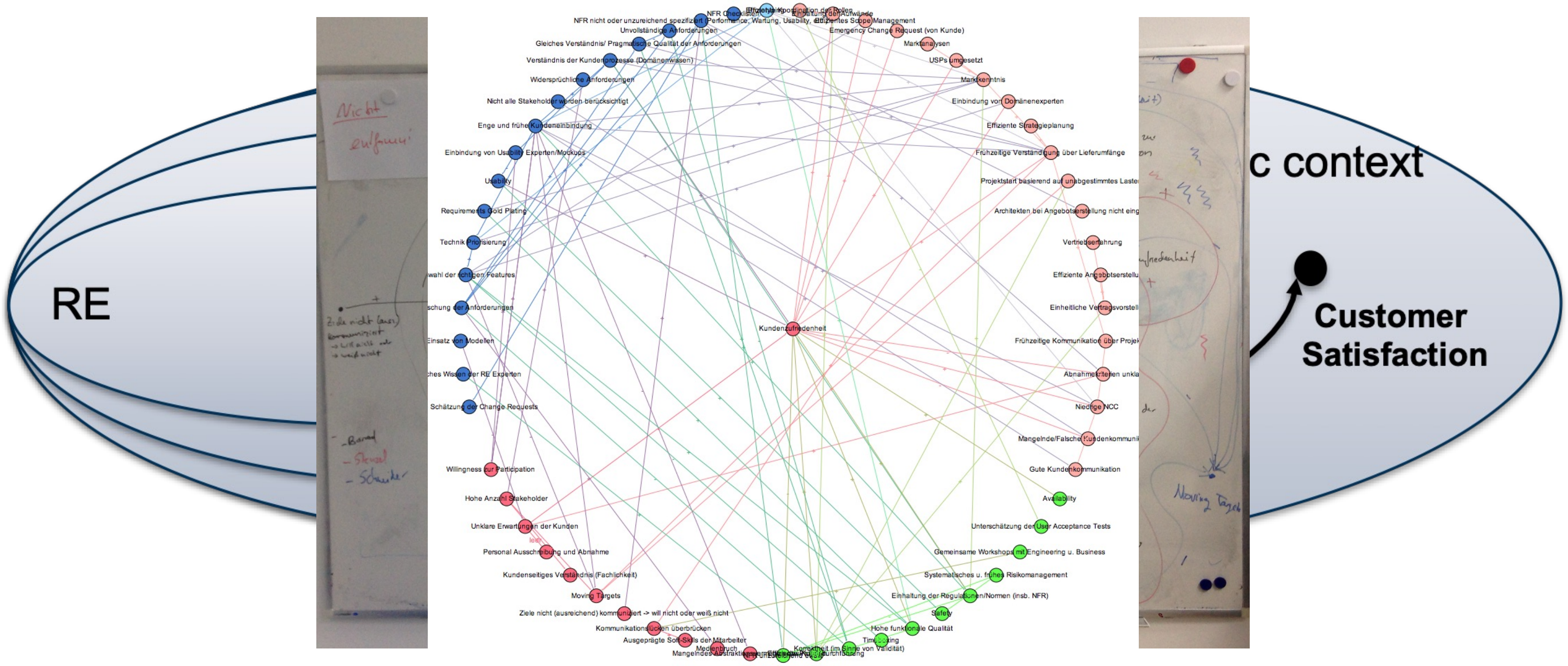


	EINFLUSSFAKTOR	DOKUMENTATION* VOLLST.	DOKUMENTATION* DETAIL	KOMMUNIKATION* VERW.	KOMMUNIKATION* PRO.
PRO ARTEFAKTE	Starke Teamumwälzungen	0.00	-0.11	0.11	**0.30**
	Management im Projekt schwer verfügbar	0.07	0.43	0.08	**0.39**
	Erhebung von Projekt-Kennzahlen erforderlich	**0.30**	**0.51**	-0.02	**0.36**
	Stakeholder schwer greifbar	-0.09	**0.30**	0.07	0.11
	Projektziel von hoher Komplexität	**-0.32**	-0.13	-0.23	-0.09
	Safety/Security für Projektziel relevant	**0.38**	0.12	0.24	**0.49**
	Lange Releasezyklen	-0.24	-0.25	0.23	**0.35**
CONTRA ARTEFAKTE	Projektpartner aus früherer Zusammenarbeit bekannt	-0.06	0.01	-0.20	-0.09
	Gute Zusammenarbeit im Team	**0.41**	0.23	0.01	-0.16
	Instabile Anforderungen	**-0.49**	-0.13	-0.12	-0.06

*Berechnet als Rankkorrelationskoeffizient nach Kendall (Einflussfaktor entweder Prozesseigenschaft o. Projekteigenschaft)

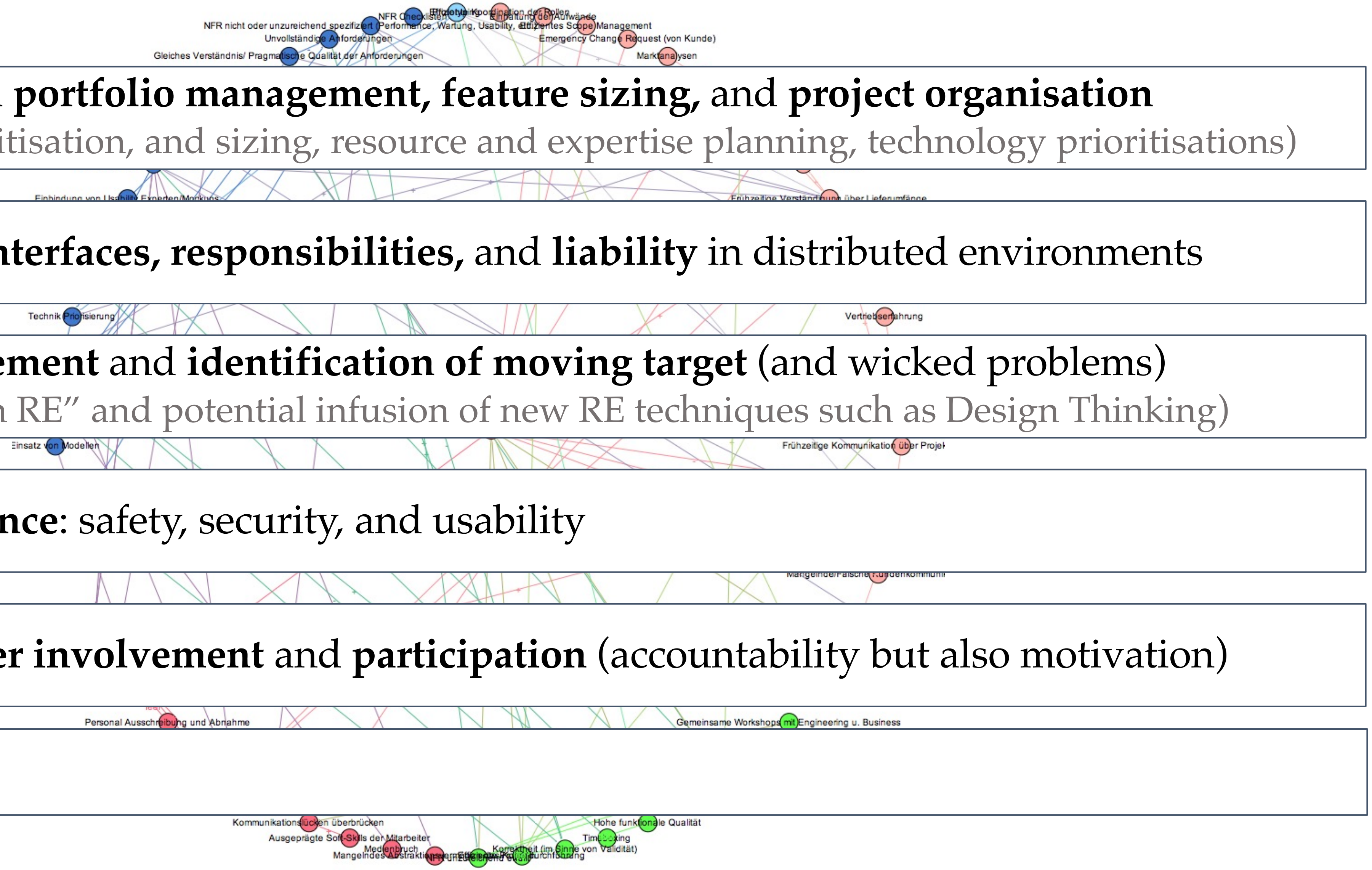
How does Customer Satisfaction depend on RE?

- Interviews of different roles in different project settings
- Root cause analysis of customer satisfaction to phenomena in RE



RE
c context
Customer Satisfaction
Nicht
NFR Checklisten
NFR nicht oder unzureichend spezifiziert (Performance, Wartung, Usability, etc.)
Unvollständige Anforderungen
Gleiches Verständnis/ Pragmatische Qualität der Anforderungen
Verständnis der Kundenprozesse (Domänenwissen)
Widersprüchliche Anforderungen
Nicht alle Stakeholder werden berücksichtigt
Enge und frühe Kundeneinbindung
Einbindung von Usability Experten/Mockups
Usability
Requirements Gold Plating
Technik Priorisierung
wahl der richtigen Features
schung der Anforderungen
Einsatz von Modellen
ches Wissen der RE Experten
Schätzung der Change Requests
Willingness zur Participation
Hohe Anzahl Stakeholder
Unklare Erwartungen der Kunden
leer
Personal Ausschreibung und Abnahme
Kundenseitiges Verständnis (Fachlichkeit)
Moving Targets
Ziele nicht (ausreichend) kommuniziert -> will nicht oder weiß nicht
Kommunikationslücken überbrücken
Ausgeprägte Soft-Skills der Mitarbeiter
Medienbruch
Mangelndes Abstraktionsvermögen
Koordination der Rollen
Einhaltung der Aufwände
Effizientes Scope Management
Emergency Change Request (von Kunde)
Marktanalysen
USPs umgesetzt
Marktkenntnis
Einbindung von Domänenexperten
Effiziente Strategieplanung
Frühzeitige Verständigung über Lieferumfänge
Projektstart basierend auf unabgestimmtes Laste
Architekten bei Angebotserstellung nicht eing
Vertriebserfahrung
Effiziente Angebotserstellu
Einheitliche Vertragsvorstell
Frühzeitige Kommunikation über Projek
Abnahmekriterien unkla
Niedrige NCC
Mangelnde/Falsche Kundenkommuni
Gute Kundenkommunikation
Kundenzufriedenheit
Availability
Unterschätzung der User Acceptance Tests
Gemeinsame Workshops mit Engineering u. Business
Systematisches u. frühes Risikomanagement
Einhaltung der Regulationen/Normen (insb. NFR)
Safety
Hohe funktionale Qualität
Timeboxing
Korrektheit (im Sinne von Validität)
Moving Targ

RE is a recognised basis for…



… effective **product and portfolio management, feature sizing,** and **project organisation** (feature planning, prioritisation, and sizing, resource and expertise planning, technology prioritisations)

… clear **(dev.) process interfaces, responsibilities,** and **liability** in distributed environments

… effective **risk management** and **identification of moving target** (and wicked problems) (basis for "good-enough RE" and potential infusion of new RE techniques such as Design Thinking)

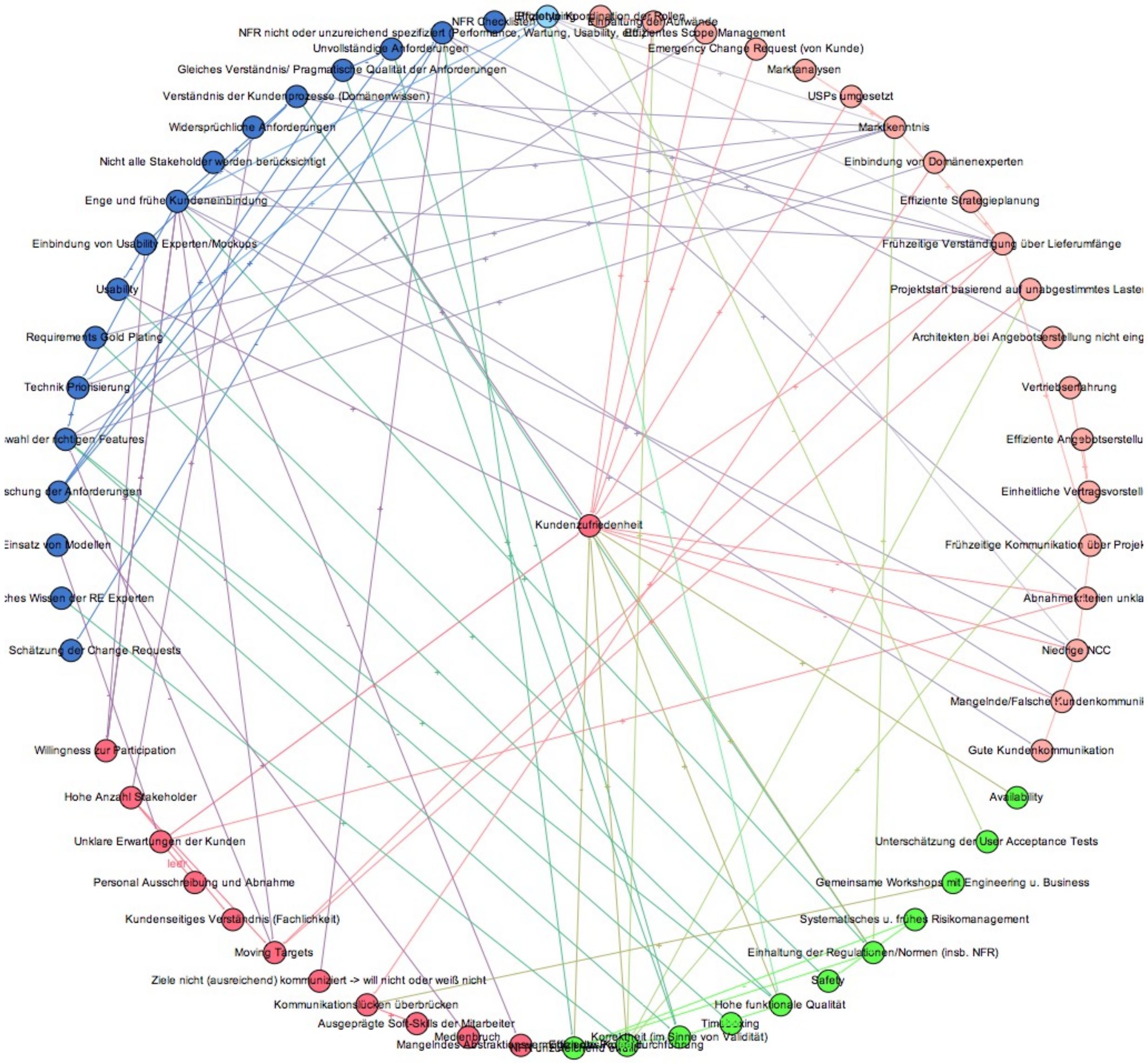
… **(regulatory) compliance**: safety, security, and usability

… **increased stakeholder involvement** and **participation** (accountability but also motivation)

… (…)

NFR nicht oder unzureichend spezifiziert (Performance, Wartung, Usability, ...
NFR Checklisten
Effiziente Koordination der Rollen
Einhaltung der Aufwände
Effizientes Scope Management
Unvollständige Anforderungen
Emergency Change Request (von Kunde)
Gleiches Verständnis/ Pragmatische Qualität der Anforderungen
Marktanalysen
Einbindung von Usability Experten/Mockups
Frühzeitige Verständigung über Lieferumfänge
Technik Priorisierung
Vertriebserfahrung
Einsatz von Modellen
Frühzeitige Kommunikation über Projek
Mangelnde/Falsche Kundenkommuni
Personal Ausschreibung und Abnahme
Gemeinsame Workshops mit Engineering u. Business
Kommunikationslücken überbrücken
Ausgeprägte Soft-Skills der Mitarbeiter
Medienbruch
Mangelndes Abstraktionsvermögen
Hohe funktionale Qualität
Timeboxing
Korrektheit (im Sinne von Validität)
Durchführung

RE is a recognised basis for…


NFR Checklisten
NFR nicht oder unzureichend spezifiziert (Performance, Wartung, Usability,
Unvollständige Anforderungen
Gleiches Verständnis/ Pragmatische Qualität der Anforderungen
Verständnis der Kundenprozesse (Domänenwissen)
Widersprüchliche Anforderungen
Nicht alle Stakeholder werden berücksichtigt
Enge und frühe Kundeneinbindung
Einbindung von Usability Experten/Mockups
Usability
Requirements Gold Plating
Technik Priorisierung
wahl der richtigen Features
schung der Anforderungen
Einsatz von Modellen
ches Wissen der RE Experten
Schätzung der Change Requests
Einhaltung der Aufwände
Effizientes Scope Management
Emergency Change Request (von Kunde)
Marktanalysen
USPs umgesetzt
Marktkenntnis
Einbindung von Domänenexperten
Effiziente Strategieplanung
Frühzeitige Verständigung über Lieferumfänge
Projektstart basierend auf unabgestimmtes Laste
Architekten bei Angebotserstellung nicht eing
Vertriebserfahrung
Effiziente Angebotserstellu
Einheitliche Vertragsvorstell
Frühzeitige Kommunikation über Projek
Abnahmekriterien unkla
Niedrige NCC
Mangelnde/Falsche Kundenkommuni
Gute Kundenkommunikation
Kundenzufriedenheit
Willingness zur Participation
Hohe Anzahl Stakeholder
Unklare Erwartungen der Kunden
leer
Personal Ausschreibung und Abnahme
Kundenseitiges Verständnis (Fachlichkeit)
Moving Targets
Ziele nicht (ausreichend) kommuniziert -> will nicht oder weiß nicht
Kommunikationslücken überbrücken
Ausgeprägte Soft-Skills der Mitarbeiter
Medienbruch
Mangelndes Abstraktions
Availability
Unterschätzung der User Acceptance Tests
Gemeinsame Workshops mit Engineering u. Business
Systematisches u. frühes Risikomanagement
Einhaltung der Regulationen/Normen (insb. NFR)
Safety
Hohe funktionale Qualität
Timeboxing
Korrektheit (im Sinne von Validität)
durchführung

Example 2:
Research initiatives

Background: Requirements Engineering research often dominated by conventional wisdom

Challenge: What research topics are of high practical relevance?

NaPiRE

What are practices and problems in practical Requirements Engineering environments?

RE Pract

How do practitioners perceive the relevance of contributions by the RE research community?

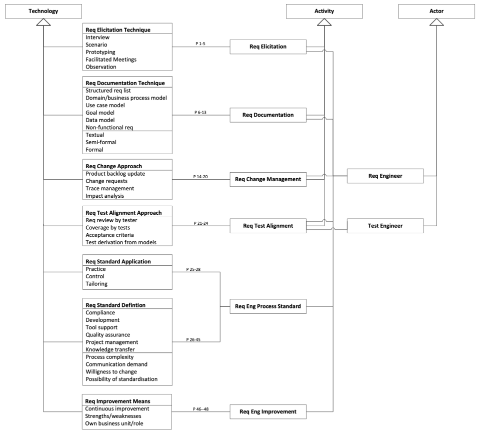
How do practitioners perceive the relevance of RE standards?

Exemplary initiatives in context of IREB working group (“IREB Research”)

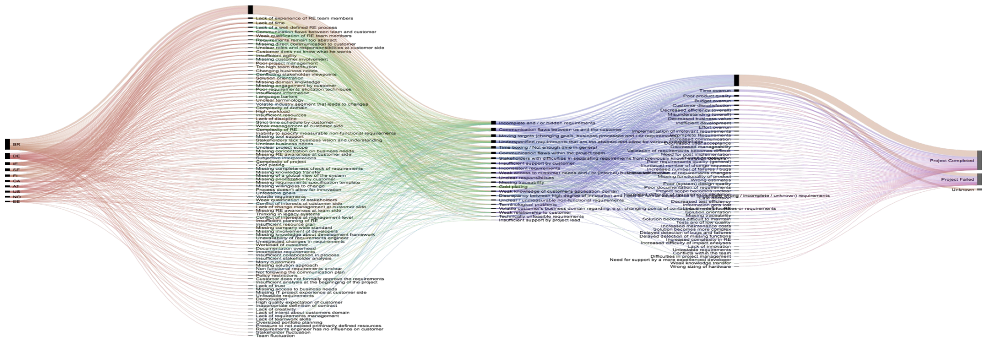
NaPiRE Naming the Pain in Requirements Engineering

Objectives: Build theory on RE practices used in industry and on problems practitioners experience

Research method: Large-scale survey research

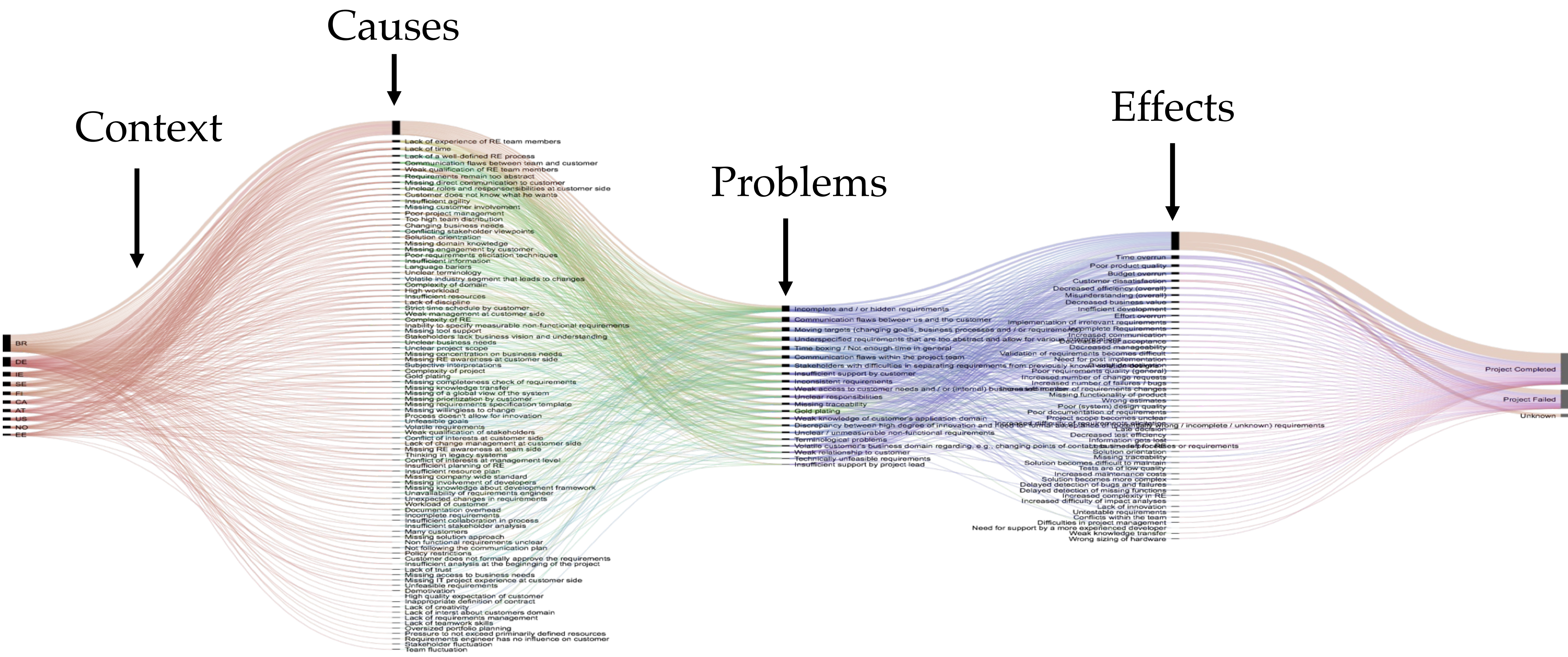


Practices



Problems, causes, and effects

First theory on Requirements Engineering practices and problems supporting problem-driven research



Context

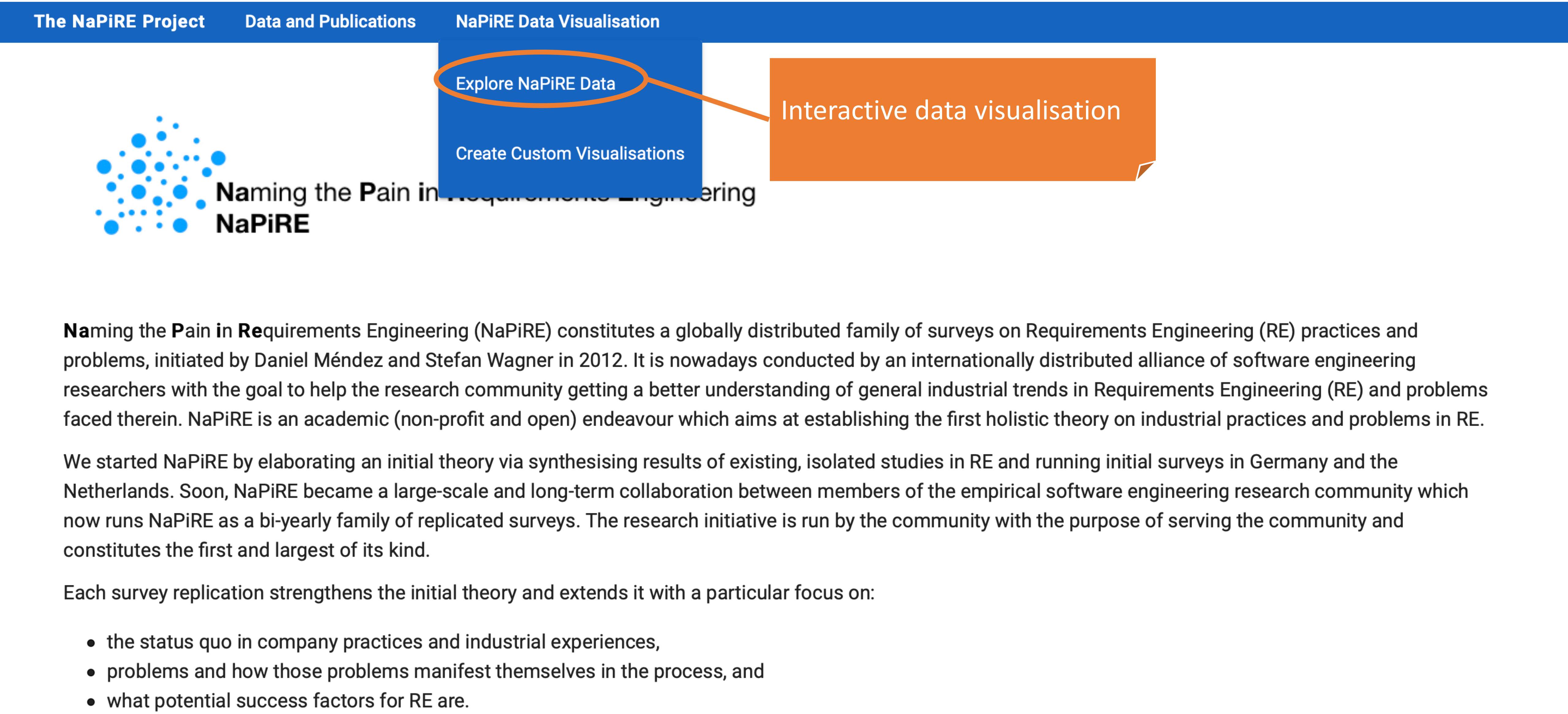
Causes

Problems

Effects

BR
DE
IE
SE
FI
CA
AT
US
NO
EE
Lack of experience of RE team members
Lack of time
Lack of a well-defined RE process
Communication flaws between team and customer
Weak qualification of RE team members
Requirements remain too abstract
Missing direct communication to customer
Unclear roles and responsonsibilities at customer side
Customer does not know what he wants
Insufficient agility
Missing customer involvement
Poor project management
Too high team distribution
Changing business needs
Conflicting stakeholder viewpoints
Solution orientation
Missing domain knowledge
Missing engagement by customer
Poor requirements elicitation techniques
Insufficient information
Language bariers
Unclear terminology
Volatile industry segment that leads to changes
Complexity of domain
High workload
Insufficient resources
Lack of discipline
Strict time schedule by customer
Weak management at customer side
Complexity of RE
Inability to specify measurable non-functional requirements
Missing tool support
Stakeholders lack business vision and understanding
Unclear business needs
Unclear project scope
Missing concentration on business needs
Missing RE awareness at customer side
Subjective interpretations
Complexity of project
Gold plating
Missing completeness check of requirements
Missing knowledge transfer
Missing of a global view of the system
Missing prioritization by customer
Missing requirements specification template
Missing willingless to change
Process doesn't allow for innovation
Unfeasible goals
Volatile requirements
Weak qualification of stakeholders
Conflict of interests at customer side
Lack of change management at customer side
Missing RE awareness at team side
Thinking in legacy systems
Conflict of interests at management level
Insufficient planning of RE
Insufficient resource plan
Missing company wide standard
Missing involvement of developers
Missing knowledge about development framework
Unavailability of requirements engineer
Unexpected changes in requirements
Workload of customer
Documentation overhead
Incomplete requirements
Insufficient collaboration in process
Insufficient stakeholder analysis
Many customers
Missing solution approach
Non functional requirements unclear
Not following the communication plan
Policy restrictions
Customer does not formally approve the requirements
Insufficient analysis at the beginnging of the project
Lack of trust
Missing access to business needs
Missing IT project experience at customer side
Unfeasible requirements
Demotivation
High quality expectation of customer
Inappropriate definition of contract
Lack of creativity
Lack of interst about customers domain
Lack of requirements management
Lack of teamwork skills
Oversized portfolio planning
Pressure to not exceed priminarily defined resources
Requirements engineer has no influence on customer
Stakeholder fluctuation
Team fluctuation
Incomplete and / or hidden requirements
Communication flaws between us and the customer
Moving targets (changing goals, business processes and / or requirements)
Underspecified requirements that are too abstract and allow for various interpretations
Time boxing / Not enough time in general
Communication flaws within the project team
Stakeholders with difficulties in separating requirements from previously known solution designs
Insufficient support by customer
Inconsistent requirements
Weak access to customer needs and / or (internal) business information
Unclear responsibilities
Missing traceability
Gold plating
Weak knowledge of customer's application domain
Discrepancy between high degree of innovation and need for formal acceptance of (potentially wrong / incomplete / unknown) requirements
Unclear / unmeasurable non-functional requirements
Terminological problems
Volatile customer's business domain regarding, e.g., changing points of contact, business processes or requirements
Weak relationship to customer
Technically unfeasible requirements
Insufficient support by project lead
Time overrun
Poor product quality
Budget overrun
Customer dissatisfaction
Decreased efficiency (overall)
Misunderstanding (overall)
Decreased business value
Inefficient development
Effort overrun
Implementation of irrelevant requirements
Incomplete Requirements
Increased communication
Decreased user acceptance
Decreased manageability
Validation of requirements becomes difficult
Need for post implementation
Overall demotivation
Poor requirements quality (general)
Increased number of change requests
Increased number of failures / bugs
Increased number of requirements changes
Missing functionality of product
Wrong estimates
Poor (system) design quality
Poor documentation of requirements
Project scope becomes unclear
Increased difficulty of requirements elicitation
Late decision
Decreased test efficiency
Information gets lost
Less time left for RE
Solution orientation
Missing traceability
Solution becomes difficult to maintain
Tests are of low quality
Increased maintenance costs
Solution becomes more complex
Delayed detection of bugs and failures
Delayed detection of missing functions
Increased complexity in RE
Increased difficulty of impact analyses
Lack of innovation
Untestable requirements
Conflicts within the team
Difficulties in project management
Need for support by a more experienced developer
Weak knowledge transfer
Wrong sizing of hardware
Project Completed
Project Failed
Unknown

See yourself and play with the interactive data visualisation: www.napire.org



Interactive data visualisation

Naming the **P**ain **i**n **R**equirements **E**ngineering
NaPiRE

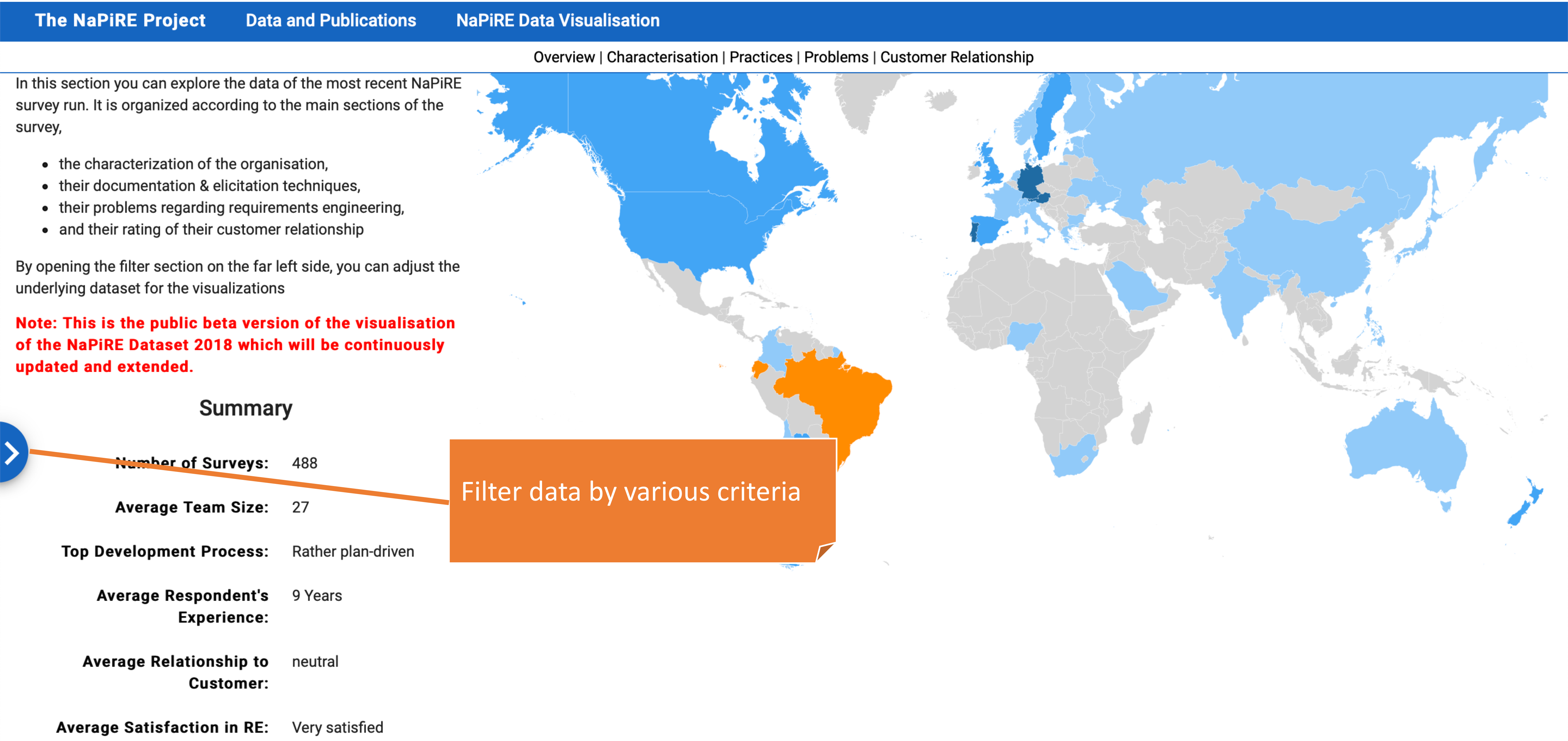
Naming the **P**ain **i**n **Re**quirements Engineering (NaPiRE) constitutes a globally distributed family of surveys on Requirements Engineering (RE) practices and problems, initiated by Daniel Méndez and Stefan Wagner in 2012. It is nowadays conducted by an internationally distributed alliance of software engineering researchers with the goal to help the research community getting a better understanding of general industrial trends in Requirements Engineering (RE) and problems faced therein. NaPiRE is an academic (non-profit and open) endeavour which aims at establishing the first holistic theory on industrial practices and problems in RE.

We started NaPiRE by elaborating an initial theory via synthesising results of existing, isolated studies in RE and running initial surveys in Germany and the Netherlands. Soon, NaPiRE became a large-scale and long-term collaboration between members of the empirical software engineering research community which now runs NaPiRE as a bi-yearly family of replicated surveys. The research initiative is run by the community with the purpose of serving the community and constitutes the first and largest of its kind.

Each survey replication strengthens the initial theory and extends it with a particular focus on:

- the status quo in company practices and industrial experiences,
- problems and how those problems manifest themselves in the process, and
- what potential success factors for RE are.

See yourself and play with the interactive data visualisation: www.napire.org



The NaPiRE Project
Data and Publications
NaPiRE Data Visualisation
Overview | Characterisation | Practices | Problems | Customer Relationship

In this section you can explore the data of the most recent NaPiRE survey run. It is organized according to the main sections of the survey,

- the characterization of the organisation,
- their documentation & elicitation techniques,
- their problems regarding requirements engineering,
- and their rating of their customer relationship

By opening the filter section on the far left side, you can adjust the underlying dataset for the visualizations

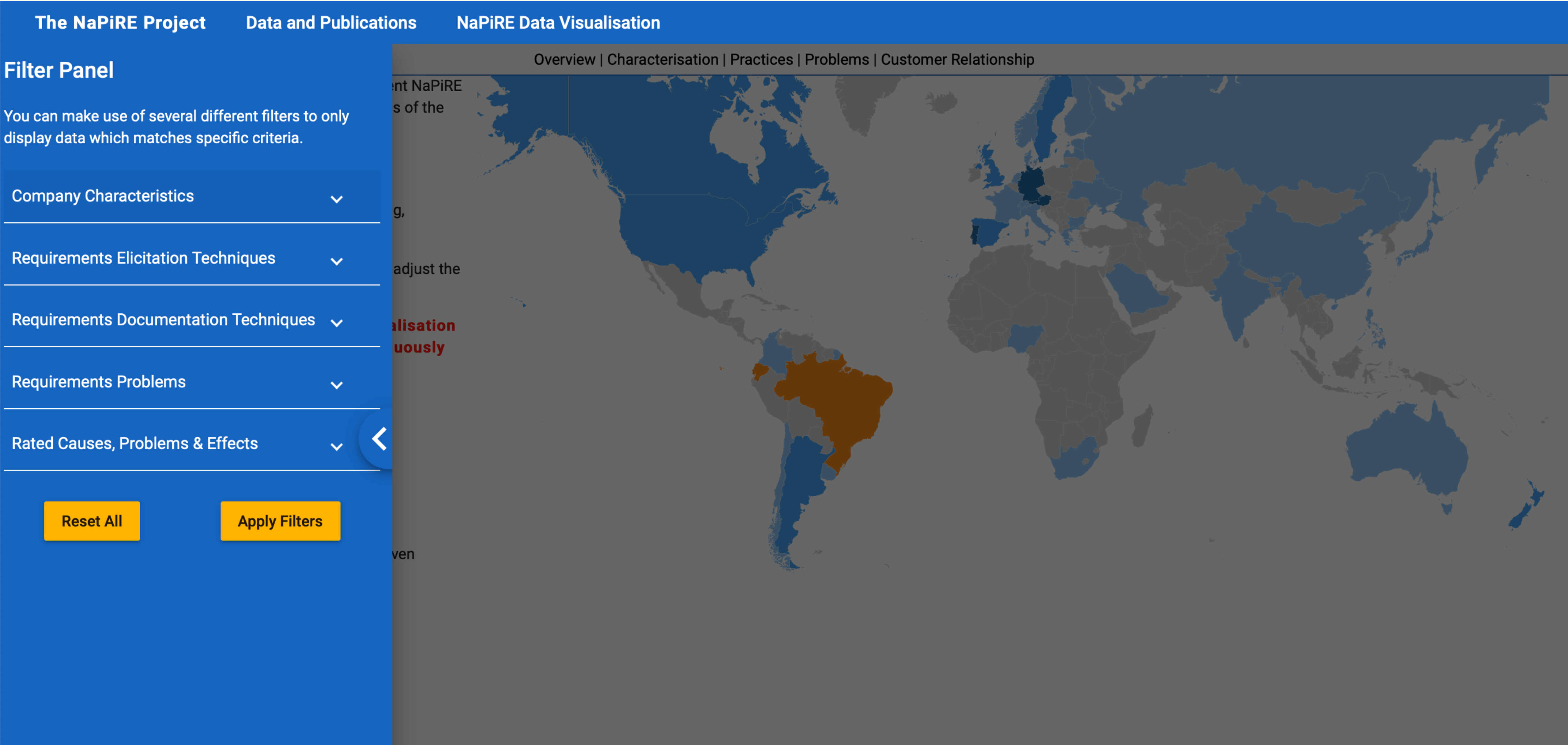
Note: This is the public beta version of the visualisation of the NaPiRE Dataset 2018 which will be continuously updated and extended.

Summary

Number of Surveys:	488
Average Team Size:	27
Top Development Process:	Rather plan-driven
Average Respondent's Experience:	9 Years
Average Relationship to Customer:	neutral
Average Satisfaction in RE:	Very satisfied

Filter data by various criteria

See yourself and play with the interactive data visualisation: www.napire.org



The NaPiRE Project | Data and Publications | NaPiRE Data Visualisation

Overview | Characterisation | Practices | Problems | Customer Relationship

Filter Panel

You can make use of several different filters to only display data which matches specific criteria.

- Company Characteristics
- Requirements Elicitation Techniques
- Requirements Documentation Techniques
- Requirements Problems
- Rated Causes, Problems & Effects

Reset All | Apply Filters

nt NaPiRE
s of the

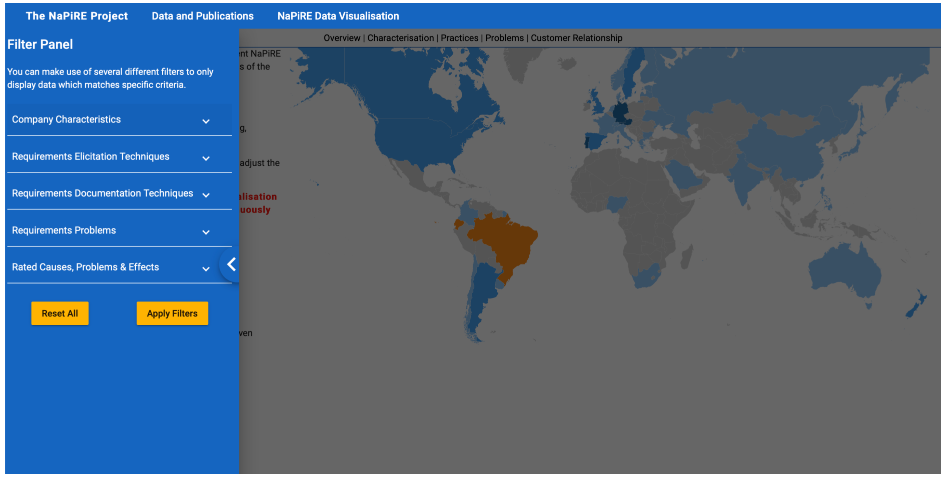
g,

adjust the

alisation
uously

ven

See yourself and play with the interactive data visualisation: www.napire.org


The NaPiRE Project
Data and Publications
NaPiRE Data Visualisation
Overview | Characterisation | Practices | Problems | Customer Relationship
Filter Panel
Company Characteristics
Requirements Elicitation Techniques
Requirements Documentation Techniques
Requirements Problems
Rated Causes, Problems & Effects
Reset All
Apply Filters
nt NaPiRE
s of the
g,
adjust the
alisation
uously
ven

You can make use of several different filters to only display data which matches specific criteria.

Quiz

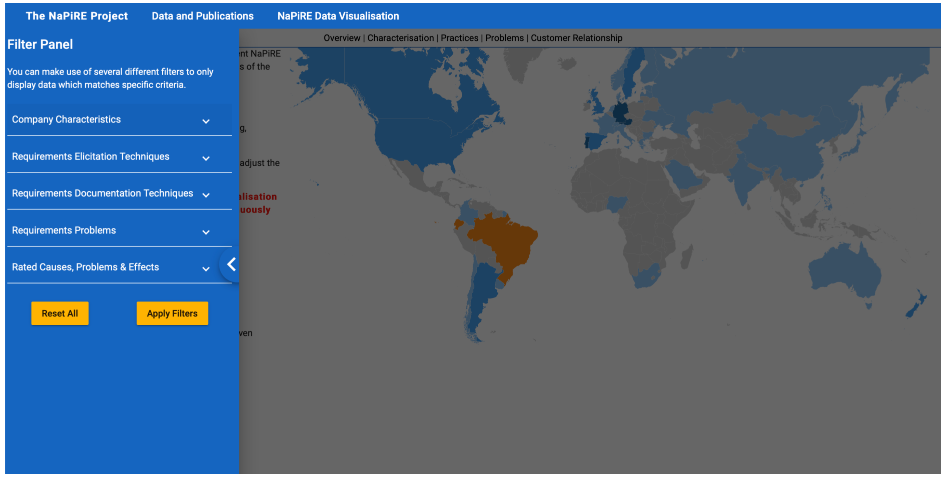
What is the most **frequently stated problem** companies face in RE when using a **rather agile** software development process model?

- ? Incomplete or hidden requirements
- ? Unprecise / Unmeasurable requirements
- ? Communication problems

* Prize

Ericsson Space sweater ☺

See yourself and play with the interactive data visualisation:
www.napire.org


The NaPiRE Project
Data and Publications
NaPiRE Data Visualisation
Overview | Characterisation | Practices | Problems | Customer Relationship
Filter Panel
Company Characteristics
Requirements Elicitation Techniques
Requirements Documentation Techniques
Requirements Problems
Rated Causes, Problems & Effects
Reset All
Apply Filters
nt NaPiRE
s of the
g,
adjust the
alisation
uously
ven

You can make use of several different filters to only display data which matches specific criteria.

Quiz

What is the most **frequently stated problem** companies face in RE when using a **rather agile** software development process model?

● **Incomplete or hidden requirements**

○ Unprecise / Unmeasurable requirements

○ Communication problems

* Prize
Ericsson Space sweater ☺

Example 3:
Publicly funded research projects

S.E.R.T

reThink▪reThought

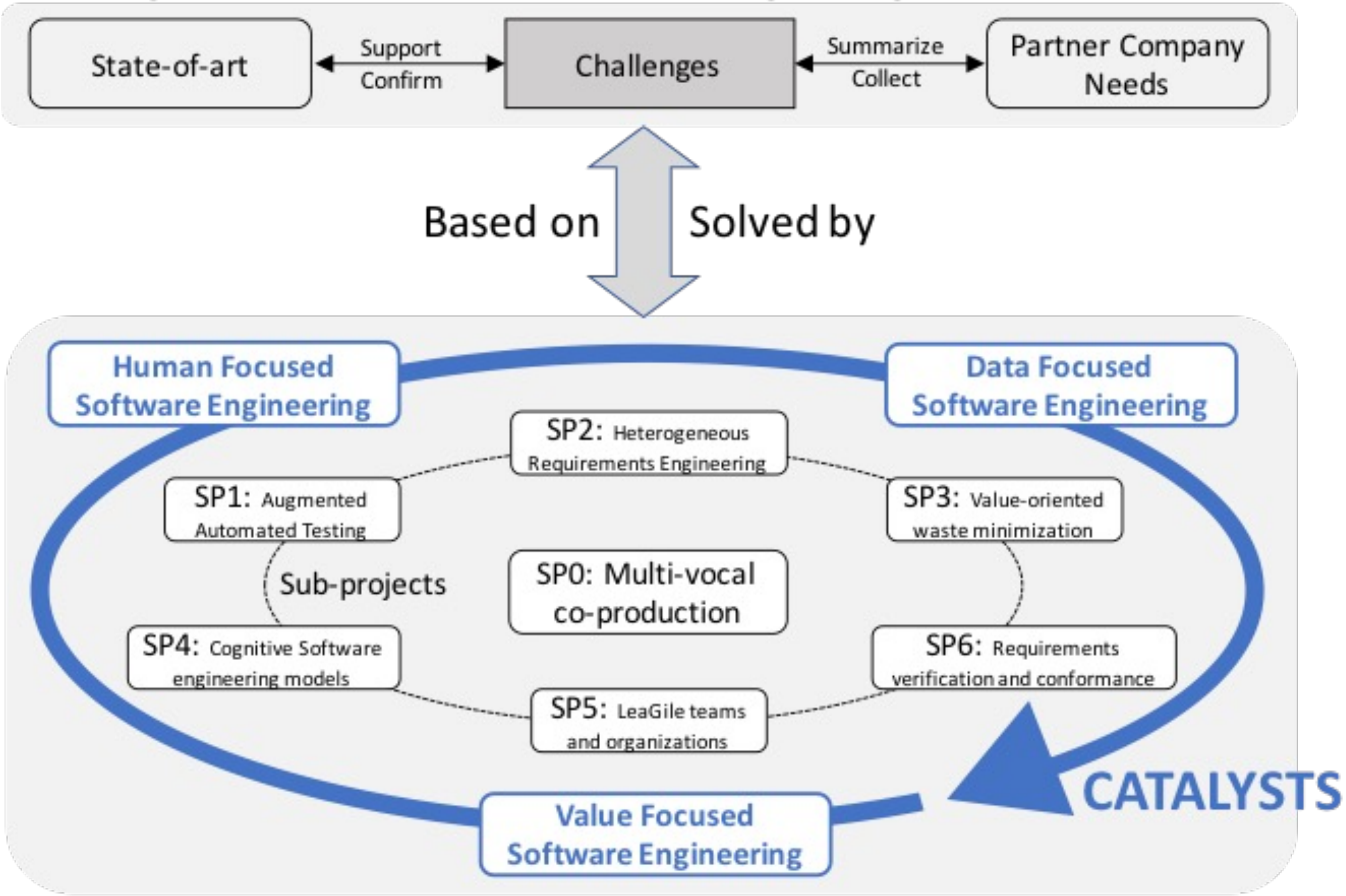
Background: Empirical software engineering is advancing already, but it needs to integrate multiple competences:

- Data-centric automation
- Value orientation
- Human-centricity

Goal: Extend empirical SE to solve next generation SE problems.

See yourself: www.rethought.se

Challenges for the Next Generation Software Engineering



State-of-art

Support

Confirm

Challenges

Summarize

Collect

Partner Company Needs

Based on

Solved by

Human Focused Software Engineering

Data Focused Software Engineering

SP2: Heterogeneous Requirements Engineering

SP1: Augmented Automated Testing

SP3: Value-oriented waste minimization

Sub-projects

SP0: Multi-vocal co-production

SP4: Cognitive Software engineering models

SP6: Requirements verification and conformance

SP5: LeaGile teams and organizations

CATALYSTS

Value Focused Software Engineering

SE Rethought: Solutions for the Next Generation Software Engineering

ERICSSON

VOLVO

Volvo Construction Equipment

QTEMA

QUALITY & TEST MANAGEMENT

Swedbank

TOLPAGORNI

product management

Handelsbanken

SONY

MaxKompetens

Telia Company

FORTNOX

Småföretagens bästa vän

TimePeopleGroup

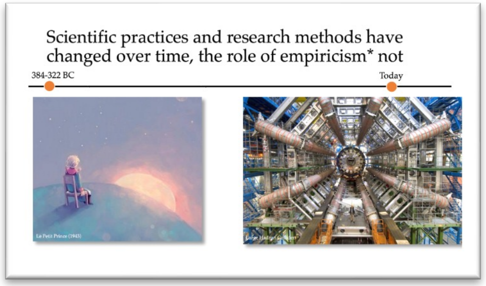
Outline

- What is Empirical Software Engineering?

- Why do we need Empirical Software Engineering?

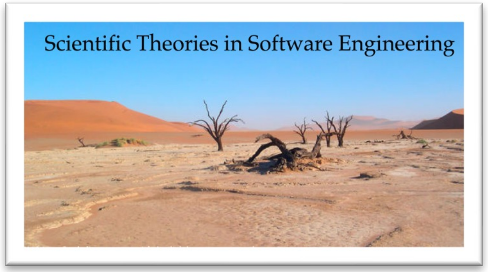
- What are the perspectives in Empirical Software Engineering?

Key Takeaways



Scientific practices and research methods have changed over time, the role of empiricism* not

Empirical research is important to turn software engineering into a scientific discipline

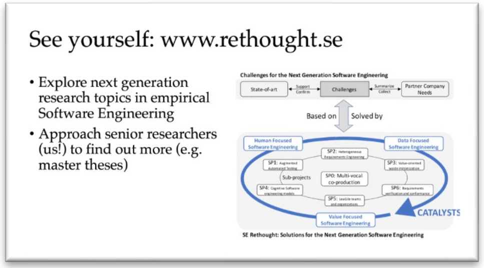


Scientific Theories in Software Engineering

The state of evidence in Software Engineering is still weak

Empirical Software Engineering Community

The growth of a community of empirical researchers and practitioners is promising



See yourself: www.rethought.se

- Explore next generation research topics in empirical Software Engineering
- Approach senior researchers (us!) to find out more (e.g. master theses)

Challenges for the Next Generation Software Engineering

State-of-art
Support
Confirm
Challenges
Summarize
Collect
Partner Company Needs
Based on
Solved by
Human Focused Software Engineering
Data Focused Software Engineering
SP2: Heterogeneous Requirements Engineering
SP1: Augmented Automated Testing
SP3: Value-oriented waste minimization
Sub-projects
SP0: Multi-vocal co-production
SP4: Cognitive Software engineering models
SP6: Requirements verification and conformance
SP5: LeaGile teams and organizations
CATALYSTS
Value Focused Software Engineering

SE Rethought: Solutions for the Next Generation Software Engineering

We are now entering the next generation of empirical Software Engineering research