Building and Evaluating Theories in Software Engineering

Daniel Méndez
🏛 Blekinge Institute of Technology, Sweden
🏛 fortiss GmbH, Germany
🌐 www.mendezfe.org
🐦 mendezfe

Ground rule

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

Frequently encountered prejudice

“Our inability to carry out truly scientific experiments and surveys [...] will yield anecdotes of limited value. Empirical studies should only be used to confirm what works in theory […]”

“Only quantitative data is real data.”

At the same time, we often see studies like this

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 Volvo (p<0.0

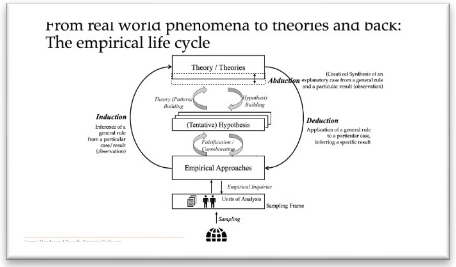
Empirical research is more than simply applying statistical equations in search for universal "truth"

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

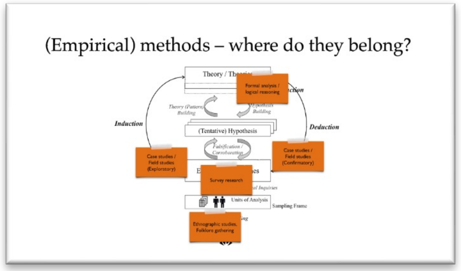
Key Takeaways

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

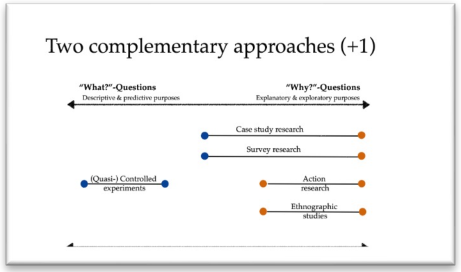
Empirical research is the backbone of every scientific discipline.


From real world phenomena to theories and back: The empirical life cycle
Theory / Theories
Abduction
(Creative) Synthesis of an explanatory case from a general rule and a particular result (observation)
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

Theory building and evaluation allow us to move forward from paradigmatic stage of an engineering discipline to a scientific one.


(Empirical) methods – where do they belong?
Theory / Theories
Formal analysis / logical reasoning
Theory (Pattern) Building
Hypothesis Building
Induction
(Tentative) Hypothesis
Deduction
Falsification / Corroboration
Case studies / Field studies (Exploratory)
Case studies / Field studies (Confirmatory)
Survey research
Units of Analysis
Sampling Frame
Ethnographic studies, Folklore gathering

Every research method has its place in a larger picture.


Two complementary approaches (+1)

Qualitative and quantitative research have complementary purposes, strengths, and limitations in building and evaluating theories.

Theory Building in Software Engineering

- Science and Theories in a Nutshell
… where we will briefly talk about the general notion of theories

- State of Evidence in Software Engineering
… where we will see why theory building is so important to our field

- Research Methods in Software Engineering
… where we will put research methods in a larger (philosophical) picture

- Qualitative "vs" quantitative research
… where we will briefly discuss different research approaches

Theory Building in Software Engineering

- **Science and Theories in a Nutshell**
  … where we will briefly talk about the general notion of theories

- State of Evidence in Software Engineering
  … where we will see why theory building is so important to our field

- Research Methods in Software Engineering
  … where we will put research methods in a larger (philosophical) picture

- Qualitative "vs" quantitative research
  … where we will briefly discuss different research approaches

Let’s start step by step….

What is scientific practice?

-- 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)

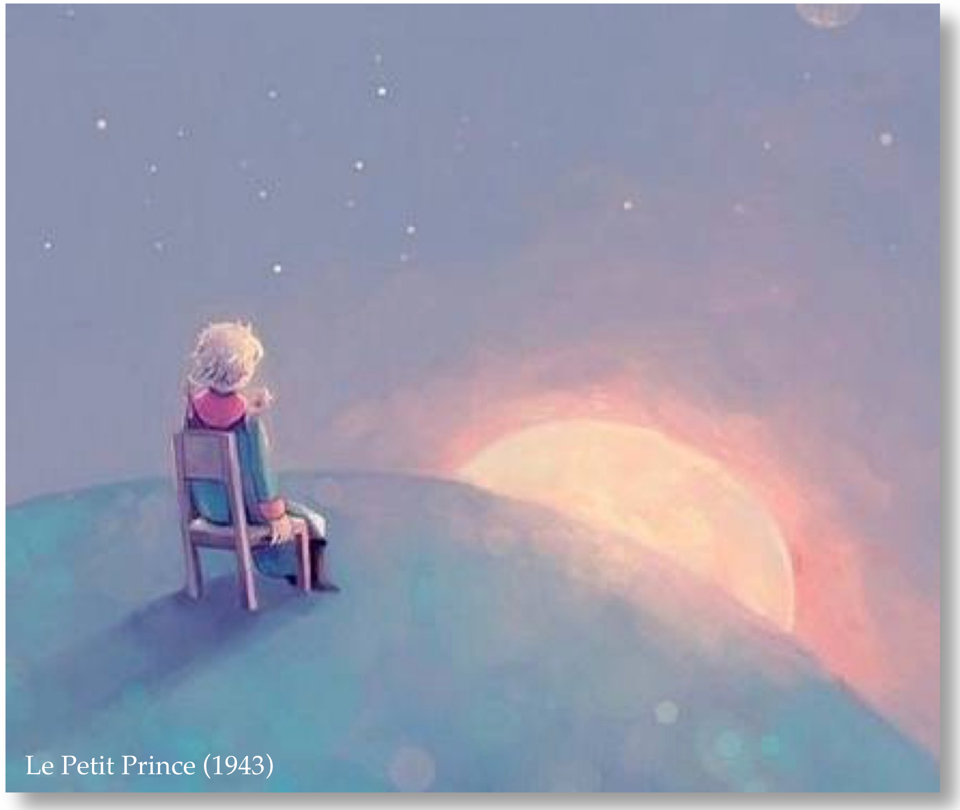
384-322 BC … 1561-1626 … 1694-1778 1724-1804 … 1896-1980 1902-1994 …

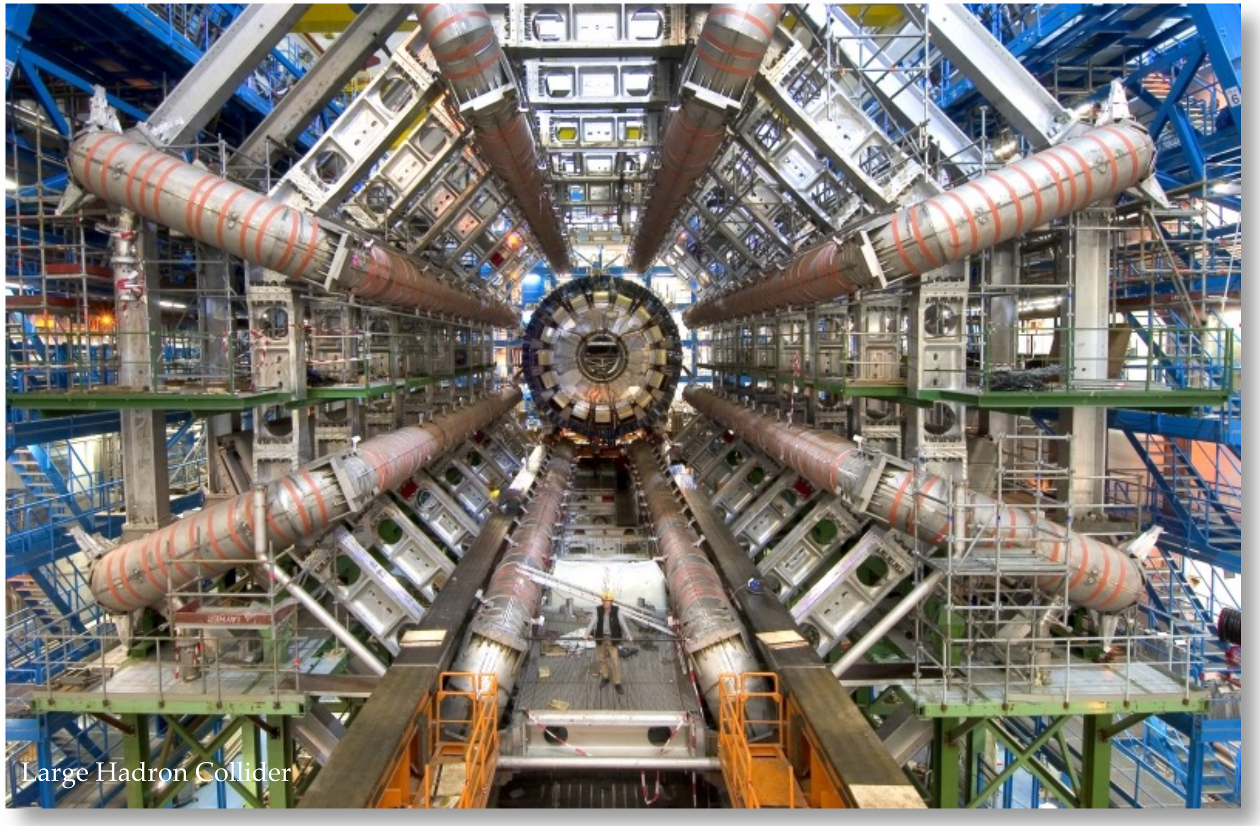
384-322 BC	**Aristoteles**	• Search for laws and reasoning for phenomena • Understanding the nature of phenomena
1561-1626	**Bacon**	• Progress of knowledge of nature (reality) • Draw benefits from knowledge growth
1694-1778	**Voltaire**	• Emancipation from gods and beliefs
1724-1804	**Kant**	• System of Epistemology (theory of knowledge)
1896-1980	**Piaget**	• Era of constructivism
1902-1994	**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:

- 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
- Although empirical observations may be faulty, it is possible (in the long run) to make reliable observations and to falsify incorrect statements about reality

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):
- "Vaccinations lead to autism"
- "Global warming is a hoax by ecologists to harm the industry"
- "Earth is flat"
- …

Are these theories scientific?

No: Speculations based on imagination, hopes and fears, and resulting in opinions that often 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)

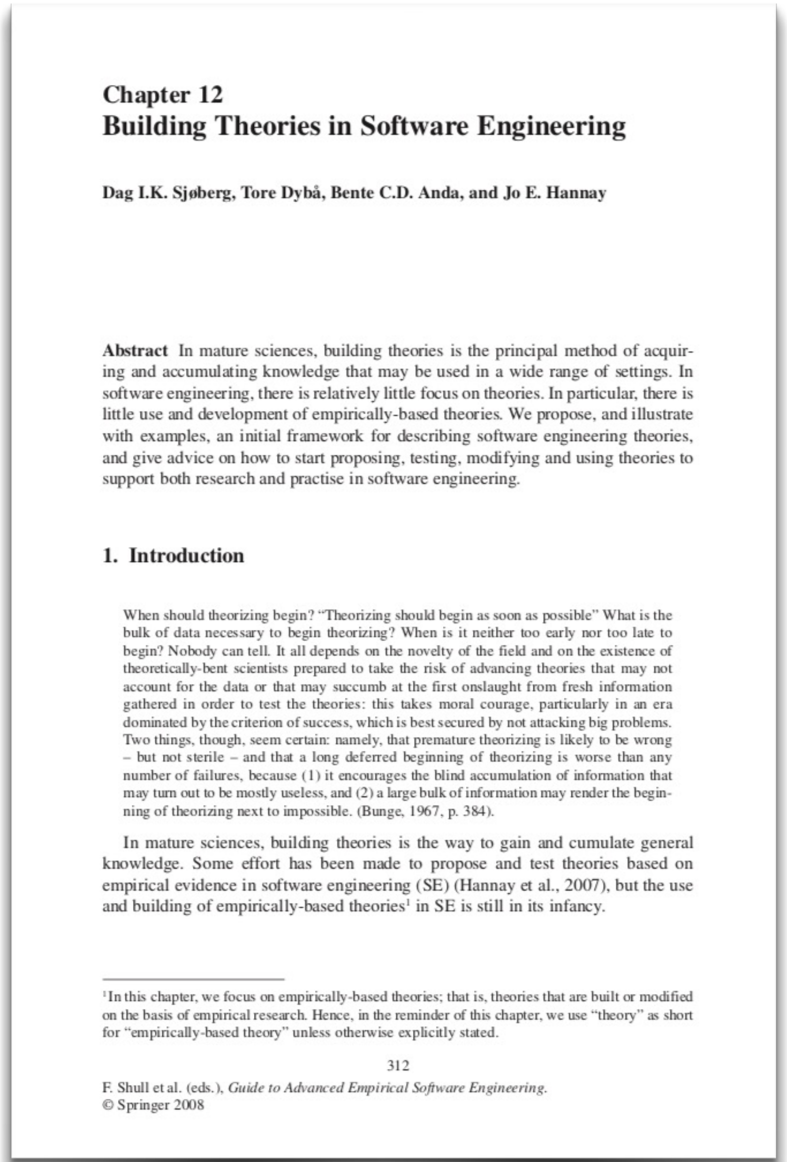
1. Tests

- Experiments, simulations, …
- Replications

In scope of empirical research methods

2. Criticism

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



Chapter 12
Building Theories in Software Engineering

Dag I.K. Sjøberg, Tore Dybå, Bente C.D. Anda, and Jo E. Hannay

Abstract In mature sciences, building theories is the principal method of acquiring and accumulating knowledge that may be used in a wide range of settings. In software engineering, there is relatively little focus on theories. In particular, there is little use and development of empirically-based theories. We propose, and illustrate with examples, an initial framework for describing software engineering theories, and give advice on how to start proposing, testing, modifying and using theories to support both research and practise in software engineering.

1. Introduction

When should theorizing begin? "Theorizing should begin as soon as possible" What is the bulk of data necessary to begin theorizing? When is it neither too early nor too late to begin? Nobody can tell. It all depends on the novelty of the field and on the existence of theoretically-bent scientists prepared to take the risk of advancing theories that may not account for the data or that may succumb at the first onslaught from fresh information gathered in order to test the theories: this takes moral courage, particularly in an era dominated by the criterion of success, which is best secured by not attacking big problems. Two things, though, seem certain: namely, that premature theorizing is likely to be wrong – but not sterile – and that a long deferred beginning of theorizing is worse than any number of failures, because (1) it encourages the blind accumulation of information that may turn out to be mostly useless, and (2) a large bulk of information may render the beginning of theorizing next to impossible. (Bunge, 1967, p. 384).

In mature sciences, building theories is the way to gain and cumulate general knowledge. Some effort has been made to propose and test theories based on empirical evidence in software engineering (SE) (Hannay et al., 2007), but the use and building of empirically-based theories[1] in SE is still in its infancy.

[1]In this chapter, we focus on empirically-based theories; that is, theories that are built or modified on the basis of empirical research. Hence, in the reminder of this chapter, we use "theory" as short for "empirically-based theory" unless otherwise explicitly stated.

F. Shull et al. (eds.), *Guide to Advanced Empirical Software Engineering*.
© Springer 2008

"There is no universally agreed upon definition of the concept of an empirically-based theory [in Software Engineering], nor is there any uniform terminology for describing theories."

Approach by characteristics

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

Scientific Theories have…

Note: Design Science Theories
A theory about artefacts in a context
[Artefact specification] X [Context assumption] → [Effect]

… 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
- Empirical support / (high) level of confidence
- Explanatory power
- Usefulness to researchers and / or practitioners
- …

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.

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.

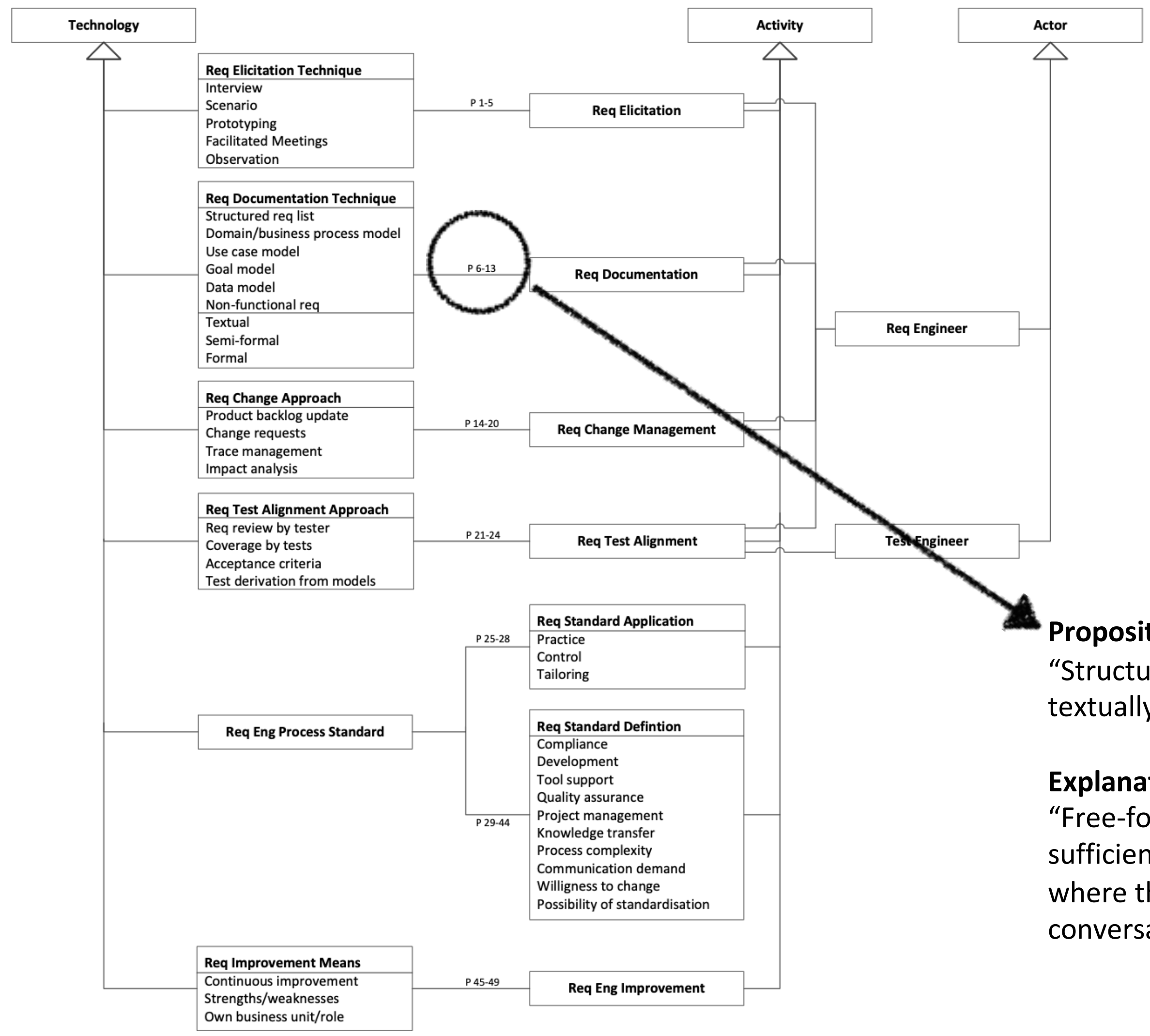
Source (example): Wagner, Mendez et al. Status Quo in Requirements Engineering: A Theory and a Global Family of Surveys, TOSEM 2018.

Exemplary framework for describing theories

xt factors)

theory is

Example



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

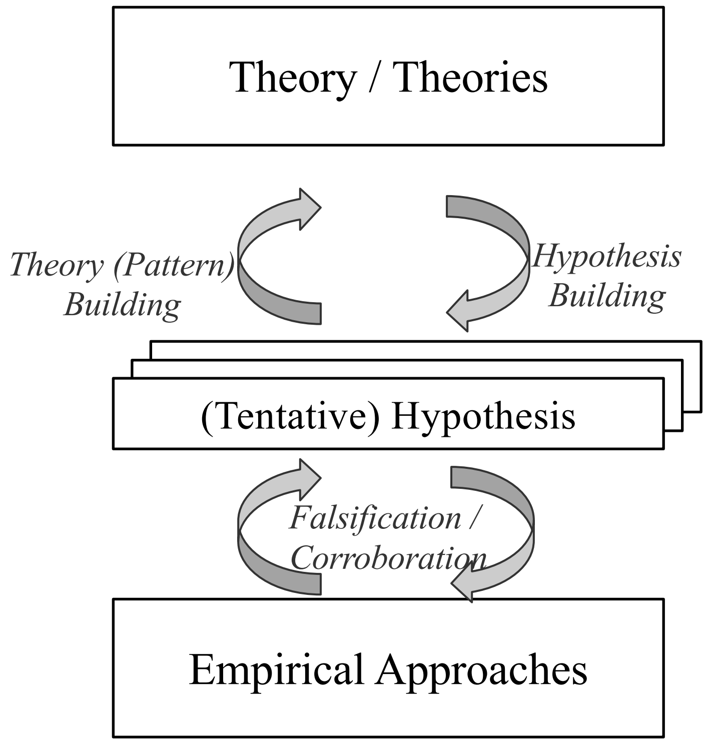
Explanation and Scope:
"Free-form and constraint textual requirements are sufficient for many contexts such as in agile projects where they only act as reminders for further conversations."

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

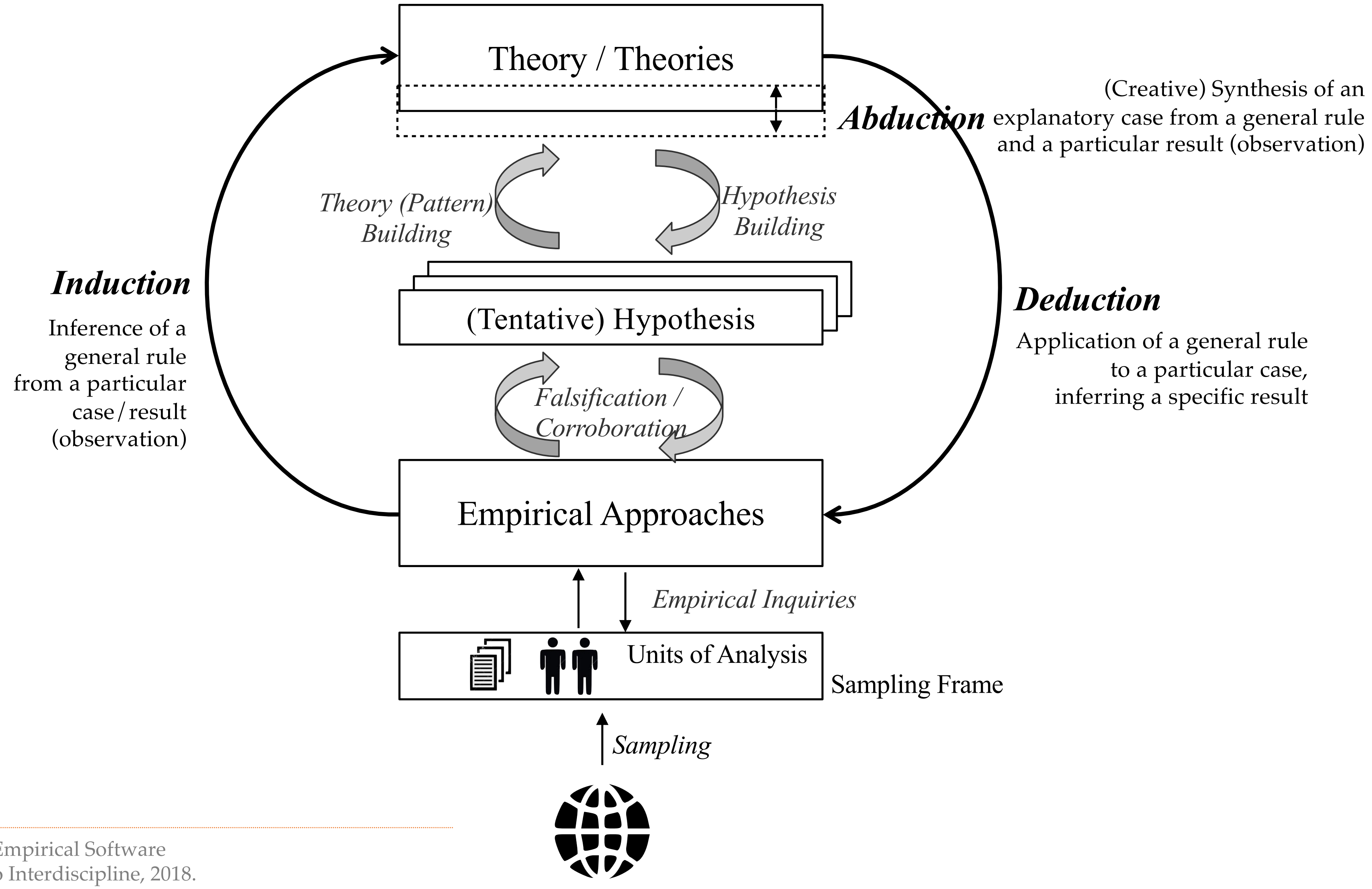
- "[…] based on hypotheses tested and verified multiple times by detached researchers" (J. Bortz and N. Döring, 2003)

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)

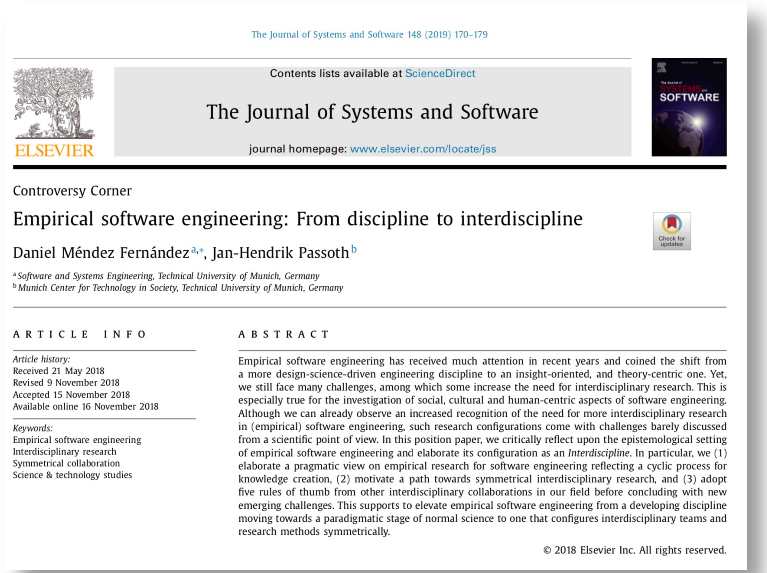
From real world phenomena to theories and back: The empirical life cycle


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

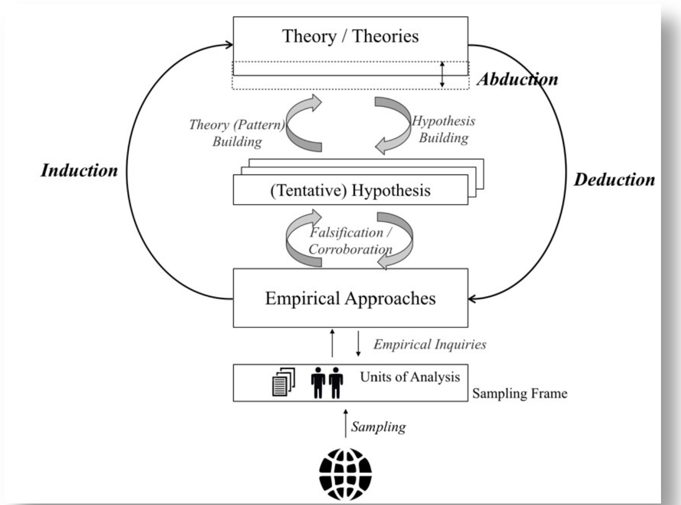
ARTICLE INFO

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.

Theory Building in Software Engineering

- Science and Theories in a Nutshell
… where we will briefly talk about the general notion of theories

- **State of Evidence in Software Engineering**
… where we will see why theory building is so important to our field

- Research Methods in Software Engineering
… where we will put research methods in a larger (philosophical) picture

- Qualitative "vs" quantitative research
… where we will briefly discuss different research approaches

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

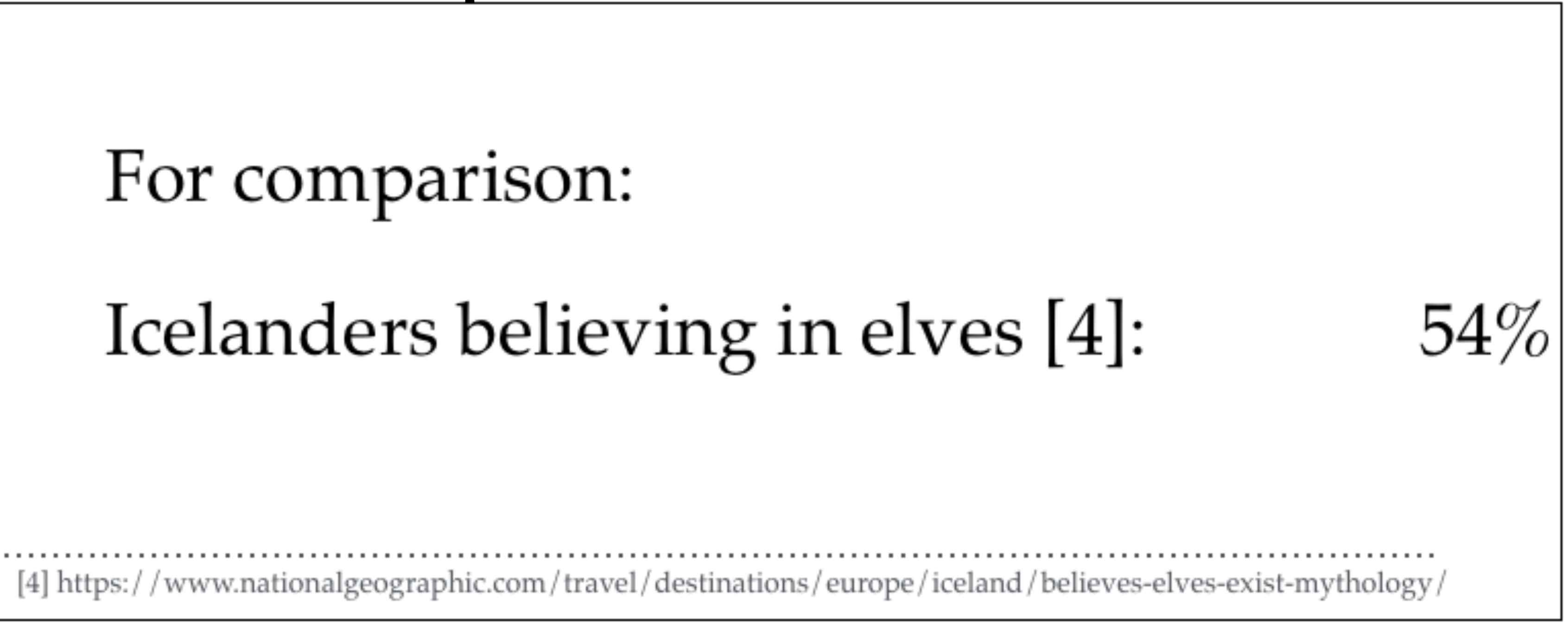
* Addressing the situation in the quantum mechanics research community, an analogy

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/

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

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

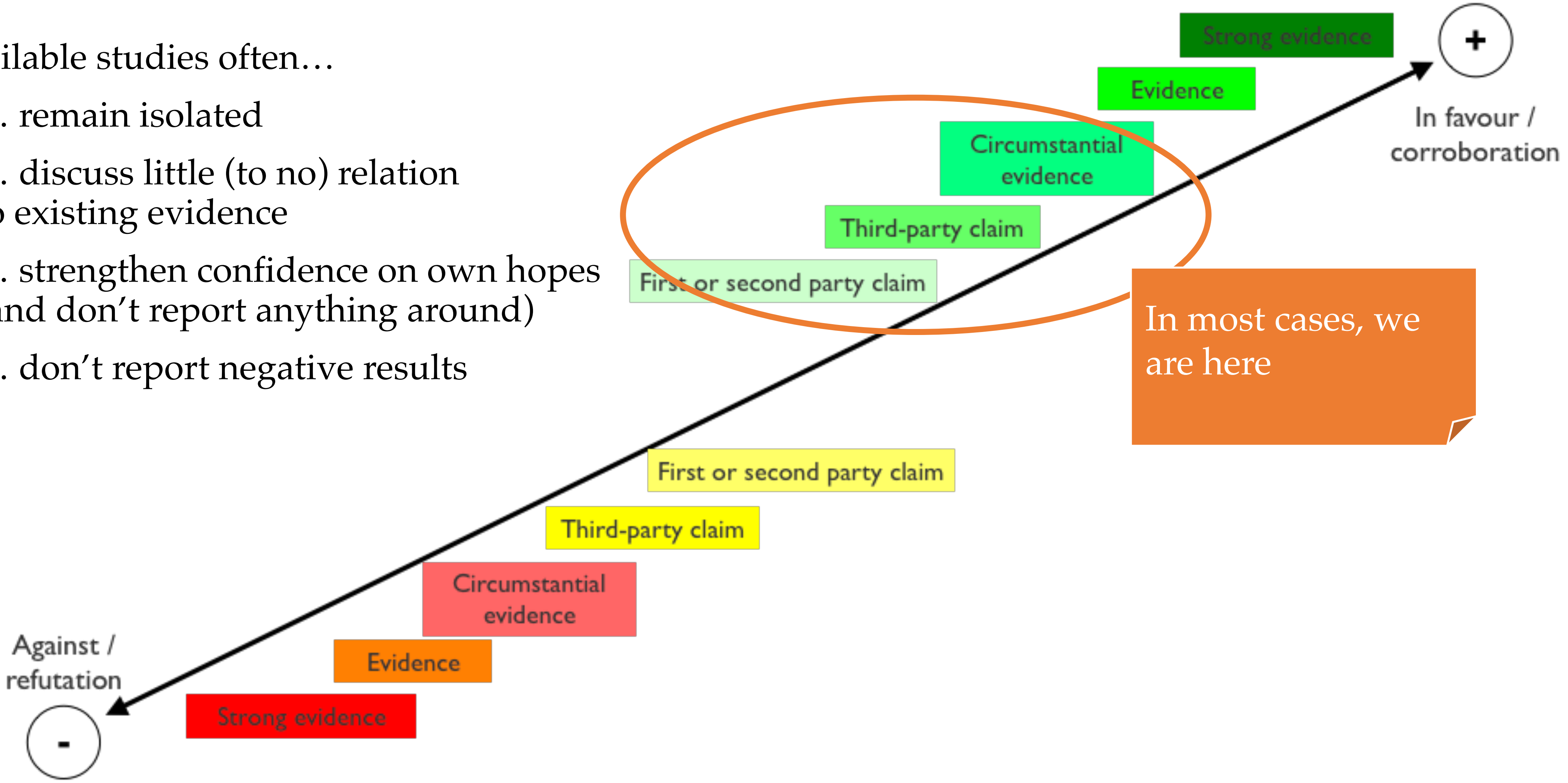
[2] Mavin, et al. Does Goal-Oriented Requirements Engineering Achieve its Goal?, 2017

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

Current state of evidence in SE

Available studies often…

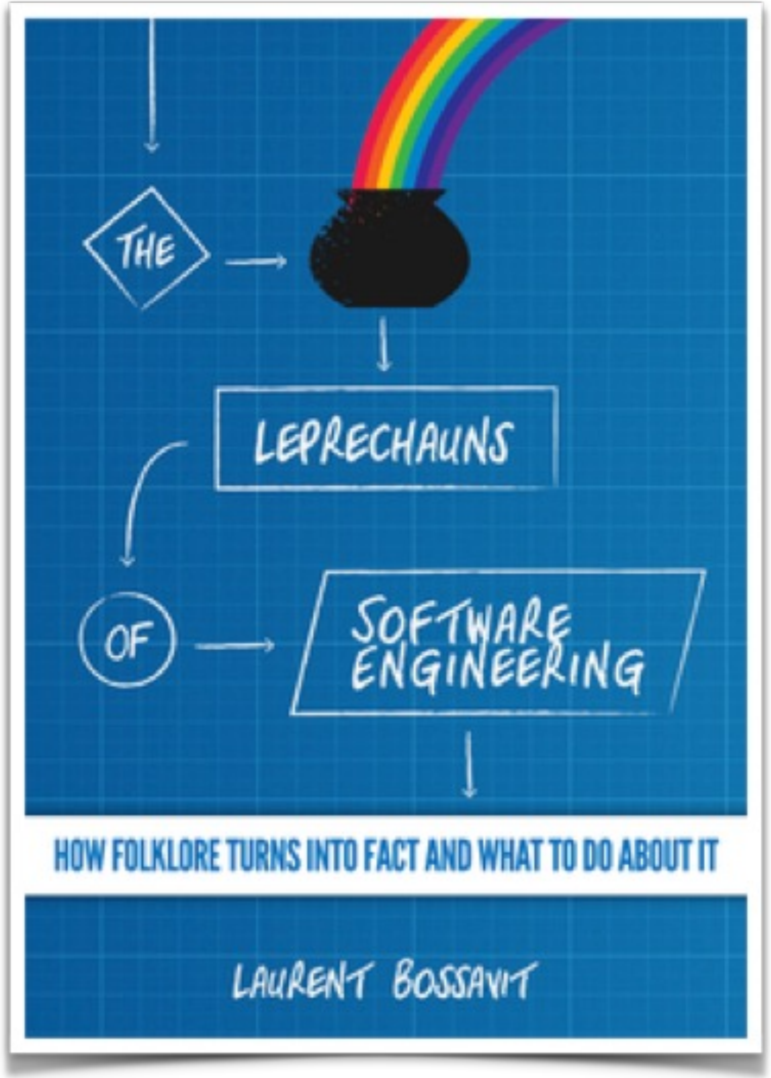
- … remain isolated
- … discuss little (to no) relation to existing evidence
- … strengthen confidence on own hopes (and don't report anything around)
- … don't report negative results



Conventional Wisdom in SE

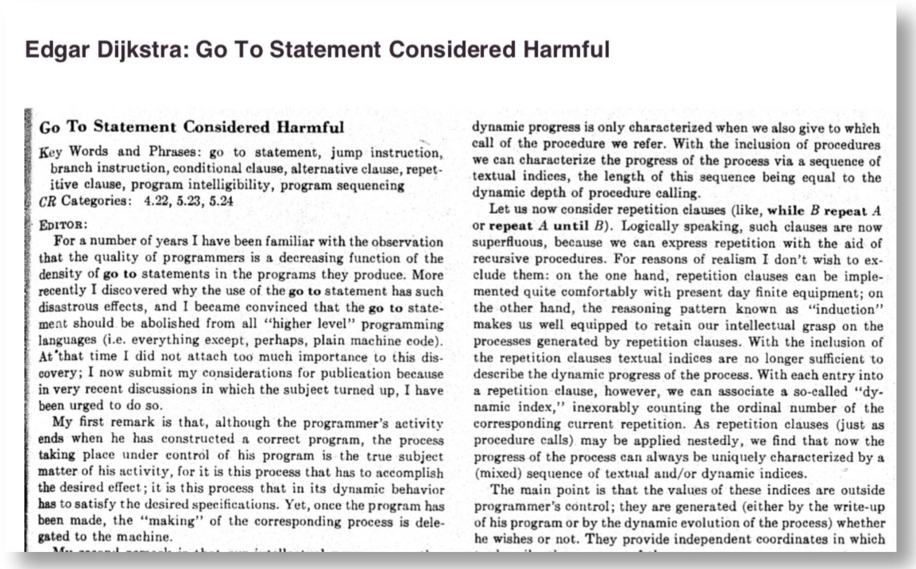
“Leprechauns”: Folklore turned into facts

- 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 “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.

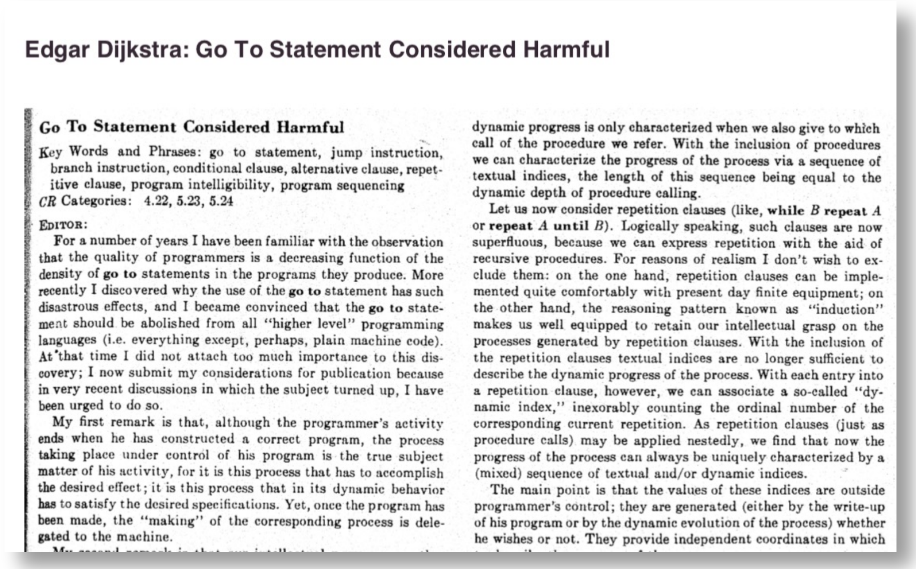
The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which

- Public exchange based on reasoning by argument (rationalist arguments)...
- … finally challenged by one single empirical study.

[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.

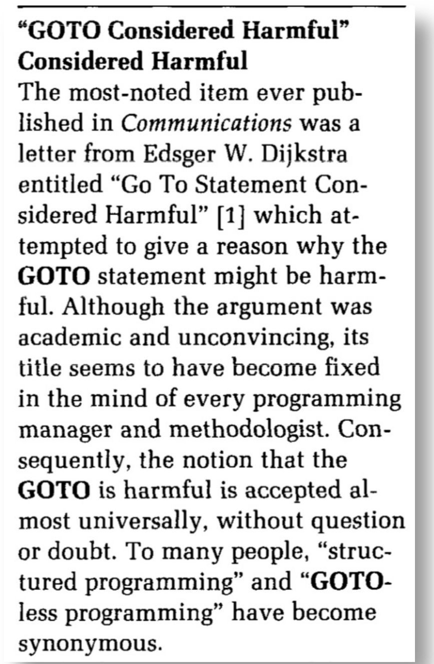
My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its 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



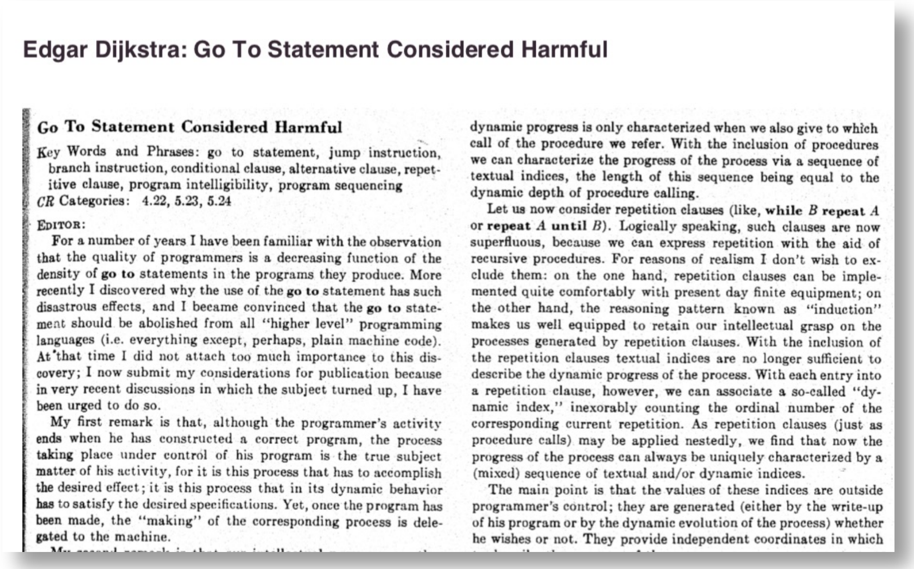
"GOTO Considered Harmful" Considered Harmful
The most-noted item ever published in *Communications* was a letter from Edsger W. Dijkstra entitled "Go To Statement Considered Harmful" [1] which attempted to give a reason why the **GOTO** statement might be harmful. Although the argument was academic and 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 challenged by one single empirical study.

[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.

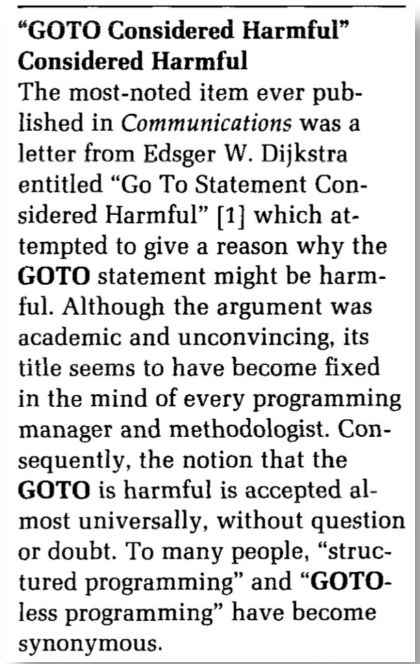
My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its 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.

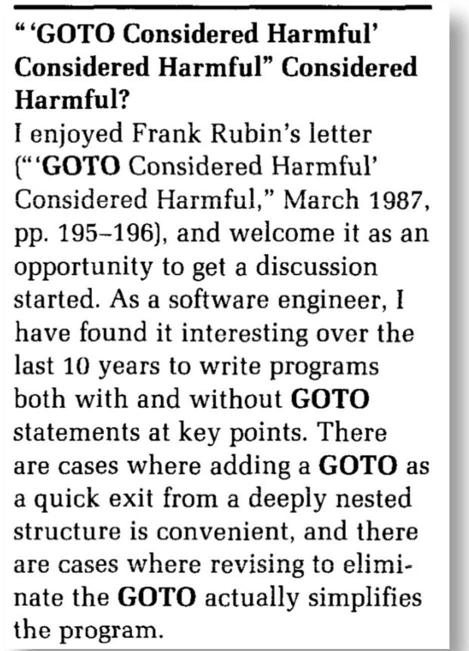
The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which

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



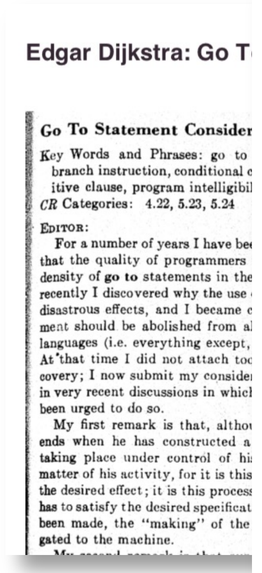
“‘GOTO Considered Harmful’ Considered Harmful” Considered Harmful?
I enjoyed Frank Rubin's letter (“‘**GOTO** Considered Harmful’ Considered Harmful,” March 1987, pp. 195–196), and welcome it as an opportunity to get a discussion started. As a software engineer, I have found it interesting over the last 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 challenged by one single empirical study.

[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 2015

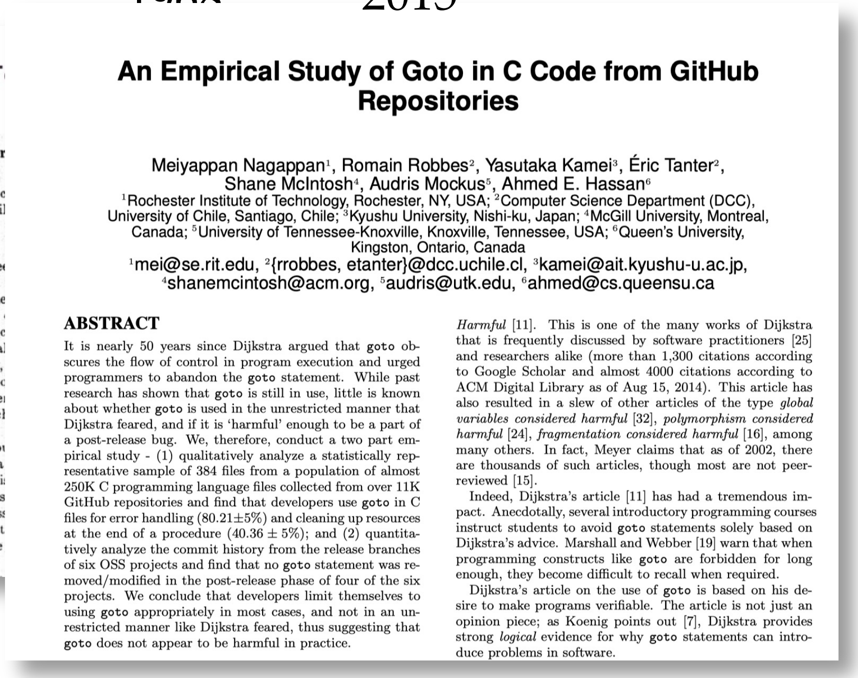


Edgar Dijkstra: Go T

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.



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 challenged by one single empirical study.

[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.

Takeaway

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

- Theory building and evaluation (i.e. empirical SE) are crucial

» Reason about the discipline and social phenomena involved

» Recognise and understand limits and effects of artefacts (technologies, techniques, processes, models, etc.) in their contexts

Theory Building in Software Engineering

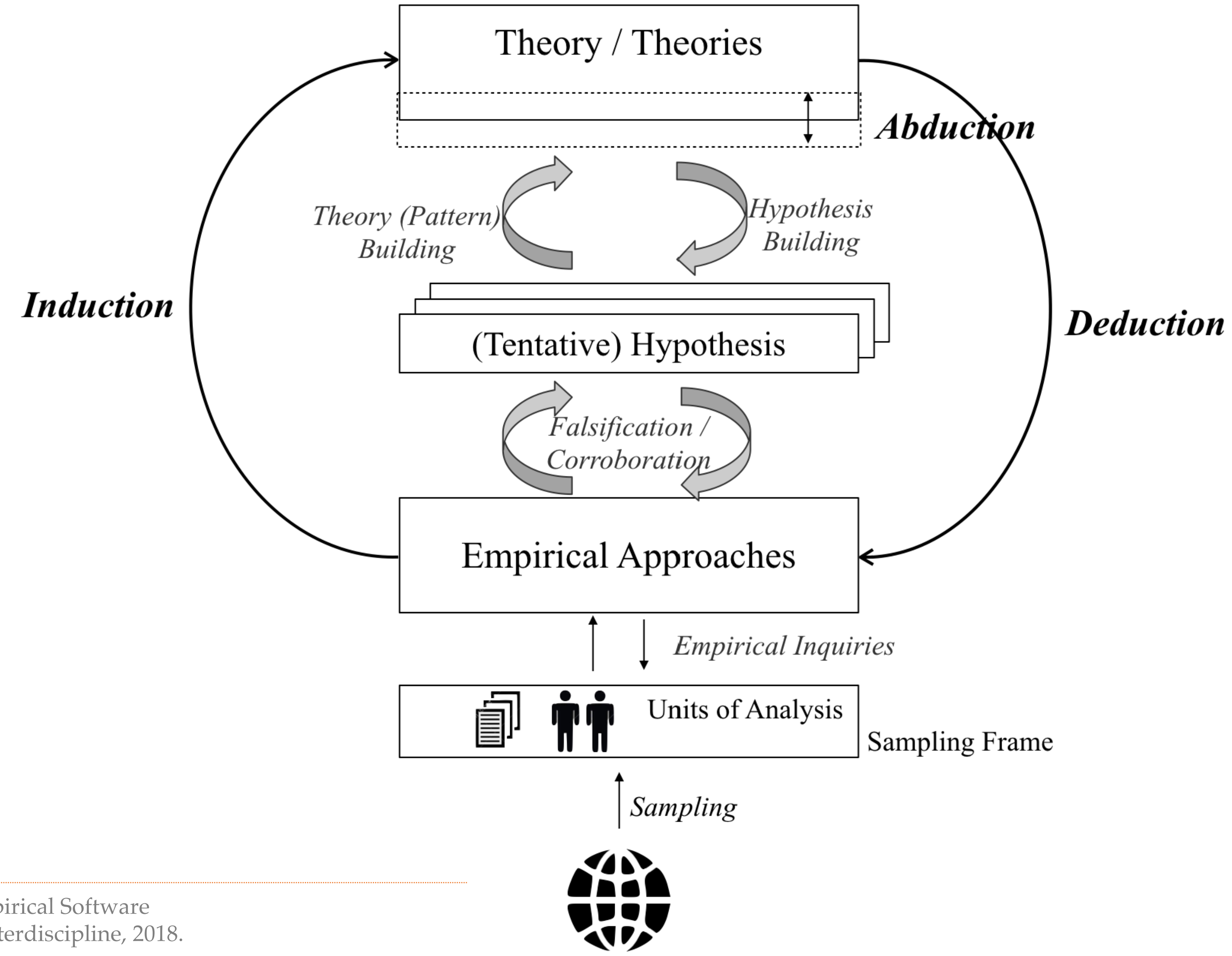
- Science and Theories in a Nutshell
… where we will briefly talk about the general notion of theories

- State of Evidence in Software Engineering
… where we will see why theory building is so important to our field

- **Research Methods in Software Engineering**
… where we will put research methods in a larger (philosophical) picture

- Qualitative "vs" quantitative research
… where we will briefly discuss different research approaches

Recap: The empirical Lifecycle

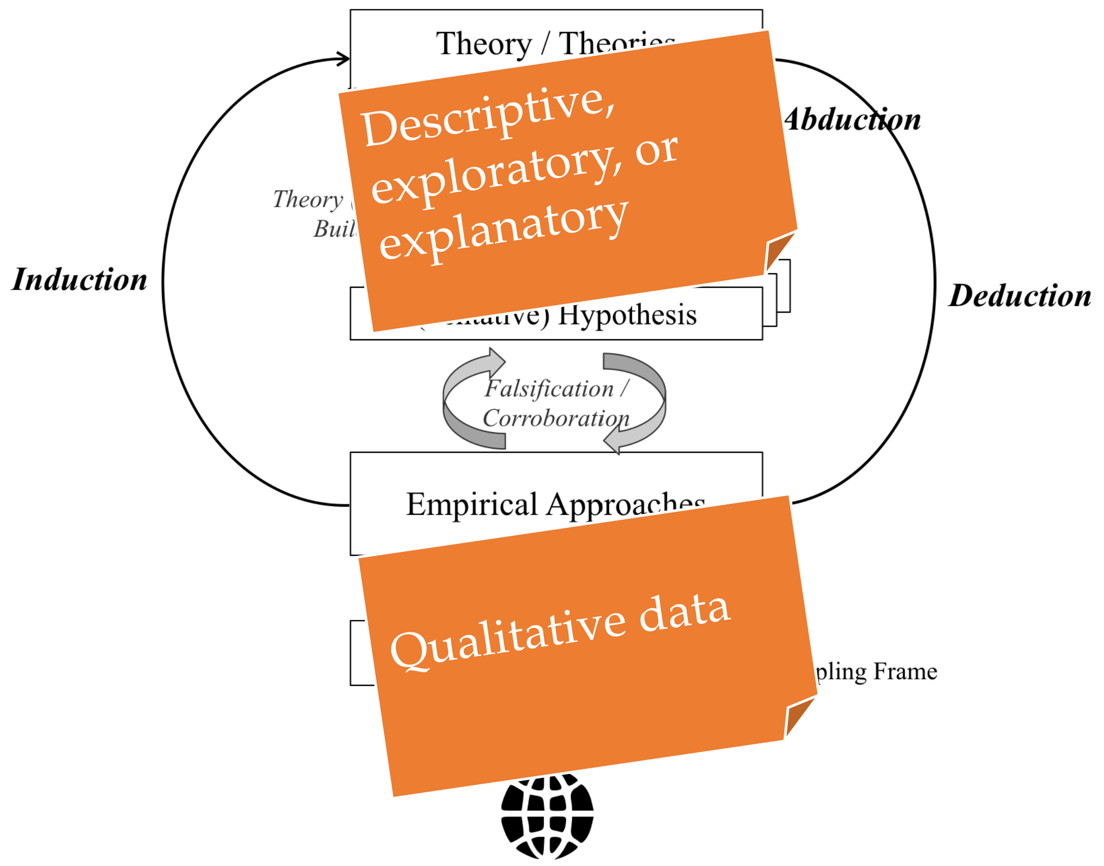


Source: Mendez and Passoth. Empirical Software Engineering: from Discipline to Interdiscipline, 2018.

(Empirical) methods

- Each method…
  - …has a specific purpose
  - …relies on a specific data type
- Purposes
  - Exploratory
  - Descriptive
  - Explanatory
  - Improving
- Data Types
  - Qualitative
  - Quantitative

Example: "Grounded Theory"



(Empirical) methods – where do they belong?

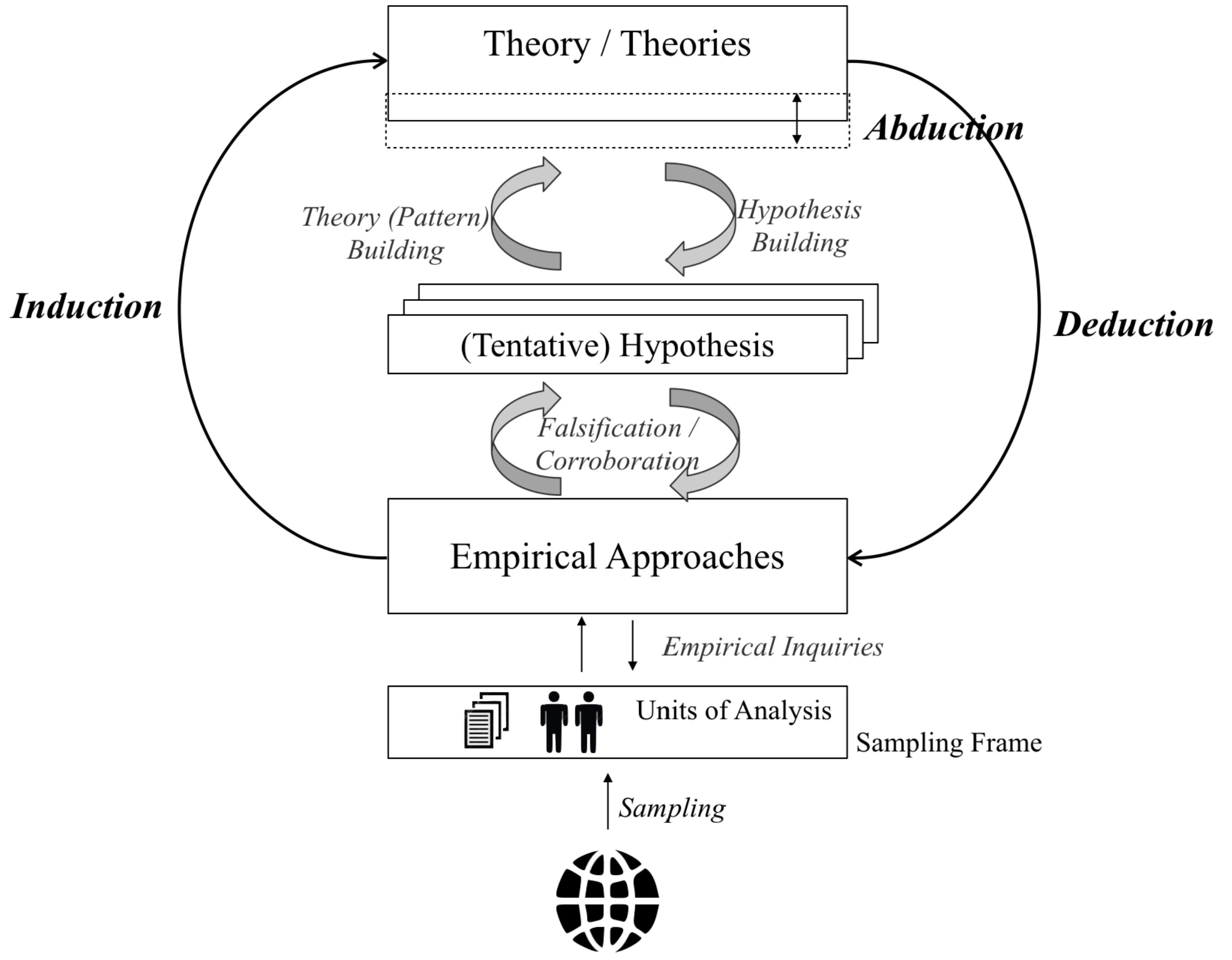
Ethnographic studies, Folklore gathering

Case studies / Field studies (Exploratory)

Case studies / Field studies (Confirmatory)

Formal analysis / logical reasoning

Survey research



Which research method(s) to use in which situation?

There is no such thing as a universal way of scientific practice

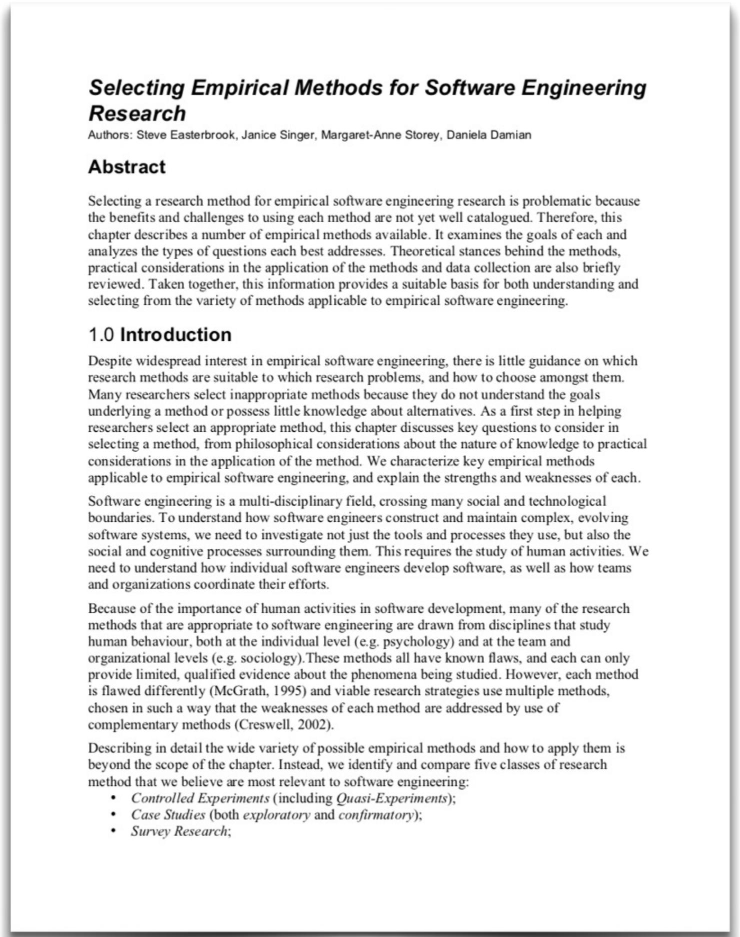
Method selection depends on many non-trivial questions

- What is the purpose of the study?
  Exploratory? Descriptive? Explanatory? Improving?
- What is the nature of the study?
  Inductive? Deductive?
- What is the relation to existing evidence?
  Building a new theory? "Testing" existing theory?
- What is the nature of the questions we ask?
  What-questions? Why-questions?
- What is the nature of the environment?
  Controlled environments? Realistic environments?
- What is the necessary sample?
  Population source?
  Units of analysis?

Criteria for selecting methods

Criteria for environment selection (and sampling)

Not trivial, but possible: checklists



Selecting Empirical Methods for Software Engineering Research

Authors: Steve Easterbrook, Janice Singer, Margaret-Anne Storey, Daniela Damian

Abstract

Selecting a research method for empirical software engineering research is problematic because the benefits and challenges to using each method are not yet well catalogued. Therefore, this chapter describes a number of empirical methods available. It examines the goals of each and analyzes the types of questions each best addresses. Theoretical stances behind the methods, practical considerations in the application of the methods and data collection are also briefly reviewed. Taken together, this information provides a suitable basis for both understanding and selecting from the variety of methods applicable to empirical software engineering.

1.0 **Introduction**

Despite widespread interest in empirical software engineering, there is little guidance on which research methods are suitable to which research problems, and how to choose amongst them. Many researchers select inappropriate methods because they do not understand the goals underlying a method or possess little knowledge about alternatives. As a first step in helping researchers select an appropriate method, this chapter discusses key questions to consider in selecting a method, from philosophical considerations about the nature of knowledge to practical considerations in the application of the method. We characterize key empirical methods applicable to empirical software engineering, and explain the strengths and weaknesses of each.

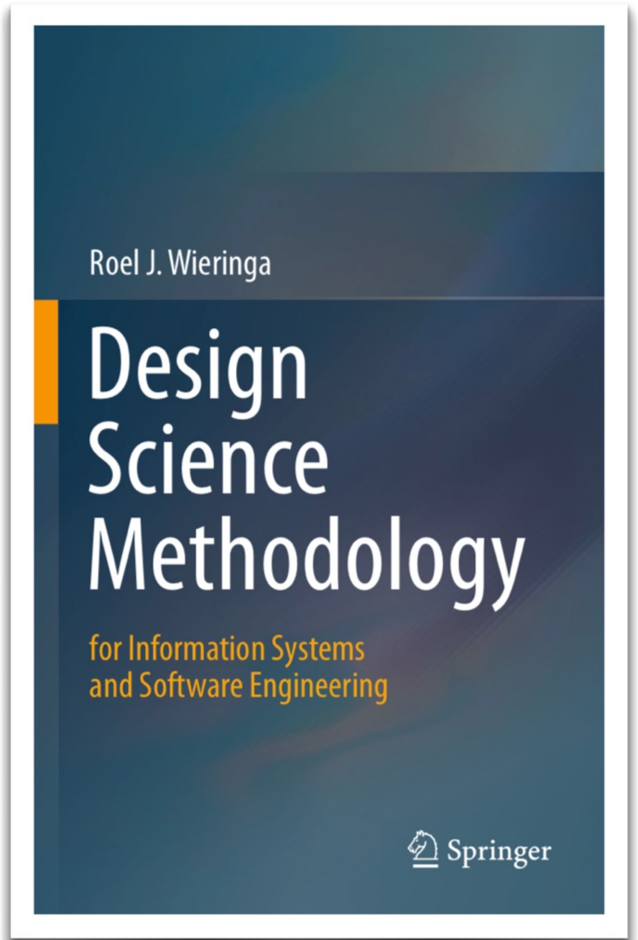
Software engineering is a multi-disciplinary field, crossing many social and technological boundaries. To understand how software engineers construct and maintain complex, evolving software systems, we need to investigate not just the tools and processes they use, but also the social and cognitive processes surrounding them. This requires the study of human activities. We need to understand how individual software engineers develop software, as well as how teams and organizations coordinate their efforts.

Because of the importance of human activities in software development, many of the research methods that are appropriate to software engineering are drawn from disciplines that study human behaviour, both at the individual level (e.g. psychology) and at the team and organizational levels (e.g. sociology).These methods all have known flaws, and each can only provide limited, qualified evidence about the phenomena being studied. However, each method is flawed differently (McGrath, 1995) and viable research strategies use multiple methods, chosen in such a way that the weaknesses of each method are addressed by use of complementary methods (Creswell, 2002).

Describing in detail the wide variety of possible empirical methods and how to apply them is beyond the scope of the chapter. Instead, we identify and compare five classes of research method that we believe are most relevant to software engineering:

- *Controlled Experiments* (including *Quasi-Experiments*);
- *Case Studies* (both *exploratory* and *confirmatory*);
- *Survey Research*;

Good starting point



Roel J. Wieringa

Design Science Methodology

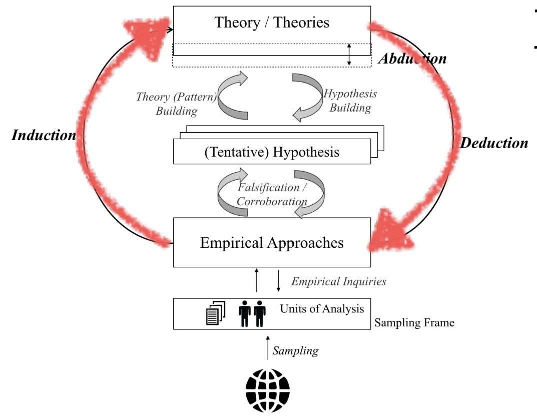
for Information Systems and Software Engineering

Springer

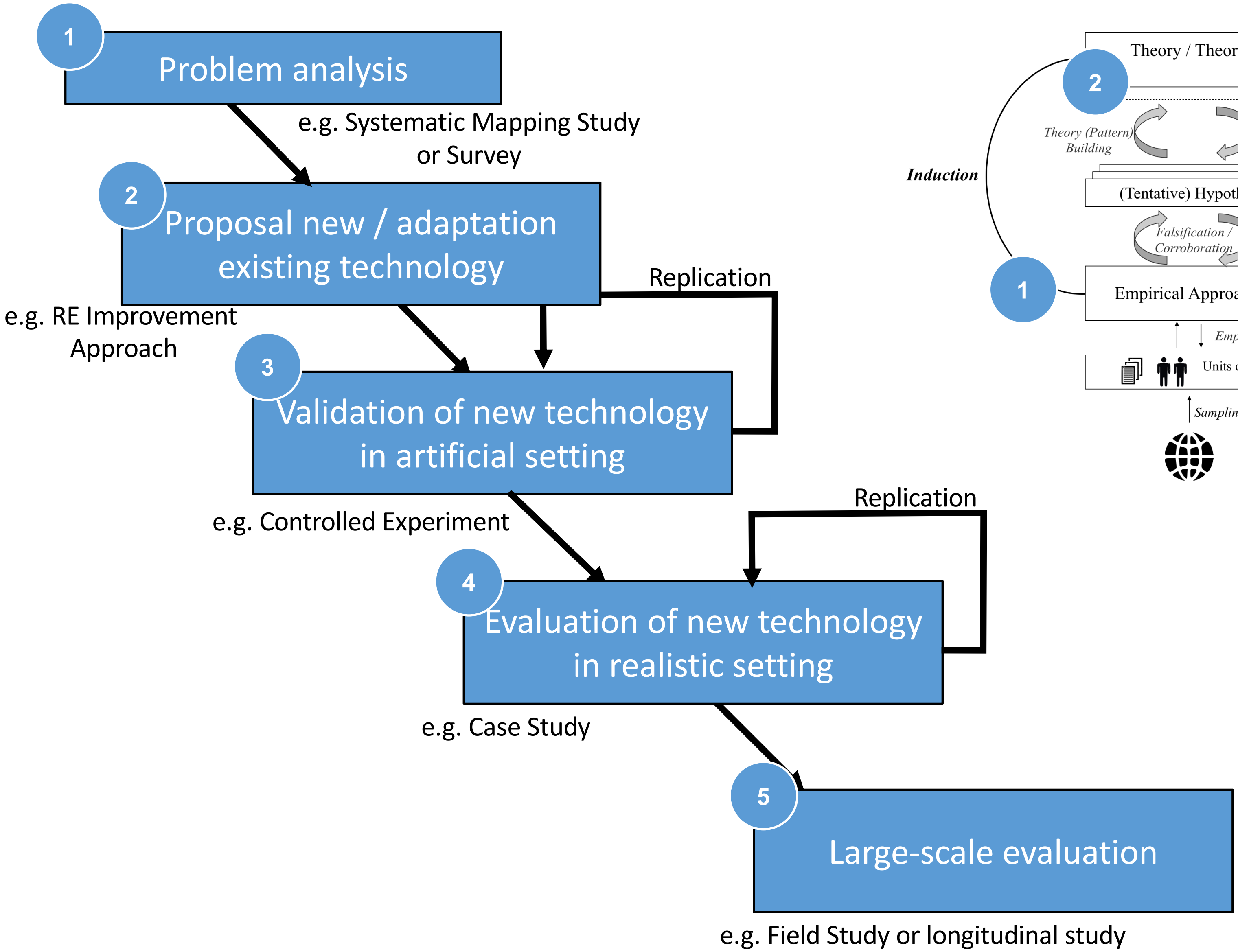
More advanced
- Chapter 16 + Appendix
- http://bit.ly/checklists-design_science

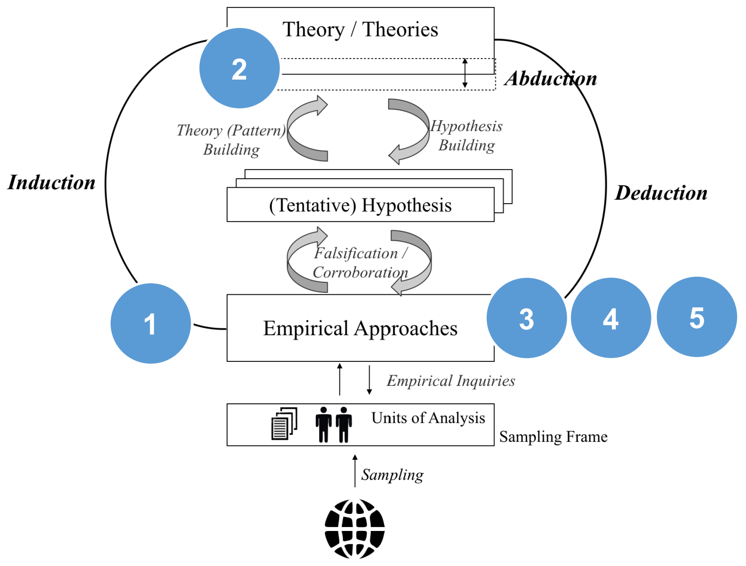
How to achieve scientific progress?

In step-wise iterations, with multiple methods
(aka “ ramme”)



Progress via multi-study approaches


1
Problem analysis
e.g. Systematic Mapping Study or Survey
2
Proposal new / adaptation existing technology
e.g. RE Improvement Approach
Replication
3
Validation of new technology in artificial setting
e.g. Controlled Experiment
Replication
4
Evaluation of new technology in realistic setting
e.g. Case Study
5
Large-scale evaluation
e.g. Field Study or longitudinal study



Theory Building in Software Engineering

- Science and Theories in a Nutshell
… where we will briefly talk about the general notion of theories

- State of Evidence in Software Engineering
… where we will see why theory building is so important to our field

- Research Methods in Software Engineering
… where we will put research methods in a larger (philosophical) picture

- **Qualitative "vs" quantitative research**
… where we will briefly discuss different research approaches

What is the difference between qualitative and quantitative research?

-- What do you think? --

Warning: EmSE emerges from natural science, thus, qualitative methods are often confronted with prejudice

"I prefer working with real data [not with qualitative data]"

— Anonymous ISERN member

"In contrast [to previous qualitative studies], this study attempts to obtain more scientific evidence in the form of objective, quantitative data."

— Anonymous ESEM 2018 author

Warning: EmSE emerges from natural science, thus, qualitative methods are often confronted with prejudice

"I prefer working with real data [not with qualitative data]"

— Anonymous ISERN member

"In contrast [to previous qualitative studies], this study attempts to obtain more scientific evidence in the form of objective, quantitative data."

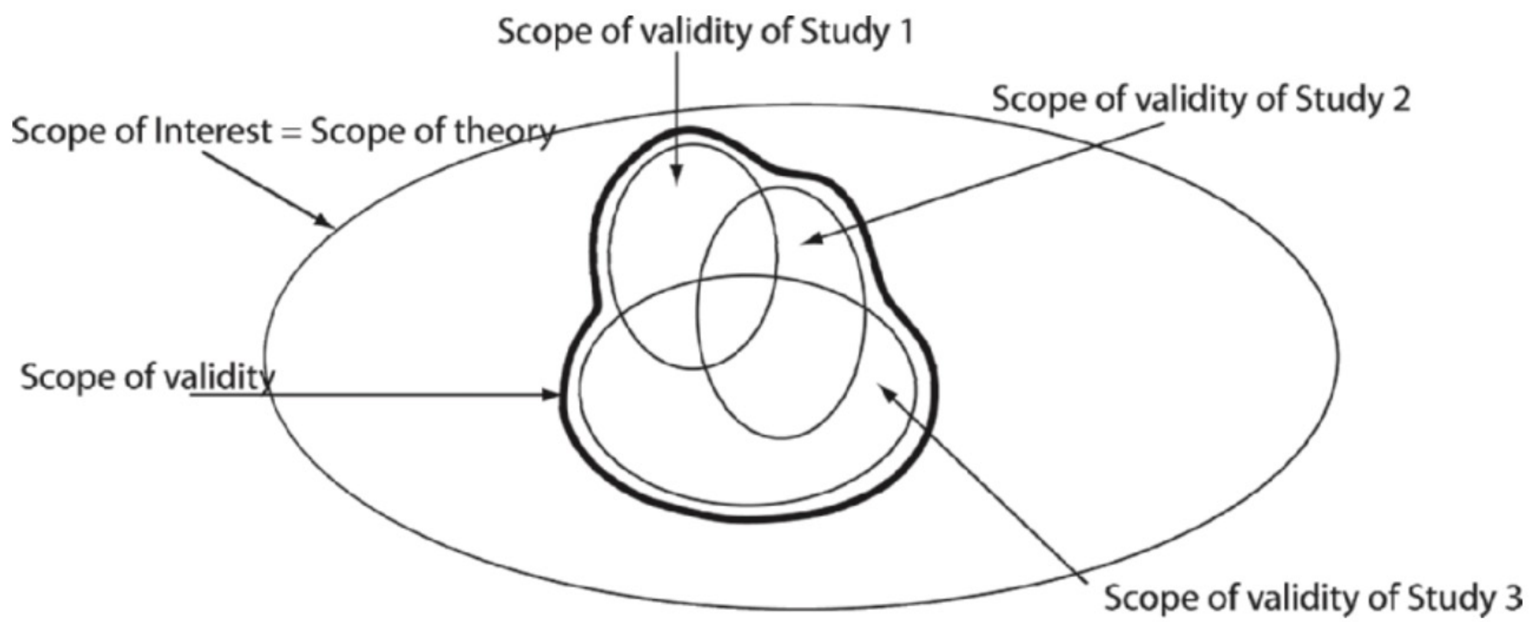
— Anonymous ESEM 2018 author

"With all due respect, DO NOT make such ridiculous claims that only quantitative studies are "scientific" and "objective" […]. There is NO more objectivity in numbers than there is in qualitative data. […] It is insulting and unnecessary and frankly naive to claim that because this study happens to use a bunch of numbers it is of better quality than qualitative studies"

— Anonymous ESEM 2018 reviewer

Postulate I

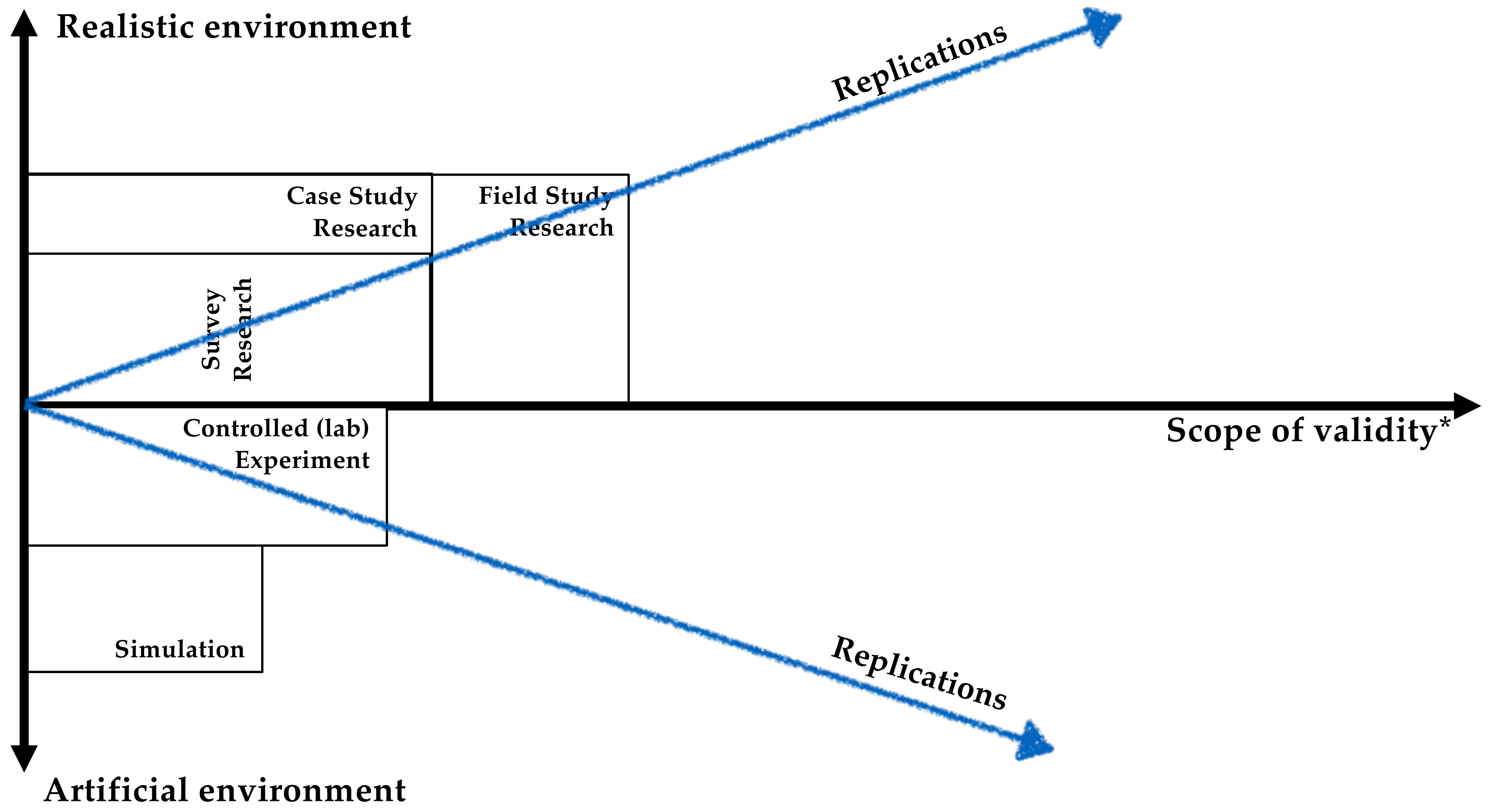
<u>Every</u> research approach has a specific scope of validity only


Scope of validity of Study 1
Scope of validity of Study 2
Scope of Interest = Scope of theory
Scope of validity
Scope of validity of Study 3

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

Postulate II

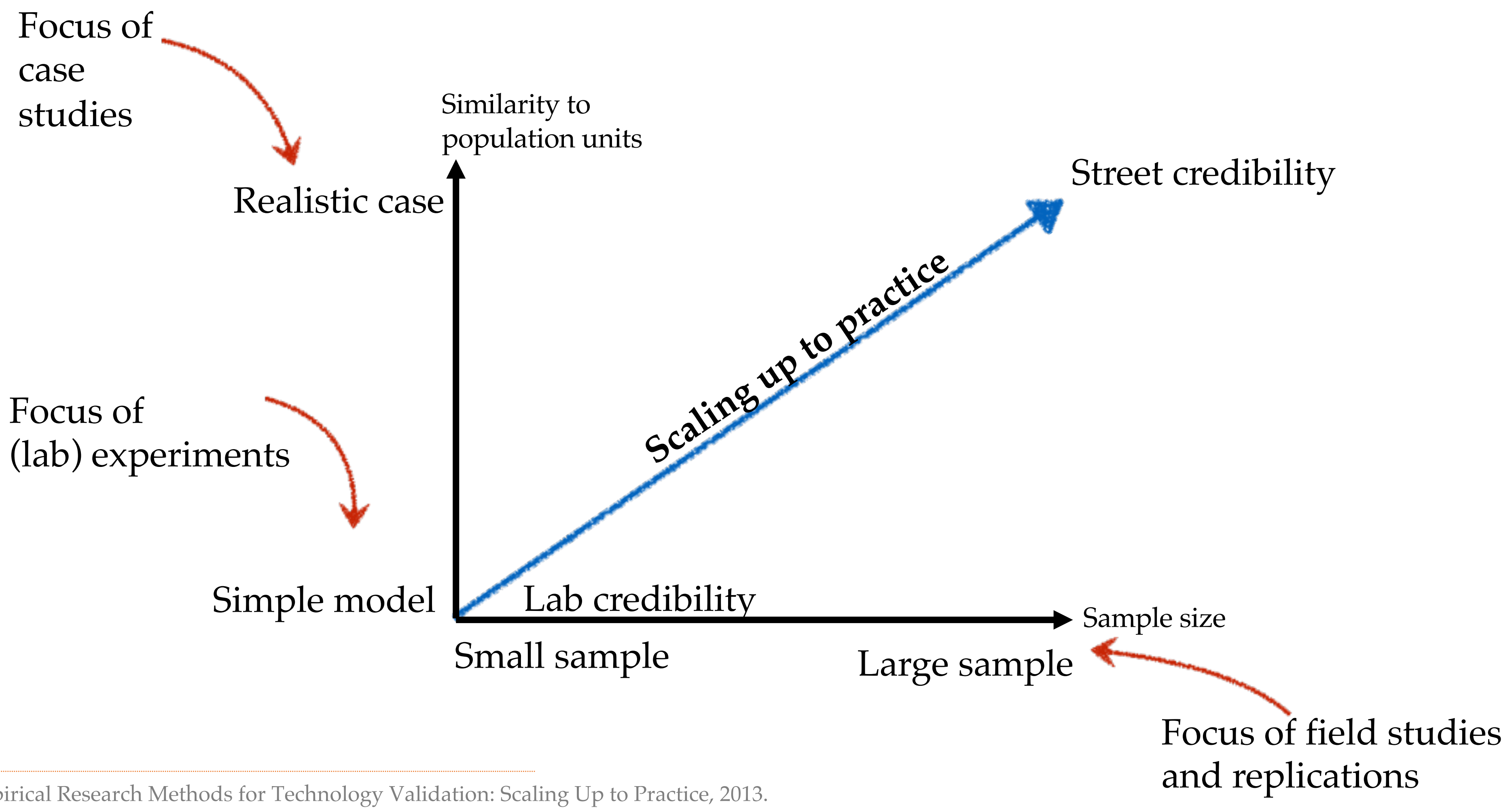
Scope of validity ≠ Degree of reality



* Extremely simplified view to orient discussions

Postulate III

Different research methods complement each other in scaling up to practice



Source: Wieringa R. Empirical Research Methods for Technology Validation: Scaling Up to Practice, 2013.

The essence

Quantitative studies focus primarily on the kind of evidence that will enable you to understand **what is going on.**

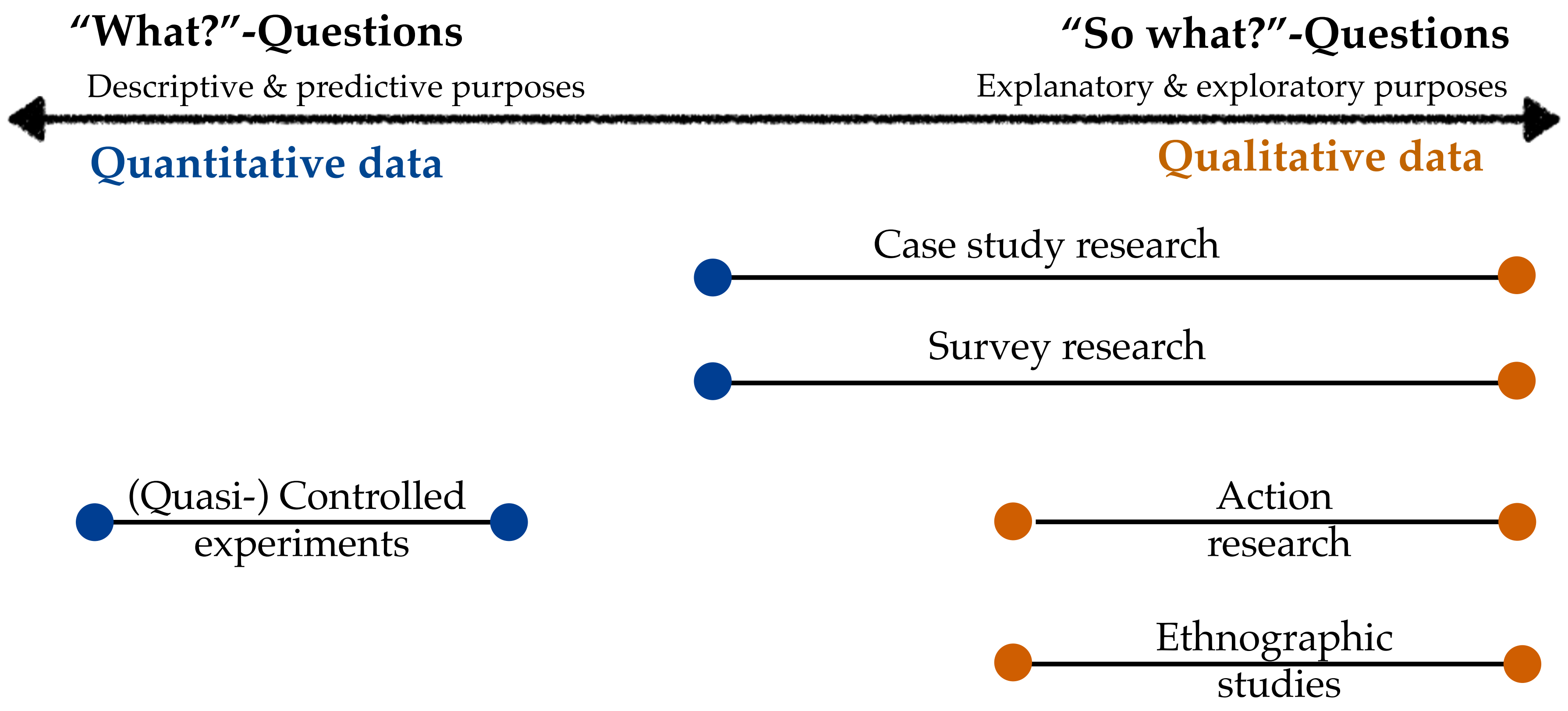
Qualitative studies focus primarily on the kind of evidence that will enable you to understand the **meaning [and purpose, reasoning, etc] of what is going on.**

Qualitative and quantitative methods have complementary purposes, strengths, and limitations in theory building.

Two complementary approaches (+1)

- **Quantitative research:** Describing events and finding causes to predict similar events in the future – "What?"-questions
  - (Typically) focus on what, how much, or how many
  - (Typically) in numerical forms
  - (Typically) descriptive purpose
- **Qualitative research:** Understanding meaning [and purpose, reasoning, etc.] of a phenomenon for those involved – "So what?"-questions
  - (Typically) focus on why/meaning, and how people interpret their experiences
  - (Typically) in variety of non-numerical forms, like texts, diagrams, etc.
  - (Typically) exploratory or explanatory purpose
- (Mix-method research)

Two complementary approaches



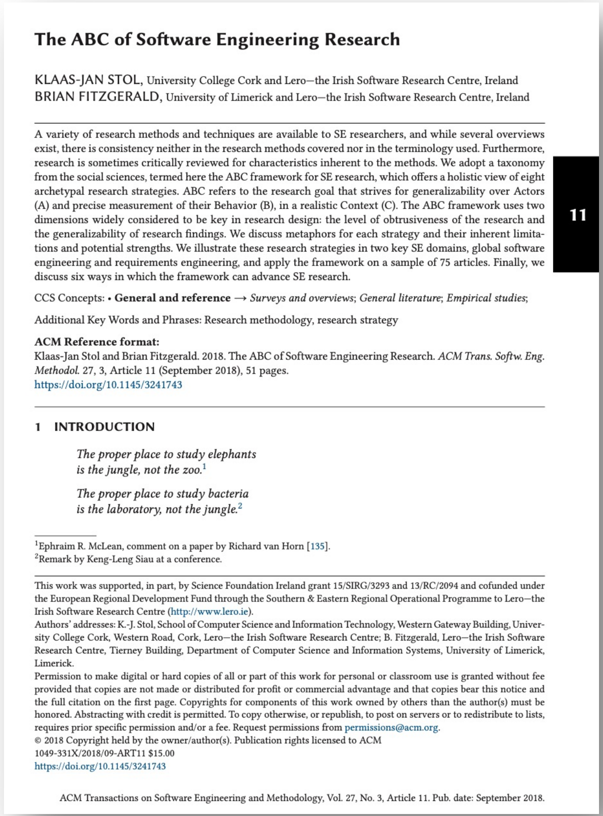
Qualitative "vs" quantitative research

* For all, you can add a "in tendency"

	Quantitative research	Qualitative research
Goals	• Description, control, prediction • Hypothesis testing (typically)	• Understanding, reasoning, explanations, descriptions, meaning (to subjects), discovery • Hypothesis generation (typically)
Design characteristics	• Predetermined, structured / fixed • Deductive, statistical	• Flexible, evolving, emergent • Inductive, constant comparative
Samples	• Large(r), random, representative	• Small, non-random (sometimes even opportunistic), purposeful, theoretical
Data collection	• Inanimate instruments (tests, surveys, questionnaires, etc.)	• Researcher often primary instrument • Interviews, observations, document analysis, …
Findings	• Precise and statistical	• Comprehensive, holistic, rich descriptions

Adopted from: Da Silva. Tutorial given at the Ibero-American Conference on Software Engineering, 2018 (Bogota, Colombia)

Further reading: Selected papers



The ABC of Software Engineering Research

KLAAS-JAN STOL, University College Cork and Lero—the Irish Software Research Centre, Ireland
BRIAN FITZGERALD, University of Limerick and Lero—the Irish Software Research Centre, Ireland

A variety of research methods and techniques are available to SE researchers, and while several overviews exist, there is consistency neither in the research methods covered nor in the terminology used. Furthermore, research is sometimes critically reviewed for characteristics inherent to the methods. We adopt a taxonomy from the social sciences, termed here the ABC framework for SE research, which offers a holistic view of eight archetypal research strategies. ABC refers to the research goal that strives for generalizability over Actors (A) and precise measurement of their Behavior (B), in a realistic Context (C). The ABC framework uses two dimensions widely considered to be key in research design: the level of obtrusiveness of the research and the generalizability of research findings. We discuss metaphors for each strategy and their inherent limitations and potential strengths. We illustrate these research strategies in two key SE domains, global software engineering and requirements engineering, and apply the framework on a sample of 75 articles. Finally, we discuss six ways in which the framework can advance SE research.

11

CCS Concepts: • **General and reference** → *Surveys and overviews*; *General literature*; *Empirical studies*;

Additional Key Words and Phrases: Research methodology, research strategy

ACM Reference format:
Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of Software Engineering Research. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 11 (September 2018), 51 pages.
https://doi.org/10.1145/3241743

1 INTRODUCTION

The proper place to study elephants
is the jungle, not the zoo.[1]

The proper place to study bacteria
is the laboratory, not the jungle.[2]

[1]Ephraim R. McLean, comment on a paper by Richard van Horn [135].
[2]Remark by Keng-Leng Siau at a conference.

This work was supported, in part, by Science Foundation Ireland grant 15/SIRG/3293 and 13/RC/2094 and cofunded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero—the Irish Software Research Centre (http://www.lero.ie).
Authors' addresses: K.-J. Stol, School of Computer Science and Information Technology, Western Gateway Building, University College Cork, Western Road, Cork, Lero—the Irish Software Research Centre; B. Fitzgerald, Lero—the Irish Software Research Centre, Tierney Building, Department of Computer Science and Information Systems, University of Limerick, Limerick.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM
1049-331X/2018/09-ART11 $15.00
https://doi.org/10.1145/3241743

ACM Transactions on Software Engineering and Methodology, Vol. 27, No. 3, Article 11. Pub. date: September 2018.

Empir Software Eng (2009) 14:131–164
DOI 10.1007/s10664-008-9102-8

Guidelines for conducting and reporting case study research in software engineering

Per Runeson · Martin Höst

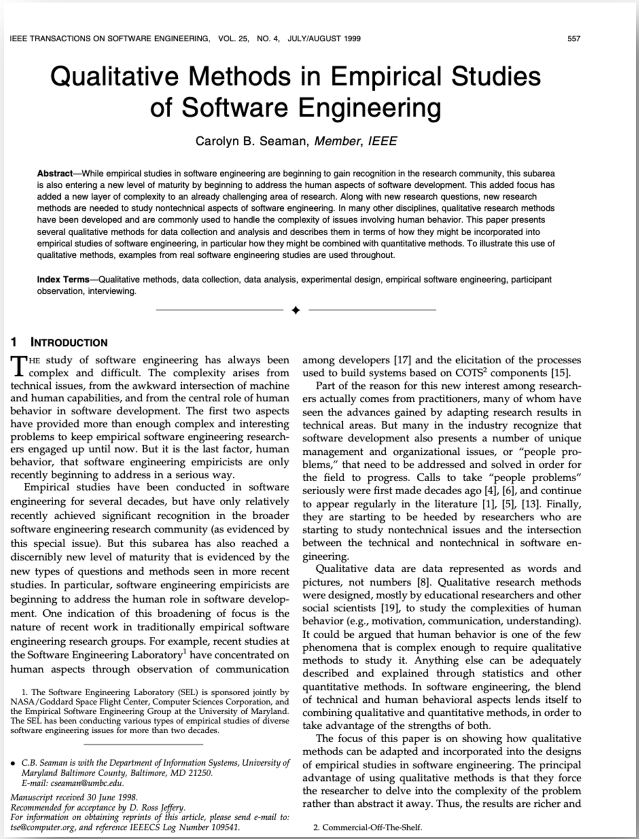
Published online: 19 December 2008
© The Author(s) 2008. This article is published with open access at Springerlink.com
Editor: Dag Sjøberg

Abstract Case study is a suitable research methodology for software engineering research since it studies contemporary phenomena in its natural context. However, the understanding of what constitutes a case study varies, and hence the quality of the resulting studies. This paper aims at providing an introduction to case study methodology and guidelines for researchers conducting case studies and readers studying reports of such studies. The content is based on the authors' own experience from conducting and reading case studies. The terminology and guidelines are compiled from different methodology handbooks in other research domains, in particular social science and information systems, and adapted to the needs in software engineering. We present recommended practices for software engineering case studies as well as empirically derived and evaluated checklists for researchers and readers of case study research.

Keywords Case study · Research methodology · Checklists · Guidelines

1 Introduction

The acceptance of empirical studies in software engineering and their contributions to increasing knowledge is continuously growing. The analytical research paradigm is not sufficient for investigating complex real life issues, involving humans and their interactions with technology. However, the overall share of empirical studies is negligibly small in computer science research; Sjøberg et al. (2005), found 103 experiments in 5,453 articles Ramesh et al. (2004) and identified less than 2% experiments with human subjects, and only 0.16% field studies among 628 articles. Further, existing work on empirical research



IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 25, NO. 4, JULY/AUGUST 1999 557

Qualitative Methods in Empirical Studies of Software Engineering

Carolyn B. Seaman, *Member, IEEE*

Abstract—While empirical studies in software engineering are beginning to gain recognition in the research community, this subarea is also entering a new level of maturity by beginning to address the human aspects of software development. This added focus has added a new layer of complexity to an already challenging area of research. Along with new research questions, new research methods are needed to study nontechnical aspects of software engineering. In many other disciplines, qualitative research methods have been developed and are commonly used to handle the complexity of issues involving human behavior. This paper presents several qualitative methods for data collection and analysis and describes them in terms of how they might be incorporated into empirical studies of software engineering, in particular how they might be combined with quantitative methods. To illustrate this use of qualitative methods, examples from real software engineering studies are used throughout.

Index Terms—Qualitative methods, data collection, data analysis, experimental design, empirical software engineering, participant observation, interviewing.

1 INTRODUCTION

THE study of software engineering has always been complex and difficult. The complexity arises from technical issues, from the awkward intersection of machine and human capabilities, and from the central role of human behavior in software development. The first two aspects have provided more than enough complex and interesting problems to keep empirical software engineering researchers engaged up until now. But it is the last factor, human behavior, that software engineering empiricists are only recently beginning to address in a serious way.

Empirical studies have been conducted in software engineering for several decades, but have only relatively recently achieved significant recognition in the broader software engineering research community (as evidenced by this special issue). But this subarea has also reached a discernibly new level of maturity that is evidenced by the new types of questions and methods seen in more recent studies. In particular, software engineering empiricists are beginning to address the human role in software development. One indication of this broadening of focus is the nature of recent work in traditionally empirical software engineering research groups. For example, recent studies at the Software Engineering Laboratory[1] have concentrated on human aspects through observation of communication among developers [17] and the elicitation of the processes used to build systems based on COTS[2] components [15].

Part of the reason for this new interest among researchers actually comes from practitioners, many of whom have seen the advances gained by adapting research results in technical areas. But many in the industry recognize that software development also presents a number of unique management and organizational issues, or "people problems," that need to be addressed and solved in order for the field to progress. Calls to take "people problems" seriously were first made decades ago [4], [6], and continue to appear regularly in the literature [1], [5], [13]. Finally, they are starting to be heeded by researchers who are starting to study nontechnical issues and the intersection between the technical and nontechnical in software engineering.

Qualitative data are data represented as words and pictures, not numbers [8]. Qualitative research methods were designed, mostly by educational researchers and other social scientists [19], to study the complexities of human behavior (e.g., motivation, communication, understanding). It could be argued that human behavior is one of the few phenomena that is complex enough to require qualitative methods to study it. Anything else can be adequately described and explained through statistics and other quantitative methods. In software engineering, the blend of technical and human behavioral aspects lends itself to combining qualitative and quantitative methods, in order to take advantage of the strengths of both.

The focus of this paper is on showing how qualitative methods can be adapted and incorporated into the designs of empirical studies in software engineering. The principal advantage of using qualitative methods is that they force the researcher to delve into the complexity of the problem rather than abstract it away. Thus, the results are richer and

1. The Software Engineering Laboratory (SEL) is sponsored jointly by NASA/Goddard Space Flight Center, Computer Sciences Corporation, and the Empirical Software Engineering Group at the University of Maryland. The SEL has been conducting various types of empirical studies of diverse software engineering issues for more than two decades.

- *C.B. Seaman is with the Department of Information Systems, University of Maryland Baltimore County, Baltimore, MD 21250.*
  E-mail: cseaman@umbc.edu.

Manuscript received 30 June 1998.
Recommended for acceptance by D. Ross Jeffery.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 109541.

2. Commercial-Off-The-Shelf.

Recommended starting point

Further reading on terminological demarcation and key characteristics of methods

Theory Building in Software Engineering

- Science and Theories in a Nutshell
… where we will briefly talk about the general notion of theories

- State of Evidence in Software Engineering
… where we will see why theory building is so important to our field

- Research Methods in Software Engineering
… where we will put research methods in a larger (philosophical) picture

- Qualitative “vs” quantitative research
… where we will briefly discuss different research approaches

Thank you!

Key Takeaways

Last but not least...

1 Let's use the breakout sessions to jointly discuss research strategies, methods, and their (case-based) application in detail.

2 I am organising a summer school on human factors in software engineering (hfse.school).
Interested? Approach me!