Empirical Software Engineering

Yann-Gaël Guéhéneuc and Foutse Khomh

Abstract Software engineering as a discipline exists since the 1960s, when participants of the NATO Software Engineering Conference in 1968 at Garmisch, Germany, recognised that there was a "software crisis" due to the increased complexity of the systems and of the software running (on) these systems. The software crisis led to the acknowledgement that software engineering is more than computing theories and efficiency of code and that it requires dedicated research. Thus, this crisis was the starting point of software engineering research. Software engineering research acknowledged early that software engineering is fundamentally an empirical discipline, thus further distinguishing computer science from software engineering, because (1) software is immaterial and does not obey physical laws and (2) software is written by people for people. In this chapter, we first introduce the concepts and principles on which empirical software engineering is based. Then, using these concepts and principles, we describe seminal works that led to the inception and popularisation of empirical software engineering research. We use these seminal works to discuss some idioms, patterns, and styles in empirical software engineering before discussing some challenges that empirical software engineering must overcome in the (near) future. Finally, we conclude and suggest further readings and future directions.

All authors have contributed equally to this chapter.

Y.-G. Guéhéneuc
Polytechnque Montréal and Concordia Universit, Montreal, QC, Canada
e-mail: yann-gael.gueheneuc@polymtl.ca

F. Khomh
Polytechnque Montréal, Montreal, QC, Canada
e-mail: foutse.khomh@polymtl.ca

S. Cha et al. (eds.), *Handbook of Software Engineering*,
https://doi.org/10.1007/978-3-030-00262-6_7

1 Introduction

Software engineering as a discipline exists since the 1960s, when participants of the NATO Software Engineering Conference in 1968 at Garmisch, Germany [39] recognised that there was a "software crisis" due to the increased complexity of the systems and of the software running (on) these systems. This increased complexity of the systems was becoming unmanageable by the hardware developers who were also often the software developers, because they knew best the hardware and there was little knowledge of software development processes, methods, and tools. Many software developers at the time, as well as the participants of the NATO Conference, realised that software development requires dedicated processes, methods, and tools and that they were separate from the actual hardware systems: how these were built and how the software systems would run on them.

Until the software crisis, research mostly focused on the theoretical aspects of software systems, in particular the algorithms and data structures used to write software systems [32], or on the practical aspects of software systems, in particular the efficient compilation of software for particular hardware systems [51]. The software crisis led to the acknowledgement that software engineering is more than computing theories and efficiency of code and that it requires dedicated research. Thus, this crisis was the starting point of software engineering research. It also yielded the distinction between the disciplines of computer science and software engineering. Computer science research pertains to understanding and proposing theories and methods related to the efficient computation of algorithms. Software engineering research pertains to all aspects of engineering software systems: from software development processes [24] to debugging methods [36], from theories of software developers' comprehension [67] to the impact of programming constructs [20], and from studying post-mortems of software development [65] to analysing the communications among software developers [21].

Software engineering research has had a tremendous impact on software development in the past decades, contributing with advances in processes, for example, with agile methods, in debugging methods, for example, with integrated development environments, and in tools, in particular with refactorings. It has also contributed to improving the quality of software systems, bridging research and practice, by formalising, studying, and popularising good practices. Software engineering research acknowledged early that software engineering is *fundamentally* an empirical discipline—thus further distinguishing computer science from software engineering—because (1) software is immaterial and does not obey physical laws and (2) software is written by people for people. Therefore, many aspects of software engineering are, by definition, impacted by human factors. Empirical studies are needed to identify these human factors impacting software engineering and to study the impact of these factors on software development and software systems.

In this chapter, we first introduce the concepts and principles on which empirical software engineering is based. Then, using these concepts and principles, we describe seminal works that led to the inception and popularisation of empirical software engineering research. We use these seminal works to discuss some idioms,

patterns, and styles in empirical software engineering before discussing some challenges that empirical software engineering must overcome in the (near) future. Finally, we conclude and suggest further readings and future directions. Thus, this chapter complements previous works existing regarding the choice of empirical methods [16] or framework into which empirical software engineering research could be cast, such as the evidence-based paradigm [30].

2 Concepts and Principles

Empirical software engineering is a field that encompasses several research methods and endeavours, including—but not limited to—surveys to collect data on some phenomenon and controlled experiments to measure the correlation between variables. Therefore, empirical studies in software engineering are studies that use any of the "usual" empirical research methods [70]: survey (including systematic literature reviews), case studies, quasi-experiments, and controlled experiments. Conversely, we argue that software engineering research works, which do not include any survey, case study, or experiments, are *not* empirical studies (or do not contain empirical studies). Yet, it is nowadays rare for software engineering research works *not* to include some empirical studies. Indeed, empirical studies are among the top most popular topics in conferences like the ACM/IEEE International Conference on Software Engineering. Therefore, we believe useful at this point in time and for the sake of the completeness of this chapter to recall the justification of and concepts related to empirical software engineering.

2.1 Justification

While it was possible some years ago to publish a research work, for example, proposing a new language mechanism to simplify the use of design patterns [61] without detailed empirical studies of the mechanism, they were followed by empirical studies to assess *concretely* the advantages and limitations of similar mechanisms [68]. As additional example, keywords related to empirical software engineering, such as "case study" or "experiment," appeared in the titles of only two papers presented in the 2nd International Conference on Software Engineering in 1976.[1] In the 37th International Conference on Software Engineering, in 2015, 39 years later, seven papers had keywords related to empirical software engineering in their titles, and a whole track of four research papers was dedicated to human factors in software engineering.

These examples illustrate that software engineering research is taking a clear turn towards empirical research. The reasons for this clear turn are twofold: they

[1] Yet, a whole track of this conference was dedicated to case studies with six short papers and one long paper.

pertain to the nature of software engineering research and to the benefits provided by empirical research. On the one hand, software engineering research strives to follow the scientific method to offer sound and well-founded results, and it thus requires observations and experimentations to create and to assess hypotheses and theories. Empirical research offers the methods needed to perform these observations and experimentations. Observations can usually be readily made in real conditions, hence rooting empirical research into industry practices. Experimentations may be more difficult to carry but allow to compare activities and tasks. On the other hand, empirical research has a long tradition in science, in particular in fields like medicine and/or physics. Therefore, empirical researchers can draw inspiration from other fields and apply known successful methods.

2.2 General Concepts

Empirical software engineering, as any other empirical field, relies on few general concepts defined in the following. We begin by recalling the concept of scientific method and present related concepts, tying them with one another. Therefore, we present concepts by neither alphabetical order nor importance but rather as a narrative.

Scientific Method The scientific method can be traced back to early scientists, including but not limited to Galileo and Al-Jazari, and has been at the core of all natural science research since the seventeenth century. The scientific method typically consists in observations, measurements, and experimentations. Observations help researchers to formulate important questions about a phenomenon under study. From these questions, researchers derive hypotheses that can be tested through experimentations to answer the questions. A scientific hypothesis must be refutable, i.e. it should be possible to prove it to be false (otherwise it cannot be meaningfully tested), and must have for answers universal truths. Once the hypothesis is tested, researchers compile and communicate the results of the experiments in the form of laws or theories, which may be universal truths and which may still require testing. The scientific method is increasingly applied in software engineering to build laws and theories about software development activities.

Universal Truth An observation, a law, or a theory achieves the status of universal truths when it applies universally in a given context, to the best of our knowledge and notwithstanding possible threats to the validity of the experiments that led to it, in particular threats to generalisability. A universal truth is useful for practitioners who can expect to observe it in their contexts and for researchers who can test it to confirm/infirm it in different contexts and/or under different assumptions.

Empirical Method At the heart of the scientific method are empirical methods, which leverage evidences obtained through observations, measurements, or experimentations, to address a scientific problem. Although empirical methods are not the only means to discover (universal) truths, they are often used by

researchers. Indeed, a fundamental principle of the scientific method is that results must be backed by evidences and, in software engineering, such evidences should be based on concrete observations, measurements, or experimentations resulting from qualitative and quantitative research, often based on a multi-method or mixed-method methodology.

Refutability Observations, measurements, or experimentations are used to formulate or infirm/confirm hypotheses. These hypotheses must be refutable. The refutability of a hypothesis is the possibility to prove this hypothesis to be false. Without this possibility, a hypothesis is only a statement of faith without any scientific basis.

Qualitative Research Qualitative research aims to understand the reasons (i.e. "why") and mechanisms (i.e. "how") explaining a phenomenon. A popular method of qualitative research is case-study research, which consists in examining a set of selected samples in details to understand the phenomenon illustrated by the samples. For example, a qualitative study can be conducted to understand why developers prefer one particular static analysis tool over some other existing tools. For such a study, researchers could perform structured, semi-structured, or unstructured interviews with developers about the tools. They could also run focus groups and/or group discussions. They could also analyse data generated during the usage of the tools to understand why developers prefer the tool.

Quantitative Research Quantitative research is a data-driven approach used to gain insights about an observable phenomenon. In quantitative research, data collected from observations are analysed using mathematical/statistical models to derive quantitative relationships between different variables capturing different aspects of the phenomenon under study. For example, if we want to investigate the relation between the structure of a program and its reliability measured in terms of post-release defects, we must define a set of variables capturing different characteristics of the program, such as the complexity of the code, or the cohesion of the code, and then, using a linear regression model, we can quantify the relationship between code complexity values and the number of post-release defects.

Multi-Method Research Methodology A multi-method research methodology combines different research methods to answer some hypotheses. It allows researchers to gain a broader and deeper understanding of their hypotheses and answers. Typically, it combines survey, case studies, and experiments. It can also combine inductive and deductive reasoning as well as grounded theories.

Mixed-Method Research Methodology A mixed-method research methodology is a particular form of multi-method research methodology in which quantitative and qualitative data are collected and used by researchers to provide answers to research hypotheses and to discuss the answers. This methodology is often used in empirical software engineering research because of the lack of theories in software engineering with which to interpret quantitative data and because of the need to discuss qualitatively the impact of the human factor on any experiments in software engineering.

Grounded Theory While it is a good practice to combine quantitative analyses, which help identify related attributes and their quantitative impact on a phenomenon, with qualitative analyses, which help understand the reasons and mechanisms for their impact on the phenomenon, such a methodology also allows researchers to build grounded theories. Grounded theories are created when researchers abstract their observations made from the data collected during quantitative and qualitative research into a theory. This theory should explain the observations and allow researchers to devise new interesting hypotheses and experiments to infirm/confirm these hypotheses and, ultimately, refine the theory.

Activities and Tasks Software engineers can expect to perform varied activities and tasks in their daily work. Activities include but are not limited to development, maintenance, evolution, debugging, and comprehension. Tasks include but are not limited to the usual tasks found in any engineering endeavour: feasibility studies, pre-project analysis, architecture, design, implementation, testing, and deployment. Typically, a task may involve multiple activities. For example, the task of fixing a bug includes the activities of (1) reading the bug description, (2) reproducing the bug through test cases, (3) understanding the execution paths, and (4) modifying the code to fix the bug.

2.3 Empirical Research Methods

When performing empirical research, researchers benefit from many well-defined methods [70], which we present here. These methods use supporting concepts that we define in the next Sect. 2.4.

Survey Surveys are commonly used to gather from software engineers information about their processes, methods, and tools. They can also be used to collect data about software engineers' behaviours, choices, or observations, often in the form of self-reports. To conduct a survey, researchers must select a representative sample of respondents from the entire population or a well-defined cohort of software engineers. Statistical techniques exist to assist researchers in sampling a population. Once they have selected a sample of respondents, researchers should choose a type of survey: either questionnaires or interviews. Then, they should construct the survey, deciding on the types of questions to be included, their wording, their content, their response format, and the placement and sequence of the questions. They must run a pilot survey before reaching out to the all the respondents in sample. Pilot surveys are useful to identify and fix potential inconsistencies in surveys. Online surveys are one of the easiest forms of survey to perform.

Systematic Literature Review Another type of surveys are systematic literature reviews. Systematic literature reviews are important and deserve a paragraph on their own. They are questionnaires that researchers self-administer to collect systematically data about a subset of the literature on a phenomenon. Researchers use

the data to report and analyse the state of the art on the phenomenon. Researchers must first ask a question about a phenomenon, which requires a survey of the literature, and then follow well-defined steps to perform the systematic literature review: identification and retrieval of the literature, selection, quality assessment, data extraction, data synthesis, and reporting the review. Systematic literature reviews are secondary studies based on the literature on a phenomenon. Thus, they indirectly—with one degree of distance—contribute to our understanding of software engineering activities and tasks. They have been popular in recent years in software engineering research and are a testimony of the advances made by software engineering researchers in the past decades.

Case Study Case studies help researchers gain understanding of and from one particular case. A case in software engineering can be a process, a method, a tool, or a system. Researchers use qualitative and quantitative methods to collect data about the case. They can conduct a case study, for example, to understand how software engineers apply a new development method to build a new system. In this example, researchers would propose a new development method and ask some software engineers to apply it to develop a new system—the *case*. They would collect information about the method, its application, and the built system. Then, they could report on the observed advantages and limitations of the new method. Researchers cannot generalise the results of case studies, unless they selected carefully the set of cases to be representative of some larger population.

Experiment Experiments are used by researchers to examined cause–effect relationships between different variables characterising a phenomenon. Experiments allow researchers to verify, refute, or validate hypotheses formulated about the phenomenon. Researchers can conduct two main types of experiments: controlled experiments and quasi-experiments.

Controlled Experiment In medicine, nursing, psychology, and other similar fields of empirical research, acknowledging the tradition established by medicine, researchers often study the effect of a *treatment* on some *subjects* in comparison to another group of subjects not receiving the treatment. This latter group of subjects is called the *controlled group*, against which researchers measure the effect of the treatment. Software engineering researchers also perform controlled experiments, but, differently from other fields of empirical research, they are rarely interested by the effect of a treatment (a novel process, method, or tool) on some subjects but rather by the effect—and possibly the magnitude of the effect—of the treatment on the *performance of the participants*. Consequently, software engineering researchers often control external measures of the participants' performance, like the numbers of introduced bugs or the numbers of failed test cases.

Quasi-Experiment In quasi-experiments, researchers do not or cannot assign randomly the participants in the experiments in the control and experimental groups, as they do in experiments. For example, researchers could perform a (quasi-) experiment to characterise the impact of anti-patterns on the comprehensibility of systems. They should select some systems containing instances of some

anti-patterns. They would create "clean" versions of the systems without these instances of the anti-patterns. They would use these versions of the systems to compare participants' performance during comprehension activities. Researchers should assign the participants to control and experimental groups using an appropriate design model (for example, a 2×3 factorial design). They could also administer a post-mortem questionnaire to participants at the end of the experiments to collect information about potential confounding factors that could affect the results.

Replication Experiment While experiments are conceived by researchers and performed to produce *new* knowledge (observations, results), there is a special form of experiments that is essential to the scientific method: replication experiments. Replication experiments are experiments that reproduce (or quasi-reproduce) previous experiments with the objectives to confirm or infirm the results from previous experiments or to contrast previous results in different contexts. They are essential to the scientific method because, without reproducibility, experiments provide only a collection of unrelated, circumstantial results while their reproductions give them the status of universal truth.

2.4 Empirical Research Methods: Supporting Concepts

We now recall the concepts supporting the empirical research methods presented before. We sort them alphabetically to acknowledge that all these concepts are important to the methods.

Cohort A cohort is a group of people who share some characteristics, for example, being born the same year or taking the same courses, and who participate in empirical studies.

Measurement A measurement is a numerical value that represents a characteristic of an object and that researchers can compare with other values collected on other objects.

Object Surveys, case studies, and experiments often pertain to a set of objects, which the researchers study. Objects can be processes, methods, or tools as well as systems or any related artefacts of interest to the researchers.

Participant A participant is a person who takes part in an experiment. Researchers usually distinguish between participants and subjects. Participants perform some activities related to a phenomenon, and researchers collect data related to these activities and phenomenon for analysis. Researchers are not interested in the participants' unique and/or relative performances. They consider the participants' characteristics only to the extent that these characteristics could impact the data collected in relation to the phenomenon. On the contrary, subjects perform some activities related to a phenomenon, and researchers are interested in the subjects'

performance, typically to understand how subjects' characteristics impact their performance.

Reproducibility The ability to reproduce a research method and its results is fundamental to the scientific method. Without reproducibility, a research method and/or its results are only circumstantial evidences that do not advance directly the state of the art. It is a necessary condition for any universal truth.

Scales A scale is a level of measurement categorising different variables. Scales can be nominal, ordinal, interval, and ratio. Scales limit the statistical analyses that researchers can perform on the collected values.

Transparency Transparency is a property of any scientific endeavour. It implies that researchers be transparent about the conduct of their research, from inception to conclusion. In particular, it requires that researchers disclose conflicts of interests, describe their methods with sufficient details for reproducibility, provide access to their material for scrutiny, and share the collected data with the community for further analyses. It is a necessary condition for any universal truth and also an expectation of the scientific method.

2.5 Empirical Research Techniques

Many different techniques are applicable to carry empirical software engineering research, depending on the chosen method, the hypotheses to be tested, or the context of the research. Surveying all existing techniques is out of the scope of this chapter. However, some techniques are commonly used by researchers in empirical software engineering and have even spawned workshops and conferences. Therefore, these techniques deserve some introduction because they belong to the concepts available to empirical software engineering researchers.

Mining Software Repositories One of the main objects of interest to researchers in software engineering research are the software products resulting from a software process. These products include the documentation, the source code, but also the bug reports and the developers' commits. They are often stored by developers in various databases, including but not limited to version control systems or bug tracking tools. They can be studied by researchers using various techniques, which collectively are called mining techniques. These mining techniques allow researchers to collect and cross-reference data from various software repositories, which are an important source of information for empirical software engineering researchers. Thus, mining software repositories is an essential means to collect observations in empirical software engineering research.

Consequently, they became popular and have dedicated forums, in particular the conference series entitled "Mining Software Repositories". Although they are various and numerous, some techniques became de facto standards in empirical software engineering research, such as the SZZ algorithm used to cross-reference

commits and bug reports [57, 69] or the LHDiff algorithm used to track source code lines across versions [4]. These techniques form a basis on which researchers can build their experiments as well as devise new techniques to analyse different software repositories.

Correlation Tests and Effect Sizes After collecting data from software repositories, researchers are often interested in relating pieces of data with one another or with external data, for example, changes to source code lines and developers' efforts. They can use the many techniques available in statistics to compute various correlations. These techniques depend on the data along difference dimensions: whether the data follow a normal distribution or not (parametric vs. non-parametric tests), whether the data samples are independent or dependent (paired or unpaired tests), and whether there are two or more data samples to compare (paired or generalisation). For example, the paired t-test can be applied on two parametric, dependent data samples. Yet, researchers should be aware that correlation does not mean causation and that there exist many threats to the validity of such correlations.

In addition to testing for correlation, researchers should also report the effect size of any statistically significant correlation. Effect size helps other researchers assess the magnitude of a correlation, independently of the size of the correlated data. They can use different measures of effect size, depending on the scale of the data: Cohen's d for means or Cliff's δ for ordinal data. They thus can provide the impact of a treatment, notwithstanding the size of the sample on which the treatment was applied.

3 Genealogy and Seminal Papers

Empirical software engineering research has become a prominent part of software engineering research. Hundreds, if not thousands, of papers have been published on empirical studies. It is out of the scope of this chapter to systematically review all of these empirical studies, and we refer to the systematic literature review of empirical studies performed by Zendler [73] in 2001 entitled "A Preliminary Software Engineering Theory as Investigated by Published Experiments". In this systematic literature review, the author provides a history of experimental software engineering from 1967 to 2000 and describes and divides the empirical studies into seven categories: experiments on techniques for software analysis, design, implementation, testing, maintenance, quality assurance, and reuse.

Rather than systematically reviewing all empirical studies in software engineering, we describe now some landmark papers, books, and venues related to empirical software engineering research. Choosing landmark works is both subjective and risky. It is subjective because we must rely on our (necessarily) limited knowledge of the large field of empirical software engineering research. It is risky because, in relying on our limited knowledge, we may miss important works or works that would have been chosen by other researchers. The following works must be taken

as reading suggestions and not as the definitive list of landmark works. This list is meant to evolve and be refined by the community on-line.[2]

3.1 Landmark Articles

The landscape of software engineering research can be organised along two dimensions: by time and by methods. Time is a continuous variable that started in 1967 and that continues to this date. Empirical research method is a nominal variable that includes surveys, case studies, and experiments (quasi- and controlled experiments), based on the methods defined in Sect. 2.3. We discuss some landmark works for each dimension although many others exist and are published every year.

3.1.1 Timeline

The software engineering research community recognised early that the software engineering endeavour is intrinsically a human endeavour and, thus, that it should be studied experimentally. Therefore, even before software engineering became a mainstream term [39], researchers carried out empirical studies. The timeline in Fig. 1 displays and relates some of these landmark articles, in particular showing the citation relation among these articles.

According to Zendler [73], Grant and Sackman published in 1967 the first empirical study in software engineering entitled "An Exploratory Investigation of Programmer Performance under On-line and Off-line Conditions". This study compared the performance of two groups of developers working with on-line access to a computer through a terminal or with off-line access, in batch mode. It showed that differences exist between on-line and off-line accesses but that interpersonal differences are paramount in explaining these differences. It generated several discussions in the community and may have been the driver towards more empirical research in software engineering, for example, by Lampson in the same year [35] or by Prechelt, who would write a longer critique of this study 12 years later, in 1999 [41].

Another empirical study published early in the history of software engineering was an article by Knuth in 1971 entitled "An Empirical Study of FORTRAN Programs" [33], in which the author studied a set of FORTRAN programs in an attempt to understand what software developers really *do* in FORTRAN programs. The authors wanted to direct compiler research towards an efficient compilation of the constructs most used by developers. As in many other early studies, the authors defined the goal of the study, the questions to research, and the measures to answer these questions in an ad hoc fashion.

[2] http://www.ptidej.net/research/eselandmarks/.

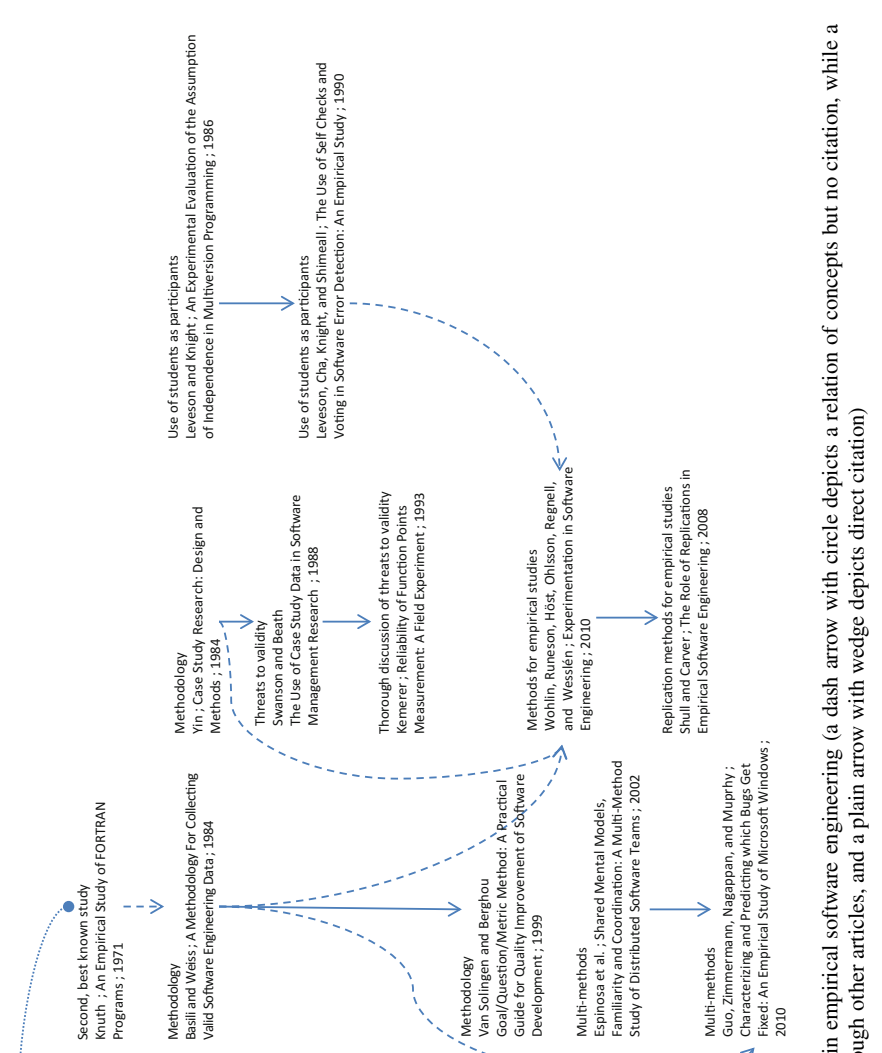

Fig. 1 Timeline of some landmark articles in empirical software engineering (a dash arrow with circle depicts a relation of concepts but no citation, while a dash arrow with wedge depicts citations through other articles, and a plain arrow with wedge depicts direct citation)

A systematic methodology to define empirical studies appeared in an article by Basili and Weiss in 1984 entitled "A Methodology For Collecting Valid Software Engineering Data" [5], which was popularised by Van Solingen and Berghou in 1999 in their book *Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development* [66]. The methodology, named Goal/Question/Metrics (GQM), came at a time of great interest in academia and industry for software measurement programs and the definition of new software metrics. It was defined by Basili and Weiss after they observed that many measurement programs and metrics were put together haphazardly and without clear goals and/or without answering clear questions. It helped managers and developers as well as researchers to define measurement programs based on goals related to products, processes, and resources that can be achieved by answering questions that characterise the objects of measurement, using metrics. This methodology was used to define experiments in many studies on software quality in particular and in empirical software engineering in general.

The GQM methodology helped researchers define empirical studies systematically. However, empirical studies cannot be perfect in their definitions, realisations, and results. Researchers discuss the goals, questions, and metrics of empirical studies as well as their relations. For example, researchers often debate the choice of metrics in studies, which depends both on the goals and questions of the studies but also on the possibility to measure these metrics at an acceptable cost. Thus, researchers must balance scientific rigor and practicality when performing empirical studies. For example, researchers often must draw participants from a convenient sample of readily available students. One of the first papers to ask students to perform software activities and to analyse their outputs was a series of article by Leveson, Knight, and colleagues between 1986 and 1990, for example, entitled "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming" [31] and "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study" [37] in which the authors enrolled graduate students to perform inspection-related tasks. They showed that self-checks are essential and stemmed a line of research works on software inspection, thus contributing to make inspection mainstream.

Finally, researchers must deal with noisy empirical data that lead to uncertain results. For example, researchers frequently obtain pieces of data with outlier values or that do not lend themselves to analyses and yield statistically non-significant results. Consequently, researchers must discuss the threats to the validity of their studies and of their results. One of the first studies explicitly discussing its threats to validity was an article by Swanson and Beath in 1988 entitled "The Use of Case Study Data in Software Management Research" [60], in which the authors discussed with some details the threats to the validity of the study by following the landmark book by Yin [71], first published in 1984.

Threats to the validity of empirical studies and of their results are unavoidable. However, they do not necessarily undermine the interest and importance of empirical studies. For example, despite some threats to its validity, the study by Kemerer in 1993 entitled "Reliability of Function Points Measurement: A Field

Experiment" [27] provided a thorough discussion of its results to convince readers of their importance. It was one of the first studies discussing explicitly their threats. Discussing threats to validity is natural and expected in empirical studies following a multi-method research methodology, in general, and a mixed-method research methodology, in particular. With these methodologies, researchers put in perspective quantitative results using qualitative data, including threats to the validity of both quantitative and qualitative data by contrasting one with the other.

The multi-method and mixed-method research methodologies provide opportunities for researchers to answer their research questions with more comprehensive answers than would allow using only quantitative or qualitative methods. One of the first studies to follow a multi-method research methodology was published in an article by Espinosa et al. in 2002 entitled "Shared Mental Models, Familiarity and Coordination: A Multi-Method Study of Distributed Software Teams" [1]. These methodologies have then been made popular, for example, by Guo, Zimmermann, Nagappan, and Muprhy in 2010 in an article entitled "Characterizing and Predicting which Bugs Get Fixed: An Empirical Study of Microsoft Windows" [19] that reported on a survey of "358 Microsoft employees who were involved in Windows bugs" to understand the importance of human factors on bug-fixing activities. Since then, most empirical studies include qualitative and quantitative results.

Qualitative and quantitative results are often collected using experiments and surveys. Yet, empirical studies can use other methods, discussed in the next section. These methods were thoroughly described and popularised by Wohlin, Runeson, Höst, Ohlsson, Regnell, and Wesslén in 2000 in their book entitled *Experimentation in Software Engineering* [70]. This book was the first to put together, in the context of software engineering research, the different methods of studies available to empirical software engineering researchers and to describe "the scoping, planning, execution, analysis, and result presentation" of empirical studies. However, this book, as other previous works, did not discuss replication experiments. However, replication experiments are important in other fields like social sciences and medicine. They have been introduced in empirical software engineering research by Shull and Carver in 2008 in their article entitled "The Role of Replications in Empirical Software Engineering" [54]. They should become more and more prevalent in the years to come as empirical software engineering research becomes an established science.

3.1.2 Methods

The main methods of empirical research are surveys, case studies, quasi-experiments, and controlled experiments. While surveys are used to understand activities and tasks, case studies bring more information about a reduced set of cases. Quasi- and controlled experiments are then used to test hypotheses possibly derived from surveys and case studies.

The first method of empirical studies are surveys. Surveys are important in software engineering research because software development activities are impacted

by human behaviour [52]. They are, generally, a series of questions asked to some participants to understand what they do or think about some software processes, methods, or tools. They have been introduced by Thayer, Pyster, and Wood in 1980 in their article entitled "The Challenge in Software Engineering Project Management" [62], which reported on a survey of project managers and computer scientists from industry, government, and universities to understand their challenges when managing software engineering projects. They also have been used to assess tool adoption and use by developers, for example, by Burkhard and Jenster in 1989 in their article entitled "Applications of Computer-aided Software Engineering Tools: Survey of Current and Prospective Users" [11]. They were popularised by Zimmerman et al. in 2010 in an article entitled "Characterizing and Predicting which Bugs Get Fixed: An Empirical Study of Microsoft Windows" [19], in which the authors used a mixed-method research methodology, as described in the previous section.

The second method of empirical studies are case studies. Case studies are regularly used by empirical software engineering researchers because researchers often have access to some cases, be them software systems or companies, but in limited numbers either because of practical constraints (impossibility to access industrial software systems protected by industrial secrets) or lack of well-defined populations (impossibility to reach all software developers performing some activities). Therefore, case studies have been used early by researchers in empirical software engineering, for example, by Curtis et al. in 1988 in an article entitled "A Field Study of the Software Design Process for Large Systems" [14]. They were popularised by Murphy et al. in 2006 in an article entitled "Questions Programmers Ask during Software Evolution Tasks" [55]. Runeson et al. published in 2012 a book entitled *Case Study Research in Software Engineering: Guidelines and Examples* [49] that includes a historical account of case studies in software engineering research.

The third method of empirical studies are quasi-experiments. Quasi-experiments are experiments where the participants and/or the objects of study are not randomised. They are not randomised because it would be too expensive to do so or because they do not belong to well-defined, accessible populations. Kampenes et al. presented in 2009 a systematic literature review of quasi-experiments in software engineering entitled "A Systematic Review of Quasi-experiments in Software Engineering" [25]. They performed this review on an article published between 1993 (creation of the ESEM conference series) and 2002 (10 years later). They observed four main methods of quasi-experiments and many threats to the reported experiments, in particular due to the lack of control for any selection bias. They suggested that the community should increase its awareness of " how to design and analyse quasi-experiments in SE to obtain valid inferences".

Finally, the fourth method of empirical studies pertains to controlled experiments. Controlled experiments are studies "in which [...] intervention[s] [are] deliberately introduced to observe [their] effects" [25]. Again, Kampenes et al. presented in 2005 a survey of controlled experiments in software engineering entitled "A Survey of Controlled Experiments in Software Engineering" [56].

The survey was performed with articles published between 1993 and 2002. They reported quantitative data regarding controlled experiments as well as qualitative data about threats to internal and external validity. Storey et al. popularised quasi-experiments and controlled experiments through their works on visualisation and reverse engineering techniques as well as human factors in software development activities, for example, in 1996 with their article entitled "On Designing an Experiment to Evaluate a Reverse Engineering Tool" [59], in which they reported an evaluation of the Rigi reverse engineering tool.

3.2 Landmark Books

The trend towards empirical studies in software engineering culminated with the publication by Wohlin et al. in 2000 of the book entitled *Experimentation in Software Engineering* [70]. This book marks an inflection point in software engineering research because it became very popular and is used to train and to guide researchers in designing sound empirical studies. This book is highly recommended to researchers interested in performing empirical studies to learn about the design and execution of surveys, case studies, and experiments.

This trend led to the publications of many books related to empirical software engineering research, for example, the book written by Runeson et al. in 2012 entitled *Case Study Research in Software Engineering: Guidelines and Examples* [49], which focuses on practical guidelines for empirical research based on case studies, or the one by Bird et al. in 2015 entitled *The Art and Science of Analyzing Software Data* [10], which focuses on the analysis of data collected during case studies. This later book covers a wide range of common techniques, such as co-change analysis, text analysis, topic analysis, and concept analysis, commonly used in software engineering research. It discusses some best practices and provides hints and advices for the effective applications of these techniques in empirical software engineering research.

Another important, seminal book was published by Yin in 2009 and entitled "Case Study Research: Design and Methods" [71]. It provides a detailed and thorough description of case study research, outlining strengths and weaknesses of case studies.

A common problem raised in empirical software engineering as well as other empirical fields is that of reporting empirical results to help other researchers to understand, assess, and (quasi-)replicate the studies. Runeson and Höst in 2009 published a book entitled *Guidelines for Conducting and Reporting Case Study Research in Software Engineering* [48] that provides guidelines for conducting and reporting results of case studies in software engineering. Thus, they contributed to the field by synthesising and presenting recommended practices for case studies.

3.3 Landmark Venues

Some authors reported low numbers and percentages of empirical studies relative to all works published in computer science and/or software engineering research [48]. For examples, Sjøberg et al. [56] found 103 experiments in 5453 works; Ramesh et al. [44] reported less than 2% of experiments with participants and 0.16% of field studies among 628 works; and Prechelt et al. [64] surveyed 400 works and reported that between 40% and 50% did not have experimental validations and, for those that had some, 30% were in computer science and 20% in software engineering. These observations date back to 1995, 2004, and 2005. They did not focus on venues publishing empirical studies, but they are a useful baseline to show how much the field of empirical software engineering research grew over the years.

Another evidence showing the importance taken by empirical studies in software engineering research was the creations and subsequent disappearances of workshops related to empirical studies in software engineering. The workshops International Workshop on Empirical Software Engineering in Practice (IWESEP) (from 2009 to 2016) and Joint International Workshop on Principles of Software Evolution and International ERCIM Workshop on Software Evolution (from 1998 to 2016) and others all focused, at different periods in time, on empirical studies for different domains. They are proofs of the interest of the software engineering research community in empirical studies and also a testimony that empirical studies are nowadays so essential to software engineering research that they are not limited to dedicated workshops anymore.

There are a conference and a journal dedicated to empirical software engineering research: the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) and the Springer journal of Empirical Software Engineering (EMSE). The ESEM conference is the result of the merger between two other conferences, the ACM/IEEE International Symposium on Empirical Software Engineering, which ran from 2002 to 2006, and IEEE International Software Metrics Symposium, which ran from 1993 to 2005. Thus, if considering the oldest of the two conferences, the ESEM conference has a history that dates back to 1993.

The Springer journal of Empirical Software Engineering was created in 1996 by Basili and has run, as of today, 21 volumes. While it started as a specialised journal, it has become in 2015 the journal with the highest impact factor in software engineering research, with an impact factor of 2.161 to be compared to 1.934 for the second highest ranking journal, the IEEE Transactions in Software Engineering.[3] It publishes empirical studies relevant to both researchers and practitioners, which include the collection and analysis of data and studies that can be used to characterise, evaluate, and identify relationships among software processes, methods, and tools.

[3] http://www.guide2research.com/journals/software-programming.

The empirical study by Knuth [33] illustrated that early studies in software engineering were already considering software developers as an important factor in software development activities (the humans in the loop). They were also considering more objectives and quantitative factors, such as the compilation and runtime efficiency of programs. This "trend" will continue to this day: interestingly, among the first four articles published in the journal of Empirical Software Engineering, in its first volume, is an article by Frazier et al. in 1996 entitled "Comparing Ada and FORTRAN Lines of Code: Some Experimental Results" [18], in which the authors compared the number of lines of code required to write functionally equivalent programs in FORTRAN and Ada and observed that, as programs grow bigger, there may be a crossover point after which the size of an Ada program would be smaller than that of the equivalent FORTRAN program.

Since then, the community has followed a clear trend towards empirical studies by publishing hundreds of studies in these two conference and journal but also in many other venues, including but not limited to IEEE Transactions on Software Engineering, Elsevier journal of Information and Software Technology, ACM/IEEE International Conference on Software Engineering, IEEE International Conference on Software Maintenance (and Evolution), and ACM SIGSOFT International Symposium on the Foundations of Software Engineering. More specialised venues also published empirical research, including but not limited to IEEE International Conference on Software Testing and ACM International Symposium on Software Testing and Analysis.

3.4 Other Landmarks

Burnett et al. have been forerunners in the study of end users of programming environments, in particular in the possible impact of users' gender on software development activities. Beckwith and Burnett received the Most Influential Paper Award from 10 Years Ago awarded by the IEEE Symposium on Visual Languages in 2015 for their landmark paper from 2004 entitled "Gender: An Important Factor in End-User Programming Environments?" [6], in which they surveyed the literature regarding the impact of gender in computer gaming, computer science, education, marketing, and psychology and stated hypotheses on the impact of gender in programming environments.

Kitchenham et al. helped the community by introducing, popularising, and providing guidelines for systematic literature reviews (SLRs) in software engineering. SLRs are important in maturing (and established) research fields to assert, at a given point in time, the knowledge gathered by the community on a topic. The article published by Kitchenham et al. in 2002 entitled "Preliminary Guidelines for Empirical Research in Software Engineering" [29] was the starting point of a long line of SLRs that shaped software engineering research. For example, Zhang and Budgen applied in 2012 the guidelines by Kitchenham et al. to assess "What [...] We Know about the Effectiveness of Software Design Patterns?" [74].

Endres and Rombach in 2003 summarised and synthesised years of empirical software engineering research into observations, laws, and theories in a handbook entitled *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories* [17]. These observations, laws, and theories pertained to all software development activities, including but not limited to activities related to requirements, composition, validation, and verification as well as release and evolution. They provided a comprehensive overview of the state-of-the-art research and practice at the time as well as conjectures and hypotheses for future research.

Arcuri and Briand in 2011 provided a representative snapshot of the use of randomised algorithms in software engineering in a paper entitled "A practical guide for using statistical tests to assess randomized algorithms in software engineering" [3]. This representative snapshot shows that randomised algorithms are used in a significant number of empirical research works but that some do not account for the randomness of the algorithms adequately, using appropriate statistical tests. Their paper provides a practical guide for choosing and using appropriate statistical tests.

Another landmark is the article by Zeller et al. in 2011 entitled "Failure is a Four-letter Word: A Parody in Empirical Research" [72], in which the authors discuss, with some humour, potential pitfalls during the analysis of empirical studies data. This landmark highlights real problems in the design and threats to the validity of empirical studies and provides food for thoughts when conducting empirical research.

Many other "meta-papers" exist about empirical software engineering research, regarding sample size [43], bias [9], misclassification [2], and so on. These papers describe potential problems with empirical research in software engineering and offer suggestions and/or solutions to overcome these problems. They also warn the community about the current state of the art on various topics in which empirical methods may have been misused and/or applied on unfit data. They serve as a reminder that the community can always do better and more to obtain more sound observations and experimentations.

4 Challenges

Since Knuth's first empirical study [33], the field of empirical software engineering research has considerably matured. Yet, it faces continuing challenges regarding the size of the studies, the recruitment of students or professional developers as participants, theories (and lack thereof) in software engineering, publication of negative results, and data sharing.

4.1 Size of the Studies

Two measures of the size of empirical studies are the numbers of participants who took part in the studies and the numbers of systems on which the studies were conducted. These measures allow comparing the extent to which the studies could generalise, irrespective of their other strengths or weaknesses. It is expected that the greater the numbers of participants and systems, the more statistically sound are the results and, thus, the more generalisable would be these results.

For example, the first study by Knuth [33] included 440 programs *on 250,000 cards*, which translated into 78,435 assignments, 27,967 if statements, etc., i.e. a quite large system. Yet, other studies included only two programs, for example, the most influential paper of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16) by Kapser and Godfrey in 2008 entitled "'Cloning Considered Harmful' Considered Harmful: Patterns of Cloning in Software" [26]. Other studies also involved few participants, for example, the study by Lake and Cook in 1992 entitled "A Software Complexity Metric for C++" was performed with only two groups of five and six participants [34]. Yet, other studies were performed with more than 200 participants, such as the study by Ng et al. in 2007 entitled "Do Maintainers Utilize Deployed Design Patterns Effectively?" [40].

Although it should not be a mindless race towards larger numbers of systems and/or participants at the expenses of interesting research questions and sound empirical design, empirical studies should strive to have as many systems and participants as required to support their results. Other fields of research in which people play an essential role, such as medicine, nursing, and psychology, have a long established tradition of recruiting statistically representative sets of participants. They also have well-defined cohorts to allow long-term studies and to help with replications. Similarly, the empirical software engineering research community should discuss the creation of such cohorts and the representativeness of the participants in empirical studies. It should also consider outsourcing the creation of such cohorts as in other fields.

4.2 Recruiting Students

Software engineering is intrinsically a human endeavour. Although it is based on sound mathematical and engineering principles, software developers play an important role during the development of software systems because, essentially, no two systems are identical. Therefore, they must use as much their creativity as their expertise and experience [58] when performing their software development activities. This creativity could prevent the generalisability of the results of empirical studies if it was not taken into account during the design of the studies.

While some early empirical studies used only few participants, for example, the study by Lake and Cook [34] involved 11 participants, other more recent studies included hundreds of participants. To the best of our knowledge, the largest experimental study to date in software engineering is that by Ng et al. in 2007 [40], which included 215 participants. This study considered the impact of deployed design patterns on maintenance activities in terms of the activities performed by the participants on the classes related to design patterns. However, this study consisted of students "who were enrolled in an undergraduate-level Java programming course offered by the Hong Kong University of Science and Technology". Hence, while large in terms of number of participants, this study nonetheless has weaknesses with respect to its generalisability because it involved students from one and only one undergraduate class in one and only one university.

There is a tension in empirical software engineering research between recruiting students vs. professional developers. The lack of accessibility to large pools of professional developers forces empirical software engineering researchers to rely on students to draw conclusions, which they hope to be valid for professional developers. For example, Höst et al. in 2000 conducted an experiment entitled "Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-time Impact Assessment" [22], in which they involved students and professional developers to assess the impact of ten factors on the lead time of software projects. They considered as factors the developers' competence, the product complexity, the stability of the requirements, time pressure, priority, and information flow. They compared the students' and professional developers' performance and found only minor differences between them. They also reported no significant differences between the students' and professional developers' correctness. This study, although not generalisable to all empirical studies, strengthens the advocates of the inclusion of students in empirical studies.

Yet, the debate about the advantages and limitations of recruiting students to participate in empirical studies is rather moot for several reasons. First, empirical software engineering researchers may only have access to students to carry their studies. They often have difficultly to convince professional developers in participating in their studies because professional developers have busy schedules and may not be interested in participating, essentially for free, to studies. Second, they usually have access to students, in particular last-year students, who will have worked as interns or freelancers in software companies and who will work immediately after graduation. Hence, they can consider students as early-career professional developers. Third, they have no access to theories that explain the internal factors (intrinsic to developers) and external factors (due to the developers' processes, methods, tools) that impact software development activities. Consequently, they cannot generalise their results obtained from any number of professional developers to all professional developers.

Several authors commented on recruiting students as participants of empirical studies (or professional in empirical studies with students [23]), and most agreed that recruiting students may not be ideal but still represents an important source of participants to carry out studies that advance the state of the art and state of the

practice in software engineering [12, 22, 50, 63]. In particular, Tichy wrote "we live in a world where we have to make progress under less than ideal circumstances, not the least of which are lack of time and lack of willing [participants]" [63]. Therefore, it is acceptable to recruit students as participants to empirical studies, notwithstanding the care taken to design the studies and the threats to their validity [12]. Moreover, students, in particular last-year student and graduate students, are "next-year" developers, trained on up-to-date processes, methods, and tools. Therefore, they are representative of junior industry developers. Finally, choosing an appropriate research method and carefully considering threats to the validity of its results can increase the acceptability of students as participants.

Thus, we claim that **students represent an important population of participants** that the community can leverage in empirical software engineering research. However, we also emphasise that replication experiments should ultimately be done with professional developers. A subpopulation of professional developers that is accessible at a low cost are freelancers. Freelancers can be recruited through several on-line platforms, such as the Mechanical Turk[4] or Freelancers for Hire.[5] Freelancers may increase the generalisability of empirical studies, but ethical consideration should be carefully weighted else they threaten the validity of the results: paid freelancers may be solely motivated by financial gain and could resort to practices that undermine conclusion validity, such as plagiarism.

4.3 Recruiting Professional Developers

Although we claimed that recruiting students is acceptable, there also have been more and more empirical studies performed in companies and involving professional developers, for example, the line of empirical studies carried out by Zimmermann et al. at Microsoft, from 2007 [75] to 2016 [15]. Yet, as with students, recruiting professional developers has both advantages and limitations.

An obvious first advantage of having professional developers perform empirical studies is that these software developers most likely have different career paths and, thus, represent better the "real" diversity of professional developers. Thus, they help gathering evidence that is more *generalisable* to other companies and, ultimately, more useful to the software industry. A less obvious advantage is that these software developers are more likely to be dedicated to the empirical studies and, thus, will help gathering evidence that is more *realistic* to the software industry. Finally, an even less-known advantage is that these software developers will likely perform the activities required by the empirical studies in less time than would students, because they have pressure to complete activities "without values" sooner.

[4] https://www.mturk.com/mturk/welcome.

[5] https://www.freelancer.ca/.

However, recruiting professional developers has also some limitations. The first obvious limitation also concerns generalisability. Although professional developers have experience and expertise that can be mapped to those of other developers in other companies, they likely use processes, methods, and tools that are unique to their companies. They also likely work on software projects that are unique to their companies. Moreover, they form a convenient sample that, in addition to generalisability, also lacks reproducibility and transparency. Reproducibility and transparency are important properties of empirical studies because they allow (1) other researchers to carry out contrasting empirical studies and (2) other researchers to analyse and reuse the data. Without reproducibility and transparency, empirical studies cannot be refuted by the community and belong to the field of faith rather than that of science. For example, the latest paper by Devanbu et al. in 2016 entitled "Belief & Evidence in Empirical Software Engineering" [15] cannot be replicated because it involved professional developers inaccessible to other researchers, and it is not transparent because its data is not available to other researchers.

Other less obvious limitations of recruiting professional developers concern the possibility of hidden incentives to take part in empirical studies. Hidden incentives may include perceived benefit for one's career or other benefits, like time off a demanding work. Therefore, researchers must clearly set, enforce, and report the conditions in which the software developers were recruited to allow other researchers to assess the potential impact of the hidden incentives on the results of the studies. For example, a developer may participate in a study because of a perceived benefit for her career with respect to her boss, showing that she is willing, dynamic, and open-minded. On the contrary, a developer may refuse participating in a study by fear that her boss perceives her participation as "slacking". Recruiting students may be a viable alternative to remove these limitations because students, in general, will have no incentive to participate in a study but their own, selfless curiosity.

4.4 Theories in ESE

Software engineering research is intrinsically about software developers and, hence, about human factors. Consequently, there exist few theories to analyse, explain, and predict the results of empirical studies pertaining to the processes, methods, and tools used by software developers in their software development activities. This lack of theories makes the reliance upon empirical studies even more important in software engineering research. However, it also impedes the definition and analysis of the results of empirical studies because, without theories, researchers cannot prove that the results of their studies are actually due to the characteristics of the objects of the studies rather than due to other hidden factors. They cannot demonstrate causation in addition to correlation. Moreover, they cannot generalise the results of their studies to other contexts, because their results could be due to one particular context.

The lack of theories in software engineering research can be overcome by building *grounded theories* from the data collected during empirical studies. The process of building grounded theories is opposite to that of positivist research. When following the process of positivist research, researchers choose a theory and design, collect, analyse, and explain empirical data within this theory. When following the process of building grounded theories, researchers start to design, collect, and analyse empirical data to observe repeating patterns or other characteristics in the data that could explain the objects under study. Using these patterns and characteristics, they build grounded theories that explain the data, and they frame it in the context of the theories. Then, they refine their theories through more empirical studies either to confirm further their theories or, in opposite, to infirm them and, eventually, to propose more explanatory theories.

The process of building grounded theories raises some challenges in software engineering research. The first challenge is that grounded theories require many replication experiments to confirm the theories, proving or disproving that they can explain some objects under study. However, replication experiments are difficult, for many reasons among which the lack of reproducibility and transparency, including but not limited to the lack of access to the materials used by previous researchers and the difficulty to convince reviewers that replication experiments are worth publishing to confirm that some grounded theories hold in different contexts.

Software engineering research would make great progress if the community recognises that, without replication experiments, there is little hope to build solid grounded theories on which to base future empirical studies. The recognition of the community requires that (1) reviewers and readers acknowledge the importance and, often, the difficulty to reproduce empirical studies and (2) researchers provide all the necessary details and material required for other researchers to reproduce empirical studies. The community should strive to propose a template to report empirical studies, including the material used during the studies, so that other researchers could more easily reproduce previous empirical studies.

The second challenge is that of publishing negative results either of original empirical studies to warn the community of empirical studies that may not bear any fruits or of replication experiments to warn the community that some previous empirical studies may not be built on sound grounded theories. We discuss in details the challenges of publishing negative results in the following section.

4.5 Publication of Negative Results

A colleague, who shall remain anonymous, once made essentially the following statement at a conference:

> By the time I perform an empirical study, I could have written three other, novel visualisation techniques.

Although we can only agree on the time and effort needed to perform empirical studies, we cannot disagree more about (1) the seemingly antinomy between doing research and performing empirical studies and (2) the seemingly uselessness of empirical studies. We address the two disagreements in the following.

4.5.1 Antinomy Between Doing Research and Empirical Studies

The time and effort needed to perform empirical studies are valid observations. The book by Wohlin et al. [70] illustrates and confirms these observations by enumerating explicitly the many different steps that researchers must follow to perform empirical studies. These steps require time and effort in their setup, running, and reporting but are all necessary when doing software engineering research, because research is a "careful study that is done to find and report new knowledge about something"[6] [38].

However, software engineering research is not *only* about building new knowledge in software engineering, i.e. "writ[ing] [. . .] novel [. . .] techniques". Research is also:

> [the] studious inquiry or examination; *especially*: investigation or experimentation aimed at the discovery and interpretation of facts, revision of accepted theories or laws in the light of new facts, or practical application of such new or revised theories or laws.

Therefore, "doing research" and "performing empirical studies" are absolutely not antinomic but rather two steps in the scientific method. First, researchers must devise novel techniques so that, second, they can validate them through empirical studies. The empirical studies generate new facts that researchers can use to refine and/or devise novel techniques.

4.5.2 Seemingly Uselessness of Empirical Studies

It may seem that empirical studies are not worth the effort because, no matter the carefulness with which researchers perform them (and notwithstanding threats to their validity), they may result in "negative" results, i.e. showing no effect between two processes, methods, or tools and/or two sets of participants. Yet, two arguments support the usefulness of negative empirical studies.

First, notwithstanding the results of the studies, empirical studies are part of the scientific method and, as such, are an essential means to advance software engineering research. Novel techniques alone cannot advance software engineering because they cannot per se (dis)prove their (dis)advantages. Therefore, they should be carefully, empirically validated to provide evidence of their (dis)advantages rather than to rely on hearsay and faith for support.

[6] http://www.merriam-webster.com/dictionary/research.

Second, notwithstanding the threats to their validity, negative results are important to drive research towards promising empirical studies and avoid that independent researchers reinvent the "square wheel". Without negative results, researchers are condemned to repeat the mistakes of previous unknown researchers and/or perform seemingly promising but actually fruitless empirical studies.

4.5.3 What Is and What Should Be

The previous paragraphs illustrated the state of the practice in empirical software engineering research and the importance of negative results. Negative results must become results normally reported by software engineering researchers. They must be considered within the philosophy of science put forward by Popper and others in which refutability is paramount: given that software engineering research relates to developers, it is impossible to verify that a novel technique works for all developers in all contexts *but*; thanks to negative results, it is possible to show that *it does not work for some developers*. Thus, no matter the time and effort needed to perform empirical studies, such studies are invaluable and intrinsically necessary to advance our knowledge in software engineering.

Moreover, the community must acknowledge that performing empirical studies is costly and that there is a high "fixed cost" in developing the material necessary to perform empirical studies. This fixed cost is important for researchers but should not prevent researchers from pursuing worthwhile empirical studies, even with the risk of negative results. In other fields, like nursing, researchers publish experimental protocols *before* they carry them. Thus, researchers are not reluctant to develop interesting material, even if they do not carry out the empirical studies for some other reasons independent of the material, like the difficulty in recruiting participants.

4.6 Data Sharing

A frequent challenge with empirical studies, be them survey or controlled experiments, is that of sharing all the data generated during the studies. The data includes but is not limited to the documents provided to the participants to recruit them and to educate them on the objects of the study as well as the software systems used to perform the studies, the data collected during the studies, and the post hoc questionnaires. It also includes immaterial information, such as the processes followed to welcome, set up, and thank the participants as well as the questions answered during the studies. It is necessary for other researchers to replicate the studies.

Some subcommunities of software engineering research have been sharing data for many years, in particular the subcommunity interested in predictive models and data analytics in software engineering, which is federated by the Promise conference

series.[7] This subcommunity actively seeks to share its research data and make its empirical studies reproducible. Other subcommunities should likewise try to share their data. However, the community does not possess and curate one central repository to collect data, which leads to many separate endeavours. For example, the authors have strived to publish all the data associated to their empirical studies and have, thus, built a repository[8] of more than 50 sets of empirical data, including questionnaires, software systems, and collected data.

However, the community should develop and enforce the use of templates and of good practices in the sharing of empirical data. On the one hand, the community must acknowledge that empirical data is an important asset for researchers and that some data may be proprietary or nominative. On the other hand, the community should put mechanisms in place to ease the sharing of all empirical data, including nominative data. Consequently, we make four claims to ease replication experiments. First, empirical data should not be the asset of a particular set of researchers, but, because collecting this data is not free, researchers must be acknowledged and celebrated when sharing their data. Second, the data should be shared collectively and curated frequently by professional librarians. Third, nominative data could be made available upon requests only and with nondisclosure agreement. Finally, the community should build *cohorts* of participants and of objects for empirical studies.

4.7 Comparisons of Software Artefacts

While many experiments in software engineering require participants and involve within- or between-subject designs, some experiments require different versions of some software artefacts to compare these artefacts with one another in their forms and contents. For example, some software engineering researchers studied the differences between various notations for design patterns in class diagrams [13] or between textual and graphical representations of requirements [53]. Other researchers studied different implementations of the same programs in different programming languages to identify and compare their respective merits [42].

One of the precursor works comparing different software artefacts is the work by Knight and Leveson in 1986 [31] entitled "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming", which reported on the design of an experiment for multiversion programming in which 27 programs implementing the same specification for a "launch interceptor" were developed by as many students enrolled at the University of Virginia and the University of California at Irvine. The work compared each version in terms of their numbers of faults, given a set of test cases.

[7] promisedata.org/.

[8] http://www.ptidej.net/downloads/replications/.

Such comparisons of software artefacts provide invaluable observations needed to understand the advantages and limitations of different versions of the same artefacts in forms and contents. However, they are costly to perform because they require obtaining these different versions and because different versions are usually not produced by developers in the normal course of their work. Thus, they require dedicated work that may limit the generalisability of the conclusions drawn from these comparisons for two main reasons. First, researchers often must ask students to develop different versions of the same software artefacts. Students risk producing under-/over-performing versions when compared to professional developers; see Sects. 4.2 and 4.3. Second, researchers often must choose one dimension of variations, which again limits the generalisability of the comparisons. For example, the Web site "99 Bottles of Beer"[9] provides more than 1500 implementations of a same, unique specification. While it allows comparing programming languages, it does not allow generalising the comparisons because one and only one small specification is implemented.

Consequently, the community should strive to implement and share versions of software artefacts to reduce the entry costs while promoting reproducibility and transparency.

5 Future Directions

In addition to following sound guidelines, empirical studies changed from ad hoc studies with low numbers of participants and/or objects to well-design, large scale studies. Yet, empirical studies are still submitted to conference and journals with unsound setup, running, and/or reporting. Such empirical study issues are due to a lack of concrete guidelines for researchers, in the form of patterns and anti-patterns from which researchers can learn, and that researchers can apply or avoid.

Consequently, we suggest that the community defines patterns and anti-patterns of empirical software engineering research to guide all facets of empirical studies, from their setup to their reporting. We exemplify such patterns and anti-patterns at different levels of abstraction. We provide one idiom, one pattern, and one style of empirical software engineering research and hope that the community complements these in future work. Such idioms, patterns, and styles could be used by researchers in two ways: (1) as "step-by-step" guides to carry empirical studies and (2) as "templates" to follow so that different research works are more easily and readily comparable. Thus, they could help both young researchers performing their first empirical studies as well as senior researchers to reduce their cognitive load when reporting their empirical studies.

[9] http://99-bottles-of-beer.net/.

5.1 Idioms for Empirical Studies

Pattern Name "Tool Comparison"

Problem Determine if a newly developed tool outperforms existing tools from the literature.

Solution

- Survey the literature to identify tools that address the same problem as the newly developed tool.
- Survey the literature to identify benchmarks used to compare similar tools. If no benchmark is available, examine the context in which evaluations of previous tools have been performed.
- Download the replication packages that were published with the similar tools. If no replication package is available, but enough details are available about the similar tools, implement these tools.
- Identify relevant metrics used to compare the tools.
- Using a benchmark and some metrics, compare the performance of the newly developed tool against that of the similar tools. When doing the comparison, use proper statistics and consider effect sizes.
- Perform usability studies, interviews, and surveys with the tools used to assess the practical effectiveness and limitations of the proposed tool. When conducting these usability and qualitative studies, select a representative sample of users.

Discussion The evaluation of the newly developed tool is very important in software engineering research. It must ensure fairness and correctness through reproducibility and transparency. Researchers and practitioners rely on the results of these evaluations to identify best-in-class tools.

Example Many evaluations reported in the literature failed to follow this idiom. They do not allow researchers and practitioners to identify most appropriate tools. However, some subcommunities have started to establish common benchmarks to allow researchers to follow this idiom during their evaluations. For example, Bellon's benchmark [7] provides a reference dataset for the evaluations of clone-detection tools.

5.2 Patterns for Empirical Studies

Pattern Name "Prima Facie Evidence"

Problem Determine if evidence exists to support a given hypothesis. An example hypothesis is that a newly developed tool outperforms similar tools from the literature.

Solution

- Survey the literature to identify tools that are similar to the ones on which the hypothesis pertain.
- Identify metrics that can be used to test the hypothesis.
- Test the hypothesis on the identified tools. When testing the hypothesis, use proper statistics and consider effect sizes.
- Survey users of the tools under study to explain qualitatively the results. When conducting this qualitative analysis, select a representative sample of users.

Discussion Prima facie evidence constitutes the initial evaluation of a hypothesis. Researchers should perform subsequent evaluations and replication experiments before building a theory. They must therefore carefully report the context in which they obtained the prima facie evidence.

Example Many empirical studies follow this pattern. For example, Ricca et al. in 2008 [46] provided prima facie evidence that the use of Fit tables (Framework for Integrated Test by Cunningham) during software maintenance can improve the correctness of the code with a negligible overhead on the time required to complete the maintenance activities. Yet, this prima facie evidence was not enough to devise a theory and, consequently, these authors conducted five additional experiments to further examine the phenomenon and formulate guidelines for the use of Fit tables [45].

Pattern Name "Idea Inspired by Experience"

Problem Determine if an idea derived from frequent observations of a phenomenon can be generalised.

Solution

- Select a representative sample from the population of objects in which the phenomenon manifests itself.
- Formulate null hypotheses from the observations and select appropriate statistical tests to refute or accept the null hypotheses.
- Determine the magnitude of the differences in values through effect-size analysis. Additionally, consider visualising the results, for example, using box plots.

Discussion Researcher must use a parametric/non-parametric test for parametric/non-parametric data. They must perform a multiple-comparison correction, for example, by applying the Bonferroni correction method, when testing multiple hypotheses on the same data.

Sample selection is key to assess the generalisability of an idea based on observations. In general, researchers may not know the characteristics of all the objects forming the population of interest. They may not be able to derive a representative sample of objects. Therefore, they cannot generalise their empirical results beyond the context in which they were obtained.

Thus, researchers must report as much information as possible about the contexts in which they performed their empirical studies to allow other researchers to replicate these studies and increase their generalisability.

Example Many empirical studies follow this pattern. For example, Khomh et al. [28] investigated the change proneness of code smells in Java systems following this pattern. They reported enough details about the context of their study to allow Romano et al. [47] to replicate their study a few years later.

5.3 Styles of Empirical Studies

Researchers can conduct different styles of empirical studies. They often have the choice between quantitative, qualitative, and mixed-method styles.

Pattern Name "Mixed-Method Style"

Problem Provide empirical evidence to support a conjecture. Mixed-method style can be applied to explore a phenomenon, to explain and interpret the findings about the phenomenon, and to confirm, cross-validate, or corroborate findings within a study.

Solution

- Collect quantitative and qualitative data from relevant sources and formulate hypotheses. The sequence of data collection is important and depends on the goal of the study. If the goal is to explore a phenomenon, collect and analyse qualitative data first, before quantitative data. If the goal is to quantify a phenomenon, collect and analyse quantitative data first, and explain and contrast this data using qualitative data.
- Examine the obtained data rigorously to identify hidden patterns.
- Select appropriate statistical tests to refute or accept the hypotheses.
- Interpret the obtained results and adjust the hypotheses accordingly.

Discussion A key advantage of a mixed-method methodology is that it overcomes the weakness of using one method with the strengths of another. For example, interviews and surveys with users are often performed to explain the results of quantitative studies, to prune out irrelevant results, and to identify the root cause of relevant results. However, this methodology does not help to resolve discrepancies between quantitative and qualitative data.

Example Many empirical studies follow the mixed-method style. For example, Bird and Zimmermann [8] followed this style during their assessment of the value of branches in software development.

6 Conclusion

Software engineering research has more than five decades of history. It has grown naturally from research in computer science to become a unique discipline concerned with software development. It has recognised for more than two decades the important role played by the *people* performing software development activities and has, consequently, turned towards empirical studies to better understand software development activities and the role played by software developers.

Empirical software engineering hence became a major focus of software engineering research. Since the first empirical studies [33], thousands of empirical studies have been published in software engineering, showing the importance of software developers in software engineering but also highlighting the lack of sound theories on which to design, analyse, and predict the impact of processes, methods, and tools.

Empirical software engineering will continue to play an important role in software engineering research and, therefore, must be taught to graduate students who want to pursue research works in software engineering. It should be also taught to undergraduate students to provide them with the scientific method and related concepts, useful in their activities and tasks, such as debugging. To ease teaching empirical software engineering, more patterns and anti-patterns should be devised and catalogued by the community. As such, the Workshop on Data Analysis Patterns in Software Engineering, which runs from 2013 to 2014, was important and should be pursued by the community.

Acknowledgements The authors received the amazing support, suggestions, and corrections from many colleagues and students, including but not limited to Mona Abidi, Giuliano Antoniol, Sungdeok Cha, Massimiliano Di Penta, Manel Grichi, Kyo Kang, Rubén Saborido-Infantes, and Audrey W.J. Wong. Obviously, any errors remaining in this chapter are solely due to the authors.

References

1. Alberto Espinosa, J., Kraut, R.E.: Kogod school of Business, and theme organization. Shared mental models, familiarity and coordination: a multi-method study of distributed software teams. In: International Conference Information Systems, pp. 425–433 (2002)
2. Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., Guéhéneuc, Y.-G.: Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, CASCON '08, pp. 23:304–23:318. ACM, New York (2008)
3. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 1–10 (2011)
4. Asaduzzaman, M., Roy, C.K., Schneider, K.A., Penta, M.D.: Lhdiff: a language-independent hybrid approach for tracking source code lines. In: 2013 29th IEEE International Conference on Software Maintenance (ICSM), pp. 230–239 (2013)

5. Basili, V.R., Weiss, D.M.: A methodology for collecting valid software engineering data. IEEE Trans. Softw. Eng. **SE-10**(6), 728–738 (1984)
6. Beckwith, L., Burnett, M.: Gender: an important factor in end-user programming environments? In: 2004 IEEE Symposium on Visual Languages and Human Centric Computing, pp. 107–114 (2004)
7. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Trans. Softw. Eng. **33**(9), 577–591 (2007)
8. Bird, C., Zimmermann, T.: Assessing the value of branches with what-if analysis. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 45:1–45:11. ACM, New York (2012)
9. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced?: Bias in bug-fix datasets. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '09, pp. 121–130. ACM, New York (2009)
10. Bird, C., Menzies, T., Zimmermann, T.: The Art and Science of Analyzing Software Data. Elsevier Science, Amsterdam (2015)
11. Burkhard, D.L., Jenster, P.V.: Applications of computer-aided software engineering tools: survey of current and prospective users. SIGMIS Database **20**(3), 28–37 (1989)
12. Carver, J., Jaccheri, L., Morasca, S., Shull, F.: Issues in using students in empirical studies in software engineering education. In: Ninth International Software Metrics Symposium, 2003. Proceedings, pp. 239–249 (2003)
13. Cepeda Porras, G., Guéhéneuc, Y.-G.: An empirical study on the efficiency of different design pattern representations in UML class diagrams. Empir. Softw. Eng. **15**(5), 493–522 (2010)
14. Curtis, B., Krasner, H., Iscoe, N.: A field study of the software design process for large systems. Commun. ACM **31**(11), 1268–1287 (1988)
15. Devanbu, P., Zimmermann, T., Bird, C.: Belief & evidence in empirical software engineering. In: Proceedings of the 38th International Conference on Software Engineering (2016)
16. Easterbrook, S., Singer, J., Storey, M.-A., Damian, D.: Selecting empirical methods for software engineering research. In: Guide to Advanced Empirical Software Engineering, pp. 285–311. Springer, London (2008)
17. Endres, A., Rombach, H.D.: A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories. Fraunhofer IESE Series on Software Engineering. Pearson-/Addison Wesley, Boston (2003)
18. Frazier, T.P., Bailey, J.W., Corso, M.L.: Comparing ada and fortran lines of code: some experimental results. Empir. Softw. Eng. **1**(1), 45–59 (1996)
19. Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B.: Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Proceedings of the 32th International Conference on Software Engineering (2010)
20. Hanenberg, S.: Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In: ECOOP 2010 – Object-Oriented Programming: 24th European Conference, Maribor, June 21–25, 2010. Proceedings, pp. 300–303. Springer, Berlin (2010)
21. Herbsleb, J.D., Mockus, A.: An empirical study of speed and communication in globally distributed software development. IEEE Trans. Softw. Eng. **29**(6), 481–494 (2003)
22. Höst, M., Regnell, B., Wohlin, C.: Using students as subjects – a comparative study of students and professionals in lead-time impact assessment. Empir. Softw. Eng. **5**(3), 201–214 (2000)
23. Jaccheri, L., Morasca, S.: Involving industry professionals in empirical studies with students. In: Proceedings of the 2006 International Conference on Empirical Software Engineering Issues: Critical Assessment and Future Directions, pp. 152–152. Springer, Berlin (2007)
24. Jacobson, I., Bylund, S. (ed.): The Road to the Unified Software Development Process. Cambridge University Press, New York (2000)
25. Kampenes, V.B., Dybå, T., Hannay, J.E., Sjøberg, D.I.K.: A systematic review of quasi-experiments in software engineering. Inf. Softw. Technol. **51**(1), 71–82 (2009)

26. Kapser, C.J., Godfrey, M.W.: Cloning considered harmful considered harmful: patterns of cloning in software. Empir. Softw. Eng. **13**(6), 645–692 (2008)
27. Kemerer, C.F.: Reliability of function points measurement: a field experiment. Commun. ACM **36**(2), 85–97 (1993)
28. Khomh, F., Di Penta, M., Gueheneuc, Y.G.: An exploratory study of the impact of code smells on software change-proneness. In: 16th Working Conference on Reverse Engineering, 2009. WCRE '09, pp. 75–84 (2009)
29. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. **28**(8), 721–734 (2002)
30. Kitchenham, B.A., Dyba, T., Jorgensen, M.: Evidence-based software engineering. In: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, pp. 273–281. IEEE Computer Society, Washington (2004)
31. Knight, J.C., Leveson, N.G.: An experimental evaluation of the assumption of independence in multiversion programming. IEEE Trans. Softw. Eng. **SE-12**(1), 96–109 (1986)
32. Knuth, D.E.: The Art of Computer Programming: Fundamental Algorithms. Addison-Wesley Publishing Company, Reading (1969)
33. Knuth, D.E.: An empirical study of fortran programs. Softw.: Pract. Exp. **1**(2), 105–133 (1971)
34. Lake, A., Cook, C.R.: A software complexity metric for C++. Technical report, Oregon State University, Corvallis (1992)
35. Lampson, B.W.: A critique of an exploratory investigation of programmer performance under on-line and off-line conditions. IEEE Trans. Hum. Factors Electron. **HFE-8**(1), 48–51 (1967)
36. Lencevicius, R.: Advanced Debugging Methods. The Springer International Series in Engineering and Computer Science. Springer, Berlin (2012)
37. Leveson, N.G., Cha, S.S., Knight, J.C., Shimeall, T.J.: The use of self checks and voting in software error detection: an empirical study. IEEE Trans. Softw. Eng. **16**(4), 432–443 (1990)
38. Merriam-Webster: Merriam-Webster online dictionary (2003)
39. Naur, P., Randell, B. (ed.): Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. Brussels, Scientific Affairs Division, NATO, Garmisch (1969)
40. Ng, T.H., Cheung, S.C., Chan, W.K., Yu, Y.T.: Do maintainers utilize deployed design patterns effectively? In: 29th International Conference on Software Engineering, 2007. ICSE 2007, pp. 168–177 (2007)
41. Prechelt, L.: The 28:1 Grant-Sackman Legend is Misleading, Or: How Large is Interpersonal Variation Really. Interner Bericht. University Fakultät für Informatik, Bibliothek (1999)
42. Prechelt, L.: An empirical comparison of seven programming languages. Computer **33**(10), 23–29 (2000)
43. Rahman, F., Posnett, D., Herraiz, I., Devanbu, P.: Sample size vs. bias in defect prediction. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 147–157. ACM, New York (2013)
44. Ramesh, V., Glass, R.L., Vessey, I.: Research in computer science: an empirical study. J. Syst. Softw. **70**(1–2), 165–176 (2004)
45. Ricca, F., Di Penta, M., Torchiano, M.: Guidelines on the use of fit tables in software maintenance tasks: lessons learned from 8 experiments. In: IEEE International Conference on Software Maintenance, 2008. ICSM 2008, pp. 317–326 (2008)
46. Ricca, F., Di Penta, M., Torchiano, M., Tonella, P., Ceccato, M., Visaggio, A.: Are fit tables really talking? A series of experiments to understand whether fit tables are useful during evolution tasks. In: International Conference on Software Engineering, pp. 361–370. IEEE Computer Society Press, Los Alamitos (2008)
47. Romano, D., Raila, P., Pinzger, M., Khomh, F.: Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In: Proceedings of the 2012 19th Working Conference on Reverse Engineering, WCRE '12, pp. 437–446. IEEE Computer Society, Washington (2012)
48. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. **14**(2), 131–164 (2008)

49. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples, 1st edn. Wiley Publishing, Hoboken (2012)
50. Salman, I., Misirli, A.T., Juristo, N.: Are students representatives of professionals in software engineering experiments? In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), May, vol. 1, pp. 666–676 (2015)
51. Sammet, J.E.: Brief survey of languages used for systems implementation. SIGPLAN Not. **6**(9), 1–19 (1971)
52. Seaman, C.B.: Qualitative methods in empirical studies of software engineering. IEEE Trans. Softw. Eng. **25**(4), 557–572 (1999)
53. Sharafi, Z., Marchetto, A., Susi, A., Antoniol, G., Guéhéneuc, Y.-G.: An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension. In: Poshyvanyk D., Di Penta M. (eds.) Proceedings of the 21st International Conference on Program Comprehension (ICPC), May. IEEE CS Press, Washington (2013)
54. Shull, F.J., Carver, J.C., Vegas, S., Juristo, N.: The role of replications in empirical software engineering. Empir. Softw. Eng. **13**(2), 211–218 (2008)
55. Sillito, J., Murphy, G.C., De Volder, K.: Questions programmers ask during software evolution tasks. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pp. 23–34. ACM, New York (2006)
56. Sjoeberg, D.I.K., Hannay, J.E., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N.K., Rekdal, A.C.: A survey of controlled experiments in software engineering. IEEE Trans. Softw. Eng. **31**(9), 733–753 (2005)
57. Sliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories MSR 2005, Saint Louis, MO, May 17, 2005
58. Soh, Z., Sharafi, Z., van den Plas, B., Cepeda Porras, G., Guéhéneuc, Y.-G., Antoniol, G.: Professional status and expertise for uml class diagram comprehension: an empirical study. In: van Deursen A., Godfrey M.W. (eds.) Proceedings of the 20th International Conference on Program Comprehension (ICPC), pp. 163–172. IEEE CS Press, Washington (2012)
59. Storey, M.A.D., Wong, K., Fong, P., Hooper, D., Hopkins, K., Muller, H.A.: On designing an experiment to evaluate a reverse engineering tool. In: Proceedings of the Third Working Conference on Reverse Engineering, 1996, Nov, pp. 31–40 (1996)
60. Swanson, E.B., Beath, C.M.: The use of case study data in software management research. J. Syst. Softw. **8**(1), 63–71 (1988)
61. Tatsubori, M., Chiba, S.: Programming support of design patterns with compile-time reflection. In: Fabre J.-C., Chiba S. (eds.) Proceedings of the 1st OOPSLA Workshop on Reflective Programming in C++ and Java, pp. 56–60. Center for Computational Physics, University of Tsukuba, October 1998. UTCCP Report 98-4
62. Thayer, R.H., Pyster, A., Wood, R.C.: The challenge of software engineering project management. Computer **13**(8), 51–59 (1980)
63. Tichy, W.F.: Hints for reviewing empirical work in software engineering. Empir. Softw. Eng. **5**(4), 309–312 (2000)
64. Tichy, W.F., Lukowicz, P., Prechelt, L., Heinz, E.A.: Experimental evaluation in computer science: a quantitative study. J. Syst. Softw. **28**(1), 9–18 (1995)
65. Tiedeman, M.J.: Post-mortems-methodology and experiences. IEEE J. Sel. Areas Commun. **8**(2), 176–180 (1990)
66. van Solingen, R., Berghout, E.: The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development. McGraw-Hill, London (1999)
67. von Mayrhauser, A.: Program comprehension during software maintenance and evolution. IEEE Comput. **28**(8), 44–55 (1995)
68. Walker, R.J., Baniassad, E.L.A., Murphy, G.C.: An initial assessment of aspect-oriented programming. In: Proceedings of the 1999 International Conference on Software Engineering, 1999, May, pp. 120–130 (1999)

69. Williams, C., Spacco, J.: Szz revisited: verifying when changes induce fixes. In: Proceedings of the 2008 Workshop on Defects in Large Software Systems, DEFECTS '08, pp. 32–36. ACM, New York (2008)
70. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering: An Introduction, 1st edn. Kluwer Academic Publishers, Boston (1999)
71. Yin, R.K.: Case Study Research: Design and Methods. Applied Social Research Methods. SAGE Publications, London (2009)
72. Zeller, A., Zimmermann, T., Bird, C.: Failure is a four-letter word: a parody in empirical research. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Promise '11, pp. 5:1–5:7. ACM, New York (2011)
73. Zendler, A.: A preliminary software engineering theory as investigated by published experiments. Empir. Softw. Eng. **6**(2), 161–180 (2001)
74. Zhang, C., Budgen, D.: What do we know about the effectiveness of software design patterns? IEEE Trans. Softw. Eng. **38**(5), 1213–1231 (2012)
75. Zimmermann, T., Nagappan, N.: Predicting subsystem failures using dependency graph complexities. In: Proceedings of the The 18th IEEE International Symposium on Software Reliability, ISSRE '07, pp. 227–236. IEEE Computer Society, Washington (2007)