Solving Problems or Enabling Problem-Solving? from Purity in Empirical Software Engineering to Effective Co-production (Invited Keynote)

Tony Gorschek[1](✉) and Daniel Mendez[1,2]

[1] Blekinge Institute of Technology, Karlskrona, Sweden
tony.gorschek@bth.se
[2] Fortiss GmbH, Munich, Germany

Abstract. Studying and collaborating with any software-intensive organization demands for excellence in empirical software engineering research. The ever-growing complexity and context-dependency of software products, however, demands for more pragmatic and solution-focused research. This is a great opportunity but it also conflicts with the traditional quest for "purity" in research and a very narrow focus of the work. In this short positioning, we elaborate on challenges which emerge from academia-industry collaborations and discuss touch upon pragmatic ways of approaching them along the co-production model which emerged from SERL Sweden.

1 Introduction

Software Engineering grew out of computer science-related fields and was baptized as a separate area to give legitimacy as a central discipline to handle challenges associated with the so-called software crisis becoming evident in the 1960s [8].[1] Empirical software engineering emerged in the 1990s and highlighted the focus on borrowing and adopting various types of empirical methods from other disciplines to conduct evidence-based research [6]. The aim was to collect evidence and build the bridge between practice in industry and research activities [1,8], thus, the discipline meant to increase the practical impact of mostly academic research contributions. The intent is explicitly stated in the name itself: "Empirical", denoting evidence collection and utilization, and "engineering" (-science) denoting the application of methods, tools, practices, and principles with the purpose of measuring the impact of said treatments in practice - this is also why many times the term "experimental" is intertwined with the "empirical" in empirical software engineering.

[1] http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html.

D. Winkler et al. (Eds.): SWQD 2021, LNBIP 404, pp. 109–116, 2021.
https://doi.org/10.1007/978-3-030-65854-0_9

In this position paper, we introduce and discuss some of the main challenges associated with scientific work in the area of *empirical software engineering* – and possible ways forward. This is based on experiences and lessons learned from the immediate applied research environment of the Software Engineering Research Lab (SERL Sweden) at Blekinge Institute of Technology, its long-term research collaborations with industrial partners, as well as through its extended collaborations with and connections to relevant research and transfer institutes such as fortiss.

Hence, the context of the discussion in this manuscript, which lays out the foundation for the keynote at the Software Quality Days 2021, is not the product or invention or innovation itself, but it is rather the methods, models, practices, tools, frameworks, processes, and principles applied as a way to enable the repeatable engineering of software-intensive products and services – in short, "ways-of-working". In terms of engineering, we include the inception of a product, through the planning, design, and realization, as well as delivery and evolution until eventual decommissioning.

We will only be able to scratch the surfaces of new ways of working, but hope with this positioning to foster an overdue debate to challenge the purity of more "traditional" ways of empirical research in order to allow for a more productive co-production.

2 The Problem(s) with "Software"

The problems of the development of software-intensive products and services can be divided, as we argue, into three main categories. These categories in themselves shed light on the complexity of the field, both from an engineering perspective, but also from an engineering research perspective:

1. The inception phase is put under the same umbrella as the engineering of the product.
2. Human-centric activities are very hard to measure objectively.
3. Delivery of product/service is not seen as a cost or effort center as in manufacturing centric instances.

2.1 Inception and Engineering

In mechanical or electrical engineering, or in the design of complex systems studied in the area of systems engineering, the inception phase (where you would "invent" and conceptualize products with long market exploration or non-technical prototyping phases) is often not seen as part of the engineering itself. Think of circuit design, for example, or maybe even more apt, the creation of clay scale models of cars that are wrapped in foil to explore designs and engineering decisions by a car manufacturer. This is often seen as "creative" endeavors and not the engineering or production of the item itself. Further, post inception, there is a translation phase where the inception meets the reality (and the constraints) imposed by the engineering and production world – translating a raw concept into something that can be eventually produced taking into account cost, scale, and repeatability of quality. This school of thought treats both engineering and creativity as two isolated, distinct, and often competing islands.

In software engineering, in contrast, all these parts are seen as "development" and more importantly, these parts are counted and measured as part of the teams' work – ranging from first product or service ideas over the exploitation of requirements and the development of first prototypes to the development, quality assurance, and deployment. Even project management activities are found under the same umbrella we call software engineering. This has significant implications for many reasons, not least when comparing plan-outcome and looking into the efficiency of the engineering organization as compared to rather traditional fields [2]. Needless to say that the challenges faced in software engineering can often not be found in other engineering activities – simply because related sub-disciplines are not associated with the engineering itself: think of challenges in human-centred and inherently volatile sub-disciplines like requirements engineering or during inherently fuzzy and uncertain tasks such as effort estimations.

2.2 Human-Centric Activities

Since software is created by humans for humans, any metrics and measurements used are largely dependent on measuring people and measuring them indirectly via artifacts associated with those people; for instance, by counting defects and their severity. If we setting aside for a moment the ethical and social aspects of human-based metrics, measuring humans and their activities remains a very hard and error-prone task. Is one engineer mentoring another something really non-productive? In addition, metrics are very seldomly information bearers. Rather, they are indicators for further analyses. To give an example, context switching and its measurement can be a good tool to gauge, for instance, team independence and the ability to focus on tasks. However, using the metric as a stand-alone part can be fraught with peril. For example, trying to push down context switch might hamper coordination efforts between teams or the ability for experienced engineers to answer questions by less experienced ones. In a traditional production environment, you can measure waste (e.g., failures or material waste); however, what constitutes waste and how this is separated from overhead in more human-centered activities is complex and very much context-dependent. In the example with context switch – the "best" solution might not be to identify the phenomenon as Waste, rather as Overhead and that a balance of not too much and not too little needs to be found – if the balance is not maintained, Waste is introduced.

2.3 Delivery

First, most "software" developed is actually not stand-alone. Rather, it is part of a product or service. The old adage that "software is free to deliver" propagates the myth that delivering software is without cost. This is of course false.

To illustrate this, let us consider two extreme examples: A purely software-based product, such as accounting software, and an embedded software product, such an infotainment software in a car. In both cases, there is a significant cost for both the development company to package and supply (in essence commissioning) the product, and more importantly, there is a cost of risk for the customer. If consumers have accounting software that works well and that they use for their business, are they risk-averse in willingly updating the software as it might cause them harm (why fix what works?). On the other hand, the development company wants to push out a homogenous (and recent) version and variant of their product – if for no other reason to lessen their support costs and the costs of maintaining versions in the field. This might seem like a trivial part, but it has vast implications for development companies as the cost of product evolution, interoperability of features, but also products are affected by the version spread. An additional difference from traditional engineering (say a car manufacturer) is that the "development organization" in the case of software has to not only evolve the products but also continuously fix issues as they come along, likely in various versions and releases maintained in parallel (interrupting the work of evolution activities).

3 Research with and in Industry

The case for empirical software engineering, beyond data-collection activities, is that the contextual aspects of a company, domain, and development organization can be taken into consideration during research. This is especially true if the researcher uses a co-production approach to the research [3,4,10] as exemplified in Fig. 1.

Figure 1 gives an overview of a co-production model that evolved at SERL Sweden over the last decades with lessons learned from collaboration with over 50 industrial partners and two dozen research projects. The model should not be seen as a recipe or a blueprint to follow. It is rather an illustration of possible steps and aspects to take into consideration to realize close industrial collaboration from a research perspective.

Below, we elaborate each step and connect them to major challenges as well as opportunities we have experienced in above mentioned contexts. For details, please follow [4].

Starting the work with any company (industry partner) involves building trust as well as knowledge about the company and its domain [5], both from a contextual perspective, and the inner workings (steps 1–2). Finding proper problems before solving them properly is essential. For a researcher, however, this is also critical since it allows access and, more fundamentally, the building of trust, and the ability to identify sources and, more importantly, input as to how relevant data (to relevant problems) can be efficiently collected and how the data can and should subsequently be analyzed and interpreted. A critical part of the initial steps is to set realistic expectations. Especially in new relationships where companies are not used to researchers, industry partners may often fall to a default "consultancy" mindset. The main difference between consultancy and research from a research perspective is quite simply to follow the question whether there is anything new to accomplish. Research should, in our view, be

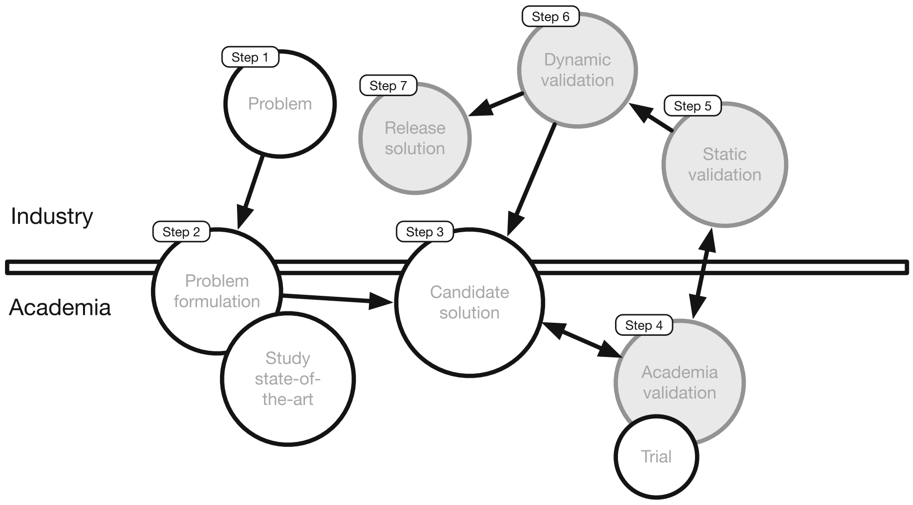

Fig. 1. SERL Co-production model. Adapted from [4].

characterized by that a new and relevant problem is solved, and/or new knowledge created (as the simplification of aiming for practical and/or theoretical impact). If this is not possible given a task, even partly, it should not be considered a viable piece of work that is equally relevant to both the participating researchers and practitioners. That being said, there are many types and flavours of research contributions. Say, for example, that a researcher aims at introducing already established (but new to their partner) ways of working. At a first glance, the researcher might not be creating new models, methods, practices, or equivalent. There might still be a research contribution as the introduction itself and the subsequent measurement of the effectiveness and efficiency of the proposed solution is a contribution. For the researcher, the trade-off is, however, the "researchabilty" of the work versus the effort put in versus the good-will created in the collaboration.

The main point here is to differentiate (steps 1–2) symptoms from actual challenge. "We have too many defects", as an example, can be a symptom of many things. Often addressing the symptom offers less researchability than addressing the underlying cause(es).

One of the main challenges here, in addition to selecting a challenge that is researchable, is that data collection often goes through people in a company, and often the measures are based on people. Setting aside ethical implications, which must be handled professionally, trust is paramount for access and to minimize the impact of the measurements. In this article, we do not even try to elaborate on this further to the extent it deserves. What is interesting to observe though is that the complexity of the case studied is compounded by both the human-centric part (as one often does not have machines to study), and the wide area of responsibilities of the teams ranging from the inception over the evolution and delivery to continuous maintenance.

Once a challenge is identified and analyzed (and compared to state-of-the-art in research), ideas for how to address challenges can be seen as "solutions" (Step 3). This is often not as clear-cut as a 1-to-1 mapping between challenges and solutions. Rather, identifying "a challenge" often escalates in itself to many discrete sub-challenges during a root cause analysis [7], and any solution proposed is in itself often a package of items that aim at addressing the challenge.

Thus the relationship is rather (Solution(many-to-many)-to-Challenge (many-to-many)) to illustrate the complexity of empirical work – at least if you intend to undertake a co-production style of research work. However, this is not necessarily something bad but rather, a solution can have unintended consequences, both positive and negative, something to be observant of during the work. The concept of applying a treatment (as in the traditional notion of experimental software engineering [11]) and taking confounding factors into account still apply. However, the control of the environment is more complex and the singular nature of the treatment is often in conflict with the usefulness of it in the case company.

Steps 4–6 should not be seen as validation only, even if their base purpose is to measure to what extent a solution impacts the company and solves the challenge(s) addressed; e.g. through a study where we aim at analyzing the extent to which we satisfied the original improvement goals. The base idea is to measure the effectiveness (fit for purpose) and the efficiency (fit for purpose in relation to cost) of a solution. An important part here which makes measurement more complicated is that every round of "validation" gives not only efficiency and effectiveness data, but also input as how the treatment (solution) can be improved to maybe give better efficiency and effectiveness. In essence, if we use this data to improve the solution as validation progresses, we make results between measurements harder to compare, and we potentially introduce confounding factors. So is the solution to keep the solution singular and pure throughout validation rounds? From a non-co-production perspective this would be best. However, from a practical perspective – if we want to continue collaborating with the company – this might not be the best or most desirable way.

These effects are impossible to remove completely, but they can at least be partly mitigated. Examples of mitigation are the selection of challenge-solutions. Smaller and faster iterations and interactions are not going to escalate in complexity as larger ones. In addition, a close collaboration beyond "data collection" is of utmost importance, in essence by adapting an action research approach [9] but only from a macro level perspective. That is to say, every step (e.g. a validation) can be conducted using any research method, e.g. via interviews, task observations, or by measuring results from tasks, just to name a few. Hence, our experience does not indicate that researchers should be involved in the actual application of the solution on a daily basis, as it is typically the case in action research, but, rather, that the close nature of introducing, training, monitoring, and measuring its application on a macro level is close to so called action research. This allows the researcher to catch contextual information as to how the solution is used, which are hard to measure, and/or even being able to catch if actions during solution validation invalidate measures that from an outside perspective look valid.

4 Discussion

What do the perspectives of inception-realization, human-centric and, and delivery have to do with the research? Studying and collaborating with any organization – (software) engineering research mandates empirical and collaborative activities – is both qualified and complicated by the contextual factors in question.

In this short position paper, we illustrated that not only are software-intensive companies not simpler than traditional ones, rather more complicated as the division of responsibilities are often less clear cut. Further, more "traditional" companies are not a reality anymore as most companies are becoming more and more software-intensive. Hence, the need for good, pragmatic, solution-focused research is growing exponentially. This is a great opportunity but also conflicts with the traditional quest for "purity" in research and a very narrow focus of the work. This can not be completely solved, but it is a balance that has to be struck out of necessity for the research to be credible and useful for the company.

This is important to understand for conveying how empirical work in software engineering is generally done, and how co-production style research should be approached in particular. We illustrated this along our co-production model which emerged from decades of academia-industry collaborations at SERL Sweden.

References

1. Basili, V.R., Selby, R.W., Hutchens, D.H.: Experimentation in software engineering. IEEE Trans. Softw. Eng. **7**, 733–743 (1986)
2. Gorschek, T.: Evolution toward soft(er) products. Commun. ACM **61**(3), 78–84 (2018)
3. Gorschek, T., Garre, P., Larsson, S., Wohlin, C.: A model for technology transfer in practice. IEEE Softw. **23**(6), 88–95 (2006)
4. Gorschek, T., Wnuk, K.: Third generation industrial co-production in software engineering. Contemporary Empirical Methods in Software Engineering, pp. 503–525. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-32489-6_18
5. Junker, M., et al.: Principles and a process for successful industry cooperation-the case of TUM and Munich Re. In: 2015 IEEE/ACM 2nd International Workshop on Software Engineering Research and Industrial Practice, pp. 50–53. IEEE (2015)
6. Mendez, D., Passoth, J.H.: Empirical software engineering: from discipline to interdiscipline. J. Syst. Softw. **148**, 170–179 (2019)
7. Pernstål, J., Feldt, R., Gorschek, T., Florén, D.: FLEX-RCA: a lean-based method for root cause analysis in software process improvement. Softw. Qual. J. **27**(1), 389–428 (2019)

8. Randell, B.: The 1968/69 NATO Software Engineering Reports. History of Software Engineering 37 (1996)
9. Wieringa, R., Moralı, A.: Technical action research as a validation method in information systems design science. In: Peffers, K., Rothenberger, M., Kuechler, B. (eds.) DESRIST 2012. LNCS, vol. 7286, pp. 220–238. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29863-9_17
10. Wohlin, C., et al.: The success factors powering industry-academia collaboration. IEEE Softw. **29**(2), 67–73 (2011)
11. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer Science & Business Media, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29044-2