Beyond the Badge: Reproducibility Engineering as a Lifetime Skill

Wolfgang Mauerer
Technical University of Applied Science Regensburg
Siemens AG, Corporate Research
Germany
wolfgang.mauerer@othr.de

Stefan Klessinger
Chair of Scalable Database Systems
University of Passau
Germany
stefan.klessinger@uni-passau.de

Stefanie Scherzinger
Chair of Scalable Database Systems
University of Passau
Germany
stefanie.scherzinger@uni-passau.de

ABSTRACT

Ascertaining reproducibility of scientific experiments is receiving increased attention across disciplines. We argue that the necessary skills are important *beyond* pure scientific utility, and that they should be taught as part of software engineering (SWE) education. They serve a dual purpose: Apart from acquiring the coveted badges assigned to reproducible research, reproducibility engineering is a *lifetime skill* for a professional industrial career in computer science.

SWE curricula seem an ideal fit for conveying such capabilities, yet they require some extensions, especially given that even at flagship conferences like ICSE, only slightly more than one-third of the technical papers (at the 2021 edition) receive recognition for artefact reusability. Knowledge and capabilities in setting up engineering environments that allow for reproducing artefacts and results over decades (a standard requirement in many traditional engineering disciplines), writing semi-literate commit messages that document crucial steps of a decision-making process and that are tightly coupled with code, or sustainably taming dynamic, quickly changing software dependencies, to name a few: They all contribute to solving the scientific reproducibility crisis, *and* enable software engineers to build sustainable, long-term maintainable, software-intensive, industrial systems. We propose to teach these skills at the undergraduate level, *on par* with traditional SWE topics.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Software and its engineering** → *Maintaining software*; *Software version control*.

KEYWORDS

reproducibility engineering, teaching software engineering

ACM Reference Format:
Wolfgang Mauerer, Stefan Klessinger, and Stefanie Scherzinger. 2022. Beyond the Badge: Reproducibility Engineering as a Lifetime Skill. In *4th International Workshop on Software Engineering Education for the Next Generation (SEENG'22), May 17, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3528231.3528359

SEENG'22, May 17, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9336-2/22/05.
https://doi.org/10.1145/3528231.3528359

1 INTRODUCTION

Since software engineering involves complex software stacks that non-trivially interact with hardware, sharing experimental setups is anything but trivial. Over the last decade, reproducibility of experimental results has become recognised as a prime aspect of computer science (CS) research. Several high-profile conferences now award badges when results can be independently verified.

Undoubtedly, reproducibility engineering (RepEng) has become a crucial skill that today's generation of PhD students has to master. In this position paper, we argue that these skills should already be taught (and practised) at the undergraduate level, and we therefore designed and conducted a course for computer science Bachelor students close to graduation. Even when students pursue an industry career, they will greatly benefit from recognising threats to reproducibility, how to tackle them, and how to build long-term reproducible code. In short, it is our conviction that students skilled in RepEng possess skills that proficient software engineers need to master (anyway).

Accordingly, we propose a multi-faceted syllabus[1] for teaching reproducibility engineering, and what we consider crucial skills. This includes best practices in computer science research and industry, such as packaging entire system software stacks for dissemination. For *long-term* reproducibility over decades (ideally, forever), we discuss why open source technologies (as massively employed in industry) are preferable to approaches crafted for research.

Structure. We recap essentials on building reproduction packages. We propose a high-level syllabus, covering social and technical best practices, as well as specific tools and technologies that are well-adopted in industry. We then discuss the literature material available for academic teaching, and conclude.

2 PRELIMINARIES

Terminology. Reproducibility is a cross-cutting theme and there are guidelines by the National Science Foundation (NSF) [17], the Association of Computing Machinery (ACM) [1], and the Institute of Electrical and Electronics Engineers (IEEE) [8]. Concepts like reproducibility and replicability receive different interpretation, depending on the community. Even large professional associations like the ACM had to revise their definitions of the terms because of prior confusion. Despite their obvious relevancy, the concepts are not yet reflected in the ACM Computing Classification System[2].

[1] We have implemented the outlined ideas in an online course, taught in the winter term 2021/22, to undergraduate students at two universities. The lecture videos are available online on YouTube (link in PDF).
[2] Available online (link in PDF), last updated 2012.

Throughout this article, we follow the ACM terminology [1] (version 1.1), and regard an experiment as *repeatable*, when the same team with the same experimental setup can confirm the results. An experiment is *reproducible*, if it is a different team, but the same setup, that confirms the results. Finally, an experiment is *replicable*, when a different team, with a different setup, confirms the result.

Reproduction Packages. Building a *reproduction package* goes beyond providing a document object identifier (DOI) to some repository hosting data, code, and setup instructions. Rather, a gold-standard reproduction package [7] bundles all research artefacts required to conduct the experiment (such as source code, libraries, or input data), and contains a dispatcher script that allows for executing and evaluating the experiment via a *single* command.

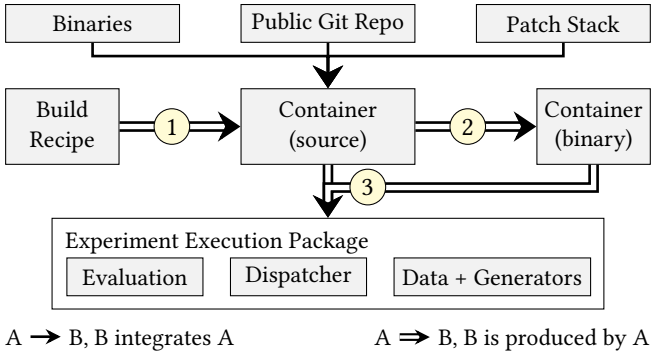

Figure 1: Building a reproduction package [11].

Figure 1 (adapted from [11]) shows a state-of-the-art setup. Based on system binaries, external and internal code in git repositories, and patch stacks with changes to existing components, a build recipe induces generation of a host-system independent Docker container as (static and immutable) build environment for measurement binaries (1). Additionally, a Docker container with pre-built binaries, devoid of any external dependencies, is created (2). The Docker container creates an experiment execution package (3) that can be deployed on cloud systems, or on local hardware, without any dependence on target-system provided artefacts. The experimental runs generate data, which are post-processed, evaluated and visualised by scripts in the experiment execution package.

3 A MULTI-LEVEL SYLLABUS

We argue that reproducibility engineering should find its way into undergraduate curricula, anchored in software engineering education. By targeting a clearly scoped audience (rather than STEM disciplines in general), we can address matters *to the point*, and provide actionable advice beyond the mechanical use of tools, or compliance to formal processes. In sketching out a syllabus, we propose a multi-level approach, and distinguish social and technical best practices. We further review specific tools and technologies.

3.1 Best practices: Social

A reproduction package should contain as many details as necessary, but must not overwhelm. Instead of minutiæ of how results were obtained, a reproduction package presents a concise and balanced view of the *outcome* of an effort. While any *structural decisions* are worth preserving, the temporal order of the *thought process* that led to intermediate results or to said decisions, is usually not.

Industry has established conventions [18] on documenting software changes (at the granularity of individual commits) to provide an understanding of the evolution of large software systems. These conventions can also be applied to documenting research progress. Such *trails of responsibility* (which persons authored changes together, who provided reviews, who participated in design decisions, etc.) are routinely created outside academia (contrariwise to the care taken in giving credit and attribution in published papers, this approach is not established in many areas of computer science). Figure 2 shows an example: It contains the technical change in form of a diff (bottom part), and metadata (unique hash, author and committer) as they are provided by version control systems like git. Apart from this information, as it is widely used in repository mining research [23], the commit also includes a summary of the change, and a rationale (brief for the sake of example) *why* the change is necessary, and *which* techniques are employed.

The commit can be seen as a form of communication with fellow humans instead of mere instructions for machines, following Knuth's seminal *literate programming* concept [10]. To create *readable histories*, we suggest to introduce the pragmatic customs developed in large, international and multi-disciplinary infrastructure projects (such as the Linux kernel) in software engineering courses.

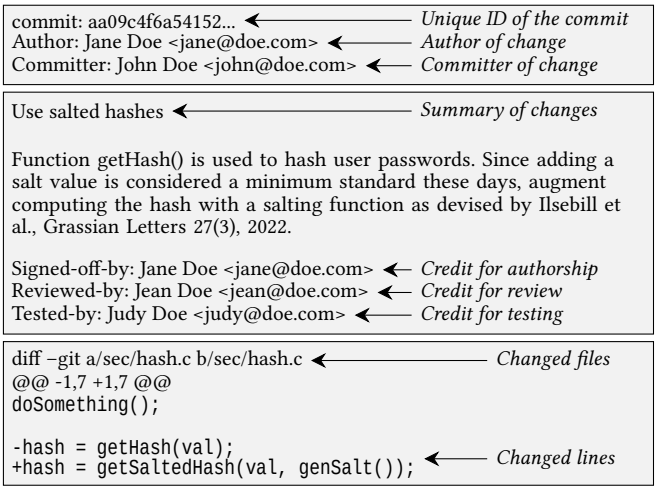

Figure 2: A technical software change accompanied by a trail of responsibility, revealing the rationale behind the change.

3.2 Best practices: Technical

We need to provide actionable guidance on how to implement the suggested procedures and approaches. This entails covering the necessary means *end-to-end*, from preparing all software components required to perform experiments, running analysis code and evaluations, and to creating insightful visualisations.

Building research artefacts depends on external sources, whose long-term availability is often not sufficiently considered. Particular care is taken to raise awareness for identifying potential issues when aiming at *reproducible builds*.

The *granularity of packaging artefacts* is an important discussion point: Should reproduction packages start directly with building the operating kernel from source, to establish absolutely identical conditions given identical hardware, or is it sufficient to package custom code that leverages any suitable execution platform? Likewise, should and can reproduction efforts re-compute all derived results, or start with data obtained from long-running calculations?

Another dimension concerns the variability of programming language, compiler and toolchain, and the distinction between build, execution, and evaluation platform. Each of the combinations that appear in practice have peculiarities worth discussing.

Furthermore, we consider the technical ramifications of different types of reproducibility introduced in Section 2: Depending on what type of quantities are handled—physical quantities like time or energy consumption, numeric results from deterministic or stochastic processes, etc.—, different means ensure that it is possible to decide whether a reproduction attempt is successful.

Using closed-source, *proprietary components* creates hurdles for other researchers, and should be avoided in ideal open science. However, relying on proprietary components cannot be completely avoided, so we need to discuss how to best handle such scenarios.

Advanced numerical techniques that require accelerator hardware such as GPUs receive increasing attention in machine learning and artificial intelligence projects. The involved software stacks do not only contain binary-only, proprietary components whose licenses place obstacles on distribution, but also do not play well with virtualisation and containerisation approaches. We need to discuss how to handle these issues specific to *dealing with hardware*.

Finally, we need to address how to properly package artefacts and ensure their *long-term availability*. Besides using well-structured hierarchies and self-documenting package formats, we address dual strategies towards short- and long-term reproducibility: The latter aims at decades of reproducibility, at a higher cost to the reproducers, while the former accepts technologies and platforms that are not certified for *DOI-safety*, but allow for easier integration into standard development workflows. This balances advantages of long-term reproducibility with the ease of continuous development.

3.3 Tools and Technologies

We need to demonstrate tools that implement the previously discussed techniques. Primarily, we focus on Linux/Unix-based command-line tools, as these are also conveniently available on standard operating systems (Windows/Mac OS). This does not necessarily hold the other way (*e.g.*, Powershell), and for GUI approaches. A small subset of the tool functionality is sufficient for reproducibility engineering, and command-line based approaches are helpful locally and on servers. We suggest starting with means for easy, but effective, *low-threshold automation* based on efficient interaction with shells, pipelined processing of data, and *glue languages* such as python, R or Matlab.

Non-linear history rewriting provided by git allows for transforming chronological records into a readable, consistent research process documentation by splitting, merging, and re-ordering. Outside of software engineering, we have encountered little knowledge of such transformations, yet they are crucial to ascertain long-term human understandability.

Virtualisation and containers [2] play a major role in our strategy: For one, they avoid having to deal with different versions and compositions of compilers, libraries, and system software when building artefacts. Also, they allow for establishing a completely self-contained environment without external internet-based dependencies that remains operational even decades after the original sources have vanished (figuratively and literally, research is even possible when trapped on a remote island). Careful engineering of containers ensures they are suitable for reproduction tasks. The appropriate techniques and patterns should therefore be introduced.

The `reprotest` tool collection is a recommended means to satisfy requirements for reproducible builds: By varying environmental parameters like user ID, folder names, or compiler settings, the tools detect issues that do not surface when a single researcher builds code on the always-same machine. Such setups lead, in our experience, to important insights on subtle sources of errors caused by implicit, yet common misconceptions. While such tools are routine for developers of distributions like Debian, and also key to long-term industrial maintainability of software, we find them not yet sufficiently integrated into SWE curricula.

We suggest to implement *preparing and documenting experiments* using `knitr`, which is not unsimilar to the paradigm of literate programming [5, 10]. It also allows for creating self-contained papers that realise end-to-end reproduction. *Electronic notebooks* like `Jupyter` are a recommended variant.

How to *describe and pin down the execution environment* is a further challenge. Typical hardware specifications in published research describe the experimental conditions along the lines of 'Linux version 5.1.92 on a Dull Powervortex 4711 with 24 GiB of RAM was used'. This is insufficient for reliable reproduction—non-standard kernel extensions that may vary widely depending on the distribution, specific settings for tuning parameters that exist in a wide variety on every system, and many other factors that may easily be dismissed as irrelevant can impact measurements by orders of magnitude. We recommend discussing means of faithfully recording the execution conditions of computational experiments.

Students should acquire hands-on experience in *reproducing experimental outcomes*. Retracing the work of others increases awareness for (and appreciation of) high-quality reproduction packages.

3.4 Special Cases

While software engineering can often be decoupled from details of the target environment (CPU architecture, OS version, …), special-purpose hardware introduces additional reproducibility requirements. We find that general-purpose graphical processing units (GPGPUs) necessitate software stacks that exceed standard compilers considerably in size, and introduce (a) strong interdependencies between software component versions and (b) dependencies on specific features that might only find intermittent support in hardware. Both stress the need to teach implementing less performant, but generic alternatives, and how to store intermediate results obtained from HW accelerators for further processing. Similar considerations hold for tensor processing units (TPUs) and other AI accelerators. Quantum computing, starting to receive interest from the software engineering community [19, 22], additionally needs to deal with globally unique hardware semi-prototypes [14].