Electronic Communications of the EASST
Volume 83 Year 2025

deRSE24 - Selected Contributions of the 4th Conference for Research Software Engineering in Germany

Edited by: Jan Bernoth, Florian Goth, Anna-Lena Lamprecht and Jan Linxweiler

Teaching Research Software Engineering Skills for Developing Simulation Software

Gerasimos Chourdakis, Hasan Ashraf, Santiago Narvaez Rivas, Tobias Neckel, Hans-Joachim Bungartz

DOI: 10.14279/eceasst.v83.2615

Teaching Research Software Engineering Skills for Developing Simulation Software

Gerasimos Chourdakis[1 2], **Hasan Ashraf**[1], **Santiago Narvaez Rivas**[1], **Tobias Neckel**[1], **Hans-Joachim Bungartz**[1]

[1]School of Computation, Information and Technology, Technical University of Munich
[2]Institute for Parallel and Distributed Systems, University of Stuttgart
gerasimos.chourdakis@tum.de

Abstract: Graduates of computational programs write research software at different stages of their career, yet software engineering skills are often assumed to be prior knowledge or a product of self-study. In order to better prepare graduates for developing larger projects, such as the next PETSc, deal.II, or preCICE, in the M.Sc. Computational Science and Engineering of the Technical University of Munich, we introduced software engineering skills in different courses in a natural way, forming a coherent curriculum track for learning how to write and maintain sustainable simulation software. This starts with the onboarding block-course “CSE Primer” (introducing the basics of Linux, Git, Matlab/Octave, C++, and teamwork via lectures and a team project), continues with “Advanced Programming” (a semester-wide C++ course touching numerical computations, software design, performance, and several tools on the way), and expands to a variety of practical courses, which involve building larger projects in a team. After acquiring experience with such larger projects, students can take advantage of courses such as “Patterns in Software Engineering” to analyze and improve the design of their codes. All of these courses have been very well received by students, and the course “Advanced Programming” has been attracting an exponentially increasing number of students from multiple engineering curricula, making the originally intended CSE audience only a small minority. This paper gives an overview of these courses, discusses their connection, and highlights several didactical and implementation elements of each.

Keywords: teaching, didactics, curricula, research software engineering

1 Introduction

Simulation software, such as codes that simulate molecules, fluids, or galaxies, or simply the numerical libraries at the core of every scientific code, is complex. As a type of research software, it shares the same challenges. Codebases such as PETSc, deal.II, or preCICE[1] are often large, with a long history of development, most commonly in academia, and by individual or teams of developers with little formal education in software engineering [WPP20, HBC+22]. Successful scientific codes excel not only in code, but also in testing, documentation, and further elements [BH13], which are often not part of an engineering curriculum. Since domain

[1] PETSc: https://petsc.org/, deal.II: https://dealii.org/, preCICE: https://precice.org/

scientists develop these codes, we need to teach them the technical skills that they miss. These include adapting to the software life cycle, creating documented code building blocks, building distributable software, using software repositories, and being able to analyze the behavior of a code [GAB+24]. In addition to that, simulation software engineering requires awareness of accuracy, performance, and maintainability implications of design choices. In simulation software, as we understand it for the context of this publication, results are always approximate and rarely bitwise reproducible (in contrast, e.g., to software for discrete or symbolic mathematics), and the runtime and required computational resources are often substantial (in contrast often, e.g., to software for statistical analysis of experimental data). Execution is typically non-interactive and often driven by configuration files and a command-line interface, while the output typically comprises several files of substantial size. Several numerical libraries are typically used in a simulation software project, often each written in a different programming language, with different build systems, incompatibilities with each other, and very different maintenance plans. We need to not only prepare students for creating more efficient simulation software, but also for creating maintainable and usable large-scale projects. With students being better prepared, we can also save significant time and effort in getting them integrated into the projects developed in our institute, while they can invest more time in the research part of their thesis.

The research software engineering community currently discusses how an RSE Master's curriculum could look like [GAB+24], and collects related teaching resources[2]. Comparing several Computational Science and Scientific Computing curricula to the above list of competencies[3], we notice that several degrees are very close to what the community currently envisions as an RSE curriculum. In particular, the M.Sc. Computational Science and Engineering program at the Technical University of Munich has traditionally invested significant time on Software Engineering, for example via the courses "Advanced Programming", "Patterns in Software Engineering", and several practical courses. Some of us have in the past participated in these courses as students, which prepared us for our current roles. Having identified skills that we wished we would have acquired earlier, in a structured education context, and having coached students to contribute into multiple research software projects, we took over the task of largely redesigning and updating the core software engineering module that CSE students experience in their first semester: the IN1503 "Advanced Programming". Investing significant effort to redesign such courses has been shown to have positive effects in student engagement [Mal10]. Identifying common implicit prerequisites shared with other courses, we also introduced the course "CSE Primer". To facilitate a learning path towards simulation software engineering, we further integrated these with other courses that students can follow later in their curriculum.

In the following, we give an overview of our design decisions and several implementation details for these courses, and we discuss their reception by students and our observations as instructors. Observations are based on pre-course surveys and midterm course evaluations from at least six years. This paper is primarily addressed to teachers that design or conduct similar courses, and further to curricula designers, thus the increased focus on implementation details. While our perspective originates from simulation software, most of the ideas are directly applicable to teaching any other type of (research) software. This perspective, and our experience with

[2] Learning and teaching RSE (The Teaching RSE project): https://de-rse.org/learn-and-teach/

[3] Study programs – Learning and Teaching RSE: https://de-rse.org/learn-and-teach/learn/#study-programs

integrating students into the development of preCICE or other projects developed at the institute[4] has also influenced the topics discussed in the courses. In Section 2, we discuss the onboarding course "CSE Primer". In Section 3, we discuss the course "Advanced Programming", with a particular interest on the scalability of its implementation. In Section 4, we give an overview of how these courses interplay with other modules in the curriculum. In Section 5, we conclude with key observations, recommendations, and next steps.

2 Harmonization: The CSE Primer module

As a graduate from a science, technology, engineering, and mathematics (STEM) degree, first-semester students of computational degrees, such as CSE, have some experience with programming, as this is a qualification requirement. However, this experience is often basic or not very recent. At the same time, first-semester courses build upon such requirements and use technologies that are not commonly part of undergraduate engineering curricula, such as Linux, version control, remote access, or systems programming. The CSE Primer[5] serves as an onboarding course, bringing the technical skills of the students to a common minimum level. It teaches technical skills beyond programming, similarly to the MIT course "The Missing Semester"[6] and the Carpentries[7], but specialized on the needs of computational degrees. After several iterations of previous work, this course was funded via the TUM Ideas Competition[8] to be extended and transformed to a project week format. It now covers the complete first week of lectures, in the format of interactive lecture in the mornings and group project work in the afternoons. In the following, we give an overview of the content in Subsection 2.1, discuss our experiences with the project component in Subsection 2.2, and discuss the reception in Subsection 2.3.

2.1 Content and format

In their first semester, CSE students go deeper into programming concepts using a systems language (e.g., C++ in the Advanced Programming course), study numerical algorithms by implementing them (e.g., in MATLAB in the Scientific Computing Lab), and are expected to work in group projects and develop software on Linux, later on connecting to remote systems such as HPC clusters. We prepare the students with these technical skills in this course.

Our pre-course surveys depict the background of a first-semester computational degree student. In the beginning of the latest iteration of the course, approximately 9 out of 10 of the students had previous experience with Python, 6/10 had some experience with C++, followed by 5/10 having some experience with C and Matlab (Table 1). In the past years, responses for C++, C, and MATLAB seem to be reducing, while responses for Python are increasing (Figure 1). Only 1 out of 4 students feels confident using Linux, while half the students have close to no experience with the Linux shell, or no experience at all. Consistently over the past four years, 3/10 students seem to have never used Git, 3/10 have used it once or twice, and 4/10 use it often.

[4] Software developments at TUM SCCS: https://www.cs.cit.tum.de/en/sccs/research/software-developments/
[5] TUM Moodle page for CSE Primer: https://go.tum.de/907573 (available via guest access)
[6] MIT course "The Missing Semester of your CS education": https://missing.csail.mit.edu/
[7] Software carpentry: https://software-carpentry.org/
[8] TUM Ideas Competition 2021: https://go.tum.de/720303

Table 1: Pre-course survey of the CSE Primer module, winter semester 2023-24. Question: "Which programming languages have you used (more than a few hours)?" (multiple-choice, fixed options) (left), and details regarding C++ and MATLAB (right).

Language	N	Share
Python	35	0.90
C++	23	0.59
C	20	0.51
MATLAB	18	0.46
Java	11	0.28
R	4	0.10
Fortran	3	0.08
Rust	3	0.08
Julia	2	0.05

	C++		MATLAB	
Experience	N	Share	N	Share
None	7	2 / 10	15	4 / 10
Knows some C	4	1 / 10	-	-
Self-taught (a bit)	9	2 / 10	5	1 / 10
Had a full course	7	2 / 10	4	1 / 10
Used it in projects	10	3 / 10	13	3 / 10
Staying up-to-date	0	0 / 10	-	-
Total	37	10 / 10	37	9 / 10

Students gather in a computer room, where they get access to systems on which everything is already installed. In the first day, after getting introduced into details of the study program, they follow an interactive lecture on Linux. Parts of the lecture are self-paced, where students follow videos and execute commands, under the supervision and help of the instructors, and with regular synchronization points for discussion. An interactive lecture on teamwork, with small group discussions, showcases common time organization methods and project management techniques, and sets the expectations for a healthy work environment in the groupwork that follows later in the semester. Students also get introduced to collaborative development with GitLab and version control with Git (in this order, to make the first contact easier). The rest of the week includes a day of Octave/MATLAB, two days of C++ basics and debugging, and completes with an extended and open Q&A session and a social event.

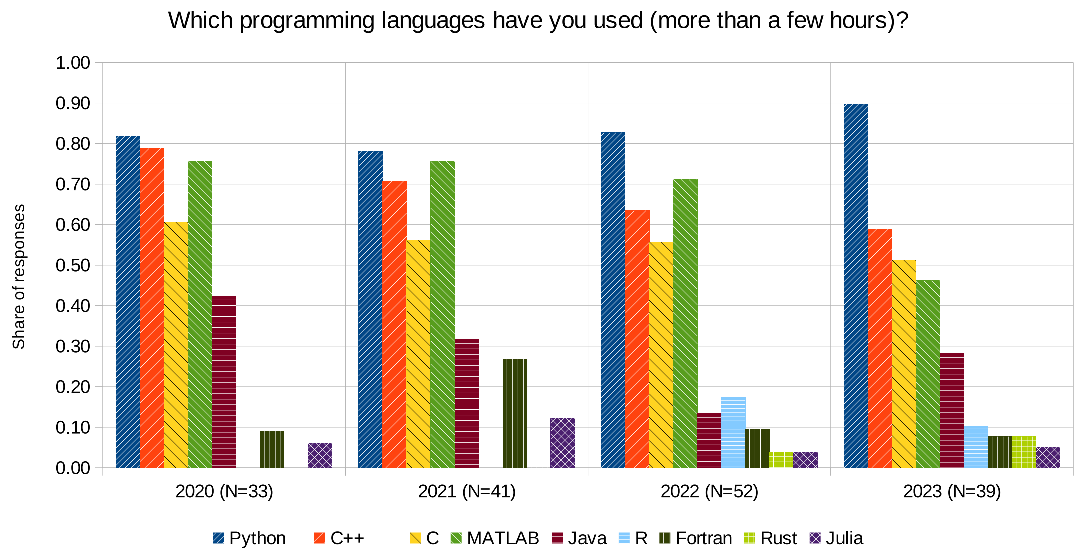
2.2 Project

The main goal of the project component is to let the students mingle and give them larger-scale hands-on experience with the tools discussed, in addition to the smaller exercises that are already part of the lecture. The students get climate-related data (sea surface temperature and sea ice surface), pre-process and plot the temperature data using Linux shell tools and MATLAB, and compute how the ice surface evolves over the years using C++. Later in the week, students can follow their own direction: some have created 3D surface plots of the temperatures, some have wrapped their application with a GUI, while others have studied the El Niño effect. To test their documentation practices and collaboration workflows, teams are forced to exchange one member during the last phase of the project. The project completes with presentations, after which students nominate the best projects.

2.3 Reception

While the course was originally designed for CSE, it is relevant for any computational degree. Due to scheduling issues (it currently blocks the complete first week of lectures for CSE students)

Figure 1: Pre-course survey CSE Primer over four years: Programming languages.



and clearly because of its name, it has so far attended primarily by CSE students. Since this is a free module, we do not have concrete attendance data, but most CSE students attend the course, and most of the students that attend also fill the pre-course surveys. While an online attendance option is offered, via streaming, Q&A via chat, student interaction via breakout rooms, and an assistant supervising the remote platform, this option remains largely unused. However, all the material is open and can be followed outside the context of the course. The course evaluations are very positive, consistently giving the lecture an overall grade of 1.3 in the German academic grading system (1: Best, 5: Worst). Students have requested similar courses for Mathematics and Parallel Programming, while similar courses for home domains (e.g., Physics or Chemistry) could help Computer Science graduates better integrate with other STEM graduates in a similar RSE curriculum. Next steps are more actively attracting students of curricula with similar needs, and introducing further interactive and self-paced elements.

3 Application and analysis: The Advanced Programming module

Assuming basic programming skills, we need to also expose students to software engineering elements common in simulation software. With the module Advanced Programming, we offer a first touch with numerical computations and accuracy, software design (focusing on object-oriented programming), and performance topics. We not only discuss these as abstract concepts, but students also apply them in practice, while learning tools common in the development of simulation software. While originally designed to be a required course for CSE students, it is now massively attended as an elective course by students of various STEM programs. This leads to a much larger audience than in the CSE Primer, as discussed in Subsection 3.8.

The module consists of a frontal lecture (90min per week, for 13 weeks), an interactive tutorial without deliverables (also 90min per week, for 12 weeks), and an optional group project. The final grade is determined by a final exam and a bonus contribution (one step of 0.3 in the German academic grading system) from the project. The course is organized in self-contained and well-integrated weeks, where each tutorial discusses concepts presented in the lecture of that week.

We give an overview of the content in Subsection 3.1 and we discuss the components (lecture, tutorial, and project) of the module in Subsection 3.2, Subsection 3.3, and Subsection 3.4, respectively, focusing on specific challenges and solutions. We discuss the assessment methods in Subsection 3.5. In Subsection 3.6, we give a glimpse of the currently employed automation, and we discuss the reception in Subsection 3.7 and open scaling issues in Subsection 3.8.

3.1 Content

The content of the course is described in the module description[9] and is openly available on the learning management system of TUM via guest access[10]. The source files of the LaTeX-based slides are currently not open, but it remains the intention of the authors to make these available as well under a permissive license, after curation and approval, as this can lead to further exposure and contributions by the community (see, for example, a C++ course by F. Busato[11]).

The course presents concepts of programming and software engineering for performance-relevant scientific applications, using C++ as a working language. Since most of the students do not already have any significant previous experience with C++, this can be seen as a C++ course tailored to the needs of scientific software development. While there are many good C++ courses already available, also as open-access, unique differences of this course are the focus on performance topics, the tutorial and project components, the demonstrating examples providing evidence for the discussed topics, and the vast range of additional tools and external references offered in addition to the lecture. Topics covered include the von Neumann architecture, data types (including precision of intrinsic types and examples of related bugs), functions (including pass-by-value/-reference recommendations), resource management (including smart pointers and move semantics), details on compiling and linking on Linux, program organization with headers and modules, as well as build-time computations. Regarding software design, topics include object-oriented programming (including class operators, inheritance, a couple of design patterns, and virtual functions), function and class templates, C++20 concepts, as well as an overview of the standard template library (including containers, algorithms, and the ranges library). Specifically regarding performance, the course covers the roofline model, several optimization techniques, and vectorization. The lecture concludes with an overview of essential legacy features often encountered in scientific code, as well as an overview of important features available or coming up in newer C++ versions. Following recommendations by the ISO C++ Group on Education[12], we use modern C++ features from the beginning, instead of following the evolution path of the language (often referred to as “C/C++”).

Tools that the students experience include a compiler (g++), a debugger (gdb), a version con-

[9] Advanced Programming module description, WS22: https://go.tum.de/841039

[10] TUM Moodle page for Advanced Programming: https://go.tum.de/897784 (available via guest access)

[11] “Modern C++ Programming Course”: https://github.com/federico-busato/Modern-CPP-Programming

[12] Christopher Di Bella – “SG20 Education and Recommended Videos for Teaching C++”: https://blog.cjdb.xyz/sg20-and-videos.html and Guidelines for Teaching C++: https://cplusplus.github.io/SG20/latest/

trol system (Git) and a collaboration platform (GitLab), a build system (Make) and a build configuration system (CMake), tools for understanding binary files (such as nm, readelf, ldd, libtree), a code formatting tool (clang-format), a static analysis tool (clang-tidy), an address sanitizer (asan), a runtime profiler (gprof), a memory inspection tool (pahole), unit testing frameworks (doctest and Catch2), and an integrated development environment (VSCode). Students also learn the importance of using libraries, doing first steps with the linear algebra library Eigen, and get a first experience with code navigation via Sourcetrail and Doxygen. These are introduced in the tutorial sessions, which refer to starting guides for each[13].

3.2 Lecture and interaction opportunities

The lecture starts a new week in the course, as the following tutorial and project deliverables build upon the content discussed in the lecture. Lectures are frontal and slides-based, but with several opportunities for interaction, including an opening quiz, a collaborative summary, occasional demonstrations with student volunteers, and several interactive code examples. Recordings are openly available on the TUM Live platform[14], where the lectures are streamed and the remote audience can ask questions.

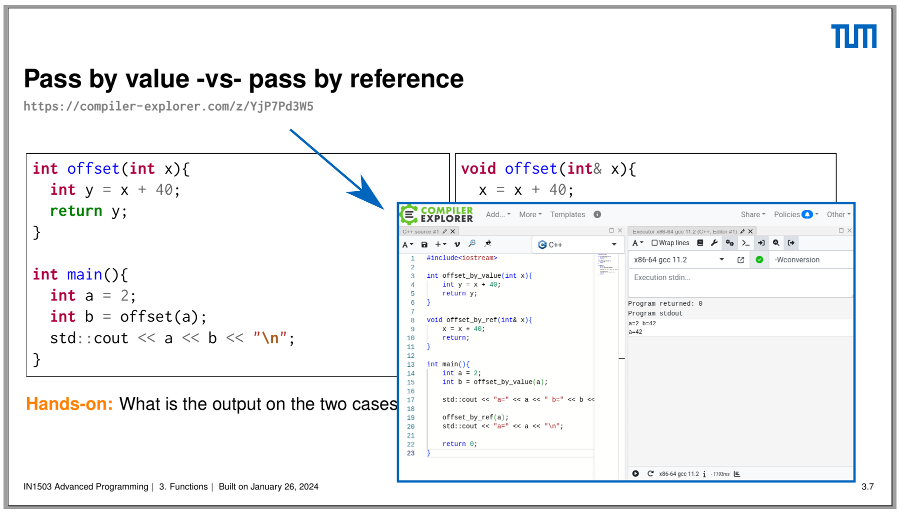
The most engaging of these add-ons has been the opening quiz, with the students requesting more of them since their introduction. While the lecturer prepares the room, students see a multiple-choice question based on content from this new lesson, which they discuss with their peers. The question triggers the interest of the students on the current topic, who can anonymously vote regarding the correct answer. Later on, and after having introduced the necessary concepts, students see how the class answered, and discuss the solution.

Every week corresponds to a stand-alone set of slides, covering one topic, but slides across weeks are consistently designed and follow similar examples, to facilitate creating relations between topics. The slide numbering scheme includes the week number, so that students can clearly refer to slides in questions, at any time. Every slide set starts with a list of intended learning outcomes, following the Bloom's taxonomy[15], to set the expectations and guide the students towards the exam. For each 90min lecture, we restrict the slides to approximately 30, and each slide contains mainly slimmed-down code examples and figures. Additional slides, available only in the distributed handout, explain side topics in more detail. The handouts are distributed one week before the lecture, allowing students to prepare or take notes during the lecture.

Even though designed for printing and to be used as reference material, the slides are also interactive. Every slide with code includes a URL to either the online execution and compiler exploration platform Compiler Explorer[16], to the analysis toolkit C++ Insights[17], or to the benchmarking toolkit Quickbench[18], as shown in Figure 2. All three are developed by the C++ community and frequently used in conference talks. References to such interactive code examples allow students to see and examine evidence for the behavior and details discussed, while they

[13] Student Starter Clues: https://gitlab.lrz.de/tum-i05/public/studentstarterclues/-/tree/master/tools (requires login)
[14] Lecture recordings: https://live.rbg.tum.de/?year=2023&term=W&slug=AdvProg&view=3
[15] Bloom's taxonomy on Wikipedia: https://w.wiki/6AK6
[16] Compiler explorer: https://compiler-explorer.com/
[17] C++ Insights: https://cppinsights.io/
[18] Quickbench: https://quick-bench.com/

Figure 2: An example slide of the course Advanced Programming. Slides mostly contain code examples and figures. Slides with code examples include links to online tools such as the Compiler Explorer, allowing students to easily and interactively study the code.



Pass by value -vs- pass by reference

https://compiler-explorer.com/z/YjP7Pd3W5

```
int offset(int x){
  int y = x + 40;
  return y;
}

int main(){
  int a = 2;
  int b = offset(a);
  std::cout << a << b << "\n";
}
```

```
void offset(int& x){
  x = x + 40;
```

```
#include<iostream>

int offset_by_value(int x){
    int y = x + 40;
    return y;
}

void offset_by_ref(int& x){
    x = x + 40;
    return;
}

int main(){
    int a = 2;
    int b = offset_by_value(a);

    std::cout << "a=" << a << " b=" << b <<

    offset_by_ref(a);
    std::cout << "a=" << a << "\n";

    return 0;
}
```

```
Program returned: 0
Program stdout
a=2 b=42
a=42
```

Hands-on: What is the output on the two cases

IN1503 Advanced Programming | 3. Functions | Built on January 26, 2024   3.7

allow quick exploration during in-class questions, and clear communication in the course forum.

Besides code and figures, slides include references to technical, often peer-reviewed material, including technical documentation and the C++ Core Guidelines[19], a collection of good practices with detailed explanations, edited by members of the ISO C++ Committee.

During the recent pandemic, the frontal lecture was temporarily replaced with a combination of pre-recorded lectures and online Q&A sessions. In that format, we observed very regular and active attendance to the Q&A sessions, in particular with written questions, which were significantly better targeted than in the case of the frontal lecture. We acknowledge the potential of the flipped classroom concept to deepen the understanding of the students, independent of the digital or presence format, and we recommend considering it when designing similar courses.

3.3 Tutorial and supervised teaching

The tutorial complements the lecture with structured exercises, provided in a worksheet. The worksheet includes a preparation checklist (with actions such as "read these code skeleton files and answer these questions"), occasional presentations about tools and additional concepts, and exercises solved in class. Students participate either in presence, or following a livestream and asking questions in writing. The exercises and their solutions are presented by second year students of the BGCE program[20], as an elective soft-skills module of their program, under the supervision and with the help and feedback of the instructors. The solutions are made available online at the end of the tutorial, but students still actively attend in order to solve the exercises in-class and to follow the discussion regarding the details, even if the pace is still too fast for a significant number of students.

[19] C++ Core Guidelines: https://isocpp.github.io/CppCoreGuidelines/
[20] Bavarian Graduate School of Computational Engineering: https://www.bgce.de/

A common issue is providing exercises that can demonstrate larger-scale challenges, without the students having to first study a new large code skeleton every week, before approaching each exercise. To solve this issue, and to give more time for discussion, we restricted the number of exercises to 1-2 per week, designing them to build up on exercises of previous weeks. We received positive feedback on this, even though students still wanted to have more opportunities to practice, which we addressed by adding optional extension ideas to the exercises, introducing the optional project, and distributing additional exercises with their solutions.

Each exercise relies on several of the tools listed in Subsection 3.1, and some exercises are designed to be fully or partially solved in pairs. While this worked relatively well in person, we have not managed to sufficiently engage online participants. We currently distribute the skeleton and solution of each exercise as archives on the learning management system, but we aim to fully distribute skeletons and solutions via the available GitLab collaboration platform, to emulate realistic and collaborative development practices early on. With this, we also hope to engage students in improving the material. The first worksheet already focuses on familiarizing with such infrastructure.

All the code skeleton archives include a `README.md` file with instructions to build the code, while later in the semester, all code skeletons include a `Makefile` or a `CMakeLists.txt`. Some codes include simple unit tests, with an expansion in this front being a clear next step, to provide further examples and normalize the practice of unit testing. All code files are formatted with the same clang-format style configuration, and developed with continuous checks of all compiler warnings and several further checks enabled, as discussed in Subsection 3.6.

Every worksheet ends with a curated list of related C++ Core Guidelines and an epilogue relating the exercises to real-world examples.

3.4 Coding project and peer review

Lectures and tutorials help comprehend and analyze new concepts, but achieving higher levels in Bloom's taxonomy and preparing students for realistic challenges requires practical experience with less concrete projects and teamwork, as is often the case in academic and industrial software development. A student cannot just see the creative process, they need to experience it themselves, and they see the bigger picture by transitioning from small-scale coding to larger-scale software engineering. While this is typically the goal of practical courses and a thesis, this optional project component allows students to make a first attempt towards a larger project, giving them the freedom to experiment, in a low-stakes environment. The project is completely optional, and students that complete it only get a small increase in their exam grade. The formal benefit is clearly lower than the experienced one, and it often does not correspond to the effort invested. Still, this is an opportunity taken by a large number of students. In the latest iteration, 120 students completed all parts of the project (50% of the students that attended the endterm or retake exam).

Students must work on the project as a group of two. This group size allows students to work in a pair programming fashion, and provides them a study partner. This is particularly important given that many are international students in their first semester. Larger group sizes would require larger projects (which is not the intention at this point), and would lead to more difficulties in group forming.

Every group chooses a topic from a pool of short proposals contributed by previous or current students, and multiple groups can work on the same proposal. Each project proposal includes three implementation phases following the schedule of the course, and each phase includes a definition of “done”, but intentionally without concrete implementation details. New proposals are accepted and reviewed in the beginning of the course, after which point they are available as options to everyone. We observed increased feeling of ownership and effort in cases of students implementing their own idea. Alternatively, a group or an individual student can contribute to an existing open-source project, after arrangement. This is an option that has not seen significant interest after two years of being offered. To simplify and clarify the procedure, we consider restricting the students to specific options, different every year.

While the idea of a coding project component is rather common, the implementation raises several questions and challenges, especially for a large audience. In particular, such a project requires scalable approaches to (a) reviewing (students receive feedback), (b) controlling (transparently checking who should get the grade bonus and who not), (c) communicating the status and progress, and (d) extracting grades for the final grade calculation.

Key components of our approach are peer-reviews of code on GitLab, and a central overview of submissions and grades on Moodle. Peer review not only reduces the effort from the perspective of the instructors, but also exposes students to other code, putting their work into perspective compared to the class, and diffusing knowledge. In the latest iteration of this course, students create a repository with their group name in a shared GitLab group, allowing read access to everyone in the class. Working directly on GitLab emulates the common development and reviewing workflows, while the transparent read access enables peer-reviewing without further actions from the reviewees, and enables the whole class to control plagiarism.

The most complicated implementation component is the submission workflow. In the following, we give a summary, while details are available on the project description page[21]. By the first deadline, students submit a link to a release merge request in their own repository. This is reviewed by the instructors and assistants, focusing on coding patterns and individual issues, and requiring approximately 30min per group. We do this to set an example of a code review and to give students an affirmation that their work is visible and checked. For the following two submissions, we use the Moodle Workshop activity[22]. After the submission deadline, reviewers are automatically matched to submissions to review, via random allocation[23]. They give a grade to a set of criteria, and they provide specific comments as code review on GitLab. In the (blind) review on Moodle, students can raise flags related to plagiarism, in which case they can skip a code review that would reveal their identity. To assign grades for the code reviews semi-automatically, reviewers then submit links to their GitLab reviews in a further Moodle activity. Every student has to review two submissions, leading to every group submission receiving feedback by in total four students. Since a new matching is computed for every project phase (versions 2 and 3), we work around the issue of groups dropping out between project phases, and we maximize the exposure of students to code.

[21] Project description page on Moodle: https://go.tum.de/477181 (available via guest access)

[22] Moodle workshop activity: https://docs.moodle.org/401/en/Workshop_activity

[23] It is possible that the random allocation results into reviewing the code of your own group, since every student submits individually. We allow students to review their own code in that case, but Moodle gives us the possibility to override the automatic allocation.

Since every submission and review step corresponds to a deliverable on Moodle, students get individual grades for each. Besides the first submission, and a quick check of the code reviews, all other grades are determined without the need of the instructor to intervene. The Moodle gradebook gives continuous feedback to the students, and a continuous overview of who is still working on the project. Special cases can be handled directly on the gradebook. While a student needs to complete all steps to receive a passing grade, having individual submission steps allows to define a grading scheme that forgives missing individual parts. At the end of the semester, grades can easily be exported to a portable CSV format for further processing in other tools.

It is crystal-clear that this is a complicated workflow. It is, however, the result of already five iterations, and every step addresses previously observed issues. For example, we previously announced a pre-defined review matching and asked students to submit a report with all their activities at the end of the semester. However, groups dropping out between project phases, and the low integration of these reports with Moodle introduced significant manual handling overhead. In our view, further simplifications to the system are only possible by reducing control, on an honor code basis, as discussed in Subsection 3.8.

3.5 Assessment

Having defined intended learning outcomes in every topic of the course, we can design exams that align with expectations set in the beginning of each lecture. However, many of these learning outcomes are challenging to assess in a written exam. The current exam format consists of multiple-choice questions, implementation exercises, and a few reasoning questions. Implementation exercises of one exam follow the same general theme (e.g., implementation of a galaxy simulation), so that students do not switch context when moving between exercises, and they mostly assess the competence of applying concepts given specific instructions. We address the analysis and evaluation levels with open-ended, free text questions. We use multiple-choice questions mostly to assess knowledge and comprehension of tools and concepts not covered by other exercises, while we avoid questions on issues that a compiler would report. Exams are designed, graded, and reviewed using the TUMExam framework and infrastructure[24].

Besides the optional project reviews and a quiz covering the first part of the course, no other forms of formative assessment are offered. Students have consistently requested automatically graded coding exercises, which we consider implementing using the Artemis platform[25].

3.6 Infrastructure

Maintaining a course with all these different components (including lecture slides, tutorial worksheets and code, and project submission workflows) is challenging without supportive infrastructure. For this, we make good use of several resources available at TUM. All the material is continuously developed and versioned on the LRZ GitLab, and we document ideas and decisions on issues labeled by course component and topic. All slides and worksheets are based on LaTeX and built with Make, allowing to easily build individual parts of the material. The continuous integration system automatically generates all the slides, worksheets, and code archives to be

[24] TUMExam: https://tumexam.de/, developed by TUM and ExatoExam: https://www.exato-exam.de/
[25] Artemis: https://docs.artemis.cit.tum.de/

uploaded on the learning management system. Additionally, it checks all the code skeleton and solution files for correct building (with warnings treated as errors and sanitizers enabled) and formatting. At the end of each year, a release archive is created.

The learning management system (Moodle) features an introduction video, which serves as a course advertisement and partially replaces the organizational information typically given in the first lecture. This allows students to revisit this information later, and allows the first lecture to dive into the interesting content faster. The course page also includes a feedback form per tutorial, a forum and a Moodleoverflow[26] for communication, a glossary of abbreviations and other terms, documentation for the various tools used in the course, and more material mainly as impulses for going deeper into each topic.

3.7 Reception and impact on learning

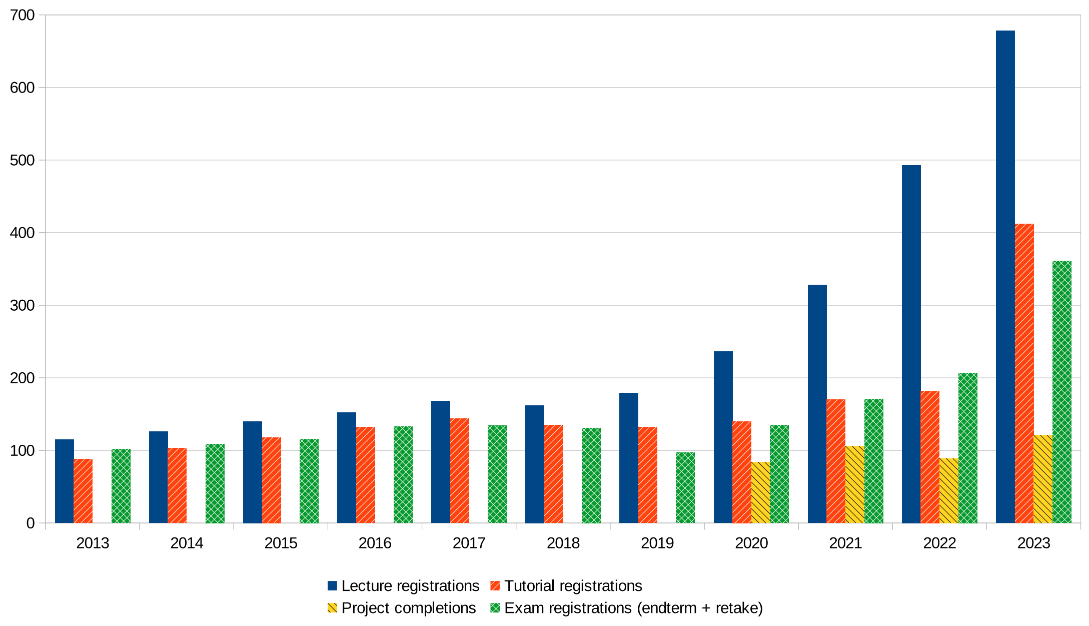
As shown in Figure 3, the course historically had under 200 students registering for the lecture, until 2020, when material changed. Compared to 2018 (when changes to several aspects of the course started), the registration numbers have more than quadrupled, and exam registrations have almost tripled. In the latest iteration (winter semester 2023/24, shown here as 2023), 678 students registered for the lecture, 412 registered for the tutorial, 121 completed the project, 176 attended the endterm exam (out of 250 registered for it), and 70 attended the retake exam (out of 111 registered). Table 2 shows where the students are coming from: in 2018, the vast majority of the students were from the M.Sc. CSE (for which the course is obligatory). In 2023, CSE accounted for less than 10% of the total registrations. While the course is offered by the School of Computation, Information and Technology, a rapidly increasing number of students now studies a program from the School of Engineering and Design. The new curricula have also gradually appeared in the population (starting with a couple of students each), hinting that students do recommend this elective to others in their environment. Interesting is also the number of informatics students, which also has increased, despite having other similar courses as options. Overall, these numbers demonstrate the demand for programming and software engineering courses in modern engineering curricula.

Independent of the popularity, is this a good course? Is it too difficult, too easy, or at the right level? While the name of the module can be understood differently by students of different backgrounds, the course does require significant previous contact time with programming, and it is designed for graduates of a STEM bachelor's program, but it is not meant to be integrated in a Computer Science graduate degree. The mid-term course evaluations consistently show that the content is at the right level, with slight deviations towards both sides of the difficulty range (Table 3). Important here is the distinction between difficulty and learning. According to the evaluation results, the content is generally new (but could be further improved), and it is delivered at the right pace and difficulty. Given that the concepts are also taught in a systems language, where students need to consider memory access, resource management, and performance implications, this is correctly considered to be an advanced programming course.

While we have received direct positive feedback from students long after the completion of the course, we do not have sufficient data to quantify the impact of our actions on the learning

[26] Moodleoverflow plugin: https://moodle.org/plugins/mod_moodleoverflow

Figure 3: Evolution of registration numbers in the lecture, tutorial, and exam, and number of students that completed the optional project. The material was redesigned in 2020.



outcome of the students. In Table 4, we provide statistics on the endterm exam, which show mostly a stable trend, with consistently lower failure rate than before. However, multiple factors are underlying: different people have been involved in the teaching and design, correction, and grading of the exam, the exam format has evolved, while the background of the audience has also changed. For a more accurate quantitative evaluation of similar efforts, we recommend an independently designed and standardized assessment, with a controlled sample of students. During the semester, this can also be monitored with formative assessments (e.g., quizzes on Moodle), which we have only implemented in a limited extent and mainly targetting self-assessment.

3.8 Scaling

While the course was originally designed for a rather homogeneous audience of 50-100 students, recent changes in the material have increased the numbers and range of previous knowledge significantly, as discussed in Subsection 3.7. Not all components of the course fit the increasing number of students equally well, while the diffusion in the level of the audience increases the challenge of helping the students in need without depriving the originally intended audience of their promised learning outcomes.

The frontal lecture scales relatively well. Students seem to ask questions even when the lecture room is densely populated, while the number of questions does not seem to create time management issues. By setting goals in the beginning of the lecture (“let’s get at least ten questions today”, with checkboxes on the blackboard) and reinforcing the questions with various treats, this behavior seems to be well-regulated. Questions after the lecture and special cases regarding

Table 2: Number of students (lecture registrations) per study program for the course Advanced Programming. Every year shows the top five programs, and programs that were included in the top five in other years. Rows are sorted by value in 2023. The course is a required module for the Computational Scinece & Engineering, and elective for everyone else.

Study program	**2018**	**2019**	**2020**	**2021**	**2022**	**2023**
Mechatronics and Robotics	0	0	1	15	112	119
Robotics, Cognition, and Intelligence	2	1	9	15	35	104
Aerospace	0	1	2	16	34	49
Computational Science & Engineering	63	59	63	85	44	45
Informatics	21	19	14	11	17	29
Mathematics in Data Science	0	0	5	27	23	21
Biomedical Computing	32	26	31	43	36	14
Physics (applied and engineering Physics)	4	11	11	7	6	12
Other	40	62	100	109	186	285
Total	162	179	236	328	493	678

Table 3: Excerpt from the evaluation results (organized by the student council). Missing data is due to varying evaluation questionnaires between years. The course has gradually improved in all of these criteria since 2018. The material was redesigned in 2020.

Criterion	**2018**	**2019**	**2020**	**2021**	**2022**	**2023**	**Scale**
Logical sequence	1.9	1.7	1.3	1.4	1.3	1.2	1:Best, 5:Worst
Content is new	2	2.6	2.2	2.6	1.9	-	1:Best, 5:Worst
Relation to research	2.5	2.2	1.9	1.8	2.1	1.5	1:Best, 5:Worst
Difficulty of the content	2.6	2.5	2.8	3.0	2.7	2.9	3:Best, 1:Difficult
Pace of delivery	2.7	2.6	2.5	2.9	2.8	2.8	3:Best, 1:Fast
Lecture grade	2.1	2.6	1.5	-	1.3	1.3	1:Best, 5:Worst
Tutorial grade	2.2	2.0	1.6	1.6	1.6	-	1:Best, 5:Worst

Table 4: Endterm exam statistics over the past ten years. Grade scale: 1:Best, 5:Worst, 4:Minimum passing grade. Different people have been involved in the teaching and in the design of the exam over the years. The exams of 2020 and 2021 were shorter and online/open-book. The (passing) grade bonus option was introduced in 2020.

Quantity	**2014**	**2015**	**2016**	**2017**	**2018**	**2019**	**2020**	**2021**	**2022**	**2023**
Attended	76	73	67	89	82	81	84	118	113	176
Average grade	2.94	3.48	3.31	2.58	2.86	2.68	2.38	2.47	2.54	2.68
Average (pass)	2.47	2.78	2.45	2.17	2.46	2.36	2.10	2.32	2.20	2.30
Fail rate	21%	34%	36%	16%	17%	14%	11%	6%	12%	15%

the exam definitely increase the communication and the effort and the time invested by the instructors, for which clearer documentation and open communication channels help. The largest scaling challenges are caused by the tutorial and the project.

The tutorial is currently offered in two small groups, in small seminar rooms, each with a tutor and a supervisor. The group size is intentionally small, so that tutors can help students with questions and technical issues. A common scaling approach is offering more rooms and hiring more tutors, resources currently in shortage. Instead of increasing the resources invested (even if that meant integrating more instructors affiliated to study programs beyond CSE), we consider offering a central tutorial, in combination with more automation (automatically corrected exercises via Artemis) and strengthening the network of students. We could take advantage of the Moodleoverflow point system[27] as alternative source of grade bonus, in order to encourage such structured communication (gamification). Using students as resources should further increase the feedback that students get, without increasing the time invested by the instructors.

The most important factor that limits the scaling of the course is the project workflow. Peer review since the first submission, given predefined examples of code reviews, would remove the instructors from the loop. A "panic button" procedure could engage the instructor only in cases where this is necessary (e.g., students identifying issues with a submission), and assuming that submissions are complete by default, would allow to fully automate the submission and review procedure. However, this is only possible as long as only a small part of the grade is defined by such automation and peer review, due to restrictions by regulations.

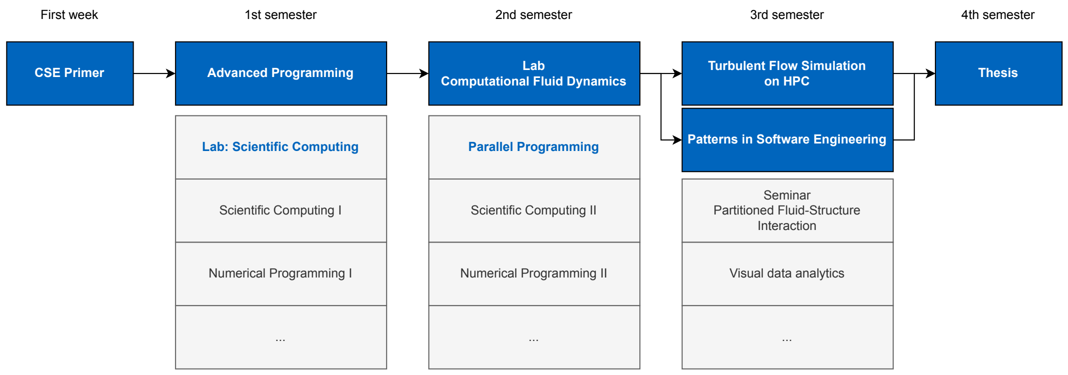
4 Diving deeper: A coherent curriculum track

After making a first contact to basic programming and teamwork skills in the CSE Primer module, and after studying deeper several software engineering concepts and trying them out in a low-stakes environment in the Advanced Programming module, students are ready to dive deeper into specific applications and work on larger team projects. They do that in practical courses, such as the Computational Fluid Dynamics Lab[28]. In that, students develop a larger simulation code over the first part of the course, before extending it further in their own direction in the second part. They also work in groups, and they also collaborate via Git on GitLab, developing C++ code. This time, however, the quality of the code directly influences the final grade of this 10 ECTS course. A similar setup is used in the course Turbulent Flow Simulation on HPC Systems[29], a course that naturally continues the CFD Lab and focuses more on software design.

Advanced Programming is helpful also in courses that do not use C++ as a working language. In the course Parallel Programming[30], which typically follows in the second semester, students are expected to be familiar with C programming. At the end of Advanced Programming, we include a section on legacy features and interoperability with C, while several of the tools introduced in Advanced Programming are also used there. Students that want to analyze and im-

[27] On Moodleoverflow, similarly to StackOverflow, students get points by writing good questions or helpful answers.
[28] CFD Lab module description: https://go.tum.de/568369
[29] Turbulent Flow Simulation on HPC systems module description: https://go.tum.de/351683
[30] Parallel Programming module description: https://go.tum.de/973494

Figure 4: Multiple courses work together to teach students software engineering skills for developing simulation software. A student starts with the very basics in the CSE Primer, learns individual concepts and skills in Advanced Programming, develops a larger project in the CFD Lab, and specializes further (towards applications and/or software engineering) in further courses and their thesis.



prove the design of their codes can continue with the course Patterns in Software Engineering[31], which uses Java for implementing the discussed concepts. While earlier courses invest more time in individual code development details, they also introduce students to broader topics of software engineering (including software requirements, software design, testing, maintenance, and more [BF14]), which gradually become relevant in later courses, where students need to work in teams and manage the complete lifecycle of gradually larger projects.

Looking at the bigger picture, and after iterations of cross-integration, these courses work together as a coherent curriculum track. An engineering graduate that wants to specialize in computational fluid dynamics can follow the courses CSE Primer, Advanced Programming, CFD Lab, Turbulent Flows on HPC and/or Patterns in Software Engineering (as shown in Figure 4), without expecting any surprises, and at the end being perfectly ready to contribute to academic or industrial simulation codes in their thesis or later career. This implicit curriculum track is embedded into the CSE curriculum, which also offers several opportunities to specialize in application courses from computational {structural mechanics, fluid dynamics, biosciences, physics, chemistry, electronics}, and more[32].

[31] Patterns in Software Engineering module description: https://go.tum.de/453490
[32] Modules: https://www.cit.tum.de/cit/studium/studiengaenge/master-computational-science-engineering/modules/

5 Conclusion and future work

The M.Sc. Computational Science and Engineering at the Technical University of Munich offers a unique emphasis on software engineering, preparing its graduates for simulation software engineering roles. This emphasis is now stronger than ever with the updated “Advanced Programming” module, the introduction of the “CSE Primer” onboarding module, and the integration with further courses in the CSE curriculum. The interactive formats, supported by technical training, established tools and references, collaborative project work, and facilitated interaction between students, has made these courses particularly helpful for students. The rapidly increasing popularity led us reconsider our design choices, leading to courses that scale better than before. In less than five years, lecture registrations have quadrupled, and exam registrations have almost tripled. Based on the latter, more than 300 students every year are now getting better prepared for contributing to the research software community, and onboarding students to our projects takes significantly less time and effort.

Looking back at the design decisions that helped us and were positively received by the students, we can conclude with some recommendations. Telling one story consistently, for longer parts of a course, helps students dive deeper into complex topics. A frontal lecture can be largely interactive, including easy to pursue impulses for further learning, such as tools running or analyzing code snippets, or curated resources. Actively seeking interaction keeps students engaged, and engagement brings more students every year. Introducing tools into lectures and exercises is also appreciated by students. Peer review can strongly increase the feedback that students receive, without significantly increasing the effort invested by the instructors, exposes them to more code, and puts their work in perspective. Last but not least, iteratively refining the material and workflows in a structured way, using the same tools and techniques as developing software, can be catalytic to long-term improvement.

These courses are now ready to be studied and criticized by a wider community, but we still see significant potential for further improving their usefulness and impact. In order to further improve the logical sequence and the relation to research, we can put further emphasis on building larger and better motivated examples during the semester. To further increase the technical skills acquired, we want to introduce more opportunities for guided debugging and automated testing. To strengthen the collaboration and networking between students, we can distribute all code via a collaboration platform and expand the collaboration cues in the exercises. To increase the feedback that students receive, we can introduce more formative assessment via automation, and we can apply gamification on the structured communication channels, building a community. To scale up the project participation, we can further replace instructor-driven steps by annotated examples and peer review. Finally, to better integrate with further courses, and to make these courses helpful for more students, further coordination with instructors of other courses and program coordinators would be a first step.

Acknowledgements: Several ideas described in this article have been developed in the context of or in other ways influenced by didactics courses organized by the TUM ProLehre. We thankfully acknowledge the funding by the Technical University of Munich, especially in the context of the TUM Ideas Competition 2019 and the project "Beyond CSE Primer: Onboarding for computational degree programs". We would like to thank Valentin Seitz for his assistance in that project. We would also like to thank the instructors of other courses that helped integrate them with this work, especially Benjamin Rodenberg for his work on the courses "CFD Lab"and "Turbulent Flow Simulation on HPC Systems". These courses have also been improved a lot by the continuous feedback of the supervised teaching tutors of the BGCE program, and the students. Finally, we would like to thank Benjamin Uekermann and Miriam Schulte for feedback during the review phase.

Bibliography

[BF14] P. Bourque, R. E. Fairley (eds.). *SWEBOK: guide to the software engineering body of knowledge, version 3*. IEEE Computer Society, Los Alamitos, CA, 2014.
http://www.swebok.org/

[BH13] W. Bangerth, T. Heister. What makes computational open source software libraries successful? *Computational Science & Discovery* 6(1):015010, Nov. 2013.
https://doi.org/10.1088/1749-4699/6/1/015010

[GAB+24] F. Goth, R. Alves, M. Braun, L. J. Castro, G. Chourdakis, S. Christ, J. Cohen, F. Erxleben, J.-N. Grad, M. Hagdorn, T. Hodges, G. Juckeland, D. Kempf, A.-L. Lamprecht, J. Linxweiler, F. Löffler, M. Martone, M. Schwarzmeier, H. Seibold, J. P. Thiele, H. von Waldow, S. Wittke. Foundational Competencies and Responsibilities of a Research Software Engineer. 2024.
https://doi.org/10.48550/arXiv.2311.11457

[HBC+22] S. Hettrick, R. Bast, S. Crouch, C. Wyatt, O. Philippe, A. Botzki, J. Carver, I. Cosden, F. D'Andrea, A. Dasgupta, W. Godoy, A. Gonzalez-Beltran, U. Hamster, S. Henwood, P. Holmvall, S. Janosch, T. Lestang, N. May, J. Philips, N. Poonawala-Lohani, P. Richmond, M. Sinha, F. Thiery, B. Werkhoven, Q. Zhang. International RSE Survey 2022. Aug. 2022.
https://doi.org/10.5281/zenodo.7015772

[Mal10] D. J. Malan. Reinventing CS50. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. SIGCSE '10, p. 152–156. Association for Computing Machinery, New York, NY, USA, 2010.
https://doi.org/10.1145/1734263.1734316

[WPP20] I. Wiese, I. Polato, G. Pinto. Naming the Pain in Developing Scientific Software. *IEEE Software* 37(4):75–82, July 2020.
https://doi.org/10.1109/MS.2019.2899838