Throw Away Student Software At Semester End? Better Not!

Jan H. Boockmann[1], Kerstin Jacob[1], Gerald Lüttgen[1]

Abstract: Software engineering is continuously evolving in order to meet the challenges faced by the ever growing complexity and longevity of digital systems. Students should thus acquire additional practical competencies wrt. software evolution and maintenance. This paper describes how student team projects at our Chair have been redesigned to meet this need, and reports on our first experiences. It also aims at initiating a discussion on the challenges we met, namely the contributions we can expect from students, the effort required from staff, and the individualization of grades.

Keywords: long-living software; project-based learning; software evolution and maintenance

1 Introduction

Today's, and even more so, tomorrow's computer scientists need sound skills in designing and implementing systems in such way that these are maintainable and can evolve over a long period of time. Especially important are competencies in reading and understanding software artifacts written by others, e.g., frameworks, source code, and documentation. As pointed out by Spinellis [Sp03], these competencies are increasingly relevant nowadays, because software products grow larger and become more complex, are developed collaboratively in large teams, and combine a variety of different technologies.

The problem of how to achieve long-living, maintainable systems is called a "grand challenge within software engineering" by Bennett; Rajlich [BR00] and remains an ongoing research topic, see, e.g., the DFG priority programme report by Goltz et al. [Go15]. As noted by Tate [Ta05], reconciling agile practices, which focus on feature-based development, and the goal of maintainable software is especially challenging and subject to ongoing research.

In the light of these circumstances, we observed that the topic of designing and implementing long-living software products had played only a subordinate role in the teaching at our Chair. Our student projects on software development spanned a single semester, development started from scratch, and the generated software was thrown away at semester end. Thus, students (and assessment) focussed on having software that works at the time of hand-in, and less so on writing maintainable code. Given the aim of teaching students the abilities to understand complex systems and write long-living software, we revised the teaching concept of our student projects two years ago. In our new project setting, students work in flexible teams to develop new projects and, even more importantly, to maintain and extend existing projects that have been initiated in previous semesters.

[1] University of Bamberg, Software Technologies Research Group, Germany, firstname.lastname@swt-bamberg.de

doi:10.18420/SEUH2022_03

The remainder of this paper is structured as follows: Sect. 2 briefly introduces our new teaching concept for software development projects, Sect. 3 summarizes the lessons learned by us, and Sect. 4 proposes questions regarding our teaching concept which might deserve a wider discussion within the community of lecturers in software engineering.

2 Revised Teaching Concept

At our Chair, the teaching of software engineering principles at Bachelor level had spanned two compulsory modules. Usually, Bachelor students in their third semester of study will take a 6 ECTS module called "FSE" that introduces the foundations of software engineering, focussing on requirements engineering, software modelling and design, and quality assurance. The lectures are accompanied by traditional practicals as well as tutorials that introduce students to popular development tools such as *JUnit* and *Cucumber* for unit and acceptance testing, resp. In the following semester, students continue with a 6 ECTS "LAB" project module in which they apply the knowledge gained in FSE to develop in teams a small Java application following a variant of *SCRUM*. The module focusses on the application of agile practices and team work. Students also familiarize themselves with the concepts of object-relational mappers and the *JavaFX* GUI framework. The project tasks are provided by industry partners who function as a customer for the student teams.

However, teaching students the need for documentation, including abstractions such as UML diagrams, is a challenging task, because the simple systems typically discussed in lectures largely do not require documentation for understanding; unfortunately, presenting a complex system, where documentation is necessary, exceeds a semester's time frame. In many classical team project modules, this problem persists to some degree, because students rarely have to use documentation written by others. Another issue is that students are closely guided by lecturers regarding agile processes, which is necessary to successfully conduct a first team project, but actually contradicts agile principles.

New concept. We now continue the LAB with a second, *elective* team project module (again 6 ECTS), called the "HUB", where students operate in flexible teams to start new projects from scratch and maintain software products developed in previous semesters. Also taught in collaboration with industry partners, HUB puts a stronger focus on programming practices and develops web-based products based on the *Spring Boot* and *Angular* frameworks. Note that this allows students to fully reuse the business logic implemented in the LAB for a continuation in the HUB module. With the addition of HUB, the technology stack of LAB was revised to also include *Spring Boot*, albeit not in a client-server setting. This turned out to be a good decision, because students now have to use the architecture and design patterns stipulated by the framework, and thus avoid typical mistakes and save valuable time.

Projects currently running in the HUB include, for example, a build monitor that calculates and visualizes software metrics obtained during automatic software builds, and a library

system for administrating the Chair's literature collection ("Handapparat"). Students are required to work on two or three projects at a time, which are however all based on the same technology stack sketched above. For each project, one student acts as the product owner, while the other students typically specialize on one or two technologies.

We believe that the HUB is suitable for teaching the competencies required to develop long-living software. The module trains the ability to read and comprehend complex foreign source code, as students need to become familiar with the software products built in previous semesters to be able to fix bugs or design flaws, to replace libraries and frameworks by current versions, and to implement new features. By having to deal with the sometimes messy source code and frequently incomplete and inconsistent documentation of software products of prior student generations, students experience first-hand the value of clean code and the importance of thorough documentation. This motivates them to strive for high-quality artifacts and to apply, e.g., modern code review.

Successes and new challenges. The combination of the FSE module and the LAB project has been running for approximately ten years at our Chair. The additional HUB project has now been offered for two years with positive student feedback. Students enjoy the development of web-based applications (in particular, learning *Angular*), the freedom of self-organization (e.g., specializing on a certain technology), and the interactions with industry partners (giving students insights into professional work practices). However, from a teaching perspective, the addition of the HUB has posed new challenges:

(1) It is difficult to estimate how much we can expect from students in our project setting. The applied technologies are not trivial to learn, and the quality and complexity of existing projects differ. We cannot easily draw from past experiences, neither ours nor others, because the HUB differs from a classical team project and focuses on different competencies.

(2) The time required for teaching the HUB is higher than for the LAB. To answer the multitude of questions raised by students, instructors need detailed knowledge about software requirements that differ among the concurrently developed software products, and about many details of the comprehensive technology stack employed. The necessary teaching resources thus depend on the student cohort and the own experience of the lecturing staff with the employed technologies, and are unevenly distributed throughout a semester.

(3) Individualizing student grades for the HUB module is challenging when compared to the SWL module, where students collaborate in fixed teams and with close supervision by staff. In the HUB, students operate in flexible teams and collaboratively work on tasks using techniques such as pair programming. Additionally, because the existing projects and applied technologies are plentiful, students need to specialize on different topics during the semester. Not all of their contributions to the projects are directly visible in terms of added or changed functionality, but also include refactoring source code, adding automated tests, communicating with stakeholders, or enhancing documentation.

3 Lessons Learned

In response to these challenges, we have adjusted the HUB module throughout the past three semesters. The remainder of this section outlines the most important lessons learned by us along the way. However, certain issues remain, which are discussed in Sect. 4.

Online-tutorials are beneficial. Due to the COVID-19 pandemic, we have shifted from face-to-face tutorials to online material, which introduces tools and techniques for practical software development and consists of video recordings, slides, and hands-on tasks. Sample tutorials focus on *"Managing the Development Process using GitLab"*, *"Persisting Data with JPA using Spring Boot"*, and *"Writing Clean Code and Assuring Quality with Code Reviews"*. Providing asynchronous online material gives the students the option to (re-)watch the tutorials at their own pace when facing the challenges addressed in the tutorials, without overwhelming them with too much content at the beginning of the semester.

A good demo project is key for well-structured projects. To ensure that the projects can be maintained by students, all projects should have a similar structure and employ similar technologies. We have learned that providing a demo project is more effective than stipulating a detailed coding convention, and have introduced two demo projects that showcase the technology stacks for the LAB and HUB modules, resp. These projects serve as a skeleton for products developed from scratch, which leads to a similar structure across all projects and minimizes the initial overhead. Previously, students spent the semester start on setting up their own project according to the coding convention, which usually required additional support and clarification by teaching staff. Our demo projects also include a CI/CD pipeline, which checks for proper formatting, builds the projects, runs automated tests, generates metrics such as code coverage, and deploys the application.

Keep it relatively simple wrt. technologies. The provision of tutorial videos and demo projects aids students in acquiring the skills for mastering today's software development technologies. While we do not forbid students to use additional external libraries, frameworks, and platforms, technology proposals now have to be discussed with the industry partner and the Chair. This discussion primarily focuses on the effect of the inclusion of the requested technology on maintainability. Before, for example, a student already familiar with the *ElasticSearch* engine proposed to use it for one product, which turned out to be too complex for other students and had to be replaced by a simpler search technology in a later semester.

Expect less functionality than in classical student projects. Compared to a project structure in which a new project is developed from scratch and thrown away at semester end, we expect less functionality to be implemented by students in the HUB. When adding

features to a new or existing software project, students need more time for planning and implementation, as they need to consider clean code and provide documentation early on. When extending existing projects, students need time to understand the codebase and product requirements. This is particularly true if the codebase contains source code of poor quality, incomplete documentation, or technologies unfamiliar to the students. For maintaining projects, students need to invest time in tasks that do not result in new functionalities, but still resemble a significant contribution. These activities include, but are not limited to refactoring source code, fixing bugs, adding automated tests, and enhancing documentation.

A journal written by each student helps to individualize grades. Determining the grade of an individual student in our HUB project requires multiple forms of assessment and is based on the student's contribution to the codebase, including documentation. For each software product developed or maintained in the HUB, students collectively submit a report that summarizes the initial state of the product, the contributions made throughout the semester, and directions for future enhancements. To be able to map contributions to individual students, each student has to document each sprint in a journal, including the time spent on each topic they worked on, and a reflection on their learnings. This information is cross-checked in an oral exam, in which individual students answer questions on design and implementation choices, quality assurance, project management, and teamwork. While this grading is time-intensive (about 5–7 person days), it is perceived as fair by students and staff alike. Each product report is also a valuable first source of information for future student generations who need to maintain the product.

Answering questions on low-level implementation details is time consuming. In past semesters, most of the teaching staff's effort in the LAB and HUB modules, and partially also the effort of our industry partners, has been spent on answering student questions concerning implementation-specific details, not at least because of the time-consuming task of inspecting the codebase in question. This winter semester 2021/22, we offer a help desk organized by a student assistant for answering those types of question. In doing so, we seek to enable our industry partners and us lecturers to focus on design-level issues and software quality aspects, which are critical for software longevity.

4 Conclusions

Despite the challenges identified in our teaching concept, we have established our current student project structure because we believe that being able to understand complex systems and writing long-living software is highly relevant when tackling current and future IT projects. After two years of experience with the HUB and based on the positive feedback from students, we have come to the conclusion that our teaching concept is suitable for making students aware of the importance of sustainability in software development and motivating them to write clean code and good documentation.

Remaining challenges. Certain challenges remain, however, which ought to be discussed by the community of lecturers involved in software engineering education. Guiding questions for discussion may be the following:

(1) Should competencies of understanding complex systems and writing long-living software be taught at Bachelor or Master level, or both (see, e.g., Zukunft [Zu16])?

(2) Is giving students practical experience with software maintenance enough, or do we need to accompany project work with additional theoretic input (see, e.g., the module "Software Evolution" taught at Univ. Heidelberg[1])?

(3) How do we grade contributions to the codebase, documentation, and quality assurance in the context of maintaining long-living software? How could a grading scheme be devised that accounts for these different types of contributions? What alternative approaches to our journal can be adopted to simplify the individualization of grades?

(4) What changes are necessary for the teaching concept to scale to large student cohorts? (So far, we have run the HUB module with up to 25 students in a single semester.)

(5) What other approaches exist for teaching the competencies needed for developing long-living software? (One approach is for students to contribute to open source projects, as noted by Spinellis [Sp21].)

Acknowledgements. We thank the reviewers and our colleague Eugene Yip for their suggestions which have helped us to improve the paper.

References

[BR00] Bennett, K. H.; Rajlich, V.: Software maintenance and evolution: A roadmap. In: ICSE 2000. ACM, pp. 73–87, 2000.

[Go15] Goltz, U.; Reussner, R. H.; Goedicke, M.; Hasselbring, W.; Märtin, L.; Vogel-Heuser, B.: Design for future: Managed software evolution. Comput. Sci. Res. Dev. 30/3-4, pp. 321–331, 2015.

[Sp03] Spinellis, D.: Reading, Writing, and Code. ACM Queue 1/7, pp. 84–89, 2003.

[Sp21] Spinellis, D.: Why computing students should contribute to open source software projects. Commun. ACM 64/7, pp. 36–38, 2021.

[Ta05] Tate, K.: Sustainable Software Development: An Agile Perspective. Addison-Wesley Professional, 2005, ISBN: 0321286081.

[Zu16] Zukunft, O.: Empfehlungen für Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen, 2016, URL: https://dl.gi.de/handle/20.500.12116/2351.

[1] https://se.ifi.uni-heidelberg.de/teaching/previous_semesters/ws_202021/softwareevolution.html (accessed 2021/12/08)