Teaching Programming at Scale

Angelika Kaplan, Jan Keim, Yves R. Schneider, Maximilian Walter, Dominik Werle, Anne Koziolek, Ralf Reussner
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
{angelika.kaplan, jan.keim, yves.schneider, maximilian.walter, dominik.werle, koziolek, reussner}@kit.edu

***Abstract*—Teaching programming is a difficult task and there are many different challenges teachers face. Beyond considerations about the right choice of teaching content and presentation in lectures, scaling practical parts of courses and the examination and grading to course sizes of around 1,000 students is particularly challenging. We believe programming is a skill that needs to be trained practically, which creates additional challenges, especially at this scale. In this paper, we outline learning goals for our undergraduate programming course and the structure for the course we derived from these goals. We report on the challenges we see when teaching programming at scale and how we try to overcome them. For example, one central challenge is how to grade a high number of students in a good, transparent, and efficient way. We report on our approach that includes automated tests as well as tool support for manual code review. Over the years, we experienced different issues and learned valuable lessons. We present corresponding key takeaways that we derived from our experiences.**

***Index Terms*—Programming, Object Oriented Programming, Software testing, Teaching, Computer aided analysis**

I. Introduction

Teaching programming is a challenging task. Different programming concepts need to be explained in an appropriate way for students to grasp them, their general application, and their concrete realization in a chosen programming language. In universities and schools, usually, the additional task of grading comes up. Grading is essential in our education systems and is equally challenging. At the Karlsruhe Institute of Technology (KIT), we teach object-oriented programming to undergraduates in their first semester using Java with different learning objectives stating that students:

- know the basic structures and details of the Java programming language, foremost control structures, simple data structures and how to handle objects,
- can implement non-trivial algorithms and can apply basic principles of programming and software engineering, and
- are able to create executable Java programs of medium size that withstand automated quality assurance including automated tests and enforcement of code conventions.

Our course follows the *learning-by-doing* principle for teaching programming which emphasizes the role of programming as skill that requires practical training. This view is also supported by research on programming education [1]–[3]. This is why we employ practical programming assignments instead of a hand-written exam or similar examination techniques. Therefore, one of our learning objectives is that students should be able to write 500 to 1,000 lines of code based on a complex and precise specification.

Based on our learning objectives, we have three major goals for the assessment of the students' solutions. First, the correctness of the program is important to us. Second, we want the students to program in a good and clean object-oriented manner. Counter-examples for that include god classes (programs with all functionality and logic in one class, cf. [4, p. 136ff]), high coupling between classes as well as low cohesion. Third, students need to submit self-made programs for assignments. No code written by another student or person is allowed in the submitted programs, including any kind of code-copying from others or similar kinds of plagiarism.

For smaller course sizes, achieving these goals is challenging and already requires some effort. However, the number of students joining our programming course increased by roughly 45% over the last five years. At present, we have about 1,000 students attending the lectures and almost all of them participate in the practical exercises. Around 500 students take part in the exam. When scaling the challenges up to this number of students, additional challenges arise, such as assessing and grading in an efficient, good, transparent, and fair way. These properties rise from the following factors: We have only limited time and personnel for grading, therefore we have to grade efficiently. However, we want to grade in a good way which means that students can learn and improve from their mistakes. Therefore, the whole grading process should be clear and transparent for students to understand. In addition, the grading should be fair in a way that submissions with similar quality get similar grades. Moreover, we have to make sure that students are submitting their own solutions without cheating.

In the following, we present our approach and the efforts to tackle the different challenges and share our experiences.

II. Course Structure

Our programming course consists of two parts: 1) a lecture that teaches the theoretical knowledge for Java development and 2) a set of practical exercises. The practical part consists of five exercise sheets and weekly tutorials, which are held by student teaching assistants. The tutorials are held in smaller groups with about 25 students, teach the practical application of concepts from the lecture, and present the solution to the exercise sheets. Our student teaching assistants rate the submitted exercise sheets regarding functional properties and coding style. The course participants have to submit their

solutions using the *Praktomat* system, a submission system for programming assignments, digitally. The structure of our programming course is partly based on previous programming courses at KIT like the structure described in [5], [6]. As previously stated, we do not have group submissions but have a strong emphasis on individual submissions. At the end of the semester, the course is concluded with two tasks that determine the grade for the course. Each task can be solved with about 1,000 lines of code [5]. Students only qualify for the final tasks if they scored over 50% of the points over all exercise sheets. The solutions for the final tasks are again digitally submitted via the *Praktomat* system, just like the exercise sheets during the semester. However, grading of these tasks is not done by tutors.

A. Communication Infrastructure with Students

During the semester, we provide different information sources for course participants. Besides the lecture and the tutorials, we also have other forms of communication. First, we provide a wiki, where we document the rules for grading to provide transparency in this regard. In the wiki, students can also find a beginner's tutorial for Java. Second, we provide forums where students can ask questions about the lecture and the exercises. To ensure fairness and equality, questions regarding the content of the exercises are only answered in the forum. Here, students have the options to either write under a pseudonym or under their real name. The main idea is that students answer each other's questions (cf. Section IV). The teaching staff only answers questions that are not answered by students. In the last semester, the student forum had about 300 threads with about 1,000 individual posts. Additionally, we provide a separate private forum for our student teaching assistants where they can exchange questions and information about their tutorial. Besides the forum, we answered around 1,000 e-mails from students regarding organizational matters.

B. How to Cope with Cheating

As all submission are only done digitally, including the final exams, students might be able to buy or copy solutions from other students. This is a well known problem in exercises that are submitted digitally [7], [8] and we experienced this issue before. Therefore, it is necessary to cope with this kind of cheating. We address this using two different methods:

First, every student needs to pass a special exercise. This exercise is similarly organized to a classical written exam. Students come to the exercise, where we check their identity and give them a simple exercise that they need to answer in writing. This exercise tests a minimal set of programming concepts like simple array operations or variable initialization. This still cannot guarantee that no bought solutions are submitted, but at least guarantees that every student has understood basic programming principles. From our experience, everyone who has understood the basic principles passes this exercise and the ones who fail lack significant knowledge. We mostly ask questions in the domain of knowledge and comprehension in regard to Bloom's taxonomy [9] that are easy to grade. However, a small part of this special exercise is always targeted to capture whether the student understood the basic principles and can apply them (Application, Analysis, and Synthesis according to Bloom's taxonomy).

Second, to detect solutions shared between multiple participating students, we use the automatic plagiarism checker JPlag [10], [11]. JPlag compares each solution against all solutions in the course using abstract syntax trees. However, before we finally mark a solution as plagiarism, we manually check them to filter out possible false positives. False positives often exist in the simpler exercises at the start of the semester, where the number of possible solutions is limited. Despite the fact that we announce the plagiarism checks publicly at the beginning of the semester, we unfortunately detect multiple cases of plagiarism each semester. In the last semester (winter term 2018/2019), almost 10% of the course participants were involved in such cases.

III. Grading Students

We differentiate between functional requirements and coding style requirements for the grading of programming tasks. Usually, the ratio of the points for functionality to style is 2 to 3. The basis of the functional evaluation is the degree to which a program corresponds to the functionality specified in the task description. This functional evaluation is carried out almost entirely by automated checking of test cases. The basis of the style evaluation is the degree to which a program meets the principles of object-oriented design taught in the lecture. This style evaluation is almost completely carried out by manual code reviews.

The following section explains our grading process for exercise sheets and final assignments. In general, this process is identical for the exercise sheets and the final tasks.

1) After a task is created, the correction scheme for evaluating the coding style is created and test cases for testing functionality are developed.
2) After the submission deadline, further automated functional and style tests can be performed on the submissions.
3) Once these automated tests have been completed, the manual correction is started. The source text files can be edited by the corrector, for example to comment on certain source code lines or to make suggestions for improvements. In addition, the corrector can add general comments to the submission.
4) The correction is published to the students after completion via the *Praktomat*. The students can take a look at all test cases with their evaluation, all comments regarding the grading, and all changes to the source code.

A. Automated Functional Tests

In almost all cases, functionality is checked automatically through program output. All final tasks and most tutorial tasks include a command line interaction. Student solutions are compiled with each submission and automatically run several of previously defined functional test cases. This automation

allows a much wider range of testing than would be possible manually in realistic time. For the two final tasks, we had, on average, 50 different test cases per task.

Test cases are usually divided into two groups: public and private test cases. The public test cases are visible to students during submission and are mandatory to pass a valid submission. These test cases give the students feedback whether they have correctly implemented the most important (basic) functionalities before their final submission. Private test cases, on the other hand, are only visible to students after the correction has been completed. These test cases automatically check additional functionalities like edge cases and more elaborated functionality and thus form the basis for grading.

The functional test cases describe the expected command line output for a given input. After the *Praktomat* compiles the submission, the *Praktomat* executes the submission and compares the solution's output to the expected output. The definition for individual test cases is done via a simple text file, in which an input and the corresponding output are specified line by line.

B. Grading Coding Style

While we check some coding styles automatically with Checkstyle [12], such as indentation or mandatory comments, some properties are manually reviewed. Before grading, we create grading guidelines that contain information about the required style and how to apply the grading guidelines. Because of the high number of solutions and the scope of the exercises, the grading cannot be done by one person but a group. The grading normally takes around 670 person-hours.

The UI for grading in the *Praktomat* is illustrated in Figure 1. On the right side, the correctors can see the solution of the student as well as the results of the automated tests. They can access the detailed results by clicking on each test. To attest the solution, they can switch to the *Attest solution* tab.

This process originally caused some trouble: This process was cumbersome because correctors had to remember the line for the deduction (the reduction of points) and all grading guidelines. Additionally, the quality and especially the traceability for deductions varied between different correctors. Moreover, it was sometimes hard to deduce the reason for the deduction because of lacking reasoning or incomprehensibility. Therefore, we developed the Praktomat Enhancement Tool Suite (PETS). This tool is a web overlay written in JavaScript for the existing *Praktomat* interface. The left frame of Figure 1 shows its parts containing additional information and features. First, it shows the automatically calculated points (*final grade*) of the student, here 20 points. Afterwards, it shows a list of the structure of the current solution, where the red icon marks the class with the main method of the solution. Below the structure, it shows the automatically calculated functionality points, here 13 points. Then, it shows for each style category (OO-Modeling, Comprehensibility, Style) the current points and a list of buttons. Each button represents a typical defect of students' solutions. For instance, the *empty JavaDoc* button is used in case an empty JavaDoc exists or no JavaDoc exists at all. In our experience, this is a widespread defect. In case of this defect, a corrector would select the line in the editor and then click the button. This automatically deduces a fixed number of points and produces an explanatory text for the student. The text contains the deduction, the reason for the deduction and the line of the deduction. In case none of the predefined deductions is applicable, a custom defect button exists. With the custom button, a corrector can type an individual comment and decide an individual deduction. It is also used for further explanation when the general description of a button is not enough. The deduction for each defect is individually configurable and correctors can reuse individually created comments within the scope of one solution similarly to the buttons. For the premade buttons, a threshold of minimum number of occurrences can be configured until points are deduced, e.g., 5 occurrences of *bad identifiers*. Buttons can also directly deduce points for the very first occurrence, such as *visibility*, which means the wrong visibility modifier was used for a class. After the first deduction for a certain type of defect, other occurrences do not produce further deductions. Moreover, in each grading category there can be no negative points, i.e., the minimum is 0 points. If all possible points are already deducted, further mistakes are only listed but do not change the score. Only when markings are deleted, e.g., when correcting the grading, previously disregarded deductions are checked and applied automatically.

IV. Key Takeaways

In this section, we review our practice and experience of our programming course with regard to (a) *teaching concepts and strategies* in lecture and practical training, (b) *grading*, and (c) *communication*.

As we stated before, one of our ongoing and overall goal is to optimize the lecture in terms of *teaching quality* and *effort reduction with (semi-)automation* in case of a big course size in the aforementioned categories (a)-(c). The key takeaways in each category is a statement from our point of view. Statements marked with (*) are confirmed by students via, e.g., the course evaluations we conduct each semester.

a) Teaching Concepts and Strategies: As we believe in the *learning-by-doing* principle, we use teaching strategies aligned to that principle. To organize the course structure regarding teaching content, we provide a semester schedule that lists the lecture units and an advance organizer (cf. [13]). Each lecture unit is outlined with the corresponding learning objectives, formulated according to Bloom's taxonomy [9]. We also use student-activating teaching methods during the lecture to support efficient teaching and learning. For this, we have experimented with online response tools, but found replies in class by hand signs more interactive and less distracting for the students. Tutorials, which take place every week in small size attending groups, support on one side repeating the lecture content on the other side prepare for the practical tasks in advance. To achieve a more effective learning experience, the learning objectives should also be made clear in the practical task sheets.

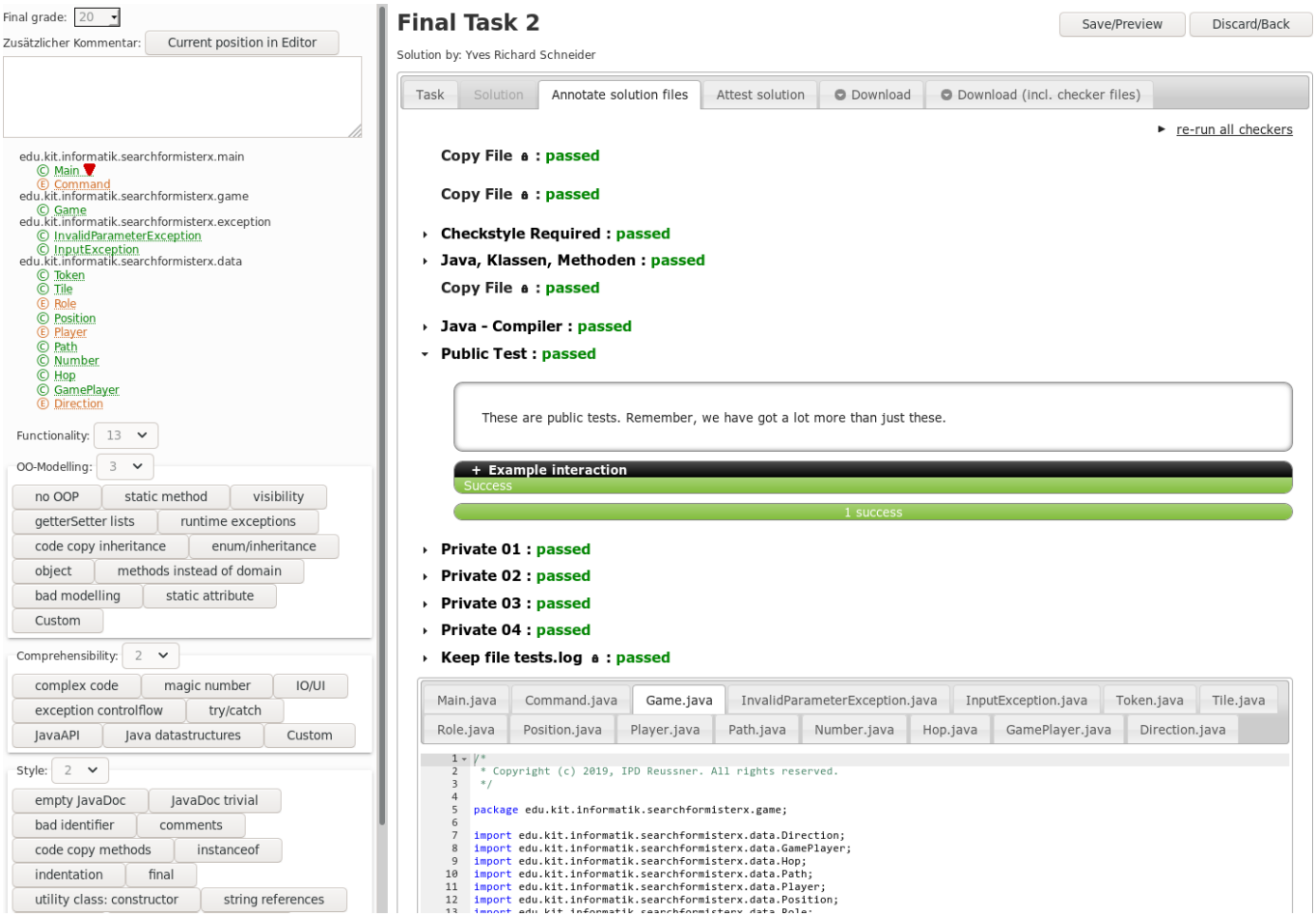

Figure 1: Attestation view with the PETS extension on the left side

Key Takeaways. Formulating learning objectives and the curricula brings many benefits: It limits and clearly determines the teaching content, thus also improves planning. Additionally, it creates a shared understanding about the expectations between lecturer and students and serves as criteria for external and self-assessment (*). Efficient teaching and learning during lecture can be achieve by student-activating teaching methods (*). Advance organizers are suitable for novice as well as more experienced programming students (*). Practical tasks encourage the *learning-by-doing* principle. In particular, implementing games is popular among students (*).

Open Issues. One of the open issues is the selection of the main teaching paradigm to use, i.e., objects first or algorithms first. Regarding the perspective of a novice programming student, we cannot decide which paradigm is more suitable. For the up-coming winter term, we plan to establish the objects first paradigm. At the beginning of the semester, we plan to use a visualization of objects and their behaviors by using an animation environment for demonstration purposes (cf. [14]).

b) Grading: During the semester, we have different phases of grading, but in this part we will focus on the grading of the final exam. At the end of the programming lecture, students should be able to write 500 to 1,000 lines of code based on a given specification. For grading, we differentiate between functional requirements and coding style requirements. Coding style is a solid part in the programming lecture and is also compactly described in our Ilias-Wiki (cf. *Communication* below). While grading referred to functional requirements is done automatically, we introduced PETS as enhancement tool for grading coding style. PETS allows us to build a coding style catalog by defining categories for grading purposes with template comments based on an informal coding style description. Additionally, graders have the opportunity to write so called custom comment for exceptional cases when a coding style convention is not considered yet.

Key Takeaways. A transparent and systematic grading scheme helps in reducing (negative) feedback from student as well as overhead concerning inspections. (Semi)-automatic grading techniques have – beside the reduction of effort (e.g., avoid repeating the same kind of feedback) – the potential to ensure fair grading in mass programming lectures.

Overall Conclusion. The alignment between teaching and examination is an ongoing optimization process. Besides effort reduction with (semi-)automatic grading, teaching quality needs continuously improvement by adapting new findings in terms of didactic concepts.

c) Communication: Regarding the novice programming course size, organizing the communication referred to organizational information and teaching content as well as teaching material is a major task. We use Ilias [15] an open source software for managing this issue. Our Ilias course for the programming lecture is organized as follows: a wiki for defining a programming style catalog, a download area to provide materials for lecture, tutorials, practical and final exam, and forums for important announcements and as a platform for questions of students. Concerning the discussion forums (one for the lecture, one for the practical tasks), we provide a set of strict rules and guidelines for interaction. For example, we use naming conventions for the titles of threads for every post concerning practical tasks. In our experience, this reduces the number of redundant questions and answers.

Key Takeaways. Learning management systems (LMS) as a central platform for all participant (i.e., teachers including student teaching assistants and students) are an effective way to keep the communication process manageable and transparent. Using open source LMS like Ilias also helps managing students and their learning activities as well as organizing a virtual learning environment (*). The E-learning course should be designed in a way that all requirements are met in the best possible way for teachers and students. In our case, regarding the novice programming course with many users, we consider a transparent communication and knowledge provision as major key element. Therefore, we use Ilias-features like file exchange, internal mail for important announcements to users, a discussion and announcement forum, and wiki functions.

Open Issues. We highly welcome the participation of students in our discussion forums at Ilias by answering questions from other students. We believe this also increases collaborative learning by building virtual learning groups. However, it is still an open issue how to support knowledge exchange among students and to encourage them even further to participate in answering discussion forums posts. In future, we will focus on this part in more detail and we plan to experiment with gamification mechanisms.

V. Conclusion

Teaching programming in lectures with around 1,000 attendants brings up some tough challenges, especially for grading. In this paper, we showed different challenges that we saw for our programming course. We then reported how we organize the course and perform grading at a scale of around 1,000 students. Automated and semi-automated techniques enable fair grading in mass programming lectures in terms of objectivity and consistency. Derived from our experiences, we listed key takeaways. Next steps for us are primarily about increasing our efficiency by further automating the grading process, especially regarding coding style. Here, we envision an extension that automatically calculates part of the grading based on the metrics provided by tools, such as SonarQube [16] or Checkstyle. To ensure correct grading, the grading is verified by a human corrector who decides on the final grade. We expect that this process not only reduces the overall human workload, but also increases the transparency as these metrics can be stated clearly and are well-defined and explained. Besides increasing the efficiency, an ongoing process is the improvement of the lecture and its paradigms, e.g., the learning goals and teaching objects first.

References

[1] M. Piteira and C. Costa, "Learning computer programming: Study of difficulties in learning programming", in *Proceedings ISDOC '13*, New York, NY, USA: ACM, 2013, pp. 75–80.

[2] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers", in *Proceedings ITiCSE '05*, ser. ITiCSE '05, Caparica, Portugal: ACM, 2005, pp. 14–18.

[3] K. Ala-Mutka and H.-M. Järvinen, "Assessment process for programming assignments", in *IEEE Conference on Advanced Learning Technologies*, 2004, pp. 181–185.

[4] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[5] J. Breitner, M. Hecker, and G. Snelting, "Der Grader Praktomat", *Automatisierte Bewertung in der Programmierausbildung*, 2017.

[6] J. Krinke, M. Störzer, and A. Zeller, "Web-basierte Programmierpraktika mit Praktomat", *Softwaretechnik-Trends*, vol. 22, no. 3, pp. 51–53, 2002.

[7] J. Sheard, M. Dick, S. Markham, I. Macdonald, and M. Walsh, "Cheating and plagiarism: Perceptions and practices of first year it students", in *Proceedings ITiCSE '02*, Aarhus, Denmark: ACM, 2002, pp. 183–187.

[8] S. Cerimagic and M. R. Hasan, "Online exam vigilantes at australian universities: Student academic fraudulence and the role of universities to counteract", *Universal Journal of Educational Research*, pp. 929–936, 2019.

[9] B. S. Bloom, "Taxonomy of educational objectives: The classification of educational goals", *Cognitive domain*, 1956.

[10] L. Prechelt, G. Malpohl, M. Philippsen, *et al.*, "Finding plagiarisms among a set of programs with jplag", *J. UCS*, vol. 8, no. 11, p. 1016, 2002.

[11] *Jplag 2.12.1*, Nov. 4, 2019. [Online]. Available: https://github.com/jplag/jplag (visited on 11/04/2019).

[12] *Checkstyle 8.18*. [Online]. Available: https://checkstyle.sourceforge.io/ (visited on 11/04/2019).

[13] D. P. Ausubel, "In defense of advance organizers: A reply to the critics", *Review of Educational research*, vol. 48, no. 2, pp. 251–257, 1978.

[14] D. Boles, *Programmieren spielend gelernt mit dem Java-Hamster-Modell*. Springer, 1999, vol. 2.

[15] *Ilias: The open source learning management system*. [Online]. Available: https://www.ilias.de/ (visited on 11/04/2019).

[16] *SonarQube*. [Online]. Available: https://www.sonarqube.org/ (visited on 11/04/2019).