Stager: Simplifying the Manual Assessment of Programming Exercises

Christopher Laß, Stephan Krusche, Nadine von Frankenberg, Bernd Brügge
Technische Universität München
christopher.lass@tum.de, krusche@in.tum.de, nadine.frankenberg@in.tum.de, bruegge@in.tum.de

Abstract

Assessing programming exercises requires time and effort from instructors, especially in large courses with many students. Automated assessment systems reduce the effort, but impose a certain solution through test cases. This can limit the creativity of students and lead to a reduced learning experience. To verify code quality or evaluate creative programming tasks, the manual review of code submissions is necessary. However, the process of downloading the students' code, identifying their contributions, and assessing their solution can require many repetitive manual steps.

In this paper, we present Stager, a tool designed to support code reviewers by reducing the time to prepare and conduct manual assessments. Stager downloads multiple submissions and adds the student's name to the corresponding folder and project, so that reviewers can better distinguish between different submissions. It filters out late submissions and applies coding style standards to prevent white space related issues. Stager combines all changes of one student into a single commit, so that reviewers can identify the student's solution more quickly.

Stager is an open source, programming language agnostic tool with an automated build pipeline for cross-platform executables. It can be used for a variety of computer science courses. We used Stager in a software engineering undergraduate course with 1600 students and 45 teaching assistants in three separate programming exercises. We found that Stager improves the code correction experience and reduces the overall assessment effort.

1 Introduction

The number of students in university courses is increasing. The number of new undergraduate students at our computer science department increased by 81 % between 2013 (1110 students) and 2017 (2005 students)[1]. Practical programming exercises are essential in computer science education and help students acquire important skills in software development [Staubitz et al., 2015]. However, a manual assessment of programming exercises in large courses can take a considerable amount of time and effort. Automatic assessment systems (also called auto-graders) aim at flexibility and scalability in large courses, and allow to integrate exercises into lectures [Krusche et al., 2017b]. These systems utilize, among others, version control systems (VCS) to store the code solutions of students in repositories and test cases that are executed on a continuous integration server to assess the solution to a programming exercise automatically [Heckman and King, 2018; Krusche and Seitz, 2018].

While automated assessment systems significantly reduce manual assessment effort, they have drawbacks. Predefined test cases cannot cover all possible solutions and therefore impose a certain solution on the students. Some students are limited in their programming skills, while other students can exploit the test cases by repetitive trial-and-error submissions. Especially first year students who are new to programming often experience problems when trying to formulate their solution and thoughts as an executable computer program [Robins et al., 2003]. Such submissions can be overly complicated, and assessment systems cannot (yet) provide enough useful feedback in that regard. Furthermore, some programming exercises cannot be assessed automatically. The automated grading of creative assignments with open problem statements is hardly possible because different solutions exist [Knobelsdorf and Romeike, 2008; Krusche et al., 2017a]. An example for such an assignment is to implement a creative collision strategy in a 2D racing game. Automated test cases could be able to validate a collision, but are incapable of assessing the creativity or code quality of the solution. As a result, manual assessment can be beneficial, even in large courses that have fully implemented automated grading solutions.

However, the process of manually assessing multiple students' solutions requires repeated manual steps. Tasks such as finding the next student's repository, downloading the source code, and renaming the folders and projects names for standardization can be time-consuming and error-prone. Determining a student's contribution is challenging when the exercise builds upon a provided code template and when the

[1] https://www.tum.de/die-tum/die-universitaet/die-tum-in-zahlen/studium

students use multiple commits in their code repository. Then it becomes difficult to separate the provided template and the final solution.

In this paper, we present Stager, a tool that is designed to support the manual assessment of programming exercises. Reviewers, e.g. teaching assistants or instructors, can automate the manual steps that are necessary to prepare the students' code repositories, for instance download all repositories at once, and thereby reduce the manual assessment time. The idea for Stager evolved during an undergraduate university course with 1600 students and 45 teaching assistants. An initial implementation was used for three separate programming assignments.

The remainder of the paper is organized as follows. We describe related work focusing on existing automated assessment solutions and the limitations of automated assessment approaches in Section 2. In Section 3, we cover Stagers' approach to automating the recurring manual steps during the correction of programming exercises. We describe design decisions, the exercise workflow with Stager, the configuration possibilities of the tool, and the concrete tasks of Stager, e.g. the *Download repositories* task. We analyze the improved code assessment experience of the teaching assistants by means of an experience report in Section 4, where we also present the results of a quantitative analysis of Stager's use in three programming exercises. Section 5 concludes the paper and provides directions for future work.

2 Related Work

Several automated assessment system approaches for programming assignments exist [Heckman and King, 2018; Knobelsdorf and Romeike, 2008; Krusche and Seitz, 2018; Pieterse, 2013]. Advantages include a decrease in the workload of course instructors and timely feedback for students [Pieterse, 2013]. Automated systems work well to grade programming assignments consistently and evaluate specific aspects, e.g. the functionality [McCracken et al., 2001] or efficiency of a system [Jackson and Usher, 1997]. However, they are missing the benefit of personal feedback which a manual grading approach could provide. The test cases used by such systems cannot assess the code quality and "elegance" of the solution [Poženel et al., 2015].

Building a robust automated assessment system amounts to a heavy workload, whereby the definition of the test cases is (usually) the most time consuming activity [Cerioli and Cinelli, 2008]. This workload is amplified when designing tasks with some degree of freedom of solutions [Chen, 2004]. The degree of freedom of solutions indicates the difficulty of the exercise [Striewe and Goedicke, 2013], meaning that a difficult exercise has more possible solutions and therefore has an increased workload to design the automated assessment system. Depending on the class size, it can therefore be less time consuming to manually assess solutions rather than to design the automated assessment system [Ala-Mutka, 2005].

Further, students can become distracted by automated feedback. For instance, students may be tempted to fix only the failing tests instead of focusing on the assignment [Heckman and King, 2018]. Automated assessment systems circumvent the detection of frequent mistakes or misunderstandings among students. The understanding and resolution of common errors is an essential learning experience for students. Semi-automated systems combine the mentioned aspects by providing automated grading, as well as manual feedback. Such systems offer personalized feedback to some extent, for instance the instructor can annotate a static assessment [Gerdes et al., 2017]. Other systems give the student instant feedback if the student's solution is correct. If it is not, the instructor reviews each solution and can give additional feedback if required [Insa and Silva, 2015]. Many systems focus on the grading itself, but not on the process the instructor has to follow to obtain the students' solutions.

Some commercially available systems and tools that are used in computer science (CS) courses offer features that aim at simplifying this process. In 2000, Jackson proposed an approach that pre-processes student submissions (sent via e-mail) by removing irrelevant information or unpacking files [Jackson, 2000]. For submissions via repositories, pull requests (also called merge requests) in GitHub[2], GitLab[3], or Bitbucket[4] allow students to commit their changes into separate branches. After requesting the code to be merged into the main branch, i.e. a submission, reviewers can highlight the student's contribution as difference to the template code and provide feedback by requesting changes. While pull requests can also be integrated with continuous integration systems, e.g. using TravisCI[5] to detect compile errors and to run automated tests, reviewers might still need to download the source code and execute it to verify if all requirements of the problem statement have been solved.

GitLab introduced a "Squash and Merge" option which "applies all of the changes in a merge request as a single commit, and then merges that commit using the merge method set for the project"[6]. This cleans up the commit history and can make it easier to identify the contribution of one particular student. Tools and services, such as Gerrit[7], support code reviews that enable the reviewer to see the code difference, and provide the option to leave in-line comments.

[2] https://github.com
[3] https://gitlab.com
[4] https://bitbucket.org
[5] https://travis-ci.org
[6] https://docs.gitlab.com/ee/user/project/merge_requests/squash_and_merge.html
[7] https://www.gerritcodereview.com

However, such tools primarily focus on continuous feedback rather than assessing a student's solution.

3 Stager's Approach

This section presents an approach that automates manual steps during the correction of programming exercises in order to prepare student repositories for easier assessment. We show how code reviewers can use Stager. Furthermore, we explain the different tasks that are automatically executed by Stager.

Figure 1 illustrates the exercise workflow including the manual assessment with the help of Stager as a UML activity diagram. As precondition for this workflow, every student must have their own repository with the code template for the exercise in a VCS[8]. After the students complete the exercise, they commit and push their solutions to the VCS (action 1.3).

Before reviewers start to work, they need to configure Stager (action 2.1). Then, they trigger Stager to process different tasks (actions 3.1 ... 3.6), such as *Download repositories* or *Normalize code style*. Finally, the reviewer can manually assess the pre-processed submissions and give qualitative feedback (action 4.2 and 5.) to the students in any arbitrary form (e.g. uploading the feedback into an exercise management system such as Moodle[9]).

The action *2.1 Configure Stager* of the *Reviewer* is described in Section 3.1. *Stager*'s actions are described as tasks in Section 3.2. The numbering in Section 3.2 aligns with the corresponding action in Figure 1.

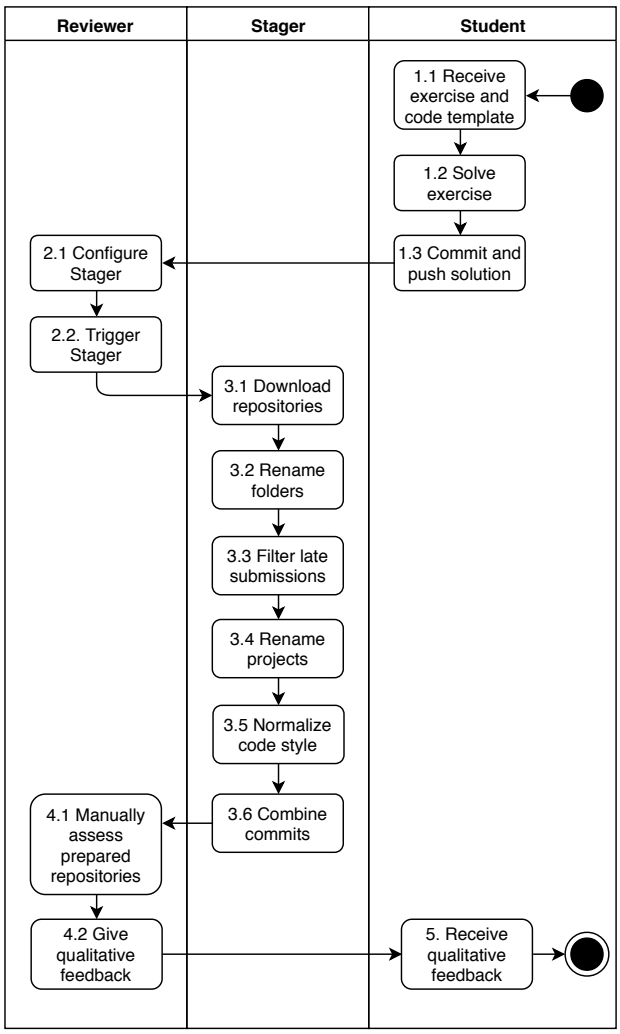

Figure 1: Exercise workflow with Stager: students complete the exercise and upload their solutions to a VCS. The reviewer configures and triggers Stager to process different tasks, e.g. *3.1 Download repositories*. Afterwards, the reviewer manually assesses the prepared repositories and gives qualitative feedback to the students.

3.1 Stager's Setup

Stager is free, open source, and available under the MIT license[10]. It is platform independent and programming language agnostic, making Stager universally applicable. It is written in the Go programming language[11] and makes use of the distributed version control system git[12]. Cross-platform executables can be downloaded from the automatic build pipeline or compiled from the source code.

Stager's configuration is separated into two files *students.csv* and *config.json*, based on how frequently the settings change. The list of students in *students.csv* might not change during the course duration, while *config.json* changes for every exercise. The configuration procedure must be completed after the code template was finished and before Stager is executed. Stager or its configuration does not add any preconditions or constraints on the students. The following settings can be edited:

1. Credentials: Remote git repositories can be accessed via the SSH or HTTP protocols [Lawrance et al., 2013]. For HTTP, the JSON keys *username* and *password* have to be set with valid credentials and access rights to the VCS. For SSH, Stager uses the operating system's global SSH settings and therefore does not require further configuration.

2. Latest commit hash of a programming exercise template: The programming exercises that are distributed to the students build upon a given code template. The SHA hash of the latest commit for the code template, meaning the latest code changes the reviewer included, must be set for the JSON key *squash_after*. This setting is required for Stager to distinguish between the given code by the reviewer and code written by the student. This configuration option is used by the task *Combine commits* and is further elaborated in Section 3.2.

3. Deadline for homework submission: Students have to submit their homework in a given time-frame.

[8] There are multiple tools available that automate this step, e.g. ArTEMiS, Github Classroom, etc.

[9] `https://moodle.org`

[10] `https://github.com/arubacao/stager`

[11] `https://golang.org`

[12] `https://git-scm.com`

For example, the homework must be submitted by Sunday midnight because the programming exercises will be discussed in class on Monday morning. However, VCSs have limitations when it comes to time-based repository access. As described in more detail in Section 3.2, the task *Filter late submissions* allows to overcome these VCSs limitations. The deadline for students submitting their homework is set with the JSON key *deadline*. The standard datetime format `YYYY-MM-DD HH:MM:SS` must be used. For example, `2018-08-31 23:59:59` is valid.

4. Remote repository URL schema: Each student has a personal repository that can be accessed with a unique URL. A general URL schema can be derived from these unique URLs, where the students' identifiers are substituted by a placeholder. For example, for the repository URL (1) of student *10001*, the derived general URL schema is (2). If the repositories are accessed using HTTP as in the example, two additional placeholders must be set for the reviewer's credentials (3). The resulting schema is set for the key *url*.

*https://repo.uni/cs101/exercise01-**10001**.git* (1)

*https://repo.uni/cs101/exercise01-**%s**.git* (2)

*https://**%s:%s**@repo.uni/cs101/exercise01-**%s**.git* (3)

5. List of students: In addition to the mentioned settings, Stager requires a list of students the reviewer wants to assess. The students' names and identifiers are defined in the *students.csv* file with the format shown in Listing 1. All mentioned people and courses in this paper are placeholder names and do not exist in reality.

Listing 1: Sample students.csv

```
name,id
John Doe,10001
Jane Roe,10002
```

After configuration, the Stager executable, *config.json*, and *students.csv* are placed in a dedicated and empty folder. Stager can then be executed via a double click or from the terminal. Listing 2 illustrates this workflow. After Stager terminates, the students' repositories are locally available and prepared by the tasks described in the following Section 3.2.

Listing 2: Folder setup and execution of Stager

```
$ cd ~/cs101/assessment3
$ ls
config.json   stager   students.csv
$ ./stager
```

3.2 Stager's Tasks

Stager provides an extendable framework which makes it easy to add or remove tasks according to the reviewer's requirements. Tasks are functions that modify the repository or its contents and have a single purpose. For example, the *Rename folders* task appends the student's name to the corresponding folder. Stager is composed of multiple tasks (shown in the Stager swimlane in Figure 1) that adhere to certain rules and are sequentially performed during the tool's execution. The implementation allows a clear distinction of tasks, such that each task addresses a separate purpose. Therefore, it is easy to add new tasks or remove existing ones conceptually and implementation-wise in the future. For example, when the reviewer does not need a certain task, only one line of code within the array of tasks has to be removed. Furthermore, tasks must be idempotent, meaning that multiple executions of the task lead to the same output. Even though tasks are independent, they are processed sequentially, i.e. the order of the tasks is relevant. For instance, repositories first have to be downloaded before other tasks have local file access.

The goal of Stager is to simplify the manual assessment of programming exercises by modifying source code, files, and repositories. Repetitive manual steps that are required for the reviewer to start the assessment should be reduced or eliminated by Stager. We identified the following relevant tasks (listed according to the order of execution) and describe each of them in detail in the following:

1. Download repositories
2. Filter late submissions
3. Rename folders
4. Rename projects
5. Normalize code style
6. Combine commits

1. Download repositories: In order to better determine the software quality and verify if all requirements of the problem statement have been solved by the students' submissions, it is necessary for the reviewer to compile and execute their homework source code locally. Hence the repositories must be available on the reviewer's computer. The initial task clones all repositories of the predefined students *as-is* and *all at once* to a given folder on the reviewer's computer. This first task takes potential existing local repositories into account and overwrites them. It ensures that each local repository is in sync with the remote repository and in a clean state.

The following tasks modify files and therefore require write access to the repositories. These modifications can only be performed when the repositories are locally available. Consequently, the *Download repositories* task must be first.

2. Filter late submissions: Homework submissions are tied to a hard deadline. With web-based VCSs like Bitbucket or GitLab, it is hardly possible to block student commits after a given deadline. Students could exploit this situation and extend their

time to finish the exercise as shown in Figure 2. The *Filter late submissions* task analyzes the commit timestamps and sets the repository to the state of the pre-configured deadline in *config.json*. Commits after the deadline are not considered anymore. This way time-based limitations of web-based VCSs are bypassed. However, this procedure is not fully forgery-proof, since commit timestamps can be manipulated.

File changes made prior to this task would be striped out, since the repository is set to the state of the pre-configured deadline. Therefore, the *Filter late submissions* task must be executed before any other task can modify files.

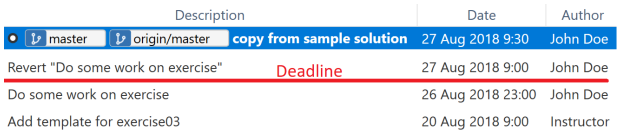

Figure 2: Filter late homework submissions by excluding commits after the homework submission deadline. The two commits above the red line are after the deadline while the two commits below the red line are before the deadline.

3. Rename folders: Depending on the naming convention, only the student's identifier is used for the repository name. The resulting folders can be hard to keep separate and to associate with the correct student. For obfuscation and identity protection this is reasonable, but counterproductive on the reviewer's local system since it is easier to identify a student by their name and not through their id. Once the repositories are locally available, the *Rename folders* task appends the student's name to the corresponding folder as illustrated in Figure 3.

```
1    - cs101-exercise03-10001
2    - cs101-exercise03-10002
1    + cs101-exercise03-10001_john_doe
2    + cs101-exercise03-10002_jane_roe
```

Figure 3: Append names to folders to better distinguish between students. Without the names john_doe and jane_roe, it would be difficult to identify which folder belongs to which student.

4. Rename projects: As precondition of Stager, each student must have their own repository for each published exercise. The content of these repositories is always identical. As a result, the project names are also identical for all students. This leads to the problem that reviewers could only import one project at the same time into Eclipse in order to review and execute the code. Renaming all projects manually is time-consuming and error-prone. Analogue to the *Rename folders* task, a student's name is prepended to the corresponding project name. This makes it possible to distinguish between students within source code editors or integrated development environments (IDEs), e.g. Eclipse[13] (Figure 4), and allows to import multiple projects at the same time. Eclipse, for instance, does not allow to import multiple projects with identical names, which makes it impossible to compare multiple solutions without renaming the projects.

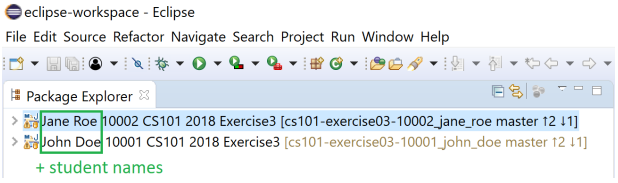

Figure 4: Prepend student names to projects so that the submissions of multiple students can be imported into Eclipse and reviewed at the same time. Jane Roe and John Doe are prepended to the project name. Otherwise the reviewer could only import one Eclipse project at the same time.

5. Normalize code style: The encoding and code style of the provided code template and the final student's contribution should be consistent. Windows and Unix-based systems use different line breaks for code files by default. Windows uses carriage return and line feed ''\r\n'' as a line ending, whereas Unix based systems use just line feed ''\n''. Also, IDEs might automatically enforce a different code style standard than desired. As illustrated in Figure 5, this could lead to non-relevant changes and obscured code differences in commits, thereby making it harder to assess the submission. To avoid these non-relevant file changes by the student, Stager invokes a *linter* that automatically normalizes the code to the same standards as the initial template. This means that all white space related changes, e.g. line breaks, empty spaces and tabs are removed, so that the reviewer does not need to analyze them. Each programming language has its own linting strategies, utilizing existing tools like *eslint*[14] for Javascript or *checkstyle*[15] for Java. This hides pure white space and encoding changes and allows code reviewers to focus on the actual contributions by the students.

6. Combine commits: Reviewers provide code templates as a starting point for the programming exercise, in which the student has to make changes across multiple files. These changes can be small compared to the provided template and consequently hard to identify by the reviewer. In order to determine the student's contribution more effectively, it is helpful to see the exact difference between the template and the final submission instead of only looking at the final submission. VCSs provide easy comparison methods

[13] https://www.eclipse.org
[14] https://github.com/eslint/eslint
[15] https://github.com/checkstyle/checkstyle

```
1   - public int getSpeed(){
2   -       return this.speed;
3   - }
1   + public int getSpeed(){
2   +       return this.speed;
3   + }
```

Figure 5: There is no visual change in the two code blocks in this figure. However, non-visible *line breaks* cause the comparison tool to show these lines. This can make it time-consuming for the reviewer to identify relevant changes.

where the difference made by a single commit is visible. However, a submission can consist of multiple commits. The reviewer would have to compare each commit and memorize the changes themselves, which makes the standard comparison method impractical and error-prone.

The *combine commits* task combines the students commits into one single commit. The reviewer does not need to review multiple changes within the same code line and can omit changes that have been added in one commit and removed again in a later commit. This single commit also contains all Stager related changes (e.g. white space changes). As a result, it is easy for the reviewer to quickly identify the student's contribution and to decide if the solution is correct. In addition to the existing branches with the complete commit history, Stager adds the combined commit into a separate branch. Thus, information is only added and not removed from the repository and the reviewer could still see the whole commit history. Web-based VCSs like GitHub also offer a squash feature, however, the reviewer would have to trigger it manually for each repository.

Figure 6 illustrates this process with an example student *John Doe* and an *Instructor*. The *Instructor* provides a code template. *John Doe* works on the given exercise. Over a period of one day, *John* submits his work separated across multiple commits. As seen in the bottom right corner of Figure 6, one assignment was to *Add new car types to the game*. Since *John* submitted multiple code changes and removed the "TODO" lines within the code, the reviewer would have to actively scan all nine commits to identify *John*'s solution. Stager solves this time-consuming process by combining all student commits into one single commit that includes all changes by *John*. This single commit is selected in the top of Figure 6. The reviewer can see every file that has been modified by the student and quickly identify, whether *John* has completed the assignment correctly.

4 Experience Report

The following experience report describes the lecture-based course Introduction to Software Engineering (EIST[16]) in which we used Stager to improve the manual assessment of programming exercises. EIST is a second semester bachelor's course with a heterogeneous group of students including computer science, business informatics, and business students.

The course assumes that students have successfully completed an introductory course in computer science (e.g. CS1) and are familiar with object-oriented programming in Java. The course's learning goals are that students are able to apply relevant concepts and methods in all phases of software engineering projects including analysis, design, implementation, testing, and delivery. Further, students know the most important terms and concepts and can apply them in modeling and programming tasks. They are aware of the problems and issues that generally have to be considered in software engineering projects. Table 1 shows the schedule and the content of the course.

Week	Content
1	Introduction
2	Model-Based Software Engineering
3	Requirements Elicitation and Analysis
4	System Design I
5	System Design II
6	Object Design
7	Model Transformations and Refactorings
8	Pattern-Based Development
9	Lifecycle Modeling
10	Software Configuration Management
11	Testing
12	Project Management
13	Repetitorium

Table 1: The course *Introduction to Software Engineering* lasts 13 weeks.

1600 students were registered for the course in 2018. One lecturer and three exercise instructors were involved in the organization of the course. 45 teaching assistants were responsible for holding 74 exercise group sessions per week. Teaching assistants were mainly bachelor students in the fourth semester, who successfully completed the same course in the previous year.

The course design is based on interaction and assumes active participation from students. The interactive parts include in-class exercises, in-class quizzes, and exercise sessions. Students need to bring their laptops to the class and to exercise sessions. Students can earn bonus points for completing in-class and homework exercises successfully. They can use these bonus points to improve their final exam grade.

[16]The German title is "Einführung in die Softwaretechnik".

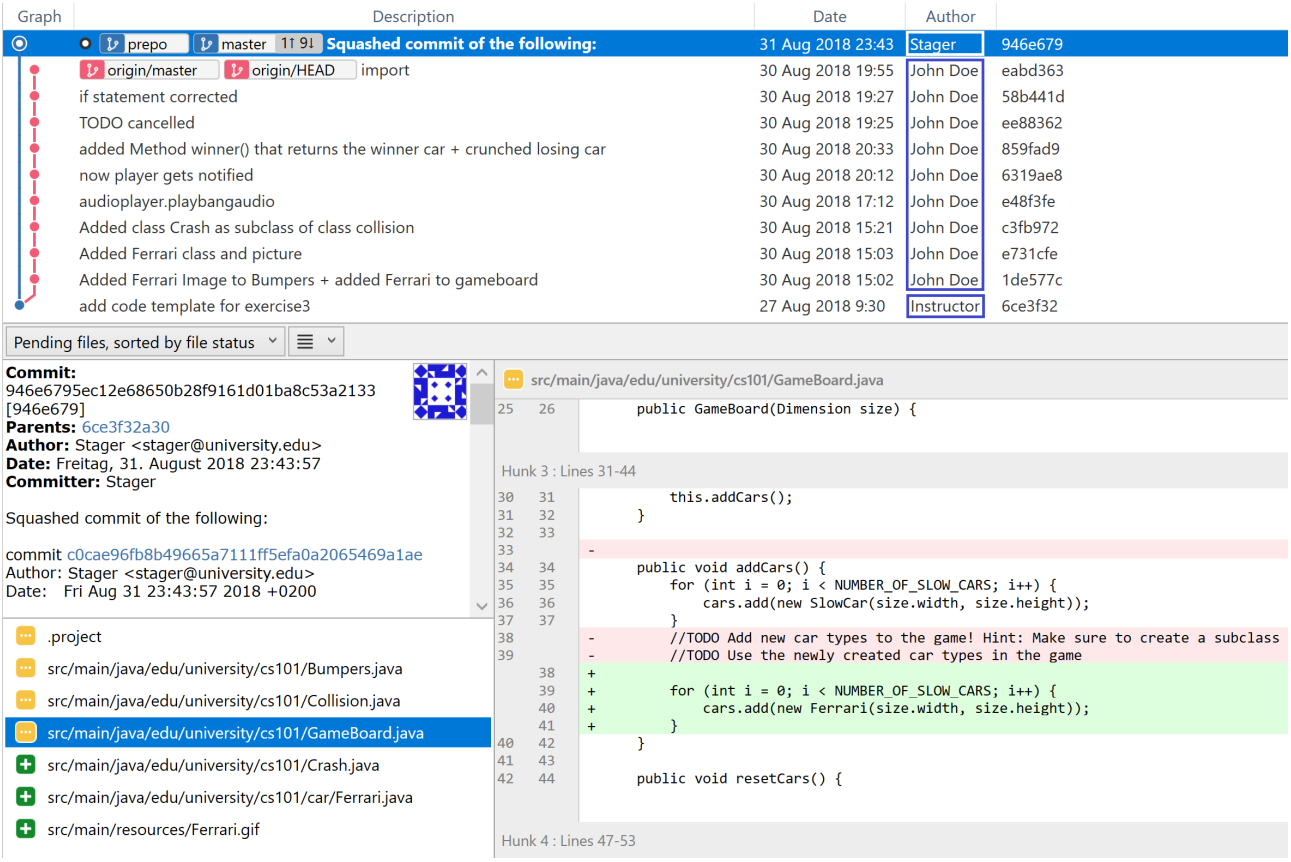

Figure 6: Student commits are combined into one discrete change set: the commit at the top highlighted in blue. This commit displays the difference between a provided code template by the instructor and the submitted solution by the student. All commits of the student John Doe are still available.

For instance, if they score more than 90 % of the total exercise points, their grade in the final exam is improved by 1.0. This possibility motivates the students to participate in the in-class exercises and in the homework exercises. In-class exercises consist of quizzes (similar to the quiz exercises described in [Krusche et al., 2017c]), modeling and programming exercises. Homework exercises include modeling, text and programming exercises.

4.1 Programming Exercises

Between 600 and 1200 students have actively participated in each programming exercise throughout the semester which is shown in Figure 7 and Figure 8. In each exercise, the students had to write new source code or adjust existing code based on a given problem statement. All students worked on the existing template code of an exercise in their individual git repository. The exercises were based on a 2D racing game called Bumpers. In the game, cars collide with each other and each collision has a winner. The course is designed so that each week's exercises focus on a different part of Bumpers in accordance with the lecture's content, e.g. in week 8, "Pattern-Based Development", exercises include the implementation of different design patterns to make the game extensible for new requirements.

To submit their solutions, the students commit their changes to a version control system. This automatically triggers test cases on a continuous integration server to verify the given solution. After the submission of their solution, students can automatically see the test results as individual feedback and improve their solution according to this feedback.

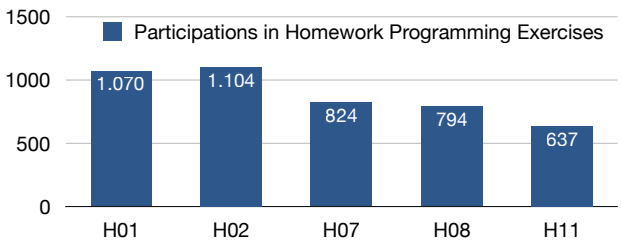

Figure 7: Number of students who submitted solutions to homework programming exercises

However, not all aspects of a problem statement can be automatically tested. Either it is difficult to test a certain aspect of a solution, for instance complex behavior tests, or the problem statement provides a

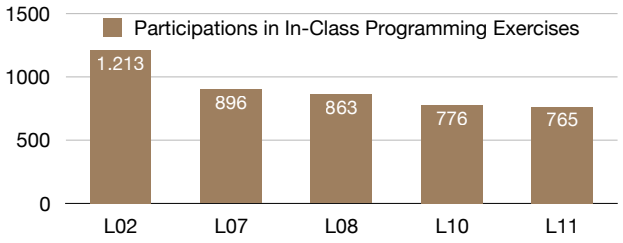

Figure 8: Number of students who submitted solutions to in-class programming exercises

high degree of freedom which makes it difficult to write test cases, e.g. open or visionary questions.

The following three homework programming exercises required manual assessment by the teaching assistants. The second and third exercises were graded semi-automated.

1. Collision Detection: The task was to implement a creative collision detection algorithm for cars in Bumpers. The students were given executable template code and had to extend it with a new class that included their solution. This exercise required manual correction to test whether the new collision algorithm performed as intended. Additionally, the most creative solutions were awarded and shown in class.

2. Serialization of Code: The students had to instantiate objects from two classes in Java. The main task was to serialize and deserialize these object using JSON. An automated assessment system was used to test the input and output of the serialization. However, the students wrote their own serialization code, so their solutions varied, e.g. in the naming of the objects or methods. This required the teaching assistants to assess the implementations manually.

3. Adapter Pattern: Based on a code template, the assignment was to extend the 2D car racing game Bumpers with legacy code using the adapter pattern. The legacy code for an existing analog speedometer panel was provided separately. An automated assessment system graded the students' solution. In addition, the teaching assistants had to verify if the speedometer panel was shown in the game user interface and displayed the velocity correctly.

4.2 Results

In order to determine how many manual steps during a homework assessment can be automated by Stager, we conducted a quantitative analysis for these three programming exercises. For the qualitative analysis we focused on:

1. Number of commits per student
2. Number of commits after the exercise deadline
3. Source code changes where only white spaces have been added or removed

Table 2 displays an overview of the number of participating students for each exercise together with submission metrics. The number of commits per student varies from 1.81 to 5.91 on average. Stager's *combine commits* task will combine student commits into one single commit so that reviewers can distinguish the difference between the provided code template and code submitted by the student immediately. There are respectively 34, 8, and 7 late submissions for the observed exercises. Stager will automatically filter commits that are contributed after the defined exercise deadline. There are between 118 and 183 students that submitted at least one commit where they only changed white spaces. While reviewing the student contributions, white space related changes are visually distracting to the reviewer (see Figure 5), since these changes are not relevant to the exercise.

In informal discussions, seven teaching assistants reported that Stager reduced their reviewing effort significantly. The workflow without Stager required the teaching assistants to first filter the repositories by student, then to check the commit dates and times, clone or download the code, and to fix potential white space problems in order to be able to assess the actual sumbission. Depending on the amount of exercise sessions, teaching assistants had to perform this manual workflow for up to 50 student submissions. Further, the repository names only include the student's identifiers, not names, so that mix-ups could occur when importing the solutions into an IDE.

4.3 Discussion

While using Stager, we identified four main advantages: (1) Combining commits is particularly helpful to review all changes of one student at a glance. This allows the reviewer to immediately identify whether the student has understood the problem statement and has implemented a proper solution. (2) Renaming the projects simplifies the assessment and comparison of multiple solutions. The reviewer can import multiple solutions at the same time with one click into an IDE. It increases the confidence of the reviewers, so that the assessment is associated with the correct student. (3) While most students follow the deadline of an exercise, some students have committed changes after the deadline. It would be possible to remove write permissions for all student git repositories at the given deadline, but this might be hard to realize. Enforcing the deadlines in Stager is easier and filters the cases where students try to circumvent the deadline. (4) Stager only depends on using git repositories for programming exercises and other instructors can use it without adaptions in their courses, e.g. in GitHub Classroom or other git environments[17]. As Stager is open-source, other instructors can adapt it to their own needs.

While Stager is easy to use as a standalone tool, reviewers need to configure it for each exercise as described in Section 3.1. It would further simplify the

[17] https://classroom.github.com

Metric	1. Collision Detection	2. Serialization of Code	3. Adapter Pattern
Total submission count	1104	657	794
Total commit count	1998	3880	2447
Average amount of commits per student	1.81	5.91	3.08
Total commits after exercise deadline	34	8	7
Total submission count with at least one white space related change	125	118	183

Table 2: Quantitative analysis of submission metrics for three programming exercises of the course

configuration if Stager would be integrated into the exercise management system, where the instructor sets up the programming exercise. Then Stager would automatically know the submission deadline, the latest commit of the instructor in the code template, and the remote repository URL. This would make the use of Stager easier and seamlessly.

4.4 Limitations

Our experience report only included three exercises that used Stager for code reviews. It would be interesting to analyze the concrete time-savings with a comparison and to use Stager throughout the whole course. While we have first indications, we did not evaluate whether the quality of the reviews improved through the use of Stager.

In addition, Stager's implementation currently has the following limitations: (1) Reviewers have to manually search for each student repository's key the first time they use Stager, before being able to use Stager for the remaining steps. The previously mentioned integration of Stager into an exercise management system would overcome this step. (2) For every exercise, the *config.json* file has to be changed accordingly with the deadline, URL-schema, and commit of the instructor. This could also be adapted to be automatically included when creating exercises by means of an exercise management system. (3) Reviewers have to install Stager on their computer and start it via a double-click or the command line interface. A web-based solution or a plugin into an IDE (e.g. Eclipse) in which the reviewers import the code would provide a more user-friendly experience.

5 Conclusion

Manual code reviews are important for the learning experience of students. While automatic tests can find typical problems and check whether code works as intended, they cannot find all problems, code smells, and implementation issues. Automatic assessment imposes certain solutions on the students and might limit their creativity. Stager supports code reviewers by automating steps in the manual assessment of programming exercises to reduce effort for the preparation and the conduction of code reviews. Stager downloads multiple students' submissions, renames folders and projects, filters out late submissions, and fixes typical white space problems. All commits of one student are combined into one discrete change-set that is easier to review. Code reviewers can better distinguish between the submissions of multiple students and identify students' contributions more quickly.

Our experience in a course with 1600 students and 45 teaching assistants shows that Stager reduced the reviewing effort and time for teaching assistants. The reviewers used the saved time to write better reviews and give more detailed feedback to the students. This improved the student's learning. A quantitative analysis in three programming exercises shows that Stager identifies several late submissions and fixes many white space issues.

Stager is free, open source, and available under the MIT license, so that other instructors can use it in their courses[18]. We will continue the development and aim to integrate the tool into the automated assessment system ArTEMiS [Krusche and Seitz, 2018]. Our future work also includes the integration of code quality metrics to support the actual code assessment. This could make it easier for reviewers to spot code quality issues in the students' solutions and be included, e.g. as a text file, into the feedback pipeline.

In addition, we would like to evaluate the quality of the code reviews when using Stager compared to pure manual reviews with respect to the completeness, helpfulness, and understandability of the review. Depending on the results of this evaluation, we could integrate strategies to semi-automatically propose common code review feedback. Automatic suggestions would further reduce the effort of reviewers but allow them to tailor these suggestions to the concrete situation.

References

[Ala-Mutka 2005] ALA-MUTKA, Kirsti M.: A Survey of Automated Assessment Approaches for Programming Assignments. In: *Computer Science Education* 15, pages 83–102, 2005.

[Cerioli and Cinelli 2008] CERIOLI, Maura ; CINELLI, Pierpaolo: GRASP: Grading and Rating ASsistant Professor. In: *Proceedings of the Informatics Education Europe III Conference*, 2008.

[18] https://github.com/arubacao/stager

[Chen 2004] CHEN, P. M.: An automated feedback system for computer organization projects. In: *IEEE Transactions on Education* 47, pages 232–240, 2004.

[Gerdes et al. 2017] GERDES, Alex ; HEEREN, Bastiaan ; JEURING, Johan ; BINSBERGEN, L. T. van: Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. In: *International Journal of Artificial Intelligence in Education* 27, pages 65–100, 2017.

[Heckman and King 2018] HECKMAN, Sarah ; KING, Jason: Developing Software Engineering Skills Using Real Tools for Automated Grading. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 794–799, 2018.

[Insa and Silva 2015] INSA, David ; SILVA, Josep: Semi-Automatic Assessment of Unrestrained Java Code: A Library, a DSL, and a Workbench to Assess Exams and Exercises. In: *Proceedings of the Conference on Innovation and Technology in Computer Science Education*, pages 39–44, 2015.

[Jackson 2000] JACKSON, David: A semi-automated approach to online assessment. In: *SIGCSE Bulletin* 32, pages 164–167, 2000.

[Jackson and Usher 1997] JACKSON, David ; USHER, Michelle: Grading Student Programs Using ASSYST. In: *Proceedings of the 28th Technical Symposium on Computer Science Education*, pages 335–339, 1997.

[Knobelsdorf and Romeike 2008] KNOBELSDORF, Maria ; ROMEIKE, Ralf: Creativity As a Pathway to Computer Science. In: *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, pages 286–290, 2008.

[Krusche et al. 2017a] KRUSCHE, Stephan ; BRUEGGE, Bernd ; CAMILLERI, Irina ; KRINKIN, Kirill ; SEITZ, Andreas ; WÖBKER, Cecil: Chaordic Learning: A Case Study. In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering Education and Training Track*, pages 87–96, IEEE, 2017.

[Krusche and Seitz 2018] KRUSCHE, Stephan ; SEITZ, Andreas: ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 284–289, 2018.

[Krusche et al. 2017b] KRUSCHE, Stephan ; SEITZ, Andreas ; BÖRSTLER, Jürgen ; BRUEGGE, Bernd: Interactive Learning: Increasing Student Participation through Shorter Exercise Cycles. In: *Proceedings of the 19th Australasian Computing Education Conference*, pages 17–26, 2017.

[Krusche et al. 2017c] KRUSCHE, Stephan ; VON FRANKENBERG, Nadine ; AFIFI, Sami: Experiences of a Software Engineering Course based on Interactive Learning. In: *Tagungsband des 15. Workshops "Software Engineering im Unterricht der Hochschulen"*, pages 32–40, 2017.

[Lawrance et al. 2013] LAWRANCE, Joseph ; JUNG, Seikyung ; WISEMAN, Charles: Git on the Cloud in the Classroom. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages 639–644, 2013.

[McCracken et al. 2001] MCCRACKEN, Michael ; ALMSTRUM, Vicki ; DIAZ, Danny ; GUZDIAL, Mark ; HAGAN, Dianne ; KOLIKANT, Yifat Ben-David ; LAXER, Cary ; THOMAS, Lynda ; UTTING, Ian ; WILUSZ, Tadeusz: A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In: *Working Group Reports on Innovation and Technology in Computer Science Education*, pages 125–180, 2001.

[Pieterse 2013] PIETERSE, Vreda: Automated Assessment of Programming Assignments. In: *Proceedings of the 3rd Computer Science Education Research Conference*, pages 45–56, 2013.

[Poženel et al. 2015] POŽENEL, Marko ; FÜRST, Luka ; MAHNIČ, Viljan: Introduction of the automated assessment of homework assignments in a university-level programming course. In: *38th International Convention on Information and Communication Technology, Electronics and Microelectronics*, pages 761–766, IEEE, 2015.

[Robins et al. 2003] ROBINS, Anthony ; ROUNTREE, Janet ; ROUNTREE, Nathan: Learning and teaching programming: A review and discussion. In: *Computer Science Education* 13, pages 137–172, 2003.

[Staubitz et al. 2015] STAUBITZ, Thomas ; KLEMENT, Hauke ; RENZ, Jan ; TEUSNER, Ralf ; MEINEL, Christoph: Towards practical programming exercises and automated assessment in Massive Open Online Courses. In: *Teaching, Assessment, and Learning for Engineering*, pages 23–30, IEEE, 2015.

[Striewe and Goedicke 2013] STRIEWE, Michael ; GOEDICKE, Michael: Analyse von Programmieraufgaben durch Softwareproduktmetriken. In: *Tagungsband des 13. Workshops "Software Engineering im Unterricht der Hochschulen"*, pages 59–68, 2013.