André Greubel, Sven Strickroth, Michael Striewe (Hrsg.)

Proceedings of the Fifth Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2021)

28. und 29. Oktober 2021
virtuelles Event

In Zusammenarbeit mit der Fachgruppe Bildungstechnologien der GI e.V.

Vorwort

Das Erlernen der Programierung ist weiterhin ein relevantes Themenfeld, vor allem in der Informatik und informatiknahen Studiengängen. Die in diesen Studienfächern auch weiterhin stark steigenden Studienzahlen stellen dabei besondere Anforderungen an die Korrektur und Feedbackgenerierung von bzw. zu Programmieraufgaben.

Die Diskussion dieser Anforderungen und neuer und bewährter Ansätze, ihnen gerecht zu werden steht auch in diesem Jahr im Zentrum des fünften Workshops zur „Automatischen Bewertung von Programmieraufgaben“. Wie in den vorherigen Auflagen des Workshops wurde dazu wieder ein zweitägiges Programm zusammengestellt, das in diesem Jahr allerdings nicht wie gewohnt als physische Zusammenkunft absolviert wurde, sondern als virtuelle Veranstaltung.

Insgesamt wurden in diesem Jahr elf Beiträge zur Begutachtung eingereicht. Aus diesen wurden schließlich sieben Beiträge für die Präsentation auf dem Workshop ausgewählt, die verschiedene Facetten dieser Anforderungen verfolgten. In einer ersten Session wurden Entwicklungen bzgl. der Architekturen verschiedener Assessment-Tools für Programmieraufgaben erläutert. Die zweite Session skizzierte neuartige Einsatzzenarien der automatischen Bewertung von Programmieraufgaben. Die Beiträge der letzten Session stießen schließlich in eine bisher wenig beachtete Richtung vor: Der Identifizierung und Quantifizierung verschiedener Eigenschaften von Programmieraufgaben und -lösungen.

Ergänzt wurde dieses Programm durch eine Keynote von Prof. Dr. Andreas Mühling, der aus Sicht eines Fachdidaktikers über Chancen und Risiken beim Einsatz verschiedener Rückmeldungssysteme sprach und einen Bogen von den Anfängen automatischer Bewertungssysteme bis zu einem Ausblick für die Zukunft schlug. Damit sorgte er für eine rege Diskussion über didaktische und praktische Ziele und wie diese in Einklang zu bringen sind.

Wir bedanken uns bei allen Einreichenden und den Mitgliedern des Programmkomitees für die Erstellung bzw. das Review der Beiträge sowie allen Teilnehmenden des Workshops. Ohne dieses gemeinsame Engagement wäre der Workshop so nicht möglich!

André Greubel, Sven Strickroth, Michael Striewe
November 2021

Organisation

André Greubel (Julius-Maximilians-Universität Würzburg)
Sven Strickroth (Ludwig-Maximilians-Universität München)
Michael Striewe (Universität Duisburg-Essen)

Programmkommitee

Oliver Bott (Hochschule Hannover)
Torsten Brinda (Universität Duisburg-Essen)
Ralf Dörner (RheinMain University of Applied Sciences)
Robert Garmann (Hochschule Hannover)
Rainer Oechsle (Hochschule Trier)
Niels Pinkwart (Humboldt-Universität zu Berlin)
Uta Priss (Ostfalia Hochschule für angewandte Wissenschaften)
Ulrik Schroeder (RWTH Aachen)
Andreas Schwill (Universität Potsdam)

Weitere Reviews:

Jens Doveren (RWTH Aachen)

Inhaltsverzeichnis

Wir können es, aber sollen wir es auch? – Ein didaktischer Blick auf automatisierte Bewertung von Programmieraufgaben

Andreas Mühling[1]

Abstract: Didaktische Entscheidungen sind in den meisten Fällen keine eindeutigen ja/nein Entscheidungen. Vielmehr sind der Kontext einer Maßnahme, das intendierte Ziel und speziell auch die Details der Umsetzung meist ausschlaggebend dafür, ob eine didaktische Maßnahme für viele oder wenige Lernende sinnvoll funktioniert, den Anderen nicht weiter schadet oder aber vielleicht für einige oder viele Lernende sogar kontraproduktiv ist. Typische und sinnvolle Ziele wie Effizienz und Effektivität, Ökonomie und Lehrqualität stehen darüber hinaus oft in einem unauflösbaren Widerspruch zueinander so dass auch hier nur eine individuelle Abwägung der Möglichkeiten zu einer Entscheidung für oder wider einer didaktischen Maßnahme führen kann.

Der Vortrag stellt zunächst relevante fachdidaktische Forschung zum Programmierunterricht und allgemeine Erkenntnisse zu Feedback und Lernen vor. Er geht dabei der Frage nach, unter welchen Bedingungen und mit welchen Zielsetzungen eine automatisierte Bewertung im Kontext von Programmieraufgaben (un-)sinnvoll erscheint. Darauf aufbauend werden Implikationen einerseits für die Technologie, andererseits aber auch für deren Einsatz speziell für die Lehre in Einführungsveranstaltungen der Informatik abgeleitet.

[1]Universität Kiel, andreas.muehling@informatik.uni-kiel.de

Improving the Scalability and Security of Execution Environments for Auto-Graders in the Context of MOOCs

Sebastian Serth[1], Daniel Köhler[1], Leonard Marschke[1], Felix Auringer[1], Konrad Hanff[1], Jan-Eric Hellenberg[1], Tobias Kantusch[1], Maximilian Paß[1], Christoph Meinel[1]

Abstract:

Learning a programming language requires learners to write code themselves, execute their programs interactively, and receive feedback about the correctness of their code. Many approaches with so-called auto-graders exist to grade students' submissions and provide feedback for them automatically. University classes with hundreds of students or Massive Open Online Courses (MOOCs) with thousands of learners often use these systems. Assessing the submissions usually includes executing the students' source code and thus implies requirements on the scalability and security of the systems. In this paper, we evaluate different execution environments and orchestration solutions for auto-graders. We compare the most promising open-source tools regarding their usability in a scalable environment required for MOOCs. According to our evaluation, Nomad, in conjunction with Docker, fulfills most requirements. We derive implications for the productive use of Nomad for an auto-grader in MOOCs.

Keywords: Auto-Grader; Scalability; MOOC; Programming; Security; Execution

1 Introduction

The acquisition of fundamental digital knowledge is often an essential prerequisite for confident and self-determined participation in today's world. Gaining programming skills, in particular, is usually seen as an integral part of this knowledge. Learners can only acquire these skills through hands-on practice. While a teacher can personally instruct individuals from small groups, frequent manual feedback becomes laborious for larger learning groups and is impossible for Massive Open Online Courses (MOOCs) with thousands of learners. Hence, automated feedback, based on unit tests provided by the course instructors, is required in these self-paced learning scenarios.

So-called auto-graders automatically assess code submissions and often additionally provide a development environment to the learners. These online tools do not require any local software besides a web browser, which allows learners to get started quickly. Those tools execute the learners' code to provide the desired functionality and grade submissions. As outlined in Sect. 2, most auto-graders run the potentially untrusted code on their servers,

[1] Hasso Plattner Institute, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany, {sebastian.serth, daniel.koehler, leonard.marschke, christoph.meinel}@hpi.de, {felix.auringer, konrad.hanff, jan-eric.hellenberg, tobias.kantusch, maximilian.pass}@student.hpi.de

doi:10.18420/abp2021-1

imposing security and scalability questions for the provider. To improve the auto-grader that we currently use in our MOOCs, we performed a requirement analysis (cf. Sect. 3) and compared existing industry-standard execution environments in Sect. 4. We conclude our paper with an outlook on future work (Sect. 5) and summarize our findings in Sect. 6.

1.1 Current Architecture of our Auto-Grader CodeOcean

Nowadays, our auto-grader CodeOcean [St16] is used by several MOOC platforms and as part of teaching activities at our faculty. It hosts more than 650 exercises and typically runs around 2.5 million code executions per month during the duration of a MOOC. At this scale, performance and security aspects are important factors to consider. As shown in Fig. 1, the system consists of two major parts: the CodeOcean application providing the frontend and handling of submissions as well as a custom micro-service called DockerContainerPool managing pre-warmed, idling containers to improve the response time for learners.

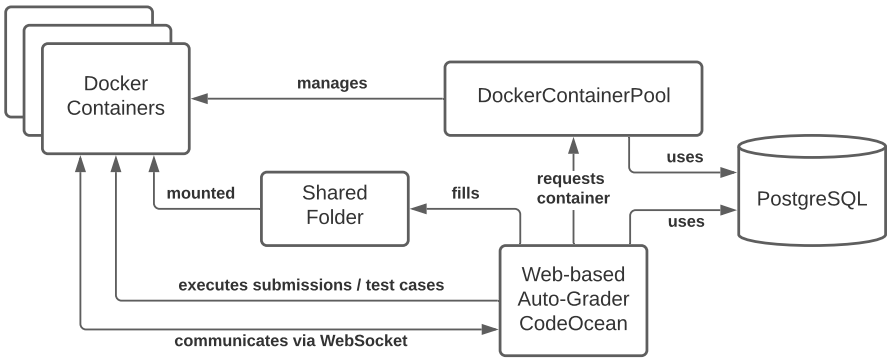

Fig. 1: The current architecture of our system consisting of two custom services for the code execution. The micro-service DockerContainerPool manages and pre-warms Docker containers while the CodeOcean application provides all user-facing interactions.

Even though the chosen architecture extracts the handling of Docker containers into a dedicated micro-service, the main application CodeOcean is tightly coupled with the DockerContainerPool. Moreover, the shared folder imposes additional security challenges for production use and makes scaling more difficult. Hence, a modernized solution is desired as a foundation for future development and educational research (cf. Sect. 5).

2 Background and Related Work

Automatically assessing code submissions using auto-graders and the implications of executing learners' code on a server have been the subject of several research projects. Two different approaches are distinguished for assessing code submissions: static and

dynamic code analysis [Ga13]. A trusted tool is used during the static analysis to inspect the submission without executing the analyzed code. For dynamic assessment, the code is run while the behavior is observed and compared with pre-defined expectations. In the context of auto-grader solutions, Garmann, as well as Strickroth, have described security implications and considerations when executing learner's Java code [Ga13, St19].

To isolate the code execution, web-based applications such as codeboard.io, the ranna code runner[2] or the grader Praktomat often rely on Docker containers [BHS16]. Docker is a containerization technique based on isolation concepts from the Linux kernel not requiring a virtual machine (VM) [An15]. Docker, as well as competing software, uses the containerd runtime[3] providing a standard interface to exchange the underlying container technologies. By introducing Firecracker as a so-called MicroVM with minimal hardware virtualization, Agache et al. increased the isolation of containers from the host intending to achieve comparable performance to containerized deployments [Ag20]. Similarly, Kata Containers provide full-featured VMs with support for container specifications and interfaces [RT19].

Scaling multiple containers across various hosts is done by so-called orchestrators to increase performance. Kubernetes and Nomad are two open-source tools scheduling and managing container clusters on multiple nodes[4]. Orchestrators allow horizontal scaling and to run jobs on-demand and are therefore relevant technologies for large-scale auto-graders.

For learning environments, even further approaches can overcome the resource and security constraints on the provider's server. Sharrock et al. demonstrated the technical feasibility of loading a sufficient Linux environment in the users' browser for use in a MOOC [SAH18]. However, to comply with our current architecture approach as well as to fulfill our interest in gathering data from the executions for further research, we aim to use approaches which allow us to keep the sovereignty about the execution with our system.

3 Requirement Analysis

Based on our existing auto-grader and experiences during past courses, we identified various requirements with the user-centered Design Thinking approach [MLP11]. Following this approach, we conducted qualitative, semi-structured interviews with a dozen representatives from identified target groups (learners, teachers, and administrators). Additionally, we inspected usage data of CodeOcean ranging back to first courses in 2015. Consequently, we identified the following three general requirements that any solution has to offer:

(1) Interactivity: Our auto-grader provides feedback for test runs and executes the learners' code in real-time. Therefore, offering a synchronous channel between the user and the execution shall allow a new command to start in less than one second.

[2] Source code of the ranna code runner on GitHub: `https://github.com/ranna-go/ranna`

[3] Homepage of the containerd project: `https://containerd.io`

[4] Kubernetes homepage: `https://kubernetes.io`, Nomad homepage: `https://www.nomadproject.io`

(2) Scalability: In the past, our auto-grader handled up to 120 execution requests per second with a mean execution time of fewer than ten seconds. While the system hits these peak usages only during a MOOC, these numbers provide a rough estimation for scalability requirements.

(3) Flexibility: Specifying runtime dependencies for containers should be as simple as with the current Docker-based solution and hence be compliant to the containerd ecosystem.

In addition to providing isolated processes and fulfilling these requirements, administrators urged the need to limit resources for code executions on their servers. Specifically, the execution time, memory consumption, and CPU should be limitable to provide a stable service for all users. As only a few exercises require network access, it should be generally restricted and only selectively enabled. Network access includes access to other hosts or forwarding specific ports to the learner. Optionally, allowing additional network orchestration with multiple nodes (e. g., for providing hands-on firewall exercises) would be a plus.

4 Evaluation of Code Execution Environments

Considering the requirements described in the previous Sect. 3, we practically evaluated containerization technologies and orchestrators (as described in Sect. 2) for the use by our auto-grader. We checked the extent to which each system meets our requirements and how extensive the effort for initial integration and following regular maintenance would be.

4.1 Execution Environments

As Docker, Firecracker, and Kata Containers share a similar feature set, we chose those technologies for our main comparison. Except for specifying an execution time limit, all three tools meet the technical demands for the required interactivity (1) once running. Docker and Firecracker also include a REST API easing the integration with a web-based auto-grader and providing a constant input and output stream to the containers.

When comparing the scalability (2), significant differences between the systems stand out. We measured some key metrics when using a Python container commonly used by our auto-grader on our production-grade system and under comparable, real-world conditions. For our experiment setup, we recorded the time for a container to complete the initialization and enter the idle waiting state (e. g., by subsequently executing the `sleep` command). Furthermore, we calculated the memory consumption of these idling containers. As shown in Tab. 1, Firecracker has the lowest memory footprint and the fastest startup time, followed by Docker. In repeating measurements with 10 and 100 concurrent container requests, we observed a linear increase in RAM usage for both. Kata Containers, in turn, had a significant increase in memory consumption, and we were unable to create 100 instances at once.

	1 Container		10 Containers		100 Containers	
	Time	RAM Usage	Time	RAM Usage	Time	RAM Usage
Docker	0.7s	47 MB	2.6s	333 MB	20.3s	3,244 MB
Firecracker	0.2s	35 MB	0.3s	375 MB	2.3s	3,900 MB
Kata Containers	3.2s	150 MB	7.0s	1,500 MB	Resource Limit Hit	

Tab. 1: The startup time for 1, 10, and 100 requested containers with different containerization technologies and the overall memory consumption when all containers are idling.

We assume that the underlying isolation technologies cause the observed differences. The setup bases Kata Containers on VMs requiring about 150 MB of RAM each. We conclude that they don't meet our second requirement, scalability, primarily due to the relatively slow startup speed. Similarly to Firecracker, they provide more robust isolation than Docker, which auto-graders would benefit from.

The third requirement, flexibility, refers to creating a new container environment for different programming languages. Docker (and Kata Containers) rely on a single so-called `Dockerfile` describing steps to build container images automatically. Each new container request uses these container images for the initialization process. Firecracker with the microVM concept requires two files: A bootable Linux kernel binary and a file system image. Even though different containers can reuse the Linux kernel and creating the required file system image can be automated, we argue that this process is more complex than the steps required for Docker. Combined with the low-level API of Firecracker, it makes using the tool more difficult even though it provides more robust isolation compared to Docker. Hence, for now, we decided against using it in our research-driven environment due to the increased setup and maintenance effort as well as the missing native support for `Dockerfiles`.

4.2 Execution Orchestrators

Similar to containerization technologies, we evaluated Kubernetes and Nomad for the desired use case in CodeOcean. Our main goal by using one of those orchestrators is to distribute the expected workload of 120 container requests per second to multiple machines with minimal manual intervention. Since the orchestrators abstract from the underlying execution technology, they also need to support the desired requirements (cf. Sect. 3). The two open-source tools evaluated both support the features mentioned above.

Therefore, we were most interested in the performance of both tools and the maintenance effort for researchers involved in the system management. For the performance evaluation, we repeated the experiment described in the previous Sect. 4.1 and focused on the startup time as the relevant metric. We deployed each tool to the same infrastructure for our assessment, with two nodes hosting Docker containers and one dedicated control plane managing the two systems. Tab. 2 shows the increase in startup times based on the number of requested containers for Nomad and Kubernetes. Interestingly, it took Kubernetes almost

eight times as long as Nomad to launch 100 containers. Even when using a managed cluster of the Google Cloud Platform with considerable more total resources, the startup times were not significantly lower or even worse as in our environment.

	Time for 1 container	Time for 10 containers	Time for 100 containers
Nomad	2.5s using 1 node	4.0s using 2 nodes	10.0s using 2 nodes
Kubernetes	3.0s using 1 node	8.0s using 2 nodes	78.0s using 2 nodes

Tab. 2: The startup time for 1, 10, and 100 requested containers using two popular container orchestrators and the number of nodes used for executing these containers.

Initially, we expected Kubernetes to outperform Nomad as it uses containerd directly rather than relying on Docker using containerd (which Nomad does). However, a Site Reliability Engineer daily working with Kubernetes explained that we were most likely observing the difference in scheduling algorithms between the two solutions.

Throughout the evaluation of Kubernetes and Nomad, we also observed the complexity of the tools during installation and configuration. Due to the vast amount of additional features which are mostly not required for our use-case, we identified Kubernetes to be more complex to configure. Since both tools show comparable performance otherwise, we decided for Nomad based on the differences in the startup time and the more straightforward configuration. The decision to use Nomad also means that we have to use Docker for now, as adapters to other containerization solutions are not yet production-ready.

4.3 Derived Implications for the Production System

Revisiting our current architecture (cf. Sect. 1.1) and the evaluation results, we created a proof of concept using Nomad and Docker containers. The architecture shown in Fig. 2 abstracts the handling of containers from our auto-grader CodeOcean and leverages this to a newly introduced middleware and the Nomad orchestrator. Most importantly, files are no longer provided to a shared folder mounted in the Docker container but instead copied to the container through Nomad. Additionally, the executor middleware abstracts from the concrete Nomad interface by providing a simplified representation of available execution environments. This detached architecture improves the system's scalability and increases the security of the containers through more robust isolation.

Hence, we plan to develop the executor middleware as a replacement for the current DockerContainerPool service. Based on the experiences we gained through our proof of concept, this middleware will primarily translate requests between the auto-grader and Nomad by abstracting the underlying container and orchestration technologies. Our goal is that the new service manages the pre-warming of containers, i. e., building and launching suitable containers for a single user. By providing separate environments for different users, their submissions cannot interfere with one another. Furthermore, the new service should enforce the time and resource constraints rather than leaving this to our auto-grader.

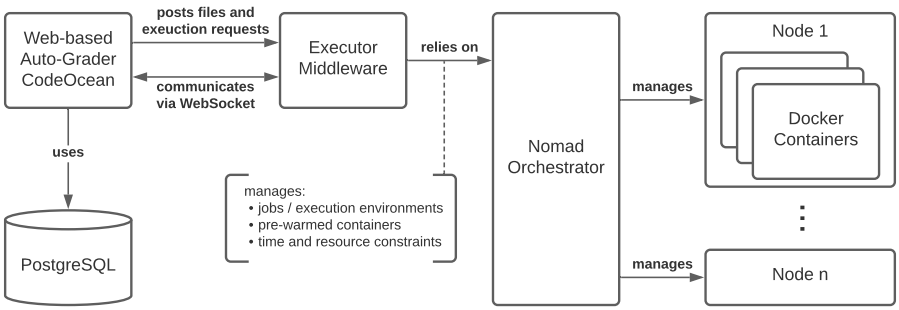

Fig. 2: The target architecture of CodeOcean consisting of the auto-grader, a custom execution middleware and a Nomad cluster. The main responsibility of the execution middleware is the management of desired states in Nomad and an abstraction of using pre-warmed containers.

5 Future Work

The proposed architecture and the insights collected from our evaluation are a foundation for future developments. The abstraction of executions allows more diverse environments as requested by various teaching teams in the past. Further, we want to investigate how course instructors can use CodeOcean in courses requiring specific hardware (such as a GPU or a Raspberry Pi). Besides that, we want to extend the auto-grading capabilities for networking and internet security courses by providing students access to network scenarios. A starting point for subsequent research could be integrating the interactive web page supplied by CodeOcean or remote shell access to one of the containers.

With the implementation of the described design and by achieving better scalability, we further provide the preconditions for longer-lasting executions. For example, we would like to offer advanced programming courses introducing learners to a debugger and give the required tools within our web-based programming environment CodeOcean. Therefore, we will add support for remote debugging of executions within a container and evaluate its impact on learners. On a technical level, we want to assess different pre-warming strategies and measure their effectiveness for handling varying workloads during a MOOC. Additionally, we plan further to compare other programming languages and their security concepts to develop our auto-grader and the new executor middleware.

6 Conclusion

In this paper, we approach enhancing our web-based, interactive auto-grader for MOOCs to enable improved security and higher scalability. From qualitative interviews with learners, teachers, and administrators, we identified a list of three main requirements: (1) interactivity, (2) scalability, and (3) flexibility. Considering these requirements, we evaluated several containerization and orchestration technologies to provide secure execution environments.

According to our analysis, Docker and Firecracker are the most promising container solutions and significantly more efficient than Kata Containers. When comparing Kubernetes and Nomad in conjunction with Docker containers for improved scalability across multiple nodes, we identified Nomad as the more performant solution for an auto-grader.

Finally, we propose a generalized architecture using Nomad and Docker for providing isolated and scalable execution environments in the context of CodeOcean. A newly introduced micro-service abstracts the management of containers and offers a simplified interface for the auto-grader. The resulting architecture provides a solid foundation for current and future use cases, thereby fostering the programming education in MOOCs.

Bibliography

[Ag20] Agache, A.; Brooker, M.; Florescu, A.; Iordache, A.; Liguori, A.; Neugebauer, R.; Piwonka, P.; Popa, D.-M.: Firecracker: Lightweight Virtualization for Serverless Applications. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20). USENIX Association, Santa Clara, CA, USA, p. 17, February 2020.

[An15] Anderson, C.: Docker [Software Engineering]. IEEE Software, 32(3):102–c3, May 2015.

[BHS16] Breitner, J.; Hecker, M.; Snelting, G.: Der Grader Praktomat. In: Automatisierte Bewertung in der Programmierausbildung. volume Digitale Medien in der Hochschullehre, Waxmann Verlag, Münster, Germany, pp. 159–172, July 2016.

[Ga13] Garmann, R.: Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito. In: Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2013). volume 1067 of CEUR workshop proceedings, CEUR-WS.org, Hannover, Germany, p. 6, October 2013.

[MLP11] Meinel, C.; Leifer, L.; Plattner, H., eds. Design Thinking. Understanding Innovation 1. Springer, Berlin, Heidelberg, 2011.

[RT19] Randazzo, A.; Tinnirello, I.: Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE, Granada, Spain, pp. 209–214, October 2019.

[SAH18] Sharrock, R.; Angrave, L.; Hamonic, E.: WebLinux: A Scalable in-Browser and Client-Side Linux and IDE. In: Proceedings of the Fifth Annual ACM Conference on Learning at Scale. ACM, London United Kingdom, pp. 1–2, June 2018.

[St16] Staubitz, T.; Klement, H.; Teusner, R.; Renz, J.; Meinel, C.: CodeOcean - A Versatile Platform for Practical Programming Excercises in Online Environments. In: 2016 IEEE EDUCON. IEEE, Abu Dhabi, pp. 314–323, April 2016.

[St19] Strickroth, S.: Security Considerations for Java Graders. In: Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2019). Gesellschaft für Informatik e.V. (GI), Essen, Germany, pp. 35–43, October 2019.

An Architecture for the Automated Assessment of Web Programming Tasks

Lara Aubele[1], Leon Martin[1], Tobias Hirmer[1], Andreas Henrich[1]

Abstract: Automatically assessing students' solutions to programming tasks in the domain of web programming requires special means due to the characteristics of web applications. This paper proposes an architecture for a web-based learning application tailored to this domain. For the implementation of the automated assessment of programming tasks, we make use of end-to-end testing and container virtualization. This allows, in contrast to other popular approaches, the coverage of tasks that include special operations like DOM manipulations, which alter the user interface of web applications, in a way that is convenient for both students and instructors. We demonstrate the capabilities and limitations of the architecture based on two common usage scenarios.

Keywords: Web programming; Automated Assessment; End-to-end testing; Docker

1 Motivation

In both academia and industry, the interest in web applications is on the rise. Due to the characteristics of web programming, students face special challenges when they approach web application development. This includes aspects like the comparatively high number of involved languages with the most prominent ones being HTML, CSS, and JavaScript, and the interaction between scripts and the user interface on the client side. Here, JavaScript is used to alter and retrieve information about the elements present in the Document Object Model (DOM), the underlying skeleton of a web application, which is defined in HTML. This relationship impedes testing web applications as the effects of scripts exceed single units. This motivates the need for a suitable learning application that is able to automatically assess students' solutions to web programming tasks that address the aforementioned aspects. Hence, in this paper, we propose an architecture for a learning application tailored to the web programming domain. The application is envisaged to complement the blended learning concept of the chair of media informatics at the University of Bamberg[2]. In addition to a literature review, a small number of qualitative interviews[3] has been conducted beforehand to gather the requirements for such an application. The architecture and technology choices

[1] University of Bamberg, Media Informatics Group, An der Weberei 5, 96047 Bamberg, Germany lara.aubele@stud.uni-bamberg.de, {leon.martin|tobias.hirmer|andreas.henrich}@uni-bamberg.de

[2] See `https://www.uni-bamberg.de/minf` (Accessed: 04.05.2021). Major parts of the present paper are based on Lara Aubele's master's thesis written at the Chair of Media Informatics.

[3] Overall eight interviews with students and one interview with a lecturer of a web technologies course have been conducted. The interviews comprise a mixture of close-ended and open-ended questions and took about 45 minutes on average.

doi:10.18420/abp2021-2

are made with respect to them. To retain focus, this paper concentrates on aspects related to the automated assessment of web programming tasks.

The paper is organized as follows: Sect. 2 investigates popular web programming learning applications and describes technologies central for our architecture. Then, the architecture itself is introduced in Sect. 3. Subsequently, Sect. 4 demonstrates its limitations and capabilities based on two scenarios, followed by remarks on administrative aspects and future work. Finally, Sect. 5 draws a conclusion.

2 Related Work & Foundations

There are many providers that offer interactive applications for learning web programming for free. However, none of them meets all of our requirements. For instance, many applications do not support adding new tasks to their curriculum, which is an essential feature for our purposes. Furthermore, the tasks in our learning application shall be part of small but still realistic projects. Providers like Codecademy[4] teach JavaScript, HTML, and CSS on a single-file level, thus focusing on the syntax and capabilities of the languages rather than the implementation of realistic projects. Another promising candidate is IntelliJ IDEA Edu[5], the educational edition of the popular IDE specifically designed for instructor-student scenarios. While instructors are able to create own courses with tasks within realistic projects, the application has to be installed locally, though. This is contrary to the interviewees' wish that they can use the application on multiple devices with their progress and code being stored but without managing multiple installations.

Another flaw of IntelliJ IDEA Edu, Codewars[6], and some approaches described in previous work [OKP17; Sc17] is their focus on unit testing [Ru06] for checking if implemented code fragments run as expected and specified in task descriptions. However, unit tests cannot cover all operations required in full web applications. The reason for this is that the effects of operations like DOM manipulations can only be observed and thus tested when the code is executed in a Browser-like environment. Here, end-to-end (E2E) testing comes to the rescue. E2E testing is capable of testing applications that comprise multiple components by observing their overall behavior [Le16; Ts01]. Applied to web programming, E2E tests specify and test the expected state of the DOM that is defined using HTML and manipulated using JavaScript. If the DOM does not show the expected effects, the tests fail, thus providing feedback in the form of failed test cases and their descriptions.

Besides testing, some approaches [Gr17] use Docker[7], a container virtualization technology that allows running applications disregarding the underlying operating system or installed programs. For this purpose, the eponymous containers provide all necessary dependencies

[4] `https://www.codecademy.com` (Accessed: 12.05.2021)
[5] `https://www.jetbrains.com/de-de/idea-edu` (Accessed: 12.05.2021)
[6] `https://www.codewars.com` (Accessed: 12.05.2021)
[7] `https://www.docker.com` (Accessed: 12.05.2021)

such that containerized applications run the same on different machines [Be14]. In contrast to virtual machines, Docker containers virtualize the operating system and not the hardware. As a result, the code within containers is not directly executed on the host machine, thus adding an extra layer of security [Bu15][8]. Thereby, Docker allows executing code produced by users in a learning application's backend with low risk. In addition, networks of and therefore interaction between Docker containers can be set up via container ports which facilitates the implementation of modular applications. For more advanced applications, technologies like Docker in Swarm mode[9] and Kubernetes[10] provide cluster management and orchestration capabilities.

3 Concept

Regarding the learning application's architecture there are two main aspects to consider. First, the central procedure for the automatic assessment of web programming tasks has to be determined (Sect. 3.1). Afterwards, the architecture of our application is introduced, in which the testing procedure is embedded (Sect. 3.2).

3.1 Testing Procedure

We consider two options for the testing procedure, i.e., the procedure that automatically assesses the users' code. For the first option, the application generates dedicated Docker images for running and testing the user code for each task upon creation by the instructor. When a user request for testing their code arrives, a container for running the user code, a container for running the tests, and a network between them are created dynamically based on the images. The usage of separate containers for running and testing follows from the way E2E testing frameworks in the web programming domain operate, where the user code has to be executed in a browser (first container) before the e2e testing framework can access the resulting application and perform the tests (second container).

For proper execution, a `docker-compose` file is used to ensure that the container for running the user code is ready before the testing container is started. This option provides the advantage that no unused Docker containers are present in the system, thus reducing the memory usage. However, experiments showed that the time required for starting the containers on demand can exceed several tens of seconds.

The second option leverages a cluster of containers that is prepared in advance for running and testing the user code. Each node of the cluster comprises a container for running the

[8] Still, Docker is not completely secure and exploits like Docker Escape Attacks can provide the advisory with root privileges [JC17] but this is beyond the scope of this paper.

[9] `https://docs.docker.com/engine/swarm` (Accessed: 31.05.2021)

[10] `https://kubernetes.io` (Accessed: 31.05.2021)

user code and a container for running the tests connected via a static network. When a user request for testing their code arrives, the assessment task is assigned to an available node of the cluster where the user code is executed and tested. While this option is significantly faster due to the pre-built containers, the application demands more memory in idle operation compared to the first option. Additionally, the pre-built containers are less flexible and require more space because the underlying images need to contain the dependencies necessary for any of the programming tasks. Otherwise, the containers would not be equally capable of running or testing user code which is necessary for this option. For this reason, it is necessary to restart the cluster with updated images if instructors add new tasks that rely on additional dependencies.

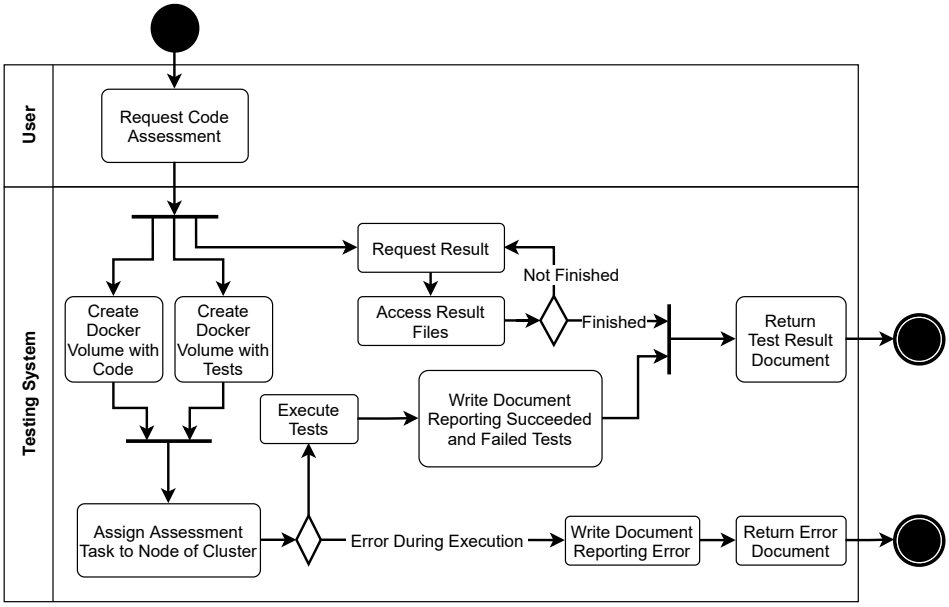

Fig. 1: An activity diagram visualizing the employed testing procedure.

Considering the application's intended use and in accordance with the statements made by the interviewees, we value faster execution higher than a bigger memory footprint. We also expect that the number of simultaneous users will not exceed 25 which is comfortably manageable memory-wise[11]. As a result we settled on the second option for our application. Fig. 1 shows an activity diagram illustrating the workflow of the application's testing procedure. As depicted, the procedure has one entry point, i.e., a user requesting the

[11] According to our tests, a Docker container running a simple React app uses about 550 MiB of memory based on the `node:16` Docker image, and a Docker container executing E2E tests about 300 MiB based on the `cypress/included:3.2.0` Docker image. Note that programming tasks related to React apps are one of the most demanding tasks in terms of both web programming proficiency and memory usage that we want to cover. Even in the unlikely case that all 25 users request the assessment of their React app simultaneously, which thus represents the worst case, only about 21 GiB of memory will be used for the containers in total.

assessment of their code which triggers the testing system. Note that aside from students, instructors shall also be able to send such requests for checking the correctness of tasks and tests. Following the request, the testing procedure is traversed, eventually yielding one of two terminal states:

Internal error This terminal state occurs when the testing system itself encounters an error, e.g., a crashing container, during running or testing the user's code. Obviously, this state should never occur which motivates the usage of proper cluster management tools. In this case, a document reporting the failure is produced and returned.

Tests completed When the assigned node of the cluster finishes the tests, a document reporting the succeeded and failed tests is produced. Based on the failed test cases and their descriptions, students can improve their solution. Hence, the test cases and descriptions have to be written with high detail and care by the instructors. The document is then returned representing the other terminal state.

3.2 Architecture & Implementation

Based on the chosen testing procedure, we derived the architecture for our learning application that comprises four components. This section describes the responsibilities of each component and the technologies that we employ to implement them.

Our envisaged learning application uses several types of data. There is user data including information about the users' progress and their credentials, among others. Furthermore, there are task descriptions, task-specific code templates, and task-specific E2E tests as well as templates for Docker images. Regarding the database, no high degree of flexibility is required for the application, suggesting the usage of a standard relational database as the *Database* component. In particular, MariaDB[12], an open source SQL database, was chosen for the implementation.

For platform independence and ease of use, the learning application itself will be implemented as a web application. As to that, the *WebInterface* component naturally provides a means of interaction between the users, i.e., students and instructors, and the learning application. Its main purposes are to present the available tasks and the according task descriptions to the students, to provide an IDE where students can prepare their solutions, to allow the submissions of solutions for the automated assessment, and to display the assessment results[13]. We use the frontend library React[14] to implement the web interface. However, instead of creating our own web IDE, we leverage the Stackblitz Platform API[15] which allows embedding Stackblitz's feature-rich IDE in other web applications.

[12] `https://mariadb.org` (Accessed: 28.05.2021)

[13] There are additional features like a forum for communication between the users. However, they are beyond the scope of this paper and therefore omitted here.

[14] `https://reactjs.org` (Accessed: 28.05.2021)

[15] `https://developer.stackblitz.com/docs/platform` (Accessed: 28.05.2021)

Regarding the automated assessment of the web programming tasks, the *TaskAssessment* component represents the most important component as it houses the cluster for running and testing the user code. For setting up and managing the cluster we use Docker in Swarm mode which provides vital capabilities like load balancing and container management. This setup is compatible to the employed E2E testing framework Cypress[16]. The testing procedure itself has already been described and illustrated in Sect. 3.1.

At the application's core, the central *BackendAPI* component acts as an intermediary between the other components. Hence, it is responsible for retrieving required data from the database via an SQL interface and sharing it with other components. For communication with the WebInterface, it exposes a RESTful API implemented using Express[17]. This way, it also receives the user requests containing their code for the automated assessment.

4 Discussion

Currently, there only exists a prototypical implementation that lacks several features required for production purposes. This limits the possible depth in terms of evaluating the application and the underlying architecture. For this reason, we first walk through two common scenarios for the learning application demonstrating its capabilities and limitations, and then summarize more administrative aspects afterwards.

Scenario 1: Description As an instructor I want to create a programming task featuring a DOM manipulation so that my students can train their understanding of the interaction between JavaScript and the DOM. The task's goal shall be to implement a function that creates an `h1`-element reading *Success!* when a button is clicked on the given web page, thereby altering its DOM.

Scenario 1: Procedure The instructor first creates a project containing the solution for the task locally. Then, they write E2E tests which test if the `h1`-element is created correctly when the test triggers the `onclick`-method of the button. The test cases and their (static) descriptions[18] have to be written with high care since they represent the feedback that students will receive. Afterwards, a duplicate project is created where the parts of the code that students will have to implement are removed. All previously successful tests should fail in this project. Then, the instructor logs into the *WebInterface* using an instructor account and navigates to the administration interface where the solution, the prepared project for the students and the tests are uploaded via file upload. The *BackendAPI* receives the files via its REST API and inserts them into the *Database*. The instructor can now add a description and ascribe the task to the appropriate section before making it available to the students.

[16] `https://www.cypress.io` (Accessed: 28.05.2021)
[17] `https://expressjs.com` (Accessed: 28.05.2021)
[18] In the future, investigating dynamically generated descriptions is worthwhile to provide more helpful feedback.

Scenario 2: Description As a student I want to continue solving a task I began earlier so that I can finish the task. The said task is the task that was created in Scenario 1.

Scenario 2: Procedure The student logs into the *WebInterface* as a regular user and selects the task from the according section. After revisiting the task description, the student uses the integrated web IDE to inspect the code they already produced which is retrieved from the *Database*. They issue the automated assessment of their solution by pressing the dedicated button. The *BackendAPI* receives the request, retrieves the tests for the task from the *Database*, and relays code and tests to the *TaskAssessment*. There, an available node of the cluster runs the code and evaluates the tests. Unfortunately, one test does not succeed. Hence, a document containing the failed test cases and their description is produced and returned, first to the *BackendAPI* and subsequently to the *WebInterface* where the information is presented to the user. Now, they can iteratively improve their code and issue the assessment until all tests succeed, i.e., the task is completed. Using the description of the failed test, they realize that their code creates a `p`-element instead of an `h1`-element. Hence, they edit their code accordingly and rerun the tests yielding a report without failed tests.

From an administrative point of view, the costs of maintenance and the computational facilities represent the central drawback of our solution. The cluster-based testing procedure and the database are the main contributors in this regard. Another open aspect is that E2E testing is a black-box testing approach, i.e., the exact steps that take place within the students' solutions cannot be tested but only the effects of the code. In contrast, white-box testing like unit testing would allow assessing the code quality automatically to a certain degree. Conveniently, Cypress also supports unit and even integration testing. Hence, one major topic for future work is to integrate these capabilities in our learning application. Finally, note that the architecture proposed in this paper is just a tool that allows testing solutions to web programming tasks while the learning success heavily depends on the quality of the test cases and test descriptions.

5 Conclusion

In this paper, we introduced an architecture for a learning application tailored to web programming. We employ the E2E testing framework Cypress and Docker in Swarm mode for implementing the automated assessment of programming tasks. This enables the application to support tasks that span multiple components of a web application like DOM manipulations where JavaScript code alters the user interface. Currently, there only exists a prototypical implementation that lacks several features required for productive usage. The missing features represent future work and include the integration of unit testing as noted in Sect. 4 and aspects omitted in this paper like a forum for communication among students and instructors.

References

[Be14] Bernstein, D.: Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Comput. 1/3, pp. 81–84, 2014, URL: https://doi.org/10.1109/MCC.2014.51.

[Bu15] Bui, T.: Analysis of Docker Security. CoRR abs/1501.02967/, 2015, arXiv: 1501.02967, URL: http://arxiv.org/abs/1501.02967.

[Gr17] Grummel, F.; Oechsle, R.; Brettle, A.; Schuster, D.: Automatische Bewertung von Python-Anwendungen(Automatic Evaluation of Python Applications). In (Strickroth, S.; Müller, O.; Striewe, M., eds.): Proceedings of the Third Workshop "Automatische Bewertung von Programmieraufgaben"(ABP 2017), Potsdam, Germany, October 5-6, 2017. Vol. 2015. CEUR Workshop Proceedings, CEUR-WS.org, 2017, URL: http://ceur-ws.org/Vol-2015/ABP2017%5C_paper%5C_04.pdf.

[JC17] Jian, Z.; Chen, L.: A Defense Method against Docker Escape Attack. In: Proceedings of the 2017 International Conference on Cryptography, Security and Privacy, ICCSP 2017, Wuhan, China, March 17 - 19, 2017. ACM, pp. 142–146, 2017, URL: https://doi.org/10.1145/3058060.3058085.

[Le16] Leotta, M.; Clerissi, D.; Ricca, F.; Tonella, P.: Approaches and Tools for Automated End-to-End Web Testing. Adv. Comput. 101/, pp. 193–237, 2016, URL: https://doi.org/10.1016/bs.adcom.2015.11.007.

[OKP17] Oster, N.; Kamp, M.; Philippsen, M.: AuDoscore: Automatic Grading of Java or Scala Homework. In (Strickroth, S.; Müller, O.; Striewe, M., eds.): Proceedings of the Third Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2017), Potsdam, Germany, October 5-6, 2017. Vol. 2015. CEUR Workshop Proceedings, CEUR-WS.org, 2017, URL: http://ceur-ws.org/Vol-2015/ABP2017%5C_paper%5C_01.pdf.

[Ru06] Runeson, P.: A Survey of Unit Testing Practices. IEEE Softw. 23/4, pp. 22–29, 2006, URL: https://doi.org/10.1109/MS.2006.91.

[Sc17] Schuster, D.; Brettle, A.; Oechsle, R.; Grummel, F.: Automatische Bewertung von JavaFX-Anwendungen(Automatic Evaluation of JavaFX Applications). In (Strickroth, S.; Müller, O.; Striewe, M., eds.): Proceedings of the Third Workshop "Automatische Bewertung von Programmieraufgaben"(ABP 2017), Potsdam, Germany, October 5-6, 2017. Vol. 2015. CEUR Workshop Proceedings, CEUR-WS.org, 2017, URL: http://ceur-ws.org/Vol-2015/ABP2017%5C_paper%5C_03.pdf.

[Ts01] Tsai, W.; Bai, X.; Paul, R. A.; Shao, W.; Agarwal, V.: End-To-End Integration Testing Design. In: 25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, 8-12 October 2001, Chicago, IL, USA. IEEE Computer Society, pp. 166–171, 2001, URL: https://doi.org/10.1109/CMPSAC.2001.960613.

Teaching Software Testing Using Automated Grading

Lukas Beierlieb, Lukas Iffländer, Tobias Schneider, Thomas Prantl, Samuel Kounev[1]

Abstract: Software testing has become a standard for most software projects. However, there is a lack of testing in many curricula, and if present, courses lack instant feedback using automated systems. In this work, we show our realization of an exercise to teach software testing using an automated grading system. We illustrate our didactic goals, describe the task design and technical implementation. Evaluation shows that students experience only a slight increase in difficulty compared to other tasks and perceive the task description as sufficient.

Keywords: automated grading; unit testing; test-driven development

1 Introduction

Over the previous decades, mandatory software testing became a de facto standard for most software projects. A current study from Jetbrains on the state of the developer ecosystem in 2020 [Je] shows that 87% of projects use some kind of software tests. However, when breaking that down to unit testing, only 68% remain, and only 48% of all developers automate half or more of their tests. Fully automated testing is only prevalent in a tenth of all software projects. Nevertheless, test-driven programming (TDD) is on the rise. Developers work a permanent iteration of the following steps: (i) write a test that fails—do not write more code that is necessary to fail (not compiling is a failure), (ii) write enough code to pass the test, (iii) if the test passes either refactor or repeat the steps for the next functionality. This approach simplifies modifications and refactoring since developers will have a test for every piece of code that they can run to check if the modification broke something.

We partially attribute this to a lack in academic education. A quote from David Carrington in 1997 still applies: “Testing is a relatively neglected component of the computer science curriculum.” [Ca96]. Most curricula only briefly touch the topics of testing inside a lecture on software engineering or project management. Testing exercises are rare often limited in scale and complexity. Furthermore, most existing approaches rely on manual grading or even printed code—the graders rarely run the submitted tests.

A manual solution to teach automated software testing is not feasible due to two reasons: (i) it hardly scales to large course sizes—a necessity to teach software testing early in a curriculum, (ii) manual graders make mistakes and often overlook errors, and (iii) the time

[1] University of Würzburg, Chair of Software Engineering, Am Hubland, 97074 Würzburg, Germany, firstname.lastname@uni-wuerzburg.de

doi:10.18420/abp2021-3

between submission and feedback is high. To this end, Stephen Edwards proposed using automatic grading [Ed]. We build upon this idea and provide the design of a programming task developed for the PABS grading system [If, ID][2].

In this task, the students have to implement test cases for an application. They first have to implement the application (a comparably small task designed to take about 90 minutes) and can check for its correctness using an external test suite. Then they have to implement the test cases that the grading environment runs against correct and flawed implementations of the application. Since we teach an introductory course, we decided to separate writing the application and writing the tests. We provide a short description of our didactic goals and how they map to the task design. We also take a look at the lecture evaluation and analyze how the students evaluate the testing task in comparison to regular tasks. In this evaluation, it received similar grades to the other tasks, showing that the students were not exceptionally overburdened by the task.

The remainder of this paper first presents the didactic goals in Section 2. Section 3 details how we realized these goals in the task design, followed by a student evaluation in Section 4. Last, we take a brief look at related works in Section 5 before Section 6 concludes the paper.

2 Didactic Goals - What the students should learn

The material presented here is for a practical programming course that students usually take between the first and second or second and third semester of their Bachelor's curriculum. It comprises a set of four tasks and an online exam. Students come from a broad range of subjects, comprising not only classical computer science but also aerospace computer science, business information systems, human-machine interaction, business mathematics, physics, and various teaching programs.

With regular programming exercises, the students get in contact with unit tests. They follow the task description, implement some code, and then let given tests check for mistakes in the behavior of the implemented code. However, the test sources are usually not accessible to students; they only receive the final report of the test results. The main goal of this programming exercise is to give the students a better understanding of how unit tests function by letting them write test code themselves. This high-level goal comprises more fine-granular aspects, which we cover in the remainder of the section.

More precisely, the target programming language is Java, with JUnit 5 as the unit testing framework. The first fundamental detail to learn is the structure of a project. This includes the placement of source and test code in designated folders, the location of resource files, and the integration of required libraries. Secondly, the students should learn that unit tests are essentially nothing more than methods that are specially tagged and treated by the JUnit

[2] The project is available at `https://oc.informatik.uni-wuerzburg.de/s/mex6TkHowZcWtTx` (Password: NM3W6Q2x). Upon acceptance we will publish the exercise on GitHub.

runner. The result status of a test depends on its execution: When an exception occurs, the test fails with an error. When the test case throws an `AssertionError` or calls the `fail()` function because of invalid code behavior, the test counts as failed. If none of those happens, the test is considered successful.

When it comes to designing test classes and methods, the exercise should give a good example to the students so that they can apply this knowledge to their future projects. One important aspect here is that tests do not have to be self-contained and, in fact, should not be. Many tests require the same or similar functionality, and implementing or copy-pasting the same code multiple times is a bad smell because it hurts maintainability. Thus, the students should recognize the possibility to move code to one or multiple test utility classes, providing helper methods, and understand when to make use of them.

Regarding the covered Java code features, the most common ones should occur. These include the testing of enums, simple classes that only store variables and provide access methods, `hashCode()`, `equals()`, and `toString()`, as well as more complex classes with considerable interaction and state change invoking methods. One key aspect to grasp here is the concept of keeping dependencies minimal, i.e., when testing one class, not to depend on the possibly incorrect implementations of other classes. The usage of mocks facilitates this paradigm considerably and accordingly presents a significant didactic objective.

3 Task Design and Implementation

With the didactic goals set, this section describes the realization of the programming exercise that fulfills these objectives. At first, we discuss the design and content of the task (3.1). Later, supplementary information about the technical details is provided (3.2).

3.1 Design

The primary goal of the exercise is to learn and practice writing unit tests. This requires the existence of some codebase whose implementation can be tested. There are multiple approaches to providing the *code under test*. One of them is to simply provide finished code. As the size of the scope of the task and expected implementation time should be consistent with other exercises, one advantage of this approach is that no time is lost writing “normal” code; more time can be spent programming tests. However, this can be deceiving. While it is not necessary to write the code base, it is still indispensable to study and understand it to be able to implement thorough test cases.

We decided to split the exercise into two parts to prevent students from skipping reviewing the given code and then having more trouble with the test programming. The first part is essentially like a usual assignment, without any testing being by the students involved. Then, the second part is concerned with writing tests for the code of the first half. Thus, by the

time the students reach the point where they have to write the tests, they are very much familiar with what their test code is interacting with—there is no code one understands better than one's own code. In PABS, the online grading system, tests are available, which examine the implementations of the first part-code, as well as the tests of the second part. A more detailed explanation of how this is achieved follows in 3.2. This way, the students can be sure that their first part-implementation is probably correct and can utilize it for local debugging when implementing the tests in the second half. We decided on this approach instead of a full TDD scenario to ensure that students are not faced with the challenge of having to write correct code and correct tests at the same time. Furthermore, the PABS test results are later used for grading.

For the first part, we decided to use an existing exercise that was used as an exam and not published afterward. Apart from the work saved to create code, tests, and task description, this approach offers several advantages. In the exam, 90 minutes are available to implement the whole task. With some experience, even less time is sufficient to complete the exercise (usually, we design the tasks to take less than 60 minutes for the teaching assistants). This is considerably less effort than a non-exam exercise has, which allows for adding testing to the task without making it of above-average size. Also, the contents of the exam covered the most important topics (already mentioned in 2). For the students, there is the additional benefit that the task implicitly prepares them for the exam.

The exam exercise selected is "Mensa ToGo", representing a food order management system for a cafeteria. The first task is to implement the `DishType` enum, defining enum values like `STARTER` and `MAIN_DISH`. Another enum `GuestType` defines different types of guests and how much discount they receive. The class `Guest` is a data class that stores the name and type of a guest. Similarly, the `Dish` class manages dish name, price, and type. One `Order` stores multiple dishes, the guest who bought it, and can calculate the total price. `equals`, `hashCode`, and `toString` implementations are mandatory for `Guest`, `Dish`, and `Order`. Finally, the `DayManager` class implements the functionality to sequentially add orders and assign them to time slots based on slot capacity and preference.

After the requirements for the classes of the first part, the task description gives an introduction and explanation of the basics of unit testing in general and JUnit in particular. Further, we explain the concept and importance of mocking using the Mockito library.

Before implementing any tests, we require two classes with helper methods. `TestUtils` provides some advanced assertion methods that are not available from JUnit. `assertThrowsWithMessage` checks whether a piece of code throws an exception of the correct type with the correct message. Testing that two collections contain the same elements in any order is realized by `assertCollectionsEquals`. Verifying that a list is not modifiable is the job of `assertListIsUnmodifiable`. The other helper class is `DataGenerator`. Because the following tests will often need data to feed to the code under test, this class provides means to generate such easily . Amongst others, there exist methods to generate lists of `Guest`, `Dish`, and `Order` mocks.

Next are the test classes for the dish and guest type enums. It is crucial for the students to learn that they cannot directly access enum values, because if they are missing in the code under test, the tests will not even compile. With the following classes for `Guest`, `Dish`, and `Order`, the tests primarily focus on creating mocks, instantiating objects to test, calling methods with prepared parameters, and comparing return value to the expected result. The focus lies on making it clear that multiple test cases, including all edge cases, are necessary for a good unit test, covering valid parameters as well as unexpected inputs. Also, emphasis is put on descriptive assertion messages. Finally, the `DayManager` requires more complex tests. For example, to validate the `getAllOrders` method, it is necessary to add orders with other methods beforehand.

3.2 Implementation

There are some technical details that are not trivial about this exercise. These are discussed in this subsection. Most importantly, it is not obvious that it is possible to let students locally write JUnit tests and still be able to use JUnit tests in the online grading system.

First off, we outline how PABS evaluates student code. PABS creates a gradle project. The student code repository is copied into the project, together with test sources, test resources, and build.gradle file, which all are stored on the PABS server. The build.gradle specifies source, resource, test source, and test resources, together with external dependencies.

Now, with the Testing exercise, the distinction between sources and test sources becomes more difficult. To avoid problems with configuration of the local student projects and to ensure compatibility with the online grading, a template is provided, which already has all folders created at the required locations. Also, a build.gradle file is provided, making manual configuration of JUnit and Mockito unnecessary. The local gradle project is configured such that the code of the first exercise part is marked as source, while the code of the second part is marked as the test sources. That way, students can easily run and debug their tests with their own solutions. The build.gradle file for online grading however marks both source folders of the student repository as sources and only the grading tests as test sources (of course, as only their results are supposed to appear in the final test evaluation report).

Now, with the structure out of the way, let us get to the actual technical challenges. Unfortunately, we didn not find a way to let the students write code 100% like they would if no grading tests existed. The local test code would instantiate objects of the classes to test (e.g., `Dish`) using a constructor. The grading tests now have to replace such constructor calls with the instantiation of one some class with known behavior to judge whether the tests test correctly, which is problematic.

We developed a minimal-invasive solution, which allows switching the class under test while keeping the test programming for the students as authentic as possible. First off, the first part of the exercise was altered from the original exam. Originally, classes with specified

names had to be implemented, and the exam tests were directly using the constructors to create objects. Now, an interface is given for each class. The students write classes that implement this interface and rewrite a static factory method in each interface to return an object of their implementation. This already allows the tests to handle different instances of the interfaces, however, the selection of the instance to test is still not possible.

We chose to implement `AbstractTestClass`, which all the student-written test classes have to derive from. The main feature of this abstract class is to provide a `construct` method to the test classes. When executed locally, the `construct` method uses the name of the test class to identify of which class an object has to be created and then uses the student-implemented factory method to construct an instance of the student's implementation. When executed in the grading system, the grading tests can create an instance of a student test class and, using features of the `AbstractTestClass`, change the behavior of `construct` to instead construct a different implementation of the interface.

In short: When the student's tests are executed on their own, they test the student code. The grading tests have the ability to change the code under test to known correct and false implementations. This comes at the cost of defining an interface for each class, inheriting the `AbstractTestClass` in each test, and using `construct` instead of directly calling factory-methods or constructors.

Another point to consider is the creation of objects of classes that are not currently under test, e.g., creating guests and dishes for an order. Theoretically, it possible to use the same approach for this; however, we chose to instead focus on the concept of mocking to prevent this situation from occurring.

4 Student Evaluation

The participants of the practical course had the chance to evaluate the course. Of 122 students admitted to the exam, 46 students followed up on this opportunity. In this section, we use their feedback to get an insight into the perceived difficulty of the testing exercise. Besides the task of interest, there were three other tasks. Hölzchenspiel was a shorter exercise that every participant had to solve within the first week implementing a simple command-line game. One of the other bigger tasks was JavAlgebra, which dealt with modeling algebraic concepts, e.g., groups and fields. The other was 2048, where students had to implement the logic and GUI for the popular 2048 game. It is important to note that, while the computer scientists had to solve all exercises, the other subjects only had a subset to solve. The number of submitted solutions gives an impression of this: 162 solutions for Hölzchenspiel, 127 for Testing, 101 for JavAlgebra, and 90 for 2048.

Participants rated the difficulty of the tasks between 1 (very easy) and 4 (very hard). Hölzchenspiel and JavAlgebra were classified as easier with an average of 2.3 and 2.4, respectively. 2048 and Testing were more difficult, with ratings of 2.8 and 3.0, respectively.

The standard deviation was very similar, 0.9 for JavAlgebra and 0.8 for the rest. One factor that can make a task hard is a bad task description. Ratings for comprehensibility also went from 1 (easily understandable) to 4 (hard to understand). The results showed Hölzchenspiel 1.9 (standard deviation=0.8), Testing 2 (sd=1), JavAlgebra 1.5 (sd=0.9), and 2048 1.5 (sd=0.6), indicating that it is unlikely to be the key factor for the difficulty. Looking at the textual feedback that students provided, we get the impression that the difficulty comes from new concepts. Hölzchen and JavAlgebra do not require programming knowledge beyond the content of introductory programming lectures, whereas 2048 requires them to implement a graphical user interface, and, of course, Testing offers the unit tests.

5 Related Work

In this section, we review related work and highlight the novelty of our contributions.

In [Ca96], the author describes his approach to teaching testing. The focus is on black and white box testing. He presents exemplary tutorial questions and assignments for both approaches. However, there no evaluation of the effectiveness of the approach is presented.

The authors of [Ed] present their approach to embedding testing more firmly in the curriculum of the Virginia Tech. Their approach is to teach students test-driven development throughout their studies. Due to the lack of a program that can automatically assess students' test suits, the authors have developed such a program themselves as an extension for Web-CAT. The self-developed extension evaluates (1) the validity, (2) the completeness (e.g., by means of code coverage), and (3) the style (with the help of static analysis tools) of the students' test suit. The authors have evaluated the effectiveness of their approach in [Ed03]. To do this, they compared programming tasks from a course before and after the introduction of the new approach. This comparison showed that students had 45% fewer bugs per thousand lines of code in their programming assignments using the new approach.

In [Ba], teaching testing is scheduled as part of the introduction to object-oriented programming. In addition to locating the topic of testing in the curriculum, the authors have developed the program PROGTEST, which can assess students' test suits. The approach was not evaluated. Next, the authors of [KP] focus on domain testing when teaching testing. For concrete realization, they developed a corresponding lecture series but without automated assessment. Their lecture series was evaluated with the help of a student survey. Also, [CRF], the authors take the approach of linking the topic of testing with security. To this end, they have developed the Code Defenders teaching program, in which students are to identify and close security gaps in the code by testing. This approach was not evaluated.

6 Conclusion

In this paper, we first showed the contents we want to convey to the students focusing on the creation of unit tests. Based on these requirements, we designed a task for a programming

course and implemented it using our automated grading system PABS. Students first implement the code using the black-box unit tests provided in PABS to assure that their implementation is correct. Then they must realize a test suite the succeeds for correct code and throws assertion errors for incorrect implementations. We evaluated the student feedback. This feedback shows that the students consider the task slightly but not significantly harder to implement and understand. However, this is also the case for other tasks that require more than the implementation of algorithms. In summary, we are satisfied with the result and plan to use similar tasks—taking improvement suggestions into account—in future iterations. In future work, we want to extend the task design allowing the teaching of test-driven development (TDD).

Bibliography

[Ba] Barbosa, Ellen F.; Silva, Marco A. G.; Corte, Camila K. D.; Maldonado, Jose C.: Integrated teaching of programming foundations and software testing. In: 2008 38th Annual Frontiers in Education Conference. IEEE.

[Ca96] Carrington, David: Teaching Software Testing. In: Proceedings of the 2nd Australasian Conference on Computer Science Education. ACSE '97, Association for Computing Machinery, New York, NY, USA, p. 59–64, 1996.

[CRF] Clegg, Benjamin S.; Rojas, Jose Miguel; Fraser, Gordon: Teaching Software Testing Concepts Using a Mutation Testing Game. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET). IEEE.

[Ed] Edwards, Stephen H.: Teaching Software Testing: Automatic Grading Meets Test-First Coding. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '03, Association for Computing Machinery, New York, NY, USA, p. 318–319.

[Ed03] Edwards, Stephen H: Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In: Proceedings of the international conference on education and information systems: technologies and applications EISTA. volume 3. Citeseer, 2003.

[ID] Iffländer, Lukas; Dallmann, Alexander: Der Grader PABS. In (Bott, Oliver J.; Fricke, Peter; Priss, Uta; Striewe, Michael, eds): Automatische Bewertung in der Programmierausbildung, volume 6 of Digitale Medien in der Hochschullehre, chapter 15, pp. 241–254. ELAN e.V. and Waxmann Verlag.

[If] Iffländer, Lukas; Dallmann, Alexander; Beck, Philip-Daniel; Ifland, Marianus: PABS-a Programming Assignment Feedback System. In: Proceedings of the secod workshop for automated grading of programming exercises (ABP).

[Je] Jetbrains: , The State of Developer Ecosystem 2020. [Online; accessed 9. Jun. 2021].

[KP] Kaner, Cem; Padmanabhan, Sowmya: Practice and Transfer of Learning in the Teaching of Software Testing. In: 20th Conference on Software Engineering Education & Training (CSEET'07). IEEE.

Akzeptanz der Nutzung von automatisiertem Assessment im Rahmen einer virtuellen Vorlesung

Florian Horn[1], Daniel Schiffner[2] und Detlef Krömker[3]

Abstract: Durch die Umstellung auf virtuelle Lehre findet auch die Verwendung von automatischen Tools zur Bewertung von Programmieraufgaben immer mehr den Einzug in die Lehre. Im Rahmen einer solchen virtuellen Vorlesung wurde eine Bewertung durch die Studierenden vorgenommen, um daraus Erkenntnisse für die zukünftige Einbettung in der Lehre zu ziehen. Die Vorlesung zielt dabei auf höhere Semester des Bachelorstudiengangs ab und nutzt dabei Vorerfahrungen der Studierenden. Insgesamt wurde Feedback von 47 Studierende durch einen Fragebogen erhoben, und daraus Rückschlüsse auf die Qualität und Einsetzbarkeit von Unit-Tests gezogen.

Keywords: UEQ, Moodle, CodeRunner

1 Einleitung

Die Verwendung von automatisiertem Assessment ist nach wie vor mit einem hohen Aufwand für beide Seiten, Lehrende und Lernende, verbunden. Anstatt ein hochgradig spezialisiertes Werkzeug für ein bestimmtes Aufgabenformat zu nutzen, zielt die vorliegende Studie darauf ab, eine Antwort auf die Akzeptanz sehr einfacher Programmieraufgaben durch die Studierenden zu geben.

In dem Untersuchungsfall wurden, im Kontext einer Vorlesung über Computergrafik, Übungen mittels des Tools CodeRunner in Moodle umgesetzt. In den vorherigen (Präsenz-)Iterationen wurden vergleichbare Aufgaben gestellt, die jedoch zum Großteil manuell bewertet wurden, allerdings nicht als Programmieraufgaben, sondern als klassische Berechnungen durch die Studierenden. Hierbei wurde jedoch stets auf eine programmatische Lösung der Aufgaben hingearbeitet. Statt einer eigenständigen Prüfungsform, waren die Übungen begleitend und konnten als Bonus in einer Klausur bzw. mündlichen Prüfung angerechnet werden.

In der neuen Variante wird methodisch auf eine automatisierte Testung von Quellcode gesetzt, welche den Studierenden bereits aus vorhergehenden Semestern bekannt ist. Jedoch wird eine höhere Selbstständigkeit erwartet und die Aufgaben sind entsprechend auch aufwändiger gestaltet. Generell wurden dabei mehrere Lernziele definiert. Zum einen

[1] Goethe University, studiumdigitale, Robert-Mayer Str. 10, 60325 Frankfurt, horn@studiumdigitale.uni-frankfurt.de

[2] DIPF | Leibniz Institute for Research and Information in Education, IZB, Rostocker Straße 6, 60323 Frankfurt, schiffner@dipf.de

[3] Goethe University, studiumdigitale, Varrentrappstraße 40-42, 60486 Frankfurt, kroemker@studiumdigitale.uni-frankfurt.de

doi:10.18420/abp2021-4

sollen Studierende durch die Programmierarbeit die einzelnen Bestandteile der Rendering Pipeline besser verstehen, da sie von den abstrakten Konzepten eine reale Umsetzung erstellen. Zum anderen soll durch die Implementierungsarbeit eine längere Auseinandersetzung mit der Thematik bewirkt werden. Hierbei eignen sich Programmieraufgaben besonders gut, da eine visuelle Ausgabe grundlegende Eigenschaft der Computergrafik ist.

Im Rahmen einer Befragung werden die folgenden Hypothesen getestet und entsprechend die Fragen darauf ausgerichtet:

1. Die automatische Validierung wird auf Seiten der Studierenden als alternative Prüfungsform akzeptiert und kann entsprechend auch eingesetzt werden.
2. Studierende sind, durch die unmittelbare Rückmeldung, motivierter selbst aufwändige Aufgaben erfolgreich zu bearbeiten.

Zur Validierung unserer Hypothesen wurde eine repräsentative Umfrage (47 von 139, ca. 33 %) unter den Teilnehmern der Veranstaltung durchgeführt. Dazu wurde auf Basis des UEQ und einzelnen Vertiefungsfragen das Feedback zu der Verwendung des Tools im Rahmen des eingesetzten LMS eingeholt. Als Motivation wird den Studierenden ein Bonus in Höhe von 4 % der zum Bestehen notwendigen Punkte für die Bearbeitung des Fragebogens gegeben.

2 Verwandte Arbeiten

Moodle [Mo21] ist ein weit verbreitetes Open Source Learning Management System (LMS). Moodle ist mittlerweile eine Community Entwicklung und erlaubt es weltweit Lehrenden diverse Features und Plugins zu verwenden, um es ihren Kursen anzupassen. Die vorhandenen Fragetypen sind dabei oft auf geschlossene Formate eingeschränkt (SC, MC, Lückentexte). Durch Plugins (bspw. H5P), können andere Formate integriert werden.

CodeRunner [CR21, LH16] ist ein Plugin für das LMS Moodle und erlaubt es Lehrenden Programmieraufgaben in diversen Sprachen zu stellen und diese automatisiert testen zu lassen. Hierbei erlaubt CodeRunner sowohl einen vordefinierten Testablauf (welcher Unit-Tests durchläuft) zu verwenden, als auch einen eigenen zu schreiben, um so beispielsweise Monte-Carlo Simulationen auf studentischen Abgaben durchzuführen. Eine Beschränkung des vordefinierten Testablaufs ist es, dass lediglich Textvergleiche zwischen der erwarteten Ausgabe des Testfalls und der studentischen Abgabe möglich sind.

In [Ul18] geben die Autoren einen Überblick über Software zur automatischen Bewertung von Programmieraufgaben und deren Auswirkung auf Programmieranfänger. Die Autoren erreichen hierbei das Fazit, dass die automatische Bewertung von Programmieraufgaben gerade für Anfänger wichtig ist, da das sofortige Feedback bei der Fehlerbehandlung hilft und somit die Studierenden motiviert, Aufgaben in mehreren Versuchen vollständig zu lösen.

In [CE20] stellen die Autoren die Ersetzung ehemaliger Self-Assesments auf MC Basis durch automatisch bewertete Programmieraufgaben und den Effekt dieser Intervention auf die Studierenden vor. Verwendet wurde hierbei auch CodeRunner. In dieser Studie zeigt sich zwar ein negativer Effekt auf die Bewertung der Self-Assessment Aufgaben. Laut den Autoren ist dies der Fall, weil MC Aufgaben die Programmier-Skill überschätzen. Zusätzlich zeigt sich auch ein Anstieg der Noten in der finalen Klausur. Zudem Bewerteten die Studierenden das erhaltene Feedback als besser, da Sie feingranularer auf Ihre Fehler hingewiesen wurden.

Ein standardisiertes Evaluationswerkzeug zur Erhebung der Usability ist das User Experience Questionaire (UEQ) [UEQ21]. Das UEQ besteht aus 27 Likert Fragen, bei denen der User seine Meinung zwischen zwei Gegensätzen verortet, etwa „übersichtlich“ und „verwirrend“. Die so dargestellte Meinung wird Kreuzvalidiert und auf 6 User Experience Dimensionen abgebildet. Für unsere Studie verwendeten wir die Kurzversion [HST17], da Usability nicht im Fokus unserer Studie stand. Diese besteht aus lediglich 8 Fragen und bildet diese auf 2 Dimensionen ab: Pragmatische und Hedonistische Qualität.

3 Durchführung

Im Rahmen der Vorlesung „Einführung in die Computergraphik“ haben wir automatisierte Programmieraufgaben, als Ersatz für Übungsblätter, als Teil des Tutoriums eingesetzt. Thematisch behandelt diese Vorlesung vor allem algorithmische Grundlagen, Datenformate und Konventionen der Computergraphik. Ziel ist es hierbei Studierenden zu ermöglichen, auch komplexe Konzepte wie Ray-Tracing und Rendering Pipelines zu verstehen und selber umzusetzen.

Die Vorlesung lief vollständig Online ab. Hierzu wurde Moodle eingesetzt und Studierenden Vorlesungsaufzeichnungen, Folien, Zusatzmaterial, sowie die Übungen zur Verfügung gestellt. Unterstützend hatten Studierende die Möglichkeit an einer wöchentlichen Sprechstunde über die Übungsaufgaben teilzunehmen, sowie ein in Moodle integriertes Forum zu verwenden, in welchem sie sowohl von den Lehrenden, als auch von Kommilitonen Antworten und Feedback erhalten konnten. Weiterhin haben die Studierenden selbstständig einen Discord-Server aufgesetzt, um dort den Austausch abseits der Lernplattform zu koordinieren.

Als Programmiersprache wurde Python verwendet, da Studierende der Universität diese bereits in anderen Veranstaltungen eingesetzt haben. Zur Auslieferung der Fragen wurde das Moodle-Plugin CodeRunner verwendet.

Um einen Eindruck vom verwendeten Testsystem zu erhalten wurde den Studierenden zunächst eine Handreichung zur Verfügung gestellt. In dieser wurde ausführlich der Test und Abgabe Prozess in CodeRunner erklärt, sowie wie die Punktevergabe erfolgt. Zusätzlich hatten die Studierenden zwei einfache Aufgaben zur Verfügung, die sie jederzeit abgeben konnten, um sich mit dem Prozess vertraut zu machen. Generell wurde

CodeRunner nur zur Abgabe verwendet. Vom aktiven Programmieren innerhalb von Moodle wurde den Studierenden abgeraten, da grundlegende Hilfsmittel, wie Code-Completion, fehlen.

Aufgaben bestanden in der Regel aus mehrere Teilaufgaben, die gemeinsam ein funktionsfähiges Programm ergaben. Studierende hatten, je nach Komplexität, 1 bis 2 Wochen Zeit zur Bearbeitung. Bedingt hierdurch sahen wir von einem „all-or-nothing“ Grading ab und gaben anteilig Punkte auf bestandene Unit-Tests.

Zusätzlich war es Studierenden erlaubt mehrmals abzugeben, allerdings mit einem Punktabzug des erreichbaren Maximums. Studierende hatten zwei Abgaben ohne Punktabzug (eine Erst-Abgabe und einen Freiversuch). Jede weitere Abgabe zog 10 % vom erreichbaren Maximum ab. Hierbei wurde sichergestellt, dass die erreichte Maximalnote verwendet wurde.

Um die Meinung der Studierenden zum automatisierten Testen, in diesem Fall spezifisch mit CodeRunner, zu erheben, wurde eine Evaluationsstudie durchgeführt. Das Ziel dieser Studie war, die subjektive Meinung der Studierenden zum Einsatz von automatisierten Testsystemen in Tutorien und als Prüfungsersatz zu erhalten. Neben geschlossenen Fragen (siehe Tabelle 1) hatten die Studierenden die Möglichkeit, zu bestimmten Fragen offenes Feedback zu geben. Zusätzlich haben wir Fragen zur Erfassung der Usability des Systems eingesetzt, da eine schlechte Usability einen signifikanten Einfluss auf die Akzeptanz von Lern- und Prüfungssoftware hat.

Der Fragebogen wurde den Studierenden der Vorlesung angeboten und als zusätzliche extrinsische Motivation zur Teilnahme an der Befragung wurden den Studierenden Bonuspunkte (4 % der zum Bestehen notwendigen Punkte) gegeben. Es wurde hierbei darauf geachtet, dass die Umfrage vollständig anonymisiert und die Speicherung der Daten zwecks Bonuspunkte Vergabe geblendet ist.

4 Ergebnisse

An der Umfrage nahmen 49 Studierende teil. Insgesamt im Kurs eingeschrieben sind 139 Studierende, von denen circa. 74 aktiv, zur Zeit der Online-Befragung, am Übungsbetrieb teilgenommen haben. 2 der Umfragen wurden ausgenommen, da diese keine Frage beantwortet haben. Die Umfrage bestand aus mehreren Likert-Fragen, einigen Freitext fragen, sowie der Kurzversion des UEQ.

Die Likert-Fragen waren skaliert von 1: „Stimme gar nicht zu“ bis 4: „Stimme vollständig zu“. Zudem hatten Studierende die Möglichkeit eine Frage auch mit „Keine Angabe“ zu beantworten. Zusätzlich hatten Fragen 7-10 die Zusatzfrage: „Ich würde meine Meinung zu Frage X ändern, wenn...“ und Studierende hatten die Möglichkeit ihre Meinung zu elaborieren. Die Ergebnisse der Quantitativen Befragung findet sich in Tabelle 1 und Abbildung 1 für unsere Fragen und Abbildung 2 für die Ergebnisse des UEQ.

Frage	**M**	**SD**	**N**	**E**
1. Ich habe die Einführungen und Beispiele in CodeRunner durch die Veranstalter als hilfreich empfunden.	3.38	0.74	45	2
2. Ich habe das direkte automatisierte Feedback von Coderunner als förderlich für den Lernerfolg erlebt.	3.02	0.84	47	0
3. Auch negatives Feedback (Anzeige von Fehlern) habe es ich als hilfreich und motivierend empfunden. Ich habe meine Lösung dann meist soweit verbessert, bis jeder Testcase erfolgreich war.	2.91	0.99	47	0
4. Der automatisierte Feedback verführt dazu, viel zu viel Zeit in die Bearbeitung der Aufgabe zu investieren, um auch die letzte Ungenauigkeit auszumerzen.	3.06	0.91	47	0
5. In unserem Fall ist Coderunner so konfiguriert, dass es nach dem zweiten Versuch zum „Prüfen“ einen Punktabzug von den maximal erreichbaren Punkten gibt. Ich verstehe, warum dieser Abzug nötig ist.	3.04	0.91	46	1
6. Ich finde den Punktabzug (Nr. 5) insofern akzeptabel.	2.67	0.86	46	1
7. Ich lehne die Nutzung von CodeRunner (grundsätzlich) ab, weil es den Lernerfolg nicht steigert.	1.72	0.95	46	1
8. Ich befürworte die Nutzung von CodeRunner zur persönlichen Information des Lernenden, also ohne Bonuspunkte zu vergeben oder es als Prüfungssystem einzusetzen.	2.67	1.00	39	7
9. Ich befürworte die Nutzung von CodeRunner (einschließlich der benutzten Bewertungsmethode für die Ergebnisse), wenn damit Bonuspunkte in geringem Umfang vergeben werden.	2.23	1.08	40	7
10. Ich befürworte die Nutzung von CodeRunner (einschließlich der benutzten Bewertungsmethode für die Ergebnisse) als (open book) Prüfungssystem, so wie in CG eingesetzt.	3.31	1.01	45	2

Tab. 1: Ergebnisse der quantitativen Befragung. Angegeben sind der Mittelwert(M), die Standardabweichung(SD), die Antwortzahl(N) und die Anzahl der Enthaltungen(E). Die Skala bewegt sich von 1 (Ablehnung) bis 4 (Zustimmung).

Wie diesen Ergebnissen zu entnehmen ist, sind Studierende generell mit einer Portfolio Bewertung durch automatisierte Tests einverstanden. Besonders bezeichnend sind hierfür die Antworten auf Frage 1 und Frage 7.

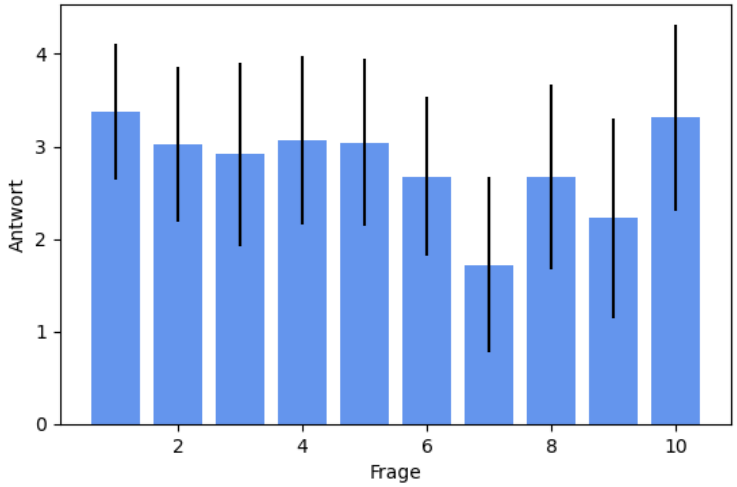

Abb.1: Ergebnisse der quantitativen Befragung. Angegeben ist für jede Frage der Mittelwert, sowie ihre Standardabweichung.

Der zweite Teil der Umfrage bezieht sich auf die einzelnen Erwartungshaltungen und Einsatzszenarien von automatisch getesteten Prüfungsaufgaben. Das Feedback der Studierenden zeigt, dass generell der Einsatz im Rahmen einer Vorlesung für fortgeschrittene Studierende hilfreich ist. Gleichfalls stimmen die meisten Studierenden dem gewählten Einsatz zu und finden dies als geeignete Prüfungsform (Fragen 2 und 7).

Für die Auswertung der UEQ Befragung (n=47) wurden die UEQ Auswertungs-Tools verwendet. Diese ergaben das CodeRunner leicht überdurchschnittlich abschneidet (vgl. Abb. 2). Entsprechend gehen wir davon aus, dass kein negativer Einfluss durch die Benutzung des CodeRunner entstanden ist. Dies ist vermutlich auch auf die dedizierte Verwendung (Nur Abgabe und Feedback) des Tools zurückzuführen.

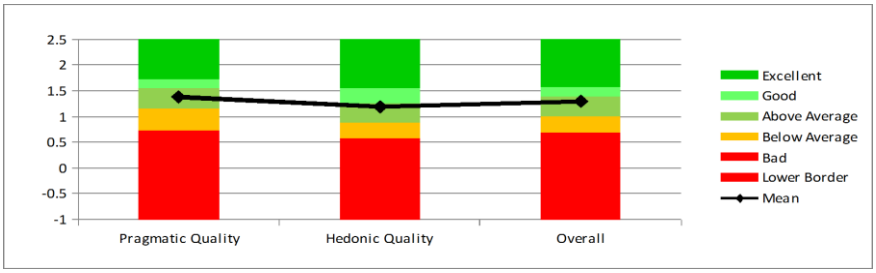

Abb. 2: Ergebnisse der UEQ Benchmark.

Die Umsetzung der Bewertung, und damit der ersten Hypothese, zeigt ein eher gemischtes Bild. Durch die Einschränkungen des Tools sind schnell Testfälle erzeugt, aber es werden lediglich textuelle Repräsentationen verwendet. Dies wird auch durch die Studierendenschaft moniert, welche sich teilweise detaillierte Tests wünschen. Dementsprechend zeigt sich auch ein eher unklares Bild in der Bewertung der Fragen 8, 9 und 10. Generell begrüßen die Studierenden, dass statt einer Klausur oder mündlichen Prüfung Programmieraufgaben, vergleichbar zu einem Praktikum, abzugeben. Sind aber unsicher, ob der Punkteabzug (Frage 6) so implementiert werden sollte. Die Motivation des Punkteabzugs widerrum wird unterstützt (Frage 5). Wir leiten aus den Fragen 8 und 9 entsprechend ab, dass eine automatisierte Bewertung für Übungsaufgaben nur dann sinnvoll ist, wenn auch ein klarer Einfluss in die Benotung vorhanden ist. Frage 10 bestätigt diesen Eindruck, ist aber auch mit einer generellen Prüfungsangst bei Studierenden konnotiert.

Zur Beantwortung der zweiten Hypothese zeigt sich eine positive Tendenz (Fragen 3 und 4). Die Erwartung, dass einzelne Studierende stets das Maximum der Punkte herausholen wollen zeigt sich auch in den offenen Antworten. Neben einzelnen, unzufriedenen Aussagen zeigt sich hier dennoch ein positiver Grundtenor, der sich insbesondere auch für ein individualisiertes Feedback ausspricht (Zitat: „Aufgaben mit weniger als 100 % Punk[t]zahl per Hand noch einmal evaluiert werden würden, um Leuten zu helfen, die eigentlich die richtige Idee hatten [...].“). Auch wäre eine geführte Herangehensweise teilweise erstrebenswert (Zitat: „Dann aber vielleicht begleitender, ähnlich eines Jupyter Notebooks. [...]“) Trotz der vielfältigen Angebote parallel zur Übung, scheinen einzelne Studierende ein weiteres Angebot zu vermissen (Zitat: „[...] Foren und Discord empfinde ich hierfür als sehr anonym und die Stimmung nicht immer wohlwollend. [...]“).

5 Fazit

Die Ergebnisse zeigen durchaus, dass Studierende der Informatik mit dem Konzept von automatisierter Bewertung von Aufgaben einverstanden sind. Es kristallisieren sich aber durchaus interessante Aspekte heraus, in denen sehr unterschiedliche Erwartungshaltungen der Teilnehmer als Ursprung zu vermuten sind.

Es zeigt sich ein unklares Meinungsbild, in dem die Teilnehmer die Art von Aufgaben als Prüfungs-Alternative zwar gerne annehmen und damit unsere erste Forschungshypothese unterstützen. Dennoch zeigt sich beim Einsatzszenario, bzw. den Möglichkeiten ein sehr unklares Bild. In vereinzelten Fällen werden hier starke Bedenken geäußert und mit einem Wunsch nach einer Vertiefung bzw. Flexibilisierung untermauert.

Für die zweite Forschungsthese hingegen zeigt sich eine positive Tendenz. Das unmittelbare Feedback hilft den Studierenden bei der Lösung von bisher nicht betrachteten Sonderfällen und schärft somit das Verständnis über die grundlegenden Algorithmen.

Ausgehend von den Erkenntnissen wird die Veranstaltungen in weiteren Iterationen angepasst, und vermehrt auch zu reinen Übungs- bzw. Self-Assessment-Zwecken automatische Aufgaben implementiert, welche dann auch eine tiefergehende Evaluation inkludieren sollen. Die Idee, personalisiertes Feedback auf die Abgaben zu geben könnte ihm Rahmen von beschränkten Aufgabenstellungen und bestimmten Programmierkonstrukten realisierbar sein. Weiterhin soll das Tool CodeRunner erweitert werden, um eine unmittelbare Interaktion zu ermöglichen und somit spezifisches Feedback und genaueres Testen zu ermöglichen.

Literaturverzeichnis

[CE20] Croft, D.; England, M.: Computing with CodeRunner at Coventry University: Automated summative assessment of Python and C++ code. In Proceedings of the 4th Conference on Computing Education Practice 2020. 2020.

[CR21] CodeRunner, https://coderunner.org.nz/, Stand: 13.06.2021

[LH16] Lobb, R.; Harlow, J.: Coderunner: A tool for assessing computer programming skills. In ACM Inroads 7.1, S. 47-51, 2016.

[Mo21] Moodle, https://moodle.org/, Stand: 13.06.2021

[SHT17] Schrepp, M.; Hinderks, A.; Thomaschewski, J: Design and Evaluation of a Short Version of the User Experience Questionnaire (UEQ-S). In: IJIMAI 4 (6), pp. 103–108, 2017.

[UEQ21] User Experience Questionaire, https://www.ueq-online.org/, Stand: 13.06.2021

[Ul18] Ullah, Z. et al.: The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. In Computer Applications in Engineering Education 26.6, S. 2328-2341, 2018.

Messung der Schwierigkeit von Programmieraufgaben zur Kryptologie in Java

Konstantin Knorr[1]

Abstract: Systeme zur automatischen Bewertung von Programmieraufgaben (ABP) werden seit vielen Jahren erfolgreich in der Ausbildung von Informatikern eingesetzt, insbesondere in Zeiten verstärkter Online-Lehre. Kryptologie gilt bei vielen Studierenden aufgrund ihrer formellen und theoretischen Natur als schwer zugänglich. Das Verständnis kryptologischer Primitiven wie Ver- und Entschlüsselung oder Signatur und ihre Verifikation kann durch die Programmierung bzw. programmatische Anwendung gestärkt werden. Der Beitrag präsentiert eine Studie mit 20 Studierenden, 20 Aufgaben zur Kryptologie und ~300 JUnit-Testfällen, die über ein ABP-System ausgewertet wurden. Die Auswertung nach der Fehlerrate und dem Lösungszeitpunkt der kryptologischen Testfälle erlaubt die Identifikation von schweren Testfällen und zeigt u.a., dass Studierende weniger Fehler bei Substitutions- als bei Transpositionschiffren machen, symmetrische Chiffren leichter fallen als asymmetrische und dass Tests zu den Konstruktoren, Exceptions und Padding deutlich früher und besser gelöst wurden als Tests zu Signaturen und deren Verifikation.

Keywords: Bearbeitungszeit, Java, JUnit, Fehlerrate, Kryptologie, Messung der Schwierigkeit

1 Einleitung

ABP-Systeme werden heute an vielen Hochschulen zur Ausbildung in Informatik-Studiengängen entwickelt und eingesetzt [EPQ07, If15, IS01, Kr20]. Vorteile dieser Systeme sind die Messbarkeit des Programmierfortschritts, das direkte Feedback an die Studierenden und die Skalierbarkeit der Systeme, die insbesondere bei großen Kohorten wichtig wird. Die Anwendung von ABP-Systemen auf Programmieraufgaben zur Kryptologie ist ein relativ junges Anwendungsgebiet [BD15, Kn20]. Der Ansatz dabei ist, Studierende die theoretischen Grundlagen der Kryptologie (z.B. aus der Zahlentheorie) bzw. die Verwendung moderner Krypto-Bibliotheken (wie z.B. [BC]) programmieren zu lassen, um so die Berührungsängste zu nehmen und das Verständnis zu stärken („Learning by coding“). Die korrekte Implementierung wird dabei über Testfälle auf einem ABP-System überprüft. Decken die Testfälle ein breites Spektrum ab, bekommt die Lehrkraft Rückmeldung darüber, an welchen Stellen die Studierenden Schwierigkeiten haben.

Der Beitrag stellt eine Studie vor, in der 20 Studierende über ein Semester 20 Aufgaben zur Kryptologie bearbeitet haben und wertet die Ergebnisse der Tests aus. Das verwendete ABP-System erlaubt den Studierenden beliebig viele Versuche, um die vorgegebenen Testfälle zu bestehen. Die Schwierigkeit der Testfälle wird (1) über die Fehlerrate und (2) über die Bearbeitungszeit, genauer über die Anzahl der notwendigen Versuche zum

[1] Hochschule Trier, Fachbereich Informatik, Am Schneidershof, 54208 Trier, knorr@hochschule-trier.de

doi:10.18420/abp2021-5

Bestehen der Testfälle gemessen. Die Auswertung von kryptologischen Themen und Kategorien von Testfällen mittels dieser beiden Schwierigkeitsmaße erlaubt Rückschlüsse auf die Bereiche der Kryptologie, die den Studierenden besonders leicht bzw. schwerfallen. Dies ist eine wichtige Hilfe zur Verbesserung der Lehre in zukünftigen Veranstaltungen zur Kryptologie.

Die Fortsetzung des Artikels hat folgende Struktur: Kap. 2 beschreibt die notwendigen Hintergrundinformationen zum verwendeten ABP-System, zum Test Driven Software Development (TDSD) und zu JUnit-Tests. Kap. 3 erläutert den Studienaufbau, den verwendeten Datensatz und seine Auswertung. Die Ergebnisse der Studie sind Gegenstand von Kap. 4. Kap. 5 diskutiert die Ergebnisse und endet mit einem Ausblick.

2 Grundlagen

2.1 Das verwendete ABP-System ASB (Automatische Software-Bewertung)

Das System ASB wird seit 2006 an der Hochschule Trier zur automatischen Bewertung von Programmieraufgaben für Informatik-Studierende genutzt. ASB ist eine webbasierte Client-Server Applikation. Studierende und Lehrkräfte besitzen unterschiedliche Rollen und Rechte und können sich über Shibboleth anmelden. Die Daten werden in einer Datenbank gespeichert und es können Ausführungsumgebungen für verschiedene Sprachen wie Java, C++ oder Python eingebunden werden. Vorbereitend erstellen Lehrkräfte die Aufgabenstellungen und Testfälle. Die Studierenden laden ihren Programmcode zur Lösung der gestellten Aufgaben in das System. Dort werden die Testfälle auf den Code angewendet und das Ergebnis direkt im Anschluss ausgegeben. Die Studierenden haben innerhalb des Bearbeitungszeitraums beliebig viele Versuche, die Testfälle der Aufgaben zu bestehen. Weitere Informationen zu ASB finden sich in [He17].

2.2 Test Driven Software Development & JUnit Tests

TDSD stammt von Beck [Be02] und ist eine Untergruppe der agilen Software-Entwicklung. Der Grundgedanke ist nur Code zu erstellen, der vorgegebene Testfälle besteht. Entwickler können so schnell und zielgerichtet Fehler entdecken und diese beseitigen. Bei vielen ABP-Systemen erstellen die Lehrkräfte die Tests, die vom Code der Studierenden bestanden werden müssen. TDSD ist unabhängig von der Programmier-sprache. Die gängigste Java-Implementierung ist das JUnit Framework [JU]. JUnit-Tests erlauben Tests von Klassen, Methoden und Attributen. Es können auch Exceptions getestet werden, was bei vielen kryptologischen Verfahren eine wichtige Eigenschaft ist (z.B. Prüfung auf korrekte Schlüssellängen bei AES). Entwicklungsumgebungen wie Eclipse bieten graphische Oberflächen für die Visualisierung der Testergebnisse an, vgl. Abb. 1. Es gibt drei Ergebnisse eines JUnit-Tests: (1) Passed, (2) Error, (3) Failure. (1) gilt im ASB-System als bestanden (=> „true“), (2) und (3) als nicht bestanden (=> „false“).

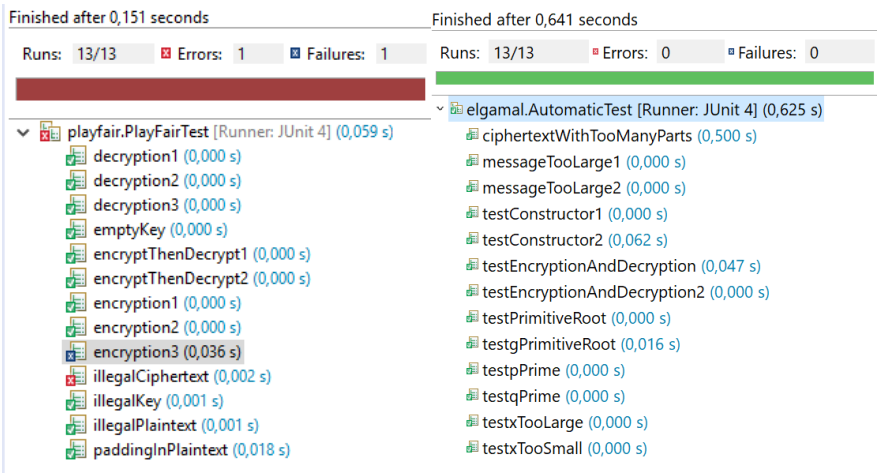

Abb. 1: Bestandene und nicht bestandene JUnit-Testfälle zur Playfair-Aufgabe (links) und zur Rabin-Aufgabe (rechts) in Eclipse

3 Studienaufbau

Das Papier basiert auf Daten des Bachelor-Wahlpflichtfachs „Kryptologisches Programmierpraktikum“ (KPP) aus dem WS 20/21. Es haben sich 20 Studierende für die Veranstaltung angemeldet, 15 haben fast alle Aufgaben bearbeitet. Die Aufgaben mussten in Java gelöst werden. Die Veranstaltung findet unregelmäßig ca. alle zwei Jahre statt, im WS 20/21 zum dritten Mal. Von den 14 Semesterwochen wurden in den ersten 10 Wochen jeweils ungefähr drei Programmieraufgaben gestellt, in den letzten Wochen bearbeiten die Studierenden ein eigenes Projekt zur Kryptologie. Die Themen der Programmieraufgaben umfassten alle gängigen Themen der Kryptologie, also u.a. symmetrische Verfahren (klassische Verfahren, Blockchiffren wie AES, Blockmodi wie CTR und GCM, Stromchiffren), asymmetrische Verfahren (RSA, Rabin, Elgamal), Verfahren zum Schlüsselaustausch (Diffie Hellman), Hash-Verfahren, Pseudozufallszahlen, X.509-Zertifikate, elliptische Kurven und ausgewählte Themen der Postquanten-Welt (NTRU, Lamport-Signaturen), vgl. [SP18]. Von den insgesamt 34 Aufgaben wurden 20 Aufgaben über ASB ausgewertet, vgl. Tab. 1. Die restlichen Aufgaben wurden händisch auf Basis des abgegebenen Codes und von dazu erstellten Videos überprüft.

Das ASB-System speicherte pro Aufgabe, Studierenden, Testfall und Zeitpunkt das Ergebnis r (für „result“) des JUnit-Testfalls. Formell:

$$r: A \times S \times T \times Z \rightarrow \{true, false\}$$

Themen	Aufgaben
Klassische Chiffren	FourSquare, Playfair, Porta (Substitutionschiffren) Jägerzaun, Skytale (Transpositionschiffren) Bifid (beides)
Asymmetrische Chiffren	BigInteger, RSA, Rabin, Elgamal, Paillier, NTRU
Signaturen	Lamport, RabinSig
Elliptische Kurven	myEC, ECDH, ECDSA
Sonstiges	Shamir Secret Sharing, Java Key Store, Padding Oracle

Tab. 1: Über ASB bewertete Themen und Aufgaben zur Kryptologie

Beispiel: *r*(Rabin, 6137, testEncryption2, 2020-10-29T13:21:35) = *true.* Dabei sind:

- *A* die Menge der Aufgaben. Über Tab. 1 kann jeder Aufgabe bei der Auswertung ein übergeordnetes Thema zugeordnet werden.
- *S* ist die Menge der Studierenden-IDs. Den Studierenden wird zur Anonymisierung eine zufällig erzeugte ID zugewiesen (im Bsp. 6137).
- *T* ist die Menge der Testfälle, typischerweise zwischen 10-20 pro Aufgabe. Die Testfälle wurden zur differenzierten Auswertung in folgende zehn Kategorien unterteilt: 1. Encryption, 2. Decryption, 3. EncDec (=encryption and subsequent decryption), 4. Signature, 5. Verify, 6. SignAndVerify (signature and subsequent signature verification), 7. Padding, 8. Constructor, 9. Illegal (Exceptions), 10. Miscellaneous. Im Beispiel: testEncryption2 ➔ Testkategorie 1, „Encryption".
- *Z* steht für die Zeit. Der Bearbeitungszeitraum für eine Aufgabe war jeweils eine Woche in der Vorlesungszeit. Die Studierenden konnten selbst wählen, wie oft und wann sie in diesem Zeitraum Lösungen ins ASB-System hochladen. Die Anzahl der Uploads schwankt zwischen 2 und 40. Für jeden Upload wurde der Zeitstempel in Sekunden gespeichert. Im Beispiel: 2020-10-29T13:21:35.

Um die Zeit zu normieren und damit vergleichbarer zu machen, wurde die Funktion *t* (für „time") verwendet:

$$t: A \times S \times T \rightarrow [0,1]$$

Dazu wurden zunächst die Testergebnisse eines Studierenden zu einem Testfall einer Aufgabe chronologisch als Liste gespeichert, bspw. [*false, false, false, true, false*]. *t* speichert den Zeitpunkt des ersten positiven Testfalls (startend von 0) geteilt durch die Gesamtzahl an Versuchen. Im Beispiel ergibt sich ein Wert von 3/5 = 0,6. Sind alle Ergebnisse „false", wird als Wert 1 gespeichert. Ist bereits der erste Versuch „true", ergibt sich als Wert 0. *t* kann daher als durchschnittliche Bearbeitungszeit interpretiert werden.

Die Fehlerrate eines Testfalls wird mit der Funktion *e* (für „error rate") gemessen:

$$e: A \times S \times T \rightarrow [0,1]$$

e teilt die Anzahl der negativen Tests durch die gesamte Anzahl an Versuchen. Beispiel: [*false, false, false, false, true*] ergibt einen Wert von 4/5=0,8. Die Funktionen *e* und *t* sind stark korreliert, erlauben aber trotzdem folgende Vergleiche bzgl. Fehlerraten und Bearbeitungszeiten:

- Vergleich innerhalb der Tests einer Aufgabe, vgl. Abb. 2.
- Vergleich zwischen Themen der Aufgaben, z.B. asymmetrischer. Chiffren vs. Signaturen oder Substitutions- vs. Transpositionschiffren, s. Tab. 1 und Abb. 3.
- Vergleich zwischen Kategorien von Testfällen, z.B. Signieren vs. Verifizieren von Signaturen oder Exception- vs. Constructor-Tests, vgl. Abb. 4.

Das ASB-System stellt die Daten in Form von Tab Separated Values-Dateien pro Aufgabe zur Verfügung. Die Dateien werden mittels Python eingelesen und ausgewertet. Die Abbildungen wurden mit Matplotlib (https://matplotlib.org) generiert. Der komplette Datensatz, die Python-Skripte sowie Beispiele für Aufgabenstellungen und JUnit-Tests stehen unter https://seafile.rlp.net/d/1776901730d14aefa869/ zum Download bereit.

4 Ergebnisse

Die Funktion *e* kann zur Auswertung der Testfälle der einzelnen Aufgaben verwendet werden. Exemplarisch wird dies an der Aufgabe zur Rabin-Verschlüsselung gezeigt, vgl. Abb. 2. Die Verschlüsselungs- und Constructor-Tests weisen deutlich niedrigere Werte für *e* und *t* auf, als die Tests zur Entschlüsselung, zur Ver- und anschließenden Entschlüsselung und zum chinesischen Restsatzes (CRT- Chinese Remainder Theorem). Dies liegt an der Definition der Rabin-Chiffre. Die Verschlüsselung ist ein einfaches modulares Quadrieren (eine Code-Zeile mit BigInteger), während die Entschlüsselung mit dem Ziehen der Quadratwurzel mod n und der Anwendung des CRT deutlich anspruchsvoller ist, vgl. [Ra79].

Abb. 3 zeigt die durchschnittliche Fehlerrate *e* pro Aufgabe. Die höchste Fehlerrate gab es bei der Aufgabe zum Padding Oracle. Diese Aufgabe war eine besonders anspruchsvolle Bonusaufgabe. Der erste bestandene Testfall lieferte hier die gesuchte Lösung. Die klassischen symmetrischen Transpositionschiffren FourSquare und Porta wurden sehr gut gelöst. Die Transpositonschiffren Jägerzaun und Skytale waren deutlich schwieriger. Die Playfair-Aufgabe war die erste im Semester, weshalb man einen gewissen Eingewöhnungseffekt berücksichtigen muss. Bei den asymmetrischen Chiffren fällt auf, dass die klassischen asymmetrischen Verschlüsselungsverfahren wie RSA, Elgamal und Rabin den Studierenden deutlich leichter fielen, als (1) die Signaturverfahren (RabinSig, Lamport) und (2) modernere asymmetrische Verfahren wie Paillier und NTRU, was evtl. auf die steigende zahlentheoretische Komplexität moderner Verfahren zurückzuführen ist. Die Aufgaben zu elliptischen Kurven liegen im Mittelfeld, was belegt, dass diese algebraische Struktur den Studierenden durchaus zugänglich ist. Bei den sonstigen Aufgaben fällt das schlechte Abschneiden der Keystore- und der BigInteger-Aufgabe auf. Java Keystores dienen dem sicheren Speichern von kryptologischen

Schlüsseln und Zertifikaten. Bei ihnen liegt die Schwierigkeit in der Komplexität der Java-Architektur dahinter, den verschiedenen Keystore-Typen und den subtilen Unterschieden zwischen diesen Typen. Die Java BigInteger-Klasse erlaubt das Arbeiten mit beliebig großen Zahlen, stellt viele wichtige Methoden für die Kryptographie zur Verfügung (wie z.B. modulare Exponentiation oder Invertierung) und eignet sich daher sehr gut als Grundlage für die Implementierung asymmetrischer Verfahren. Die Testfälle hatten keinen Bezug zu einander und wurden daher vermutlich oft chronologisch bearbeitet.

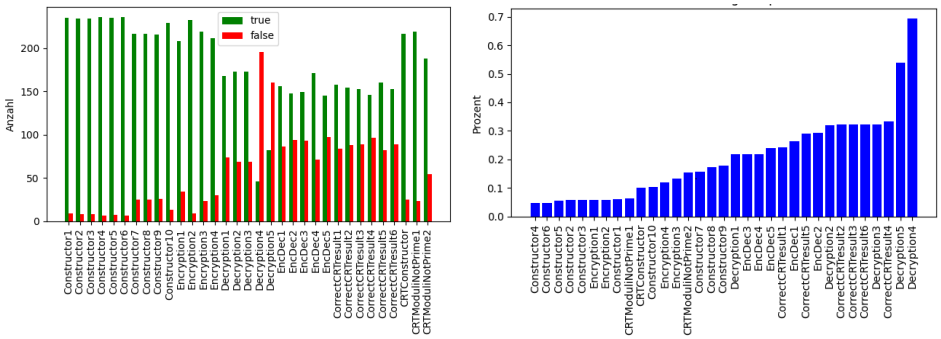

Abb. 2: Fehlerraten absolut (links) und durchschnittliche Bearbeitungszeit *t* (rechts) der Testfälle in der Aufgabe zur Rabin-Verschlüsselung

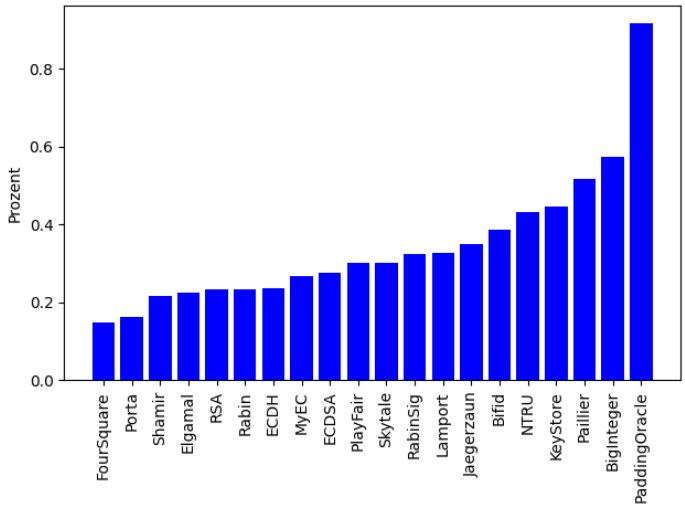

Abb. 3: Durchschnittliche Fehlerrate *e* pro Aufgabe

Abb. 4 zeigt *e* und *t* sortiert nach den zehn Testkategorien. Auffällig ist der geringe Unterschied: Tests mit hoher Fehlerrate wurden später gelöst. Tests zu den Konstruktoren und Exceptions fallen den Studierenden am leichtesten. Padding-, Encryption- und

Decryption-Tests rangieren im Mittelfeld. Die „Kombinations“-Tests EncDec und SignAndVerify schneiden schlechter ab, da hier in den Tests meist zufällige und keine fixen Testwerte verwendet wurden. Encryption fällt leichter als Decryption, Signieren leichter als die Signaturverifikation, Vertraulichkeitsschutz leichter als Integritätsschutz.

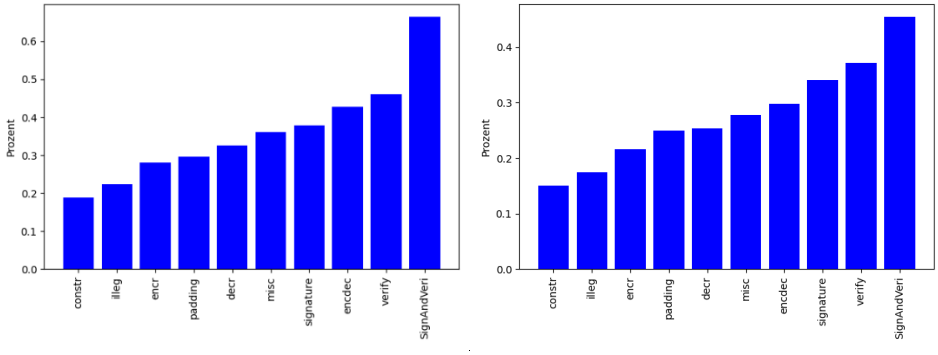

Abb. 4: *e* (links) und *t* (rechts) über alle Aufgaben gruppiert nach Testkategorie

5 Diskussion und Ausblick

Die Auswertung in Kap. 4 liefert neue Erkenntnisse bzgl. der auftretenden Schwierigkeiten bei der Programmierung von Kryptologie in Java. Lehrkräfte können so Testfälle und Aufgaben identifizieren, mit denen die Studierenden Probleme hatten und gegensteuern. Wichtige Erkenntnisse sind z.B.: Substitution fällt leichter als Transposition und Exceptions und Padding fallen leichter als Signieren und deren Verifikation

Bei der Studie sind allerdings folgende störenden Einflüsse zu nennen. **Unterschiedliche Voraussetzung bei den Studierenden**: Die Fehlerrate bei den Testfällen wird durch die Vorkenntnisse der Studierenden in der Kryptologie und der Java-Programmierung beeinflusst. Viele Studierende hatten zu Beginn noch keine intensiven Erfahrungen in Java und steigerten diese im Verlauf der Veranstaltung. Eine gewisse Affinität zu Themen der Kryptologie und Java war bei allen Studierenden gegeben. **Plagiate**: Ein Plagiatsschutz ist zwar im ASB-System vorgesehen, war aber für KPP nicht aktiviert. Plagiate mussten im Verdachtsfall manuell geprüft werden. Der Source Code jeder Abgabe ist dafür in ASB vorhanden. Da viele Aufgaben zeitaufwändig waren und Einfluss auf die Note hatten, kann ein gewisses Maß an „Teamarbeit“ zwischen den Studierenden nicht ausgeschlossen werden. **Chronologie der Testfälle**: ASB präsentiert die Testfälle pro Aufgabe in einer fixen Reihenfolge. Einige Studierende versuchten zunächst, die Testfälle genau in dieser Folge abzuarbeiten.

Effenberger et al. [ECP19] unterscheiden zwischen „Difficulty“ und „Complexity“ von Programmieraufgaben. Für die Difficulty werden wie hier Fehlerrate und Bearbeitungszeit

empfohlen. Als Komplexitätsmaß werden u.a. die Anzahl der Codezeilen und der Kontrollflussstrukturen genannt, die auch hier zum Einsatz kommen könnten. Damit ließen sich die Probleme bei der Lösung der Aufgaben noch genauer analysieren.

Für die Zukunft von KPP sind weitere Aufgaben und eine bessere Kategorisierung der Testfälle und Aufgaben geplant. Bisher sind die Kategorien und Themen in einigen Fällen nicht disjunkt. Bei den ASB-Aufgaben überwiegen die asymmetrischen Verfahren. Auch Aufgaben zu modernen symmetrischen Verfahren sollen automatisiert in ASB überprüft werden. In ASB bietet sich eine Erweiterung um eine Auswertung gemäß Abb. 2 an. Außerdem könnten die Aufgaben auch in Python gestellt werden. Dies würde einen Vergleich der Programmiersprachen Java und Python erlauben. Vorarbeiten dazu laufen [Me21].

Literaturverzeichnis

[BC] Bouncy Castle Homepage, https://www.bouncycastle.org, letzter Zugriff am 2.6.2021.

[BD15] Braga, A., Dahab, R.: A Survey on Tools and Techniques for the Programming and Verification of Secure Cryptographic Software, Proceedings of XV SBSeg, 2015.

[Be02] Beck, K.: Test-Driven Development: By Example. Addison Wesley, 2002.

[ECP19] Effenberger, T., Čechák, J., Pelánek, R.: Measuring Difficulty of Introductory Programming Tasks. Proc. of the 6th ACM Conf. on Learning@Scale, pp. 1-4, 2019.

[EPQ07] Edwards, S., Pérez-Quiñones, M.: Experiences using test-driven development with an automated grader. Journal of Computing Sciences in Colleges 22(3), pp. 44-50, 2007.

[He17] Herres, B., Oechsle, R., Schuster, D.: Der Grader ASB. In: "Automatisierte Bewertung in der Programmierausbildung", Waxmann-Verlag, S. 255-271, 2017.

[If15] Iffländer, L. et al.: PABS – a Programming Assignment Feedback System. In: Proc. of the 2. Workshop Automatische Bewertung von Programmieraufgaben, 2015.

[IS01] Isong, J.: Developing an automated program checker. J. Computing in Small Colleges, 16, 3, pp. 218-224, 2001.

[JU] JUnit Homepage, https://junit.org, letzter Zugriff am 7.6.2021.

[Kn20] Knorr, K.: Learning and Grading Cryptology via Automated Test Driven Software Development. In: Proc. of the 13th IFIP WG 11.8 World Conference on Information Security Education (WISE), WISE 13, Maribor, Slovenia, pp. 3-17, 2020.

[Kr20] Krugel, J. et al.: Automated Measurement of Competencies and Generation of Feedback in Object-Oriented Programming Courses. In: 2020 IEEE Global Engineering Education Conference, EDUCON 2020, Porto, Portugal, April 27-30, S. 329-338, 2020.

[Me21] Meiser, J.: Automatische Bewertung von kryptologischen Programmieraufgaben in Python. Projektarbeit Informatik, Hochschule Trier, 2021.

[Ra79] Rabin, M.: Digitalized Signatures and Public-Key Functions as Intractable as Factorization. MIT Laboratory for Computer Science, 1979.

[SP18] Stinson, D., Paterson, M.: Cryptography: Theory and Practice. 4th edn. CRC, 2018.

Plagiarism Detection Approaches for Simple Introductory Programming Assignments

Sven Strickroth [1]

Abstract: Learning to program is often perceived as hard by students and some students try to cheat. Plagiarisms are reported to be a huge problem particularly for summative-like assignments (e.g., crediting courses or bonus points). It is important to fight plagiarisms from early on – even for simple assignments. Especially for larger courses tool support is required. This paper provides an overview of features for commonly used plagiarism detection tools, discusses how these can be integrated into existing assessment systems, and how their results relate to each other for two data sets of quite simple assignments. Additionally, these specialized tools are compared with a simple Levenshtein distance approach. The paper also outlines limits on very simple assignments.

Keywords: automatic assessment; plagiarism detection; programming education

1 Introduction

At universities programming is often taught in lectures with accompanying homework assignments where learners are asked to write (small) programs from scratch, implement interfaces, or to expand existing programs. However, many learners face severe difficulties in turning the theoretical knowledge into code [La18]. A common approach is to use special tools that handle the submission of the learner's solutions and automatically provide feedback [KJH18; SP17]. However, not all learners solve the assignments on their own or even hand in plagiarized solutions. Plagiarism and other forms of cheating are often reported as a huge issue [La18]. This is particularly true in scenarios in which assignments are graded and used e.g. for crediting a course or for bonus points in the exam.

In smaller courses where one lecturer or tutor can assess all submissions, plagiarisms can be detected manually. However, when dealing with large courses with more than 100 students and multiple tutors grading subgroups of the cohort, it is not effectively possible to detect plagiarism that occur in different groups. Particularly for these scenarios specialized plagiarism detection tools are required.

From the experience of the author, it is very important to establish an environment of good scientific conduct and to fight plagiarism from early on. Therefore, it is necessary to have detection mechanisms that can handle also simple assignments. However, experience has shown that most of the plagiarism detection tools seem to be developed for more advanced

[1] LMU München, Institut für Informatik, Oettingenstraße 67, 80538 München sven.strickroth@ifi.lmu.de

doi:10.18420/abp2021-6

assignments or even programming projects. The problem is that for quite simple assignments lots of false positives are reported that need to be inspected manually (cf. [MPH16]).

The contribution of this paper is to show preliminary research results on comparing existing tools regarding the effort to integrate these into existing assessment systems and the results produced by different plagiarism detection approaches on quite simple assignments written in Java. This should also help to select proper thresholds for the tools. The existing specialized tools are also compared with a simple Levenshtein approach because it is hypothesized that this works better for more simple assignments regarding the number of reported plagiarisms.

2 Related Research

[NJK19] performed a comprehensive systematic literature review on programming plagiarisms in academia. Their research includes definitions of academic plagiarisms, used detection approaches, reported learner's obfuscation methods, evaluation methods, and used data sets. In research there are several plagiarism detection methods developed. As mentioned in [NJK19] the "top-five" most-mentioned tools are JPlag, MOSS, Sherlock-Warwick, Plaggie, and SIM-Grune. These methods are also sometimes directly integrated into assessment tools [SP17]. It is salient that a lot of tools are not available for use.

There are also several papers systematically comparing these methods. The majority of the research [CLS21; HRV11; Lu14; Ma14] seems to focus on the robustness of plagiarism detection evasion with "simulated undergraduate plagiarism" and/or a comparison on features and performance. A notable exception is the comparison of [MPH16] where a real data set was used to analyze the tools regarding false positives and false negatives. However, a problem there is that it is not clearly stated when a found potential plagiarism (as the detection tools output a percentage) is counted as a found one. Also, the used versions are not specified frequently. Hence, there is a research gap for comparing plagiarism detection methods on real-world data sets of introductory programming courses.

3 Plagiarism Detection Methods

The tools used for comparison in this paper are JPlag, SIM, Plaggie, Sherlock-Sydney, and Moss. The rationale for the selection of these tools was the availability of the tools. Additionally, a very simple approach based on the Levenshtein distance is also analyzed because a hypothesis is that this simple approach is particularly helpful for simple assignments. For a short comparison see Tab. 1. With the exception of Moss all tools can be used offline and are available as source code. Only JPlag and Plaggie seem to rely on fully-fledged parsers for their token-based approach and both might ignore submissions they were not able to parse. Plaggie seems to be most critique here: Usage of broken Java code (e.g., missing "}") or newer Java features such as the diamond operator with Generics (introduced in

Java 1.7) or lambda expressions (Java 1.8) make the parser fail – JPlag seems to be a bit more resistant and also doesn't seem to complain for newer language features but, still, a significant number of parse errors could be observed in practice [cf. AM19]. Other tools do not seem to have issues with unparseable Java code or unknown syntactic constructs. All tools support submissions to be organized in separate folders – one for each submission and the outputs are per cent values as an indicator for the degree of similarity. Except for Moss all reported results are symmetric. Apart from JPlag and Moss (and the GATE implementation of the Levenshtein approach) the tools do not seem to be actively maintained.

Tool	Release	Algorithm	Java support	source available	offline
JPlag[a]	2.12.1 (2019)	Greedy String Tiling (token-based)	yes ($\leq$ 9)	yes (GPLv3)	yes
Levenshtein distance	(GATE)[b]	Levenshtein distance	text-files	yes	yes
Moss[c]	n/a[g]	winnowing (token-based)	yes[g], incorrect syntax ok	no	no
Plaggie[d]	2006	Greedy String Tiling (token-based)	yes ($\leq$ 6), only correct syntax	yes (GPL)	yes
Sherlock-Sydney[e]	n/a[g]	winnowing (token-based)	text-files	yes (public domain)	yes
SIM[f]	3.0.2 (2017)	token-based	yes[g], incorrect syntax ok	yes (BSD)	yes

[a] `https://github.com/jplag/jplag`
[b] [SOP11], `https://github.com/csware/si/tree/5618eeefdf0395db86046ec1f5e7a5e0e8d45338`
[c] `https://theory.stanford.edu/~aiken/moss/`
[d] `https://www.cs.hut.fi/Software/Plaggie/`
[e] `https://github.com/diogocabral/sherlock`
[f] `https://dickgrune.com/Programs/similarity_tester/`
[g] no version specified

Tab. 1: The evaluated approaches/tools

JPlag is written in Java and can be downloaded as a JAR file. Besides Java several other languages are supported and boilerplate code can be specified. It can be used from the command-line and provides the results in form of two CSV files (average and maximum similarity) but can also be used by directly via an API. For matches HTML files are generated. A special feature is the clustering of similar submissions. By default, reports only show the top 20 clusters/similar submissions (i.e., default threshold). **Levenshtein distance** is a generic algorithm, here the implementation from the GATE system was used. The similarity is returned as per cent values (1 minus the number of deletions/additions divided by then maximum length) when comparing two files of two submissions. Before comparing, a normalization can be applied: deleting Java comments, lower casing all characters, and converting/reducing tabs, spaces, new lines into a single space. **Moss** is only available as a web-based service and can handle multiple languages. There are different submission clients available. The result is a link to a web page where a full table containing the per cent values as well as separate pages for each match can be obtained – this interface is not optimal for

automatic processing. Also, Moss was not available multiple times for several hours during the tests. Moss seems to have a threshold of 250 reported pairs. **Plaggie** is written in Java. It can be downloaded as source code and can only handle Java code. Plaggie can be used from CLI as well as other Java programs. HTML files for matches can be generated and boilerplate code can be ignored. The detection works on a per submission basis. Plaggie only handles syntactically correct Java 1.6 source code. By default, it reports all results with $\geq 50\,\%$ similarity. **Sherlock-Sydney** is implemented in C and handles generic text documents. It works on a per file basis and reports all per cent values to STDOUT in a CSV style. By default, it reports all results with a similarity $\geq 20\,\%$. **SIM** is written in C and binaries for "MS-DOS" are available (the Makefile does not work at least on current *nix systems). It can handle different languages and works on a per file basis, however, files from the same submission are not compared (see SIM documentation) and boilerplate code cannot be specified. SIM outputs similarities to STDOUT, optionally as per cent values.

4 The Data Set & Method

The focus in this research lay on quite simple assignments written in Java (1.8). Assignment 1 is from an Introductory Programming Course for learners studying business administration, assignment 2 is from a first semester programming course for computer scientists. **Assignment 1:** Calculation task (LoC: 25, main method, variables, calculations, 209 submissions, 60 known plagiarisms with 63 pairs): For an installment payment the difference of paying cash and total price incl. interests as well as the zero-difference case should be calculated and printed out with rounded numbers (2 decimals). Input (arg[]): cash price, cash down, rate per month and period in months. **Assignment 2:** main method containing a single "if else-if else"-construct or nested if-statements and returning given strings based on "arg[0]" to distinguish 3 cases. Approx. 20 lines of code, 616 submissions.

Special in this data set is that the design of the exercises of the first lecture allowed couples of learners to submit solutions and also included the explanation of the submitted solutions in presence of a tutor. Before the audition the tutors tried to detect plagiarisms (supported by Plaggie and the Levenshtein approach) and could, then, confront the learners – for the course explaining and being able to re-code a solution allowed them to get the points under certain conditions. For the second lecture, no plagiarisms are known (cf. next section).

For the evaluation all tools were used with default parameters despite that we tried to get all pairs of solutions that were not classified as completely different (i.e., threshold $\geq 1\,\%$). JPlag and Moss have hard-coded limits of 1000 and 250 results. Moss' asymmetric results are converted to symmetric ones by using the maximum of the two similarities. The found similarities are then used to calculate the number of reported results, the number of found known plagiarisms, the histograms for reported similarities, the inter-rater agreement (Cohen's κ) on the classification that a tuple of two submissions has a similarity $\geq 80\,\%$, and some descriptive statistics on the reported similarities. As a ground truth the known plagiarisms are used that were found by the tutors and confirmed by learner interviews.

5 Results

All tools needed less than 15 minutes to complete. The detection results for the assignments are shown in Tab. 2 and 3. LV means Levenshtein distance and LVN Levenshtein distance with the mentioned normalization. The columns are the number of results (≥ 1 %) that a tool reported, the number of results with the default threshold (cf. Section 3), the number of reported pairs with a similarity ≥ 80 %, the maximum reported similarity of a tool, and the mean as well as the median of all similarities (on the full data set, including all 0 % similarities). For the first assignment, the number of found known plagiarisms respective the maximum/mean per cent value on the found plagiarisms is reported in parenthesis. The data show for the first assignment that all tools detected all similarities in some way, however, the assigned per cent values seem to be very different when the mean and median values are compared. Here, JPlag, Plaggie and SIM have a median of ≥ 99. Particularly, Sherlock-Sydney stands out here, only detecting 24 plagiarisms with the default threshold and general low mean scores (22.3 for the known plagiarisms). Exemplary, Fig. 1 shows the histogram of reported similarity scores for assignment 1 and allows to get an impression on how many results are reported for different thresholds. For the second assignment Tab. 3 shows that most approaches report a high number of high similarities – even with a reported similarity score ≥ 80 %. Particularly, the simple Levenshtein distance is notable here with an average ≥ 60 % as well as the low numbers for JPlag and Moss due to the low hard-coded thresholds. Interestingly, also SIM has very low numbers compared to the others.

Tool	No. results	No. def. thr.	No. ≥ 80 %	Max %	Mean %	Median %
JPlag	1000 (63)	40 (26)	190 (59)	100 (100)	3.2 (93.8)	0 (100)
LV	21528 (63)	21528 (63)	8 (8)	100 (100)	25.8 (52.3)	0 (46)
LVN	21528 (63)	21528 (63)	43 (36)	100 (100)	36.4 (81.7)	0 (65)
Moss	250 (63)	250 (63)	17 (14)	99 (99)	0.6 (62)	0 (58)
Plaggie	7647 (63)	227 (53)	90 (46)	100 (100)	8 (80.6)	0 (100)
Sherlock	11217 (63)	174 (24)	4 (4)	100 (100)	2.9 (22.3)	0 (17)
SIM	317 (63)	317 (63)	31 (27)	100 (100)	0.6 (51.2)	0 (99)

Tab. 2: Reported similarities of the analyzed tools on assignment 1 (in parenthesis the numbers for found known plagiarisms); LV: Levenshtein distance; LVN: Levenshtein distance with normalization

Tab. 4 and 5 show the inter-rater agreements. A value $\leq .2$ is considered as "no", $.21 - .39$ as "minimal", $.4 - .59$ as "weak", $.6 - .79$ as "moderate", $.6 - .8$ as "strong" and above as "almost perfect" inter-rater agreement [Mc12]. For assignment 1 (Tab. 4) the agreements of Plaggie/JPlag and LV/Sherlock are "moderate", SIM/LVN and Moss/SIM are "weak". Others such as Sherlock/ JPlag&Plaggie are very low. The human classification seems to agree "moderately" with Plaggie and LVN. For assignment 2 (Tab. 5) most agreements are very low – notable exceptions are Plaggie/LVN (strong) as well as LVN/LV and Plaggie/LV (weak). In general, most tools do not seem to agree well on quite simple tasks for a ≥ 80 % plagiarism classification and, therefore, cannot be interchanged easily. A more in-depth analysis is necessary, also whether this is different for more complex assignments.

Tool	No. results	No. def. thr.	No. ≥ 80 %	Max %	Mean %	Median %
JPlag	1000	288	1000	100	0.5	0
LV	189420	189420	31355	100	61.9	2
LVN	189420	189420	79165	100	73	3
Moss	250	250	15	99	0.1	0
Plaggie	99326	75431	52209	100	38.5	0
Sherlock	164968	81883	1051	100	19.4	0
SIM	978	978	290	100	0.3	0

Tab. 3: Reported similarities of the analyzed tools on assignment 2

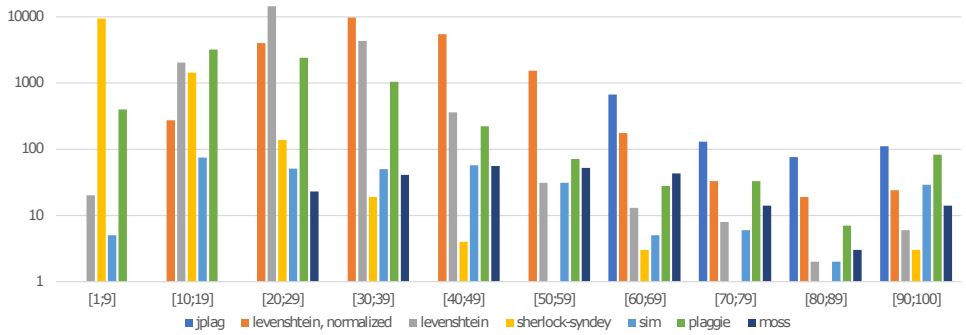

Fig. 1: Histogram of reported similarity per cent values for assignment 1

	Plaggie	Moss	SIM	Sherlock	LVN	LV	Human
JPlag	0.641	0.144	0.279	0.041	0.358	0.080	0.464
Plaggie		0.242	0.429	0.063	0.480	0.142	0.600
Moss			0.583	0.190	0.366	0.400	0.350
SIM				0.228	0.594	0.410	0.574
Sherlock					0.170	0.667	0.119
LVN						0.274	0.678
LV							0.225

Tab. 4: Inter-rater agreement on assignment 1 for ≥ 80 % classification

	Plaggie	MOSS	SIM	Sherlock	LVN	LV
JPlag	0.027	0.000	0.024	0.019	0.015	0.017
Plaggie		0.000	0.007	0.016	0.681	0.472
Moss			0.039	0.000	0.000	0.001
SIM				0.022	0.004	0.008
Sherlock					0.011	0.024
LVN						0.431

Tab. 5: Inter-rater agreement on assignment 2 for ≥ 80 % classification

6 Discussion, Conclusions and Future Work

In this paper different plagiarism detection tools were compared. Many of the published tools are, however, not available for use. Most analyzed tools are programmed in Java or can be started from CLI in order to integrate them into assessment systems. Moss, however, does not provide a good machine-readable result format (HTML). The often-mentioned tool Sherlock-Warwick was left out because it failed to run for assignments 1 and became unresponsive for assignment 2. This should be further investigated.

In practice a concrete threshold needs to be chosen. The threshold of 80 % was (arbitrarily) chosen for comparison in Tab. 2 and 3 because it showed to be a good value in practice. It allows (in combination with Fig. 1) to compare the different approaches to get a better feeling regarding the number of reported results and what the tools define as 80 % similar.

In several papers recall and precision or related metrics were calculated [NJK19]. However, a general problem is how to find a good ground truth – particularly for simple real world assignments. On the one hand we can never be sure to know all real plagiarisms. On the other hand the author's experience has shown that is also not easy for a human to be absolutely sure if a submission is a real plagiarism or just a very close implementation. This started to happen broadly by around 80 % similarity of all tools for assignment 1 and much earlier for assignment 2. Therefore, calculating false positives might not be possible at all. Consequently, no recall and precision are calculated but the absolute numbers of reported similar solutions as well as the number of the known plagiarisms among these are provided.

Based on the comparison for the more complex assignment 1 most tools were able to detect all known plagiarisms, however, the average reported similarity values vary a lot. For Sherlock-Sydney the average similarity per cent value on the known plagiarisms was just 22 whereas for JPlag it was 94. As most tools reported similarities from 1 to 100 % (exceptions are the ones with a hard-coded threshold) it seems to be more than just a scaling issue. These differences could also be observed for the inter-rate agreements in general where most tools do not agree well. For the very simple assignment 2 basically just consisting of a if-clause most tools reported a huge amount of highly rated similarities that is not usable in practice. Counterintuitively, this was also true for the Levenshtein approach (even without normalization). Here, maybe a hard limitation for too simple assignments does exist or the calculation formula needs to be improved. In general, it might be interesting to see if the combination of different approaches or whether other factors such as shared unique errors or considering multiple submissions at the same time yield better results. A more systematic analysis is planned (also including other languages).

Of course using plagiarism detection tools are just one way to counter plagiarism and have their drawbacks. None of these tools is a substitute for the academic's own intervention [MPH16]. Other approaches could be to (automatically) asks learners questions regarding their own solution to check their understanding [LSS21; SF20]. However, its is not clear yet, what minimum complexity of the assignments is required here.

The author would like to thank Iana Klinitska for building the basis of this research with her Bachelor's thesis.

References

[AM19] Ahadi, A.; Mathieson, L.: A Comparison of Three Popular Source code Similarity Tools for Detecting Student Plagiarism. In: Proc. Australasian Computing Education (ACE). ACM, 2019.

[CLS21] Cheers, H.; Lin, Y.; Smith, S. P.: Evaluating the robustness of source code plagiarism detection tools to pervasive plagiarism-hiding modifications./, Feb. 8, 2021, arXiv: cs.SE/2102.03997v1.

[HRV11] Hage, J.; Rademaker, P.; van Vugt, N.: Plagiarism Detection for Java: A Tool Comparison. In: Proc. CSERC '11. Pp. 33–46, 2011.

[KJH18] Keuning, H.; Jeuring, J.; Heeren, B.: A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. TOCE 19/1, 2018.

[La18] Luxton-Reilly, A.; et al.: Introductory Programming: A Systematic Literature Review. In: Proc. ITiCSE '18. ACM, pp. 55–106, 2018.

[LSS21] Lehtinen, T.; Santos, A. L.; Sorva, J.: Let's Ask Students About Their Programs, Automatically. In: ICPC. IEEE, pp. 467–475, 2021.

[Lu14] Luke, D.; Divya P.S; Sony L Johnson; Sreeprabha S; Varghese, E. B.: Software Plagiarism Detection Techniques: A Comparative Study. IJCSIT 5/4, 2014.

[Ma14] Martins, V. T.; Fonte, D.; Henriques, P. R.; da Cruz, D.: Plagiarism Detection: A Tool Survey and Comparison. In: Symposium on Languages, Applications and Technologies. Pp. 143–158, 2014.

[Mc12] McHugh, M. L.: Interrater reliability: the kappa statistic. Biochemia medica 22/3, pp. 276–282, 2012.

[MPH16] Modiba, P.; Pieterse, V.; Haskins, B.: Evaluating plagiarism detection software for introductory programming assignments. In: Proc. CSERC '16. ACM, 2016.

[NJK19] Novak, M.; Joy, M.; Kermek, D.: Source-code Similarity Detection and Detection Tools Used in Academia. TOCE 19/3, pp. 1–37, June 2019.

[SF20] Salac, J.; Franklin, D.: If They Build It, Will They Understand It? Exploring the Relationship between Student Code and Performance. In: Proc. ITiCSE '20. ACM, pp. 473–479, 2020.

[SOP11] Strickroth, S.; Olivier, H.; Pinkwart, N.: Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben? In: Proc. DeLFI '11. GI, pp. 115–126, 2011.

[SP17] Strickroth, S.; Pinkwart, N.: Eine Übersicht. In: Automatisierte Bewertung in der Programmierausbildung. Waxmann, pp. 17–38, 2017.

On the Influence of Task Size and Template Provision on Solution Similarity

Tobias Haan, Michael Striewe[1]

Abstract: In most cases of programming education, there is not a single correct answer to a given task. Instead, the same problem can be solved by two or more pieces of program code that look very different. At the same time, two or more pieces of program code that look very similar may actually solve very different problems. It is thus not easy to foresee which degree of similarity one can expect for all or at least the correct submissions to a given programming task. Since several applications may benefit from some kind of prediction of the similarity, this paper presents first, preliminary results from research on that topic. In particular, it presents results from an empirical study on the influence of exercise size and template provision. Results indicate that both factors are not suitable as simple predictors and that other factors have to be taken into account as well. Nevertheless, the results help to generate hypothesis for more detailed subsequent studies.

Keywords: Programming Education; Program Code Similarity; Solution Space

1 Introduction

Computer programming is known to be a domain in which there is typically not a single correct answer to a given programming problem. Depending on the type and complexity of the problem, there might be (1) different algorithmic strategies to solve it (e. g. recursive or iterative), (2) different possible implementations for the same strategy (e. g. a for-loop or a while-loop), (3) different syntactical representations of the same implementation (e. g. `i++` or `i=i+1`), and (4) virtually unlimited options for naming identifiers (e. g. variable names). Consequently, two pieces of program code can look very different but both solve a given problem. At the same time, two pieces of program code can look very similar, but produce entirely different results due to a subtle but important deviation.

While this reduces the value of a direct comparison between two pieces of program code, programming education knows nevertheless many useful applications for measuring program code similarity on the large scale. Similarity of program code can be used to cluster the “solution space” and consequently provide similar feedback to similar solutions [Gr12]. There are also approaches for automated grading support that suggest what manually crafted feedback can be applied to which submission based on similarity [He17]. The gains in efficiency of such methods directly correlated to the similarity of submissions. Approaches

[1] Universität Duisburg-Essen, paluno - The Ruhr Institute for Software Technology, Gerlingstraße 16, 45127 Essen, Deutschland, michael.striewe@paluno.uni-due.de

doi:10.18420/abp2021-7

that require the preparation of several sample solutions for different solution strategies (like e. g. [Ol19]) may also benefit from a reliable prediction of the expected similarity or diversity of submission. An analysis of the solution space can also be used to determine the relative difficulty of a task or the degree of freedom it allows [SG13]. Another quite common application is the search for plagiarism [PPM00]. Finally, structural similarity of program code can also be used for grading [Na07, TRB04]. In any of these cases, the result of a single comparison needs careful interpretation as already mentioned above. However, also general properties of the programming task may have a remarkable influence on the results. A solution template that is provided with the task may obviously cause some minimum of similarity between all solutions, regardless of their correctness. Detailed specifications within the task description like asking for a recursive solution or asking to implement a particular class inheritance structure also limit the solution space at least for correct solutions. However, the size of such effects seems not to be entirely predictable and thus it is hard to foresee what degree of similarity or diversity among solutions can be expected for a particular task. The possibility for better predictions in that area can be considered beneficial particularly in the context of technology-enhanced assessment. Knowledge about the size of the solution space can e. g. help to determine the quality of tests and feedback rules. At the same time, clustering of similar submissions can take into account if some minimal similarity is enforced due to the use of large templates.

The goal if this paper is to systematically look for possible correlations between the size of solutions, the size of provided code templates and the kind of programming task in a sample set of exercises from a programming lecture. To do so, it applies three different measures for program similarity to hundreds of solutions for six programming exercises from an introductory lecture on object-oriented programming in Java. The goal is to generate first useful insights that help to generate hypotheses for more detailed subsequent studies.

2 Measurement of Program Code Similarity

There are several ways to measure the similarity of program code. One class of techniques is based on graph comparison that is applied to the syntax tree or control flow graph of a program. Another class is based on lexical comparison that is applied directly to the program code. Finally, there is also the class of metric based comparisons, in which abstract representations of programs in terms of metric values are compared. For the sake of brevity, only the three approaches and tools used throughout this study will be explained in more detail, followed by an overview on other approaches.

Clone Doctor (CloneDR)[2] is a commercial tool for detecting “code clones”, which are pieces of code within a larger program that are identical or very similar. Code clones are usually considered problematic with respect to software maintainability. Software quality assurance thus tries to detect such clones. The approach used by CloneDR is based on the

[2] http://www.semdesigns.com/Products/Clone/

structure of abstract syntax trees [Ba98] and thus does not care about lexical properties like variable names or code formatting. Syntax trees are cut into sub-trees and hash values are computed for these trees. Sub-trees with identical hash values are considered identical, while other sub-trees are ranked by similarity. For the purpose of this paper, we use the number of code lines that are identified as clones by CloneDR and relate it to the total size of a program. The result is a similarity value in percent.

Deckard is another tool that searches for code clones based on syntax trees. Different to the previous approach, it introduces the notion of "characteristic vectors" to represent sub-trees [Ji07]. Based on these vectors and the ability to merge several vectors it is able to ignore additional, intermediate tree nodes like blocks or parentheses. Besides that, it also uses clustering and hashing of vectors for similar sub-trees. It appears to be both faster and more accurate than CloneDR [Ji07].

LAV[3] is a general-purpose tool for static program verification. Among other techniques, it uses control flow analysis and constructs a control flow graph for that purpose. That graph can also be used to analyse the similarity of two pieces of program code using "neighbour matching" to determine the degree of similarity [Vu13]. Different to both previous approaches, using the control flow graph not only ignores lexical differences but also other syntactical differences in the implementation that have no or only very limited influence on the control flow.

There are several other approaches that have not been used in this study. TBCCD [Yu19] and Holmes [Me20] are also tree-based clone detectors but use additional information from a lexical analysis of program code or the analysis of program dependency graphs, respectively. Both approaches are quite new and seem to perform better than existing approaches, but there is only little documentation available and they could thus not be applied easily on our set of test data. Deltacon [KVF13] is an algorithm for measuring the similarity of general graphs. It could thus also be applied to syntax trees, but it is specialised on comparing graphs with known node correspondence and thus focuses on the detection of differences in the connectivity. Hence, it is not suitable for the purpose of our study.

3 Empirical Study

The overall plan of the study is to use existing submissions for programming exercises, compute similarity values for these and relate these to other characteristics of the exercises and submissions, i. e. to the code (template) size and the correctness of the respective submission. No sophisticated statistical methods will be used, as it is the intention to look for obvious correlations (or their absence) and to generate hypothesis for future research.

[3] urlhttp://argo.matf.bg.ac.rs/?content=lav

3.1 Data and Method

The empirical study is based on submissions to programming exercises from winter term 2019/20. We used six exercises from an introductory lecture on Java programming. Submissions were collected via an e-assessment system with automated feedback and students were free to submit as often as they wanted. Consequently, we collected incomplete and incorrect submissions, but also submissions that received full credit. Table 1 gives an overview on the size of each exercise (measured in lines of code for template and average submission) and the number of submissions. The templates followed the usual style of writing Java code with not more than one statement per line and the vast majority of submissions are written the same way. Hence, lines of code is indeed a usable approximation for the size of the code.

	Avg. total submission size (LOC)	Template size (LOC)	Share of template on avg. submission	Number of total submissions	Number of correct submissions
Exercise 1	45	15	0.33	1352	452
Exercise 2	228	145	0.64	4058	1182
Exercise 3	158	46	0.29	3078	795
Exercise 4	529	365	0.69	4790	652
Exercise 5	69	34	0.49	5422	1897
Exercise 6	211	116	0.55	3432	1005

Tab. 1: Overview on the six sample exercises used in the empirical study.

Exercise 1 is concerned with variables and the `if`-statement. The code template provides method signatures and students are asked to implement simple calculations within these methods. Exercise 2 focuses on constructors, arrays and loops. Students receive two Java files as code template with predefined method signatures and are asked to perform some operations (like searching, sorting or summarizing) on 1- and 2-dimensional arrays. Exercise 3 asks the students to implement a list data structure and perform operations on it. Students received a single file as code template and an additional file with code that must not be altered. Exercise 4 is similar to the previous one, but deals with a binary search tree as data structure. Exercise 5 is the one with the most files to be edited since it is concerned with inheritance structures. Students receive six files and must make changes and additions to four of them to realize the correct inheritance of classes and methods. Exercise 6 deals with enumerations, interfaces and maps. Again, student receive six files, but only need to alter and extend three of them. The size of the code templates given in Table 1 only refers to files that need to be touched by students and ignores files that must not be altered.

For each exercise, we created a clone detection report with CloneDR and performed a pair-wise comparison of all submissions with Deckard and LAV. We normalized all results

on a scale from 0 (no similarity) to 1 (full similarity). For Deckard and LAV, we used the average and median similarity values for a first analysis. Admittedly, both measures are weak as they can produce misleading results if the distribution of similarity values has more than one peak. However, both measure are very close to each other in most cases and a more detailed inspection of the distribution of similarity values revealed no cases in which there were two or more clear peaks. Nevertheless, the following results are only a first step towards a more detailed analysis. Some indicators for the need for a more detailed search for peaks or clusters are discussed below.

3.2 Results

The results from all comparisons are summarized in table 2. As a general observation, CloneDR produces clearly higher similarity values than the other tools, while LAV produces slightly higher values than Deckard in all cases except for exercise 2.

There is no obvious trend on the difference between correct submissions and all submissions. CloneDR produces lower similarity values for correct submission for all exercises except for exercise 5. Deckard and LAV produces lower values in the same three cases and higher values in the other three cases. There is no obvious correlation that these are exercises with a remarkable high or low share of correct submissions. In exercises 5 and 6 the difference between the values for all and correct submissions is larger than 0.05 in all cases (except for exercise 6 in CloneDR), while it is below 0.02 in almost all other cases.

Moreover, there is no obvious correlation between the similarity of submissions and the size or share of the code template. Exercise 1 and 3 have a low share of the code template (0.33 and 0.29) and also the lowest similarity values in all tools. However, exercise 5 has a moderate share of the code template (0.49) but high similarity values in all tools, including the highest for correct submissions in CloneDR and LAV. Exercises 1, 3 and 5 are the ones with the smallest code templates, but they span the entire range of similarity values. In turn, exercise 4 has the highest share of the code template (0.69), but remarkable lower similarity values in Deckard and LAV than exercise 2, in which the code template has a quite similar share (0.64).

There is also no obvious correlation between the exercise size in terms of code produced by students (which is the difference between the average submission size and the size of the code template) and the similarity values. Exercises 1 and 5 have a similar amount of code produced by students (30 and 35 lines), but very different similarity values. At the same time, Exercise 4 and 5 have similarity values that are close to each other, but require an entirely different student contribution (164 vs. 35 lines).

	CloneDR		Deckard		LAV	
	all	correct	all	correct	all	correct
Exercise 1	0.480	0.448	0.224	0.221	0.238	0.235
			0.196	0.195	0.209	0.207
Exercise 2	0.744	0.743	0.447	0.467	0.442	0.462
			0.447	0.468	0.442	0.463
Exercise 3	0.713	0.695	0.196	0.195	0.207	0.207
			0.156	0.174	0.166	0.186
Exercise 4	0.830	0.817	0.386	0.366	0.408	0.389
			0.370	0.360	0.394	0.383
Exercise 5	0.759	0.842	0.398	0.454	0.422	0.481
			0.394	0.454	0.421	0.486
Exercise 6	0.799	0.779	0.367	0.422	0.388	0.448
			0.337	0.420	0.359	0.449

Tab. 2: Similarity values for all and only correct submissions. For CloneDR, the overall share of clones is listed as provided by the tool report. For Deckard and LAV, average (first row per exercise) and median (second row per exercise) values from the pair-wise comparison of submissions are reported.

3.3 Discussion

The missing obvious correlations in the results indicate that neither exercise size in terms of code created by students nor template size in terms of lines of code provided to students are good predictors for the size of the solution space. This is not surprising on the first glance, since there are other factors like the task descriptions (that may prescribe or disallow certain patterns) and the knowledge level of students that may also increase or limit the solution space. Nevertheless, it is interesting to see that exercise 1 is among the exercises with the lowest similarity values although it is small, has a clearly defined scope and is used at the beginning of the lecture where students are expected to have a low level of knowledge. Different to that, exercise 2 is used slightly later in the lecture and theoretically gives much more possibilities to implement algorithms on arrays, but is nevertheless among the exercises with the highest similarity value.

The relative small differences between the similarity values for all and correct solutions in exercises 1 to 4 and the missing trend towards an increased or decreased similarity for correct submissions seems to hint towards a lack of discrimination between the two categories. In fact, there may be two competing factors, one for each category: On the one hand, correct submissions can be expected to be similar if student stick closely to solution patterns and strategies they have seen in the lecture or other sources. Deviations from these patterns may cause errors and thus increase the diversity of all submissions, but not the

correct ones. On the other hand, incorrect submissions may be intentionally incomplete if students have only worked on some part of the exercise so far. Consequently, they have not yet touched and filled parts of the provided code template. That may cause a higher similarity in all submissions that will vanish once they work towards a more complete submission. For exercises 5 and 6 a larger difference with higher similarity values for correct submissions could be observed. This may indicate that there are only few correct possible solutions on these exercises. For further studies, we may need to create more categories, ignore submissions that are incomplete attempts, or apply advanced statistical methods to find clusters that can be explained by other means.

With regard to tools, the results did not match our expectations we had based on the underlying approaches. Although CloneDR and Deckard are somewhat similar in their approach, they produce quite different results. At the same time, Deckard and LAV produce quite similar results although their approaches are different. Since the relative high absolute values from CloneDR do not match our subjective impression of submission similarity, we will most likely focus on the other tools in subsequent studies.

4 Conclusions

We investigated the influence of exercise size and template provision on submission similarity. Results indicate that neither exercise size nor template size seem to be simple predictor for the size of the solution space.

Nevertheless, the study provides useful insight that helps to generate hypotheses for subsequent studies. The first is, that a more fine-grained discrimination of submissions (e. g. creating groups of submission also for partially correct submissions or partially unchanged templates) helps to describe better how the similarity between submissions evolves over time. This may be particularly useful in the context of automated grading, where scores are easily available for each submission and where grouping by scores is more or less meaningful depending on the actual similarity of submissions. A second hypothesis is, that exercise size and template size have to be studied in conjunction with other factors like task instructions and previous knowledge of students to come up with a predictor for the size of the solution space. This is important for the further development of tool support, because some factors are much easier to measure and quantify than others. Finally, there is also room for the hypothesis that existing approaches for computing the similarity of program code are not entirely suitable for the context of (small) programming exercises. Further studies on the agreement between tools as well as on the agreement with a subjective human understanding of code similarity may help to develop approaches that are specifically tailored for the use in conjunction with programming exercises.

Bibliography

[Ba98] Baxter, Ira D.; Yahin, Andrew; Moura, Leonardo; Sant'Anna, Marcelo; Bier, Lorraine: Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance. IEEE, pp. 368–377, 1998.

[Gr12] Gross, Sebastian; Mokbel, Bassam; Hammer, Barbara; Pinkwart, Niels: Feedback Provision Strategies in Intelligent Tutoring Systems Based on Clustered Solution Spaces. In (Desel, Jörg; Haake, Joerg M.; Spannagel, Christian, eds): DeLFI 2012: Die 10. e-Learning Fachtagung Informatik. Hagen, Germany, pp. 27–38, 2012.

[He17] Head, Andrew; Glassman, Elena; Soares, Gustavo; Suzuki, Ryo; Figueredo, Lucas; D'Antoni, Loris; Hartmann, Björn: Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In: Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale. pp. 89–98, 2017.

[Ji07] Jiang, Lingxiao; Misherghi, Ghassan; Su, Zhendong; Glondu, Stephane: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In: 29th International Conference on Software Engineering (ICSE'07). pp. 96–105, 2007.

[KVF13] Koutra, Danai; Vogelstein, Joshua T.; Faloutsos, Christos: Deltacon: A principled massive-graph similarity function. In: Proceedings of the 2013 SIAM International Conference on Data Mining. SIAM, pp. 162–170, 2013.

[Me20] Mehrotra, Nikita; Agarwal, Navdha; Gupta, Piyush; Anand, Saket; Lo, David; Purandare, Rahul: Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks. CoRR, abs/2011.11228, 2020.

[Na07] Naude, Kevin Alexander: , Assessing Program Code through Static Structural Similarity. Master's Thesis, Faculty of Science, Nelson Mandela Metropolitan University, 2007.

[Ol19] Olbricht, Christoph: Trace-Vergleich zur Feedback-Erzeugung im automatisierten E-Assessment-System JACK. In: Proceedings of the Workshop Automatische Bewertung von Programmieraufgaben (ABP 2019). pp. 11–18, 2019.

[PPM00] Prechelt, Lutz; Philippsen, Michael; Malpohl, Guido: JPlag: Finding plagiarisms among a set of programs. publikation, Universität Karlsruhe, Fakultät für Informatik, Germany, January 2000.

[SG13] Striewe, Michael; Goedicke, Michael: Analyse von Programmieraufgaben durch Softwareproduktmetriken. In: SEUH. pp. 59–68, 2013.

[TRB04] Truong, Nghi; Roe, Paul; Bancroft, Peter: Static Analysis of Students' Java Programs. In (Lister, Raymond; Young, Alison L., eds): Sixth Australasian Computing Education Conference (ACE2004). Dunedin, New Zealand, pp. 317–325, 2004.

[Vu13] Vujošević-Janičić, Milena; Nikolić, Mladen; Tošić, Dušan; Kuncak, Viktor: Software verification and graph similarity for automated evaluation of students' assignments. Information and Software Technology, 55(6):1004–1016, 2013.

[Yu19] Yu, Hao; Lam, Wing; Chen, Long; Li, Ge; Xie, Tao; Wang, Qianxiang: Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In: IEEE/ACM 27th International Conference on Program Comprehension (ICPC). pp. 70–80, 2019.