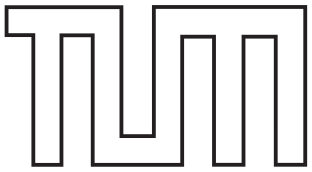


TECHNISCHE UNIVERSITÄT MÜNCHEN

Computing Education Research Group
School of Social Sciences and Technology

Bachelor Thesis

for the attainment of the academic degree
Bachelor of Education (B.Ed.)

Comparing different ways to give automated feedback on programming exercises in K-12 education

Valentin Titus Herrmann

2024

TECHNISCHE UNIVERSITÄT MÜNCHEN

Computing Education Research Group
School of Social Sciences and Technology

Bachelor Thesis

for the attainment of the academic degree
Bachelor of Education (B.Ed.)

Comparing different ways to give automated feedback on programming exercises in K-12 education

Author: Valentin Titus Herrmann
Examiner: Prof. Dr. rer. nat. Tilman Michaeli
Supervisor: Prof. Dr. rer. nat. Tilman Michaeli
Submitted: 15.09.2024

Declaration on the paper according to § 29 (para. 6) LPO I

This thesis replaces the written paper according to § 29 para. 1 sentence 1 no. 1 LPO I, which is required for admission to the first state examination for the teaching profession at grammar schools in Bavaria, according to § 29 para. 12 sentence 1 no. 3 LPO I. In accordance with § 29 Para. 6 LPO I, I hereby declare that I have written this Bachelor's thesis independently and that I have not used any aids other than those specified. The passages in the thesis that are taken from other works in terms of wording or meaning are marked as borrowings in each case, stating the source.
This declaration also extends to any graphics, drawings, sketch maps and pictorial representations contained in the work.

Erklärung zur Hausarbeit gemäß § 29 (Abs. 6) LPO I

Die vorliegende Bachelorarbeit ersetzt die schriftliche Hausarbeit nach § 29 Abs. 1 S. 1 Nr. 1 LPO I, die für die Zulassung zur ersten Staatsprüfung für das Lehramt an Gymnasien in Bayern gefordert ist, gemäß § 29 Abs. 12 S. 1 Nr. 3 LPO I. Entsprechend § 29 Abs. 6 LPO I erkläre ich hiermit, dass die vorliegende Bachelorarbeit von mir selbstständig verfasst worden ist und keine anderen als die angegebenen Hilfsmittel benutzt worden sind. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen sind, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.
Diese Erklärung erstreckt sich auch auf etwa in der Arbeit enthaltene Grafiken, Zeichnungen, Kartenskizzen und bildliche Darstellungen.

Munich, 15.09.2024,

Valentin Titus Herrmann

Abstract

Autograding systems that provide automated feedback to the students are indispensable in most university courses. Novices are provided with tailored feedback that supports them with solving exercises and scaffolds their learning process.

Several studies have been conducted on a high level in those courses, e.g., comparing overall course results and exercise scores. However, only a few considered design principles and strategies for automated feedback. Besides that, no literature is present about autograding systems in a K-12 educational context. On the other hand, the field of providing offline scaffolding and giving feedback in K-12 education has been investigated intensively.

This thesis tries to close the gap between these two research fields by conducting a small study on programming exercises with TUM's autograding system ArTEMiS [1] with three 10th grade courses at a Bavarian grammar school.

The results indicate a high impact of the exercise topics and the type of mistakes the students made on the effectiveness of feedback strategies. Unfortunately, the sample was too small to discover differences between the feedback strategies for all exercise topics and mistake types. However, the results are a good starting point for further research.

Contents

1 Introduction

Learning programming seems to be quite a complex topic. Novices lack not only knowledge but also strategic skills to solve problems [2]. Additionally, learning groups suffer from a high heterogeneity in and between groups [3]. Learning to program in schools and universities usually happens within a fixed timeframe of semesters and examinations, leading to a high learning pace and teachers to continue when only 25% of the students understood the topic (Lundgren-effect [4]). Another factor contributing to these difficulties is that students lack the skill to determine if their solution is correct. Especially weaker students need a teacher who can always help immediately [2]. This leads to the regular phenomenon of teachers running around the classroom to help students with their problems. This is so regular that the term *sneaker didactics* has been established in German cs education research [5].

Facing these problems, universities developed the idea of automatic assessment of (programming) exercises to handle the vast novice courses. Today, no learning to program in university without an automatic assessment system (also called autograders) is imaginable [1, 6, 7]. Although the use of autograders sometimes leads to lousy learning behavior [6], the possibility of receiving individual feedback in a short time significantly supports students to create better solutions, keep motivation high and not give up fast [2] is highly desirable.

The effectiveness of automated feedback is - like every type of scaffolding - highly dependant on the form and goal of it [8, 9, 10, 11]. Besides supporting students when working on tasks about already known topics

As there has been a lot of research on the general approach of using autograders in university courses and some research on giving feedback in K-12 education, the goal of the thesis will be to determine which way of designing exercises and automatic feedback is the most effective in a K-12 educational context when using TUM's automatic assessment platform ArTEMiS [1, 12].

2 Related Work

2.1 Difficulties in Learning to Program

As mentioned, learning to program is a highly complex topic [2]. This issue starts with understanding underlying concepts and learning a language's syntax and reaches to designing exercises for repetition or construction of new knowledge. Additionally, high heterogeneity in and between groups, as well as negative expectations about the subject caused by forwarded traumas of previous generations, cause low motivation [2, 13].

2.1.1 Multitasking

Learning to program is not only learning one but several skills at once. First, students have to learn how to use a computer and its software properly. It begins with basic skills like opening and saving files and using IDEs and expert tools like the version management software Git. Therefore, using an IDE developed for education purposes is highly recommended [3].

Furthermore, programming and modeling tasks are always performed in a certain context. These contexts can range from mathematics and business administration to everyday topics in the student's lives. Students confused about a task's topic - because they lack knowledge in that field - face a tough time during modeling and implementation [2, 3]. Therefore, when designing exercises, the relevant knowledge of other subjects must be considered to reduce the cognitive load.

Last but not least, students have to learn computer science concepts and a programming language when learning to program. Even these skills are multilayered: Students must learn concepts, modeling, syntax, semantics, and style simultaneously because they all depend on each other [14].

2.1.2 Motivation

The motivation and will to learn build the foundation for success [2]. Therefore, this aspect's impact must be considered when looking for reasons for the low success of programming novices.

Research shows several possible reasons for a low motivation to learn programming. The already mentioned complexity and the multiple layers of that topic require strong persistence, making it difficult to build motivation if it does not already exist. Furthermore, programming is a trial-and-error intensive subject, a learning style students are usually not familiar with from other subjects and therefore has to be learned to prevent frustration [2].

Additionally, if students miss something in class or don't understand an aspect, they won't be able to follow upcoming lessons. This leads to them thinking they cannot program at all and develop a pattern of learned helplessness. For the students who were left behind due to the pace of the class, the problem worsens as the framework of our education system sets the timeline according to the need for assessment and not the students' learning speed. Therefore, it would be essential to find a way to allow students to learn at their own pace [3].

2.1.3 Typical Mistakes of Novice Programmers

Identifying and addressing typical mistakes of students is a key part of every teacher's competence. However, unlike other STEM subjects, the research on typical mistakes and misconceptions is not fully developed in the field of computer science education [15]. Unfortunately, many educators still see mistakes or misconceptions as obstacles to learning. However, being able to specifically address identified problems is crucial to improving the students' skills. Especially as it is known that many

instructors show a weak understanding of potential misconceptions their students have, identifying and collecting them is essential [15]. In the following section, typical conceptual mistakes that are made by programming are collected to gain a better understanding of what to look for when designing test cases and exercises based on a literature review by Qian and Lehman [15]:

Variable Misconceptions can cause trouble because students do not understand that variables can only store one value at a time, how values are assigned (e.g. $a = 5 \neq 5 = a$), that the name of the variable does not affect the value (e.g. *largest*), the scope of a variable (e.g. global vs. local), where data comes and from and where it is stored and that print statements do not change variable values (e.g. 'value of X is 1') [15, 16].

Conditional Misconceptions can be that both if and else blocks will be executed or that if a condition is false, the whole program will stop instead of skipping a certain part [15].

Loops Misconceptions are such where students can not identify correctly which lines will be repeated and think that one loop type (for vs. while) is superior based on their familiarity with them. Additionally, some students have the misconception that the loop body is executed whenever the condition comes true, no matter what code is currently executed [15].

Sequential Execution Misconceptions describe a case where students struggle to identify the order of code execution. A typical case is the call of functions where students do not know where the parameter values come from and where the return value goes. Students also often struggle with the three-line-variable value swap, where the values of two variables are swapped using a third helper variable. [15, 16].

OOP Misconceptions include several problems related to object-oriented programs. This ranges from thinking classes are a collection of objects, objects being a subset of a class, the relationship between objects, confusion between static and non-static elements, call-by-ref vs. call-by-value, and the general approach of decentralized architecture [15].

Natural Language can be confusing for programming novices. This can lead to problems where variable names and string values are mixed up, some words have opposing everyday and scientific meanings, and there is often confusion about the boolean operators *and/or* [15].

Math Last but not least, algebraic operations sometimes work differently than in maths, a subject students are usually familiar with when beginning to program. Therefore, students often get confused by assignments of values (programming) that look similar to algebraic equations (maths) and division of integers, leading to integers instead of floats [15].

Unfortunately, properly identifying misconceptions requires a lot of work and data. Therefore, this thesis will analyze concrete mistakes that occur instead of trying to identify which misconceptions cause these mistakes. However, the design of the test cases to assess the students' submissions will be based on the above-mentioned misconceptions to get results that are as fine-grained as possible.

2.2 Scaffolding

Scaffolding has been studied extensively in the past years in cs and general education. Nevertheless, no consensus concerning the definition of scaffolding seems to exist [17, 18].

The **definition by Wood et al. [19]** described it as a process that enables a child or a novice to solve a problem, carry out a task, or achieve a goal beyond his unassisted efforts [18].

Whereas **Stone [20, 21] defined scaffolding** as an interactive process that occurs between teacher and student who must both participate actively in the process [17].

Yet independent from the exact definition of scaffolding all papers agree that scaffolding is effective [17, 22, 23]. Additionally, some common characteristics of scaffolding definitions have been identified by van de Pol et al. [17]:

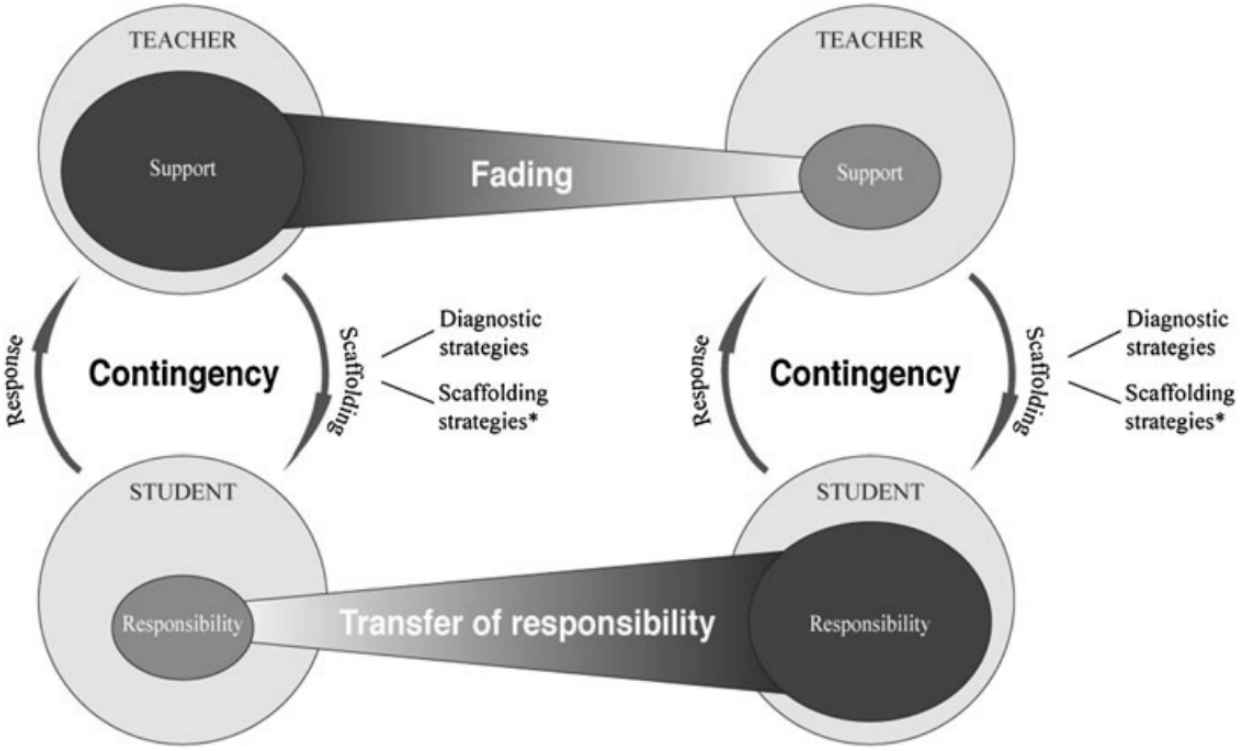
Contingency The Level of support is adjusted to the student's level. Therefore, the level of competence must be determined.

Fading Support is decreased over time.

Transfer of responsibility The responsibility for the learning success is transferred from the teacher to the student when the student is taking increasing control.

The correlation of these common scaffolding characteristics is visualized in Figure 2.1.

Figure 2.1 Conceptual scaffolding model visualizing the common characteristics [17]



This thesis will thereby follow the definition of Wood et al. [19] combined with the common characteristics identified by van de Pol et al. [17].

2.2.1 Effectiveness of (computer-based) Scaffolding

A more specific type of scaffolding - the most relevant type in the context of cs education - is computer-based scaffolding, which uses computers to provide the scaffold for learning. Even though (computer-based) scaffolding is less effective than one-to-one human tutoring [22] it still produces a medium effect on learning (g=0.53) [22]. The impact of this effect is increased as the more effective

one-to-one human tutoring can not be realized in most classroom settings as it usually requires a huge time investment by teachers [7] who mostly tend to help the students close to them and therefore oversee others [24].

A more differentiated approach of investigating the effectiveness of scaffolding is shown in the meta-analysis by Kim et al. in 2020 [23]. Considering the different options to provide scaffolding to students, three types of scaffolding have been identified: *conceptual, strategic, and metacognitive.* Additionally, expecting that individual or group work also affects the effectiveness of scaffolding, the effectiveness has also been partitioned by group sizes. The meta-analysis also tried to determine the effectiveness of motivational scaffolding but could not find mentionable results [23].

Table 2.1 Effectiveness (Hedge's g and 95% Confidence Interval) of computer-based scaffolding (own table according to Kim et al. [23])

Group	overall		conceptual	strategic	metacognitive
Single	0.47 (0.41,0.53)		0.48 (0.40,0.56)	0.47 (0.37,0.57)	0.38 (0.20,0.56)
Pair	0.59 (0.43,0.76)	0.44 (0.35,0.53)	0.46 (0.35,0.57)	0.39 (0.25,0,52)	0.48 (0.05,0.91)
Triad	0.41 (0.25,0.57)				
4-6	0.34 (0.20,0.48)				

Table 2.1 provides an overview of how effective different types of scaffolding applied to which group sizes are. The results indicate that the highest effect of all kinds of summarized scaffolding can be achieved when students work in pairs (g=0.59). If Students work in triads, scaffolding is slightly less effective (g=0.41) than when students work individually (g=0.47). In settings where students work individually, the most effective types of scaffolding are conceptual and strategic (g=0.48 and g=0.47), whereas metacognitive scaffolding is considerably less effective (g=0.38). When working in groups, students benefit the most from metacognitive scaffolding (g=0.48). However, the 95% confidence interval of this finding is very big (0.05,0.91), which indicates that the effectiveness of metacognitive scaffolding in groups has to be further researched. The second-best results in group settings could be achieved with conceptual scaffolding (g=0.46), while strategic scaffolding is less effective (g=0.39) [23].

The results of Kim et al. [23] indicated that if scaffolding shall be applied in group settings, it's best to have students working in pairs or, at most, triads while using conceptual or metacognitive scaffolding.

Considering the setting in which the effectiveness of scaffolding is studied, the findings of Heather Leary et al. [22] show that in authentic settings - which have more threads to internal and external validity - scaffolding achieves higher effects. Thus, educators can have confidence in the effectiveness of scaffolds even if the studies suffered from threads to validity [22].

2.2.2 A Framework to describe Scaffolding

Wood et al. [19] speak of six scaffolding functions: *recruitment, reduction of degrees of freedom, direction maintenance, marking critical features, frustration control, and demonstration* [17, 19] Additionally several papers suggest a further distinction between *tools/means* and the *goals/intentions* [17]. Combining these ideas into a common framework to explore scaffolding van de Pol et al. [17] described a framework that distinguishes

Five Intentions of Scaffolding

Direction maintenance refers to keeping the learning on target and maintaining the learner's pursuit of a particular objective. This intention is mostly metacognitive.

Cognitive structuring is an intention where the teacher provides explanatory and belief structures that organize and justify.

Reduction of the degrees of freedom describes taking over those parts of a task the student is not yet able to perform and thereby simplify the task.

Recruitment helps to get students interested in a task.

Frustration Control describes a system of reward or punishment as well as keeping students motivated via the prevention or minimization of frustration.

Six Means of Scaffolding

Feedback involves providing information regarding the student's performance to the student.

Giving Hints describes the provision of clues or suggestions to help the student proceed while not revealing the entire solution or detailed instructions.

Instruction involves the teacher telling the students what to do.

Explaining refers to providing more detailed information or clarification.

Modeling is the process of offering behavior for imitation.

Questioning involves asking students questions that require an active cognitive answer.

Based on this framework, any combination of scaffolding means with scaffolding intention can be constructed as a *scaffolding strategy* [17].

2.2.3 Feedback

Similar to scaffolding, the literature defines Feedback in several different ways, and no consensus seems to exist. A common approach is to define feedback as information communicated to the learner to modify his or her thinking or behavior to improve learning [7]. As this definition seems too generic for detailed research on how to design automated feedback, this thesis will stick to the definition in the scaffolding framework by van de Pol et al. [17]: *'Feedback involves providing information regarding the student's performance to the student.'* - enriching it with other means of scaffolding to find out how to provide the best possible experience to the students.

Besides the positive effects of scaffolding, the effectiveness of feedback as a means of scaffolding has been examined in several papers. As a very versatile means of scaffolding, it does not surprise that feedback can impact all described intentions of scaffolding positively [11]. Additionally, to the advantages of scaffolding, feedback strengthens low-confidence answers [25] and can significantly improve the student's learning experience by boosting the novices' motivation [10].

Considering our definition of feedback, it provides information about the student's performance. This information can be presented in several ways:

Binary Feedback provides information about the correctness of a submission or the correct percentage but no details about what is wrong [7, 11]. Unfortunately, this type of feedback is useless to improve a submission as it does not provide information that allows students to understand the gap between their current and desired state [7, 8].

Elaborated Feedback includes information about the student's mistake [11]. Elaborated feedback can be subdivided into:

Knowledge about Concepts containing explanation of the subject which has been applied wrongly or examples which illustrate the concept [7].

Knowledge about Mistakes provides detailed information of what is wrong. This can be done by describing unit test cases that failed, displaying compiler errors, hints to timeout or code style. More specifically, this type of feedback can be implemented using *location hints* (Where is something wrong?), *data hints* (Which input leads to which wrong output?) or an *example hint* (Which sample input should lead to which output?) [7, 11].

Knowledge about How to Proceed can contain hints on how an expert would usually solve a certain bug pattern or suggest code snippets for how to solve a part of the task that blocks proceeding [7, 11].

Knowledge about Task Constraints includes mostly general hints on the requirements and hints not considering the student's submission [7, 11].

Knowledge about Meta-Cognition providing students meta-cognitive strategies and knowledge for self-regulating their learning (e.g., hints about useful information sources) [11].

Knowledge about Solution is an almost binary version of elaborated feedback and shows the student (a part of) the solution [7, 11]

Scaffolded Feedback is feedback that contains more information with every wrong submission. For instance, when the exercise is to translate words into another language, one more letter of the correct answer is revealed with every wrong try [11]. Scaffolded Feedback can be implemented with any means of elaborated feedback.

2.2.4 Application of Feedback

Whenever a certain type of feedback is deployed, the structure is the same:

An error is detected - based on its type, a feedback strategy is selected - a feedback item of that strategy is deployed to the student. [26]

The crucial part of this process is selecting a proper feedback strategy. Knowledge about the (dis)advantages of each strategy is important when selecting a strategy. Especially when considering that wrong knowledge constructed based on feedback can hardly be corrected [25].

Whenever feedback is given it's important that it implementable and thus contains information about a certain mistake [9]. This is also visible by a closer look at binary feedback, which is often found to lead to lousy behavior like cheating or working on fewer exercises, especially when applied in an automatic assessment context [7]. Therefore, some kind of elaborated feedback should be used. Hereby, the best results have been achieved with feedback that required cognitive processing [25] while being corrective, especially novices benefitted from procedure hints [10]. Another important finding is that besides the mentioned constraints, a good amount of variety and a deep level of detail lead to more effective feedback [7].

Another important aspect is that feedback is deployed fast enough for students not to get frustrated but still delayed enough to encourage them to think about it [9]. Surprisingly, according to a meta-analysis by Heather Leary et al. [22], students did better when the scaffold was not faded.

Considering the instructiveness of feedback, the findings of Greifenstein et al. [9] implicate that direct instructions have a slightly negative (r=-0.13) effect on learning, while hints have a positive

effect (r=0.17). This is explained by the positive effect of autonomy [9]. Besides the positive effect of corrective feedback on the construction of knowledge, it has also been noted that too much of it reduces the students' motivation [10].

2.3 Designing exercises

2.3.1 Types of (programming) tasks

Simões and Queirós [27] collected different types of programming exercises with their individual up- and downsides:

Code from scratch where students start with an empty project. Students often struggle with focusing on important aspects, violating OOP patterns and the blank page syndrome, not knowing where to begin. The exercise preparation is fairly simple, as no template or solution has to be created. However, it is very complicated to create unit tests to verify the correctness of a submission for this type of exercise.

Code Skeleton describes an exercise where the teacher provides a code skeleton to scaffold students to focus on the task relevant to the intended learning goal. Yet preparing such exercises is costly for the teacher as a full solution has to be generated and disassembled for the template. This type of exercise can be unit-tested easily.

Code Baseline exercises are such that provide a working piece of code that has to be modified to fulfill certain requirements. To do so, students have to understand the existing code before being able to modify it. Such exercises can also easily be unit-tested and provide implicit gamification. However, this type does not work for all cs concepts.

Find the Bug is an exercise pattern where - similar to the code baseline - a piece of code is provided. This code contains certain bugs that must be found but have not been fixed. This kind of task can be solved unplugged and is, therefore, a good option if no computer can or shall be used.

Fix buggy Code extends the Find-the-Bug type to fix a bug. It can be scaffolded by telling the students where the bug is. In any case, the code has to be read and understood before the bug can be fixed.

Compiling Errors are shown to students with the corresponding code. This supports the careful reading of compiler and error output instead of relying on automated features of the used IDE and improves an understanding of how the compiler works.

Code Interpretation refers to tasks where students are given a piece of code, and they have to answer questions like 'What does this code?', 'What is the output?' or 'Which option is correct?'. This approach forces students to read (others) code carefully. This approach is also suitable for cases where no computers are available.

Fill the Blanks exercises are a version of the code skeleton with smaller gaps and no use of computers/IDEs. These exercises prevent students from brute force coding exercises.

2.3.2 Further for Designing Programming Exercises

When designing programming exercises based on either of the types listed above, it is important to keep in mind that there is not the *one* best type and design, and multiple ways of assessment are important [28]. However, when mixing the perfect exercise cocktail some general aspects must be considered.

For many students, shifting from passive learning to active problem solving is difficult if they are already familiar with traditional teacher-led instruction [23]. Yet scaffolding this by using the exercise type of modifying code samples does not lead to success as students tend to skip these kinds of tasks because they think they're supposedly boring [29]. Therefore, it has to be ensured that students do not skip exercise parts while avoiding continuous grading which causes a lot of pressure on the students [3]. Rather, emphasizing comprehension instead of exercise outputs can support their learning. However, it must be remembered that a correct solution does not equal understanding of concepts - yet, the understanding could be verified by questioning students about their code [30].

Additionally, tasks should inspire to keep interest high [3] but still ensure that if, e.g., the algorithms aren't the key concept to be learned, they should be as simple as possible [29]. Another problem to be avoided is sub-tasks stacking up on each other, preventing students from solving the following tasks if previous ones could not be solved correctly [2].

When choosing the context of an exercise, it has to be made sure that students can copy-paste as little code as possible from lecture notes or the internet because the procedure of just copy-pasting code is a reason for the loss of concepts and often occurs if students are overwhelmed by the size of an exercise [29]. To keep the size small even if a bigger context is desired, the *many-small-assignments-approach* where a big task is split into sub-tasks which can be treated individually and altogether form a bigger piece of software is promising [31]. Additionally, it is highly recommended to teach and use computer science vocabulary to communicate exercises efficiently [28].

Finally, it has been found that students tend to code without testing [29] and ask for help very late, which leads to big amounts of untested code that sometimes has to be completely rewritten, which then leads to frustration [29].

2.4 Autograders

Facing a huge amount of novice programmers struggling in programming courses and a lack of capacity to provide constant fast support by teachers [1, 2, 7] universities started to use autograding systems that became irreplaceable [6]. The term autograder refers to any platform where students submit a solution for a programming exercise and receive some feedback about the correctness of their submission.

2.4.1 Potential Problems of Autograder Use

As described in section 2.1.1, programming novices are facing a lot of skills that have to be learned simultaneously. Learning how to use an autograding system contributes to this problem. Therefore, it is highly recommended to intensively familiarise students with the autograder before starting work on complicated exercises [8].

Another common problem is that students rely on instant feedback instead of their own testing and careful thinking [6, 8]. Different approaches to handle this misbehavior have been given a try. The suggestion by Greifenstein et al. [9] and Chevalier et al. [32] to delay the feedback have been found to hold the danger of reducing motivation even though they lead to better results in the domain of computational thinking [9, 32]. Additionally, in the application of this approach is not supported by all autograding platforms. Another common approach is to introduce a penalty for each try to reduce the amount of submissions and avoid students relying on the feedback of the autograder and encourage them to test the code themselves [6, 8]. This approach has been

shown to lead to the desired results by reducing the number of submissions per Team by about 50% [6]. However, a negative side effect of the approach is that it puts severe pressure on students. Baniassad et al. [6] reported that 20% were experiencing additional stress due to the penalty, while 15% felt that the risk due to it was unnecessarily high, and 5% assumed that their overall grade was worse because they did not want to take the risk to get their grade reduced because of more uploads. Another strategy to reduce the over-reliance on the feedback of the autograding system is to make the test cases for some edge cases hidden, which means that they are executed after the exercise's deadline. Hereby, students need to properly test their code as they can only rely on the visible test cases for the general correctness of their approach [8]. However, this approach leads to stress factors similar to the submission penalty.

2.4.2 Advantages of Autograders

Despite the list of potential problems, autograding systems can be a good help in implementing computer-based scaffolding in the form of automated feedback to the students' submissions and, therefore, relieve the teacher's resources [1, 7] to enable them to focus on struggling students while the others receive their required help from the autograding system. Another strength of autograding systems is that they are fairly efficient when solving the problem that, in many cases, students do not suffer from problems solving a mistake but finding it [15].

Additionally, autograding systems collect a lot of data about the students' performances in general (like an overall percentage) and in detail when, for example, discovering that several students face the same problems, allowing the teacher to intervene even if the students did not call for help [15, 27]. Besides providing information for instant intervention, the performance data of all students at any time can be used by researchers to gain a better understanding of how students work on programming exercises and what critical points can be interesting for further research.

2.4.3 TUM's autograder ArTEMiS

For this thesis, the open source autograding platform ArTEMiS [1], which's development is managed by TUM, will be utilized. It has been chosen because of the wide range of features and freedom in designing exercises. Another criterion is the fast support of the development team in case of errors and the author's positive experience with ArTEMiS as a student.

Amongst other features, ArTEMiS supports programming, modeling, quiz, and file upload exercises. The thesis will focus on programming exercises whose workflow is described in Figure 2.2.

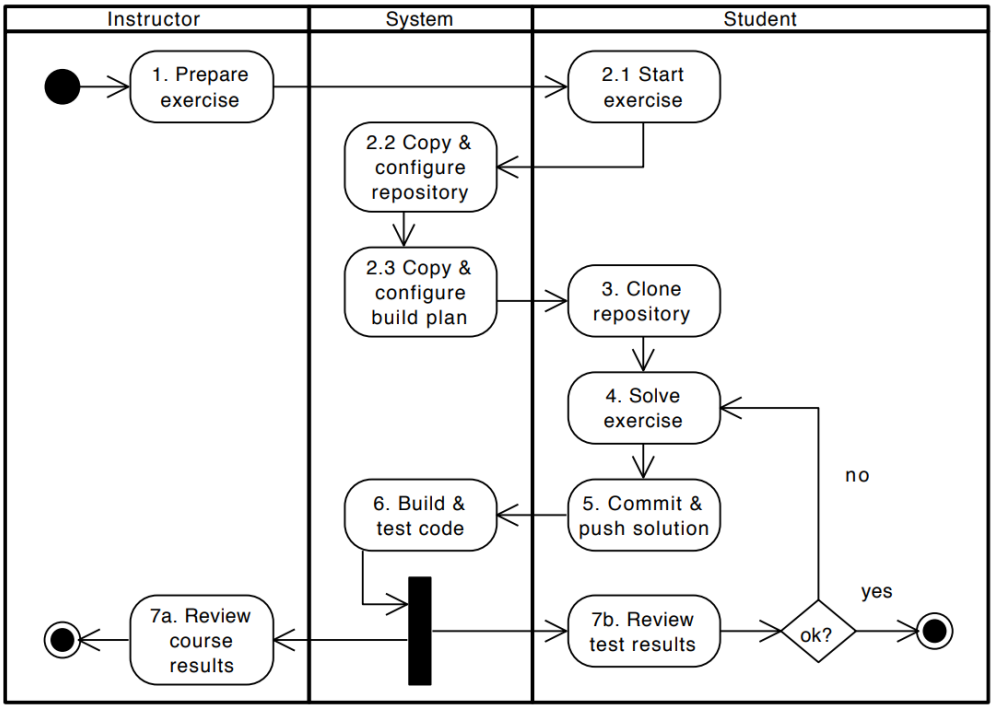
The automatic assessment is performed by unit tests utilizing the JUnit5 framework extended by the Ares test framework to provide more robust and secure testing tailored to efficiently test student submission. Therefore, instructors are free to design the test cases' fail messages to their needs and run certain test cases only if others were successful (e.g., structural tests first, behavior tests second, etc.)[33].

To provide students a better overview of what has already been completed, what still needs to be done, and which feedback belongs to which subtask, the problem statements can be designed interactively. Even parts of class diagrams can be linked with the results of test cases and are displayed in green or red according to the test results (see Figure 2.3).

Another feedback feature available is hints that appear after a certain number of times a test case fails and, therefore, work like scaffolded feedback (section 2.2.3) or delayed feedback (section 2.4.1). Unfortunately, this feature seemed buggy, so it will not be used in the exercises - even though the results of these two types of scaffolding would be interesting to assess for the purpose of the thesis.

ArTEMiS also provides a static code analysis similar to a code linter that looks for code style mistakes, reports them to the students, and (optionally) reduces their score accordingly. Although teaching a good code style to the students is desirable, keeping the cognitive load as low as possible

Figure 2.2 Process for automated assessment in ArTEMiS [1]



is essential to enable students to focus on the critical CS concepts [15, 29] - especially as they are also massively distracted by learning Java's syntax [2, 3]. Therefore, the static code analysis will also not be used in this thesis. The platform would also support using hidden test cases and a submission penalty or a limited number of submissions. These features will also not be used to keep stress factors low.

Besides the possibility of interacting with the code repositories via Git, ArTEMiS provides an online code editor that supports basic syntax highlighting and highlighting the position of compiler errors. However, the syntax is only checked when submitting a new solution, and the online editor can not execute the code. Therefore, the students will be recommended to work on their code offline and use the online editor as a fallback only [33]. By analyzing Git's commit messages, it can be determined if a submission has been uploaded from an offline IDE or the online editor.

Figure 2.3 Example of an interactive problem statement [12, 33]

Sorting with the Strategy Pattern

In this exercise, we want to implement sorting algorithms and choose them based on runtime specific variables.

Part 1: Sorting

First, we need to implement two sorting algorithms, in this case `MergeSort` and `BubbleSort`.

You have the following tasks:

1. ⊗ **Implement Bubble Sort** 0 of 1 tests passing
   Implement the method `performSort(List<Date>)` in the class `BubbleSort`. Make sure to follow the Bubble Sort algorithm exactly.

2. ⊗ **Implement Merge Sort** 0 of 1 tests passing
   Implement the method `performSort(List<Date>)` in the class `MergeSort`. Make sure to follow the Merge Sort algorithm exactly.

Part 2: Strategy Pattern

We want the application to apply different algorithms for sorting a `List` of `Date` objects. Use the strategy pattern to select the right sorting algorithm at runtime.

You have the following tasks:

1. ⊗ **SortStrategy Interface** 0 of 2 tests passing
   Create a `SortStrategy` interface and adjust the sorting algorithms so that they implement this interface.

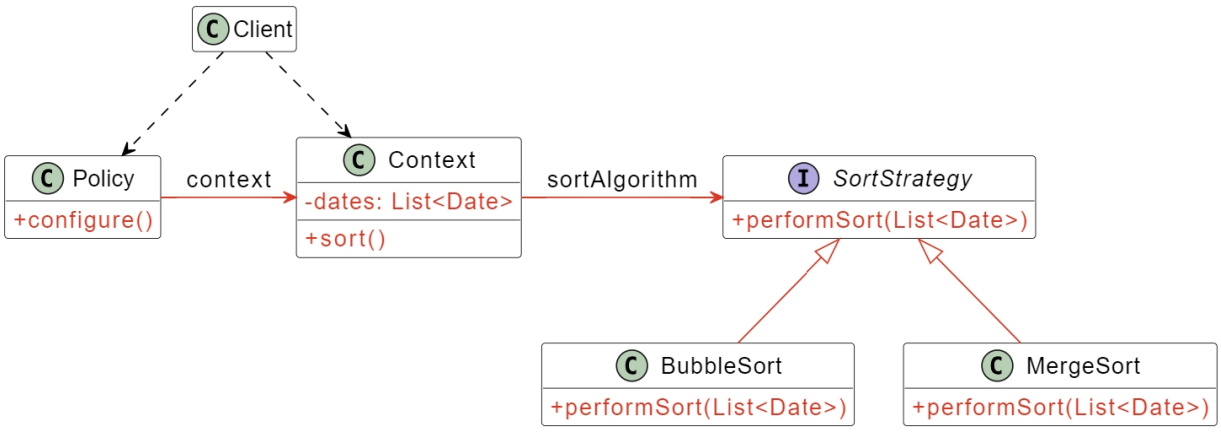
2. ⊗ **Context Class** 0 of 2 tests passing
   Create and implement a `Context` class following the below class diagram

3. ⊗ **Context Policy** 0 of 3 tests passing
   Create and implement a `Policy` class following the below class diagram with a simple configuration mechanism:

   1. ⊗ **Select MergeSort** 0 of 2 tests passing
      Select `MergeSort` when the List has more than 10 dates.

   2. ⊗ **Select BubbleSort** 0 of 2 tests passing
      Select `BubbleSort` when the List has less or equal 10 dates.

4. Complete the `Client` class which demonstrates switching between two strategies at runtime.



2.5 Measuring Success in Learning to Program

Measuring how successful a method of learning programming has always been a challenge - especially when using process data without separate written or oral exams. Including detailed code inspection, counting keystrokes over time, several visualizations of behavior, and measuring how much time was needed for a specific task, different approaches have been tried [24, 27, 34, 35, 36]. Hereby, two approaches excel:

2.5.1 Time-on-Task

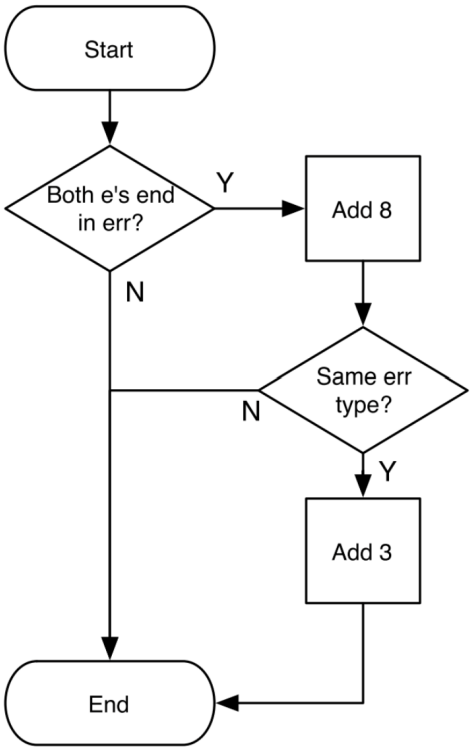
The concept of time-on-task describes the measuring of how much time has been spent working on a task. It is interesting to investigate as it has been found to moderately correlate to course results. Time-on-task is an approach that sounds fairly simple but has certain pitfalls in detail when trying to determine what counts as time that has been spent on a task and what is not. Especially when working offline or in an unrecorded environment, it is almost impossible to know what has been done between two submissions [35] and how much time has passed before the first submission of a session. One common approach is to define a time difference between two submissions, and if the actual time difference is greater, the work is split into two separate sessions [31]. Besides this, another common possibility is to try to approximate the time-on-task by other metrics. Doing this Leinonen et al. [35] found a high correlation between session time and the number of change events and suggested that this could be a good metric for approximating time-on-task. Such data can then be analyzed for change over time, which could provide insights about a student's performance [34]. Nevertheless, it is highly desirable to calculate the time-on-task with precise fine-grained data, as it was noted that there is only a moderate correlation between time-on-task calculated with fine-grained data like keystroke analysis or screen recording and time-on-task approximated from coarse-grained data like submission times [31]. This can be explained as the time between submissions could be used for tasks like looking up lecture notes, or it could be used for bad behavior like scrolling social media [35].

2.5.2 Error Quotient

The error quotient is another possibility to measure a novice's programming progress. Broken down to the core, this approach measures if consecutive submissions contain errors and if they are the same errors, which indicates that a certain problem has not been solved [34, 36].

The error quotient is usually defined on a scale ranging from 0.0 if no directly consecutive submissions both contain an error to 1.0 if all submissions ended in the same error [36]. The exact distribution of values between these limits is usually determined based on the collected data to achieve a well-differentiated output. Calculating the error quotient usually follows the pattern described in figure 2.4 where the term *Both e's* refers to two consecutive executions or submissions. The calculated value is afterward projected onto the interval $0, 1$ by dividing by the number of pairs of consecutive submissions.

Figure 2.4 Sample of an algorithm to calculate the EQ [36]



3 Research Questions

Autograding As autograding systems have become an irreplaceable part of teaching programming in higher education, much research has been carried out in this area. Unfortunately, this research is usually performed on a high level, e.g., looking at the results of a final exam compared to the scores achieved in programming exercises when varying the circumstances like introducing penalties for submission or after introducing an integrated AI tutoring system [1, 6, 8, 37]. However, to understand what's important when designing exercises and automated feedback in a K-12 education context, this aerial view of an exercise's properties is insufficient, and more precise data must be collected.

Scaffolding and Feedback Much research about providing scaffolding and giving feedback fitting a K-12 educational context has been performed, and guidelines on designing both have been developed. However, especially the feedback was either provided by an offline teacher or, if it was computer-based, the exercises were mostly simple tasks like multiple choice or fill-the-gap exercises but no highly complex programming exercises [9, 17, 18, 22, 23, 38].

Research Questions and Expected Results Therefore, the following research questions have been developed to close the gap between the research on autograding systems and feedback as a computer-based scaffold for programming exercises:

RQ1. How does the choice of the strategy of giving automated feedback - knowledge about mistakes with and without instructional characteristics on how to proceed - about a submission to a programming exercise influence the repetition of similar mistakes in subsequent exercises?

→ The findings of Greifenstein et al. [9, 10] indicate that direct instruction reduces intrinsic motivation - which would be crucial for successful learning - and that giving students partial autonomy by providing feedback without direct instruction may stimulate their thinking about solutions and, therefore, improve their cognitive learning process [9, 10, 25].

Hypothesis H1: According to these findings, it is expected that **feedback without direct instruction** leads to better learning and, therefore, **fewer repeated mistakes in future exercises**.

RQ2. How does the choice of the strategy of giving automated feedback - knowledge about mistakes with and without instructional characteristics on how to proceed - about a submission to a programming exercise influence the speed of solving mistakes in the same and subsequent exercises?

→ The results of Greifenstein et al. [10] that novices seem to need clear guidance/instruction for debugging and that direct instruction is very effective are in line with the intuition that direct instruction leads to fast problem-solving. However, according to the findings of Finn and Metcalfe [25], in future exercises, quicker solutions are produced if the provided feedback needs more cognitive processing.

Hypothesis H2.1: Giving **direct instruction** as feedback is expected to show the **quickest solution for an occurring mistake**.

Hypothesis H2.2: **In future exercises, quicker solutions** are expected if the provided feedback in previous exercises had no **no direct instruction**.

4 Research Methods

To answer the research questions, a small study was conducted among 10th-grade students of a grammar school in Munich. The students work on three programming exercises per topic. In the first two exercises, two different feedback strategies are carried out, while in the third exercise, the same feedback strategy is used for all students.

4.1 Circumstances and General Setting

4.1.1 Population and Groups

The study was conducted within the scope of the regular computer science class of three 10th-grade courses, which will later be referred to as courses a, b, or c. Due to the findings of Kim et al. [23] that computer-based scaffolding was more effective when students worked in pairs and only slightly less effective when working in triads than when working alone, students were instructed to build teams of two to three people to work on the upcoming exercises. These teams were then divided into two groups (A and B), the results of which will be compared later. Table 4.1 provides a detailed overview of how the courses were divided into groups. Each course has two lessons of 45 minutes per week, split by a 20-minute break. Additionally, the students have access to all required tools from home and are encouraged to work on exercises from home if they notice that they need more time for the exercises.

Table 4.1 Distribution of the courses to the groups

Course	Students		Teams	
	A	B	A	B
a	24	0	8	0
b	0	20	8	0
c	12	10	4	4
Total	36	30	12	12

4.1.2 Curriculum and Prerequisites

In 9th grade, the students learned basic Java programming concepts and object-oriented modeling. Therefore, they are already familiar with classes, methods, attributes, and constructors, as well as return values, parameters, local variables, primitive data types with algebraic operations, conditions with if-else-statements, and for- and while-loops. Before continuing with the 10th-grade curriculum, this prior knowledge was repeated for about five weeks to ensure the presence of the basic knowledge.

The new topics in 10th grade are inheritance, arrays, and classes as datatypes resp. object references (later referred to as reference attributes).

4.1.3 Tools and Technology

IDE BlueJ [39] has been chosen as IDE because most students were already familiar with it. Besides this, BlueJ is the only possibility to use Git with only minor restrictions by the school's firewall. As a fallback in case of problems with the interaction of IDE and autograder, an online code editor integrated into ArTEMiS is available; even so, it does not allow running code online.

Autograder Due to the reasons outlined in section 2.4.3, TUM's autograder ArTEMiS will be used for deploying feedback. The students in courses a and b were already familiar with the platform from 9th grade while working with it was new for students in course c. To reduce the negative effects of students not being familiar with autograders while learning new matters [8], the repetition of the previous year's knowledge was combined with precisely teaching how to use the autograding system, and all repetition exercises were deployed using ArTEMiS .

4.2 Lesson Structure and Feedback Strategies

Figure 4.1 Lesson structure

Application in Jump'n'Run Project	Context, individual Features, Design, etc.			no Feedback
	Mandatory Features (Keyboard Control, inherited Entity-Types, etc.) that require the skills obtained in the practice exercises			uniform Feedback
Practice	Exercise 06c (Wk. 3-5)	Exercise 07c (Wk. 5-7)	Exercise 08c (Wk. 6-8)	uniform Feedback
	Exercise 06b (Wk. 1-3)	Exercise 07b (Wk. 4-6)	Exercise 08b (Wk. 6-8)	different Feedback for Groups A and B
	Exercise 06a (Wk. 1-3)	Exercise 07a (Wk. 4-6)	Exercise 08a (Wk. 5-7)	
Knowledge (free choice)	20-25 Minute Lecture+Tutoring by the Teacher			offered 3 Times/Topic
	(no Videos available)	Videos	Videos	always available
Topic:	**Reference Attributes**	**Inheritance**	**Felder/Arrays**	

To minimize problems caused by students being unable to work at their own pace, like the danger of skipping exercises [3], a flexible lesson structure has been developed. The structure consists of three parts that build up: *Knowledge* → *Practice* → *Application*.

1) Knowledge In the first step, students have the free choice if they want to gain theoretical knowledge from hand-picked videos or a teacher-led lecture-tutoring session. They can choose one or both once or multiple times. A lecture on each topic is offered every week, where at least one exercise on the topic is available.

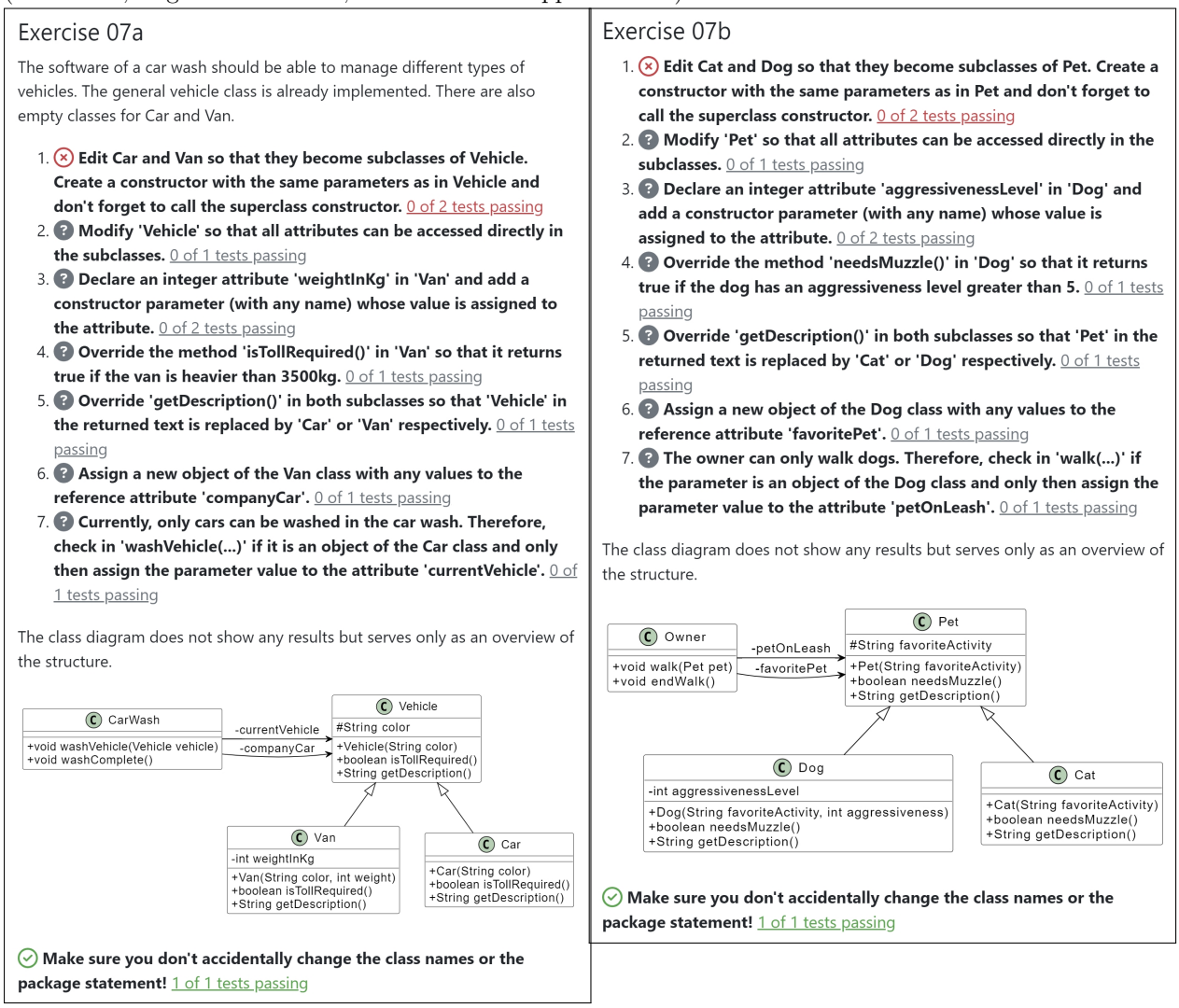
2) Practice Exercises After or parallel to the transfer of knowledge, students work on practice exercises. For each topic (reference attributes, inheritance, and arrays), three exercises with similar structures are available. A typical sample of two such exercises is provided in figure 4.2. The similar structure in different contexts is expected to improve the effect of practice [2] and has the potential to make similar mistakes in consecutive exercises, which can, therefore, be better compared.

As code skeleton exercises ([27], section 2.3.1) are very suitable for use with unit tests and, therefore, also with autograders and are also applicable for all programming-related computer science concepts, all exercise follow this design approach.

Each exercise is available for three weeks with a slight overlap (see Figure 4.1 for details). Each exercise's time frame was designed to be solvable in about 30-45 minutes of concentrated work by an average student.

3) Software Project After and parallel to the practice exercises, the learned concepts shall be applied in a jump'n'run software project. The project's baseline consists of some mandatory features like keyboard controls and inheritance of different entity types to ensure that the newly learned concepts are used in the project. However, the game's context, individual features, design, etc., are up to the students. Besides the motivational aspects and the application of skills learned in

Figure 4.2 Interactive exercise statements of similar structured exercises 07a and 07b in ArTEMiS (translated, original in German; all exercises in Appendix A.2)



closed, isolated practice exercises, another intention of the software project is to reduce the effect of students relying on the autograder and not test code themselves. However, they still do not lose motivation or get overly stressed due to a penalty for every new submission [6], by delayed feedback [32] or hidden test cases [8].

Automated Feedback Students receive automated feedback on all practice exercises and the project's mandatory features. After a student's submission has been uploaded, the feedback is displayed on the ArTEMiS page in the internet browser. The feedback for all incorrect parts of the exercise is usually displayed at once. However, feedback for subtasks whose preconditions are not fulfilled - e.g., implementing a method signature is a precondition for implementing the method behavior - is hidden from the students to reduce their cognitive load. The feedback items are also not collected for analysis.

In each topic's exercises and b, the two student groups, A and B, receive feedback based on different strategies. In contrast, in exercise c and the project exercises, feedback is deployed based on the strategy used for group A. The Feedback strategies have been developed based on the findings of section 2.2. Both strategies have the intention of reducing the degrees of freedom and some maintenance of direction by providing mainly conceptional scaffolding and some metacognitive support, as these have been found to be the most effective in group settings [23].

Feedback Strategy A implements these intentions by providing information about the student's performance (=feedback). More specifically, the elaborated feedback type *knowledge about mistakes* is applied using *location hints* and *data hints*.

Feedback Strategy B is based on strategy A but extends it with the instructional characteristics of *Knowledge about how to proceed*.

Feedback Sample The following sample illustrates a typical case for both feedback strategies in an exercise where a part of the task is to complete a method that returns the last element of a given array (without error handling):

Strategy A	**Strategy B**
The last element of the Array is not returned correctly.	The last element of the Array is not returned correctly. The last element of an array named *values* can be returned like this: *return values[values.length-1];*

4.3 Data

4.3.1 Collected data

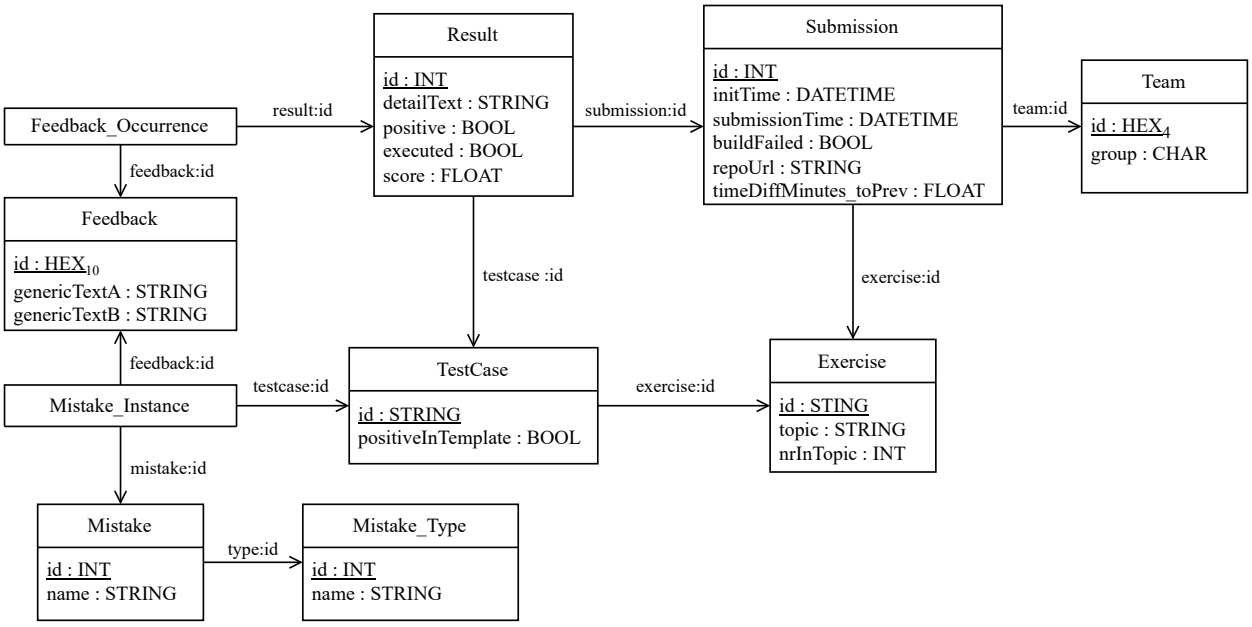
Figure 4.3 provides an overview of the structure of the collected data. In this, every exercise is assigned to one of the three main topics and a number in that topic ($a \rightarrow 1, ...$). The project exercise is listed as exercise *4* in the topic *all*. Each exercise has multiple test cases that are equivalent to the JUnit test cases executed on the ArTEMiS server after every submission. For each test case, the information if it was already passing in the template (e.g., to check if the class name from the template is correct) is stored.

Each submission to an exercise can be assigned to a student team with a pseudonymized ID and feedback group. For each submission, the time when a team has started working on the exercise, the time of the submission itself, if the compilation (=build) has been successful, and a git repository URL is available. Additionally, the time difference to the previous submission or - if no such submission exists - to the initiation time has been calculated. Each submission consists of multiple test case results. For each result, the information if the corresponding test case has been executed (which would not be the case if preconditions of the test case are not fulfilled), if it passed successfully and the achieved score is stored. Most importantly, the detailed assertion message is available if a test case fails. The detailed text contains one or more feedback IDs unique for any possible assertion message in a test case. This is necessary as a test case can fail for several reasons, e.g., the check if an array contains the correct values can fail due to a wrong length or value. If a certain feedback item (identified by its feedback ID) occurred in a test result, it is called a *feedback occurrence*. The combination of a feedback ID and a test case is a distinct mistake instance of a distinct mistake. Each mistake belongs to a certain mistake type. To illustrate this division, the following example can be helpful:

The *mistake* 'wrong return type in a method signature' can occur in multiple test cases (e.g., one for each method in the exercise). However, a wrong return type of a specific method is an *instance of this mistake*. All mistakes due to wrongly implementing elements described in a class diagram - like wrong access modifiers, wrong data types of attributes, etc. - form the *mistake type* 'element from class diagram wrong'.

Unfortunately, no structured questioning of the students about their experience and solutions and determining their understanding of concepts was possible. It would have been interesting to gain a better understanding of how good their conceptual knowledge is compared to the autograder results[29, 30]. Therefore, the only data available besides the one described, provided by ArTEMiS , to answer the research questions is the author's unstructured observation from the classroom.

Figure 4.3 Structure of the collected data



Quantity of Collected Data In total, *24 teams* of *66 students* worked on *10 exercises*, *submitted 3.853 times*, resulting in *93.896 test case results* of *190 test cases*, which, combined with *215 feedback items* (that *occurred 20.706* times in test results), form *474 mistake instances* that were assigned to *74 different mistakes* that are categorized into *13 mistake types* of which one collected *8 irrelevant mistake types* like feedback wrappers that contain another feedback item or notifications about missing requirements. A detailed list of mistakes and mistake types can be found in Appendix A.3

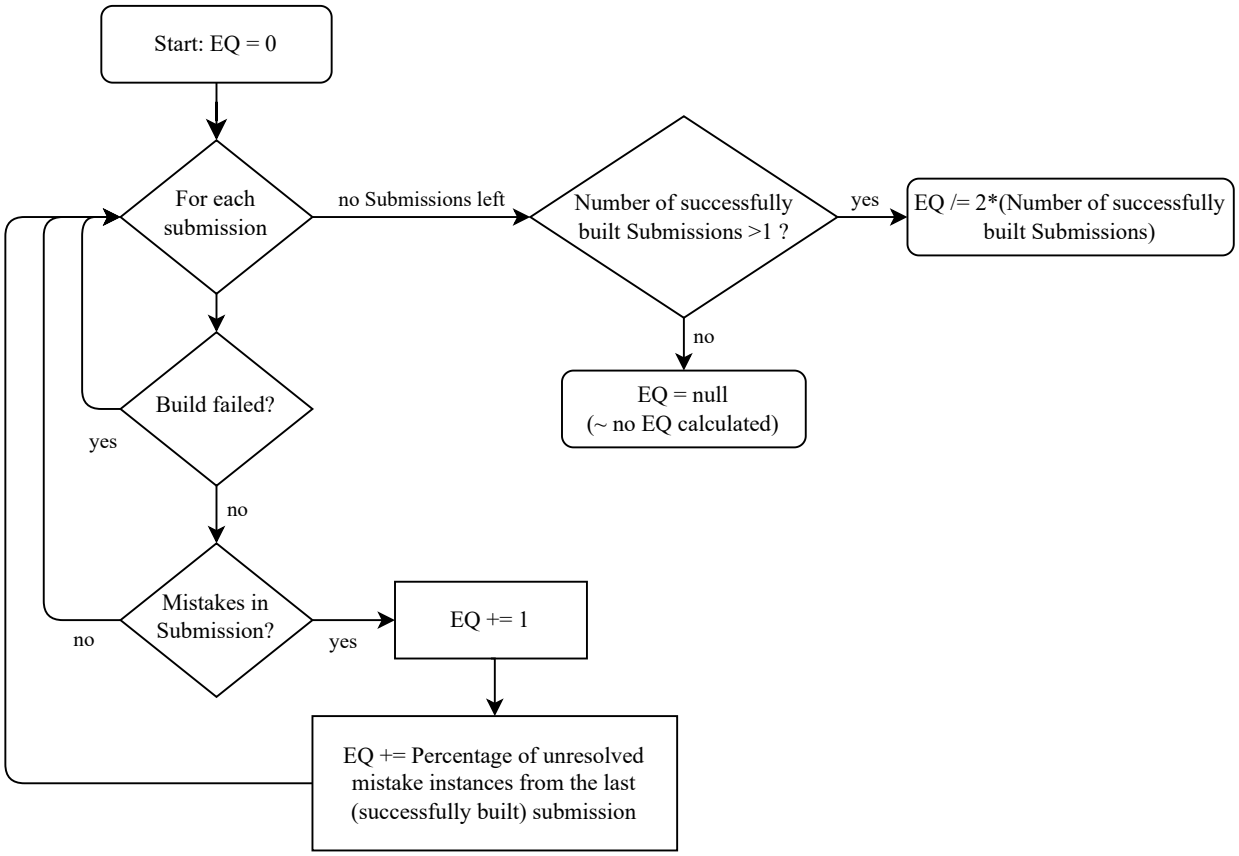
4.3.2 Calculation of Progress Measures

Based on the prototypical algorithm for calculating the error quotient described in section 2.5.2 [36] two versions of error quotients have been calculated. While the described approach considered different types of mistakes but only one mistake at once, it had to be modified to fit the requirements of being able to depict the possibility of making multiple mistakes at once.

Generic Measure: Error Quotient The EQ maps the overall progress of a team in one exercise. As the autograder can only provide elaborate feedback if a submission was successfully built, the EQ considers only successful builds to represent success after elaborate feedback has been received about a mistake. This assumption was verified by comparing the U-test results of EQs calculated with and without penalty for failing builds. If a build fails, the mistakes from the previous submission are forwarded to the next comparison. Similar to the process described by Jadud [36] after some tinkering, the weighting between making any mistakes and repeating mistakes has been fixated. For making any mistake, 1 is added to the EQ, and for not solving a mistake instance, the percentage of repeated mistake instances is added. This means for every pair of consecutive submissions, the maximum value that can be added to the EQ is 2. Therefore, the last step is to project the value to the interval $0,1$ by dividing 2 times the number of successfully built submissions. If one or fewer submissions can be successfully built, the dataset will be excluded from the process. The calculation process is additionally illustrated in the flowchart in figure 4.4.

Time on Task The first approach to answer RQ2 was to calculate a value similar to the time on task described by Leinonen et al. [31]. Unfortunately, the splitting of sessions did not lead to

Figure 4.4 Flowchart of EQ calculation for one Exercise of one Team



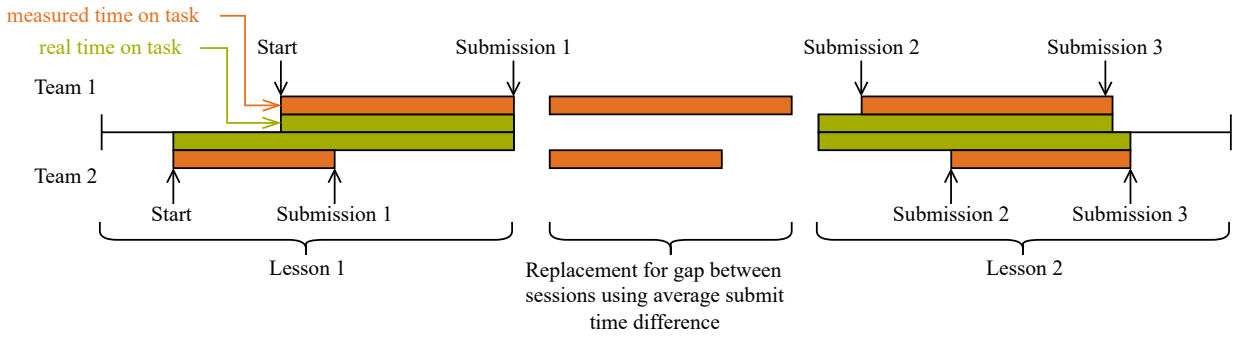
valid results. By checking random samples, it was quickly discovered that it was very common for students to submit in the middle of a lesson but not at the end and continue in the next lesson. Using the approach of leaving out the time exceeding a certain threshold between two sessions/lessons, therefore, did not seem sensible. Replacing this gap with the average time between two submissions did also not solve the problem as it could lead to a significant distortion of the results as described in figure 4.5. The exemplary case in the figure shows two different behaviors of students where one team submits at the end of a lesson and continues in the next lesson, while the other team submits in the middle of the lesson but still works till the end of it and submits the work from the second half in the next lesson. From looking at the data, both behaviors were present, and therefore, calculating the time on task or a time-based speed measure for solving problems this way can not be done with sufficient precision.

Measure per Mistake Type Because no way of calculating an expressive error quotient per mistake could be found in literature or brought up in another way, the number of mistake instances occurring for one mistake type and the number of tries to solve those instances will be compared.

A mistake instance can only occur once per exercise per team. If it occurs again after not being present for at least one submission, it is assumed that it has been overshadowed by another mistake, e.g., missing preconditions for solving the subtask of that mistake. The two occurrences are, therefore, combined into one occurrence.

The number of tries to solve a mistake is also calculated per team per exercise. The perfect case for trying to solve a mistake instance has - similar to the error quotient - the value zero. It describes the case where the mistake instance occurs only once and is instantly solved in the next submission. If a mistake instance did not occur in an exercise, no value is calculated. In any other case, the submissions containing the mistaken instances except the first one are counted and taken as the

Figure 4.5 Exemplary case that would lead to significantly distorted time on task values



value for tries to solve. If two occurrences of a mistake instance are combined into one occurrence, the same algorithm is applied. The first submission of the second occurrence (before combining) is then rated as one try for solving the mistake instance.

From the number of occurrences and tries to solve an occurrence of a mistake instance, the following change rates (later referred to as *gradient*) have been calculated per team, exercise, and mistake type: $a \to b; a \to c; b \to c; Avg\{ab\} \to c$. Each gradient shows the percentage of the change from first to second value, having zero as *no change*, -1 as *none left*, and 1 as *value doubled*. The generic gradient function from the value of exercise x to y is calculated as

$$grad\{x \to y\} \mapsto \frac{y-x}{x}$$

The gradient combines the advantages of the approach of the error quotient that the progress of a single team can be examined with the advantages of the observation of the number of occurrences and tries to solve an occurrence where detailed insights into the differences between certain mistake types can be achieved.

4.3.3 Modification of faulty data

Double Submissions is a phenomenon where students submit their solutions twice without changing anything. This can be caused by a buggy result page that does not automatically refresh after the first submission, which might cause the students to think that their submission did not reach the server. Double submission is assumed if the time difference between two consecutive submissions of one team is *smaller than ten seconds* and the test results are identical (this includes both builds failing). In both cases, the first submission is discarded, and its time difference from the previous submission is added to the respective difference of the second submission.

No test result generated Due to a bug in ArTEMiS , no result was generated on very few submissions. If that was the case, the students also received no results and had to submit again. In that case, the time difference between the previous submission and the submission without a result was added to the time difference of the next submission, and the faulty submission was discarded.

Identification invalid data To answer the research questions, the described indicators of groups A and B were compared using hypothesis tests for independent samples. As the research questions focus on comparing the feedback strategies A and B, which are varied in exercises a and b, datasets of students who did not receive feedback about a mistake type in exercises a *and* b were not considered in the comparison of mistake specific indicators occurrence count and tries per occurrence to reduce falsification by students who did never receive varied feedback about a certain type of mistake. No such filter could be applied to the error quotient. In comparisons focused on exercises

of only one topic, the datasets were filtered by a mistake type not occurring in the exercises a and b of that topic. In comparisons that consider exercises of all topics, datasets were filtered if a mistake type was found in none of all a and b exercises.

One dataset refers to all values (occurrences, tries per occurrence) of one team for one mistake type in one of three topics or one mistake type in all exercises. Table 4.2 gives an overview of how many datasets were valid (at least one occurrence of the mistake in exercises a or b). The low proportion of valid datasets is most likely since not all types of mistakes can be made in all exercises as they were designed to focus on specific concepts, e.g., no array aspects were present in the first two topics while almost no aspects of inheritance and polymorphism were used in array exercises. Additionally, some mistake types, like unintended template modifications, would only occur in edge cases. In all following examinations of mistake occurrences and tries to solve those occurrences, only valid datasets were used.

Table 4.2 Proportion of valid datasets for number of occurrences and tries per occurrence

Topic	All		Valid	
	A	B	A	B
Reference Attributes	144	144	28	38
Inheritance	144	144	33	42
Arrays	144	144	15	21
All Topics	144	144	60	77

4.3.4 Statistical Methods

Selection of hypothesis tests To choose the correct test procedure, the distribution of the variables had to be determined using the Shapiro-Wilk test. The Shapiro-Wilk test was chosen over the Kolmogorov-Smirnov test as it has the higher statistical power - especially with small samples [40, 41].

Due to the resulting information about the probability distribution, the Mann-Whitney-U-Test for independent samples was chosen for all tests to determine if and what kind of a significant difference between groups A and B exists. The U-test was used for all comparisons to make results more comparable than if some comparisons would have been made with U-tests and some with T-tests. Additionally, the effect strength, namely the Pearson correlation coefficient r, was determined for a more detailed understanding of the strength of the difference between the groups. Usually the r-value is calculated from the U-test results as $r = |\frac{Z}{\sqrt{N}}|$ [40, 41, 42, 43]. To provide a better overview of what feedback strategy leads to bigger or smaller values, the r-value is enriched with a positive sign if the mean rank of group B is bigger and a negative one if the mean rank of group B is smaller than the one of group A. Mathematically expressed:

$$r > 0 \Leftrightarrow M_{Rank}B > M_{Rank}A \quad ; \quad r < 0 \Leftrightarrow M_{Rank}B < M_{Rank}A$$

To gain a differentiated view of the answers to the research questions, the calculated variables have been examined using the following partitioning:

- All exercises at once.
- Partitioned by topic of the exercise (reference attributes, inheritance, arrays, project).
- Partitioned by the number in the topic (a-exercises, b-exercises, c-exercises, project).
- Partitioned by type of feedback (different vs. uniform for groups A and B)
- All exercises on their own.

For answering RQ1 the number of occurrences of a mistake instance is the focus of the examination while for RQ2 the number of tries to solve a mistake instance is focussed.

5 Results

5.1 Error Quotient: Distribution Test, U-Test and Boxplot

5.1.1 Error Quotient by Exercise Topic

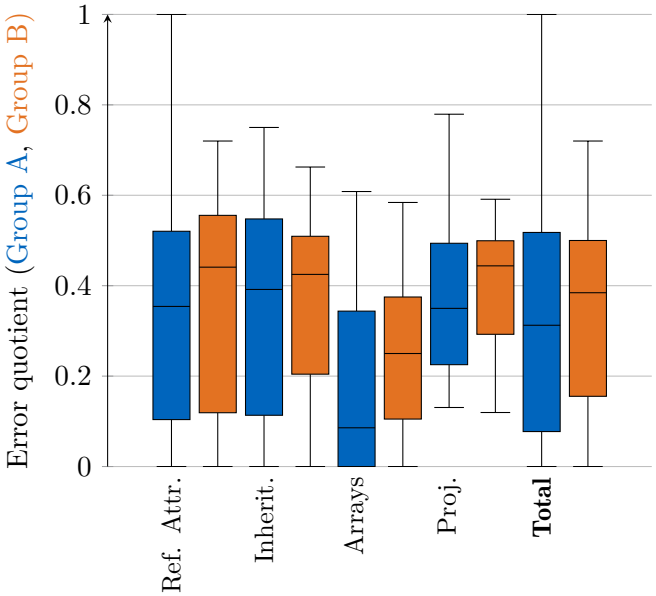
The results for normal distribution in table A.2 show that only the project exercise is normally distributed for both groups A and B and can be compared with a t-test. For all other tests, a Mann-Whitney U-test must be used. To ensure consistency of results, all values are tested with a U-test.

Table 5.1 Mann-Whitney U-Test Results of EQ of Group A against B partitioned by exercise topic

	Ref. attributes	Inheritance	Arrays	Project
N	61	60	54	23
$M_{Rank}A$	29,78	30,76	23,80	11,55
$M_{Rank}B$	32,11	30,29	30,24	12,42
$p_{2tailed}$	0,612	0,920	0,136	0,786
r	0,066	-0,013	0,204	0,064

$r > 0 \Leftrightarrow M_{Rank}B > M_{Rank}A$; $r < 0 \Leftrightarrow M_{Rank}B < M_{Rank}A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

Figure 5.1 Boxplot: EQ comparing groups A and B, partitioned by exercise topic



5.1.2 Error Quotient by Number in Topic

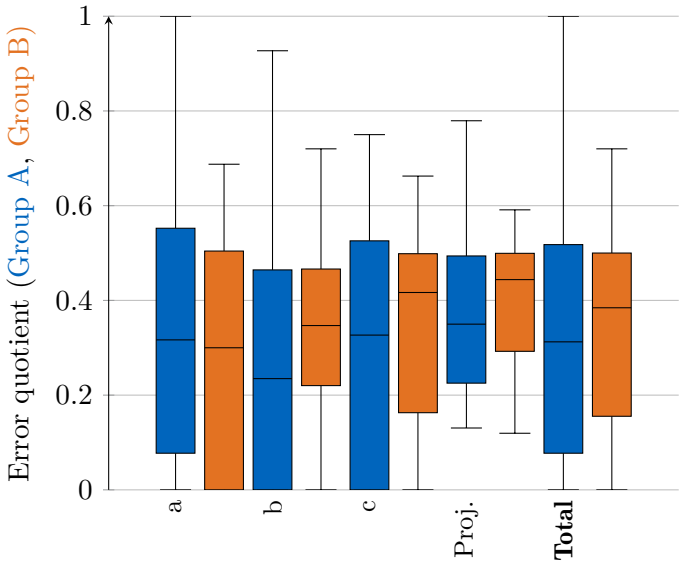
The results for normal distribution in table A.3 show that only the project exercise is normally distributed for both groups A and B and can be compared with a t-test. For all other tests, a U-test must be used. To ensure consistency of results, all values are tested with a U-test.

Table 5.2 Mann-Whitney U-Test Results of EQ of Group A against B partitioned by number in topic

	a	b	c	Project
N	59	60	56	23
$M_{Rank}A$	32,06	27,79	26,83	11,55
$M_{Rank}B$	28,27	33,03	29,67	12,42
$p_{2tailed}$	0,400	0,247	0,529	0,786
r	-0,111	0,151	0,086	0,064

$r > 0 \Leftrightarrow M_{Rank}B > M_{Rank}A$; $r < 0 \Leftrightarrow M_{Rank}B < M_{Rank}A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

Figure 5.2 Boxplot: EQ comparing groups A and B, partitioned by number of exercise in topic



5.1.3 Error Quotient by Exercise

The results for normal distribution in table A.4 show that only the exercises 06a, 06b, 07b, 08c, and the project are normally distributed for both groups A and B and can be compared with a t-test - for all others a U-test must be used. To ensure comparability, all values are tested with a U-test.

Table 5.3 Mann-Whitney U-Test Results of EQ of Group A against B partitioned by exercise

	06a	06b	06c	07a	07b	07c	08a	08b	08c	Proj.
N	21	19	21	21	20	19	17	21	16	23
$M_{Rank}A$	11,80	9,44	9,90	11,55	9,75	11,00	9,29	10,00	5,25	11,55
$M_{Rank}B$	10,27	10,50	12,00	10,50	11,25	9,42	8,80	11,91	10,45	12,42
$p_{2tailed}$	0,590	0,719	0,467	0,715	0,589	0,576	0,885	0,499	0,033	0,786
r	-0,124	0,094	0,169	-0,085	0,128	-0,136	-0,048	0,154	0,533	0,064

$r > 0 \Leftrightarrow M_{Rank}B > M_{Rank}A$; $r < 0 \Leftrightarrow M_{Rank}B < M_{Rank}A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

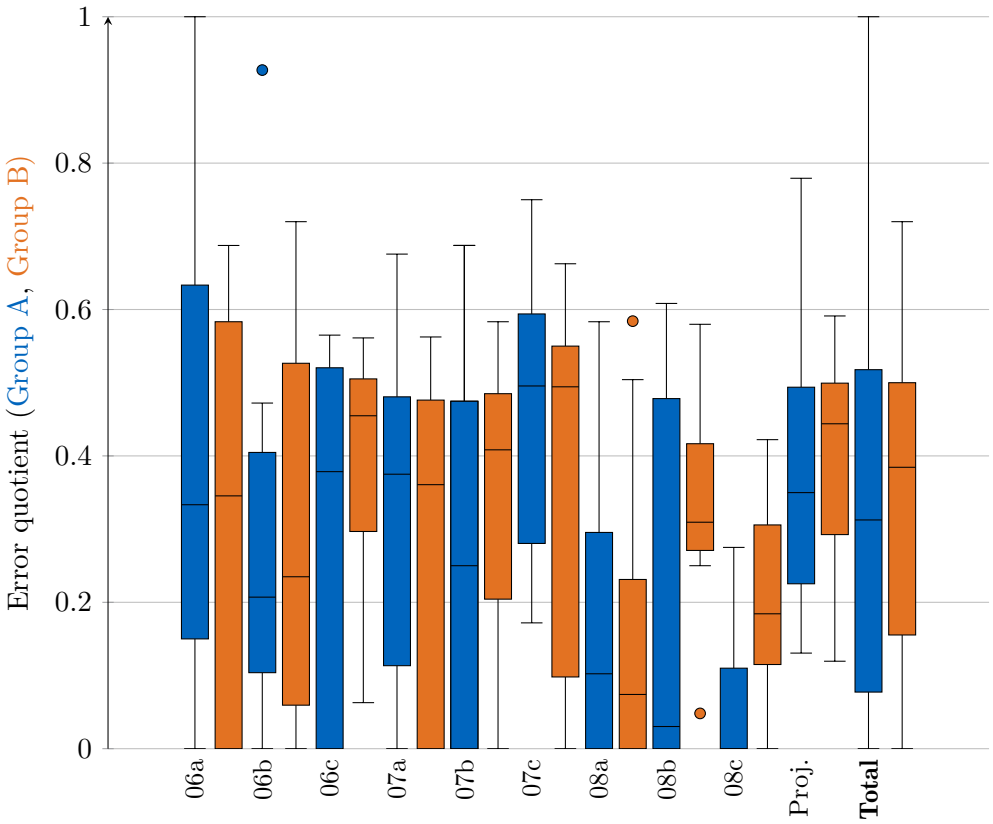


Figure 5.3 Boxplot: EQ comparing groups A and B, partitioned by exercises

5.1.4 Error Quotient by Feedback Type

The results of the test for normal distribution in table A.5 show that only one EQ value is normally distributed. Therefore, all values must be compared using U-tests.

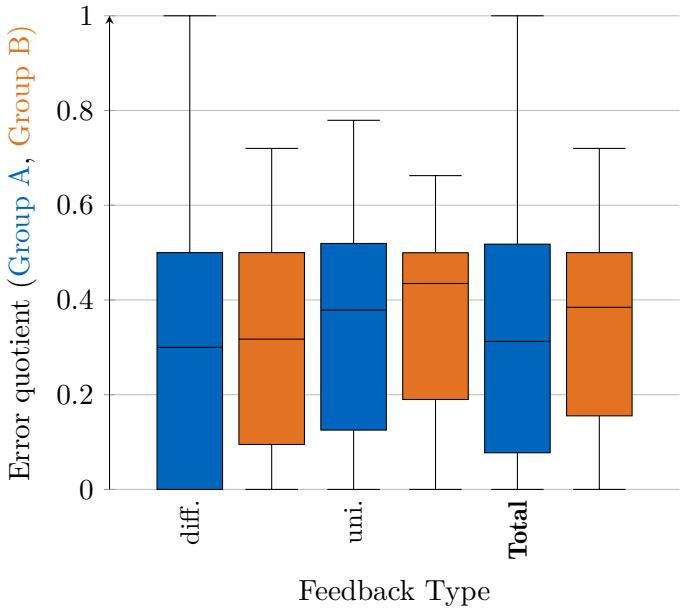


Figure 5.4 Boxplot: EQ comparing groups A and B, partitioned by feedback type

Table 5.4 Mann-Whitney U-Test Results of EQ of Group A against B partitioned by feedback type

	different	uniform
N	119	79
$M_{Rank}A$	59,13	38,32
$M_{Rank}B$	60,77	41,27
$p_{2tailed}$	0,797	0,578
r	0,024	0,064

$r > 0 \Leftrightarrow M_{Rank}B > M_{Rank}A$; $r < 0 \Leftrightarrow M_{Rank}B < M_{Rank}A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

5.1.5 Error Quotient by Feedback Type and Exercise Topic

The results for normal distribution in table A.6 show that only the array exercises with uniform feedback and the project exercise with uniform feedback are normally distributed for both groups A and B and can be compared with a t-test. For all other tests, a U-test must be used. To ensure consistency of results, all values are tested with a U-test.

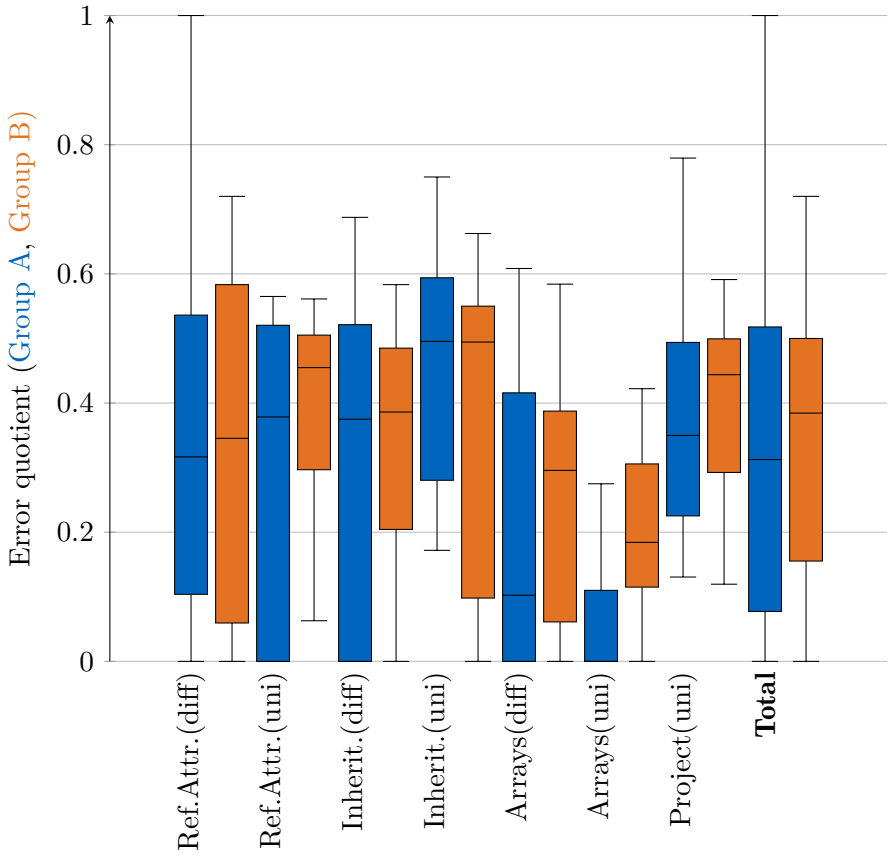


Figure 5.5 Boxplot: EQ comparing groups A and B, partitioned by exercise topic and feedback type

Table 5.5 Mann-Whitney U-Test Results of EQ of Group A against B partitioned by feedback type and topic

	Ref. Attributes		Inheritance		Arrays		Project	
	diff.	uni.	diff.	uni.	diff.	uni.	diff.	uni.
N	40	21	41	19	38	16		23
$M_{Rank}A$	20,89	9,90	20,63	11,00	18,09	5,25		11,55
$M_{Rank}B$	20,14	12,00	21,36	9,42	20,64	10,45		12,42
$p_{2tailed}$	0,845	0,467	0,851	0,576	0,487	0,033		0,786
r	-0,032	0,169	0,031	-0,136	0,115	0,533		0,064

$r > 0 \Leftrightarrow M_{Rank}B > M_{Rank}A$; $r < 0 \Leftrightarrow M_{Rank}B < M_{Rank}A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

5.2 Mistake Instances: Occurrences and Tries to Solve

5.2.1 Mistake Instances by Exercise Topic

Table 5.6 Mann-Whitney U-Test Results Comparing Group A against B partitioned by Topic of Exercise

	Number of Mistake Instances				Tries to Solve a Mistake Instance			
	a	b	Avg(ab)	c	a	b	Avg(ab)	c
All Exercises								
N	137	137	137	137	102	94	137	83
$M_{Rank}A$	67,43	65,58	66,33	65,80	49,63	42,32	63,86	40,33
$M_{Rank}B$	70,22	71,66	71,08	71,49	53,04	50,86	73,01	43,10
$p_{2tailed}$	0,676	0,360	0,479	0,389	0,559	0,131	0,176	0,610
r	0,036	0,079	0,061	0,074	0,058	0,156	0,116	0,057
Exercises on the Topic Reference Attributes								
N	66	66	66	66	32	46	66	39
$M_{Rank}A$	30,98	35,43	33,41	32,98	18,77	15,70	28,05	15,31
$M_{Rank}B$	35,36	32,08	33,57	33,88	14,95	29,50	37,51	23,26
$p_{2tailed}$	0,323	0,475	0,976	0,848	0,248	<0,001	0,042	0,031
r	0,122	-0,089	0,004	0,024	-0,208	0,518	0,250	0,344
Exercises on the Topic Inheritance								
N	75	75	75	75	67	54	75	43
$M_{Rank}A$	37,92	32,79	34,12	35,23	31,93	26,18	34,08	23,44
$M_{Rank}B$	38,06	42,10	41,05	40,18	35,68	28,21	41,08	21,15
$p_{2tailed}$	0,981	0,053	0,159	0,311	0,434	0,647	0,166	0,569
r	0,003	0,225	0,164	0,119	0,097	0,063	0,161	-0,089
Exercises on the Topic Arrays								
N	36	36	36	36	19	19	36	8
$M_{Rank}A$	20,17	16,73	19,17	18,13	8,10	12,75	18,73	2,33
$M_{Rank}B$	17,31	19,76	18,02	18,76	12,11	8,73	18,33	5,80
$p_{2tailed}$	0,398	0,361	0,758	0,885	0,096	0,121	0,922	0,071
r	-0,144	0,153	-0,060	0,041	0,385	-0,372	-0,021	0,689

$r > 0 \Leftrightarrow M_{Rank}B > M_{Rank}A$; $r < 0 \Leftrightarrow M_{Rank}B < M_{Rank}A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

5.2.2 Mistake Instances by Mistake Type

Table 5.7 Mann-Whitney U-Test: Group A against B partitioned by Mistake Type and Exercise (1)

	Occurrences of Mistake Instances					Tries to Solve a Mistake Instance				
	a	b	Avg(ab)	c	Proj.	a	b	Avg(ab)	c	Proj.
Mistakes when Creating Arrays										
N	6	6	6	6	6	5	2	2	6	6
$M_{Rank}A$	4,75	4,00	5,50	4,00	3,50	4,00	2,00	5,00	1,00	4,25
$M_{Rank}B$	2,88	3,25	2,50	3,25	3,50	2,33	1,00	2,75	2,00	3,13
$p_{2tailed}$	0,533	1,000	0,067	1,000	1,000	0,400	1,000	1,000	0,600	0,400
r	-0,559	-0,228	-0,894	-0,228	0,000	-0,596	-0,707	-0,707	0,288	-0,645
Mistakes relating to Array Indices										
N	13	13	13	13	13	7	7	13		
$M_{Rank}A$	7,60	5,30	6,10	7,00	5,50	2,88	4,50	5,90		
$M_{Rank}B$	6,63	8,06	7,56	7,00	7,94	5,50	3,92	7,69		
$p_{2tailed}$	0,747	0,224	0,584	1,000	0,231	0,114	1,000	0,434		
r	-0,129	0,366	0,195	0,000	0,413	0,630	-0,099	0,236		
Mistakes relating to Iteration of Arrays										
N	8	8	8	8	8	2	6	6	7	8
$M_{Rank}A$	4,50	4,50	4,50	3,88	4,25	1,00	4,17	4,75	2,00	3,00
$M_{Rank}B$	4,50	4,50	4,50	5,13	4,75	2,00	2,83	4,25	4,25	4,75
$p_{2tailed}$	1,000	1,000	1,000	0,571	1,000	1,000	0,700	0,267	0,400	1,000
r	0,000	0,000	0,000	0,265	0,117	0,707	-0,385	-0,567	0,401	0,112
Mistakes during Modification of Arrays										
N	3	3	3	3	3	3		3		
$M_{Rank}A$	2,00	2,00	2,00	2,00	1,00	1,50		0,00		
$M_{Rank}B$	2,00	2,00	2,00	2,00	2,50	2,25		1,50		
$p_{2tailed}$	1,000	1,000	1,000	1,000	0,333	1,000		1,000		
r	0,000	0,000	0,000	0,000	0,816	0,408		0,408		
Mistakes relating to Attribute values, Getters, and Setters										
N	13	13	13	13	13	13		13	8	
$M_{Rank}A$	5,60	7,00	5,60	6,60	5,50	7,10		7,10	3,67	
$M_{Rank}B$	7,88	7,00	7,88	7,25	7,94	6,94		6,94	5,00	
$p_{2tailed}$	0,293	1,000	0,293	0,883	0,231	0,965		0,965	0,536	
r	0,336	0,000	0,336	0,088	0,413	-0,022		-0,022	0,267	

Table 5.8 Mann-Whitney U-Test: Group A against B partitioned by Mistake Type and Exercise (2)

	Occurrences of Mistake Instances					Tries to Solve a Mistake Instance				
	a	b	Avg(ab)	c	Proj.	a	b	Avg(ab)	c	Proj.
Element from Class Diagram missing										
N	18	18	18	18	18	13	18	15	18	18
$M_{Rank}A$	8,00	9,56	8,67	9,22	8,50	6,17	8,17	7,39	9,42	8,22
$M_{Rank}B$	11,00	9,44	10,33	9,78	10,50	7,71	10,83	11,61	7,06	10,78
$p_{2tailed}$	0,297	1,000	0,541	0,832	0,421	0,571	0,299	0,340	0,328	0,097
r	0,313	-0,011	0,158	0,054	0,206	0,216	0,255	0,261	-0,239	0,399
Element from Class Diagram implemented wrongly										
N	18	18	18	18	18	18	18	9	18	18
$M_{Rank}A$	9,94	7,00	8,06	10,44	8,61	9,94	6,33	10,22	5,10	10,22
$M_{Rank}B$	9,06	12,00	10,94	8,56	10,39	9,06	5,88	8,78	4,88	8,78
$p_{2tailed}$	0,779	0,043	0,256	0,499	0,493	0,748	0,873	0,968	0,605	0,582
r	-0,088	0,495	0,284	-0,192	0,170	-0,084	-0,056	-0,041	-0,135	-0,138
Mistake in or relating to Constructor										
N	14	14	14	14	14		13	14	7	
$M_{Rank}A$	5,50	7,00	5,50	5,40	6,50		4,10	4,80	2,50	
$M_{Rank}B$	8,61	7,78	8,61	8,67	8,06		8,81	9,00	4,25	
$p_{2tailed}$	0,221	0,835	0,198	0,266	0,505		0,034	0,080	0,857	
r	0,454	0,124	0,449	0,431	0,293		0,594	0,490	0,316	
Mistake relating to Reference Attributes										
N	22	22	22	22	22	22	19	19	19	22
$M_{Rank}A$	8,25	10,35	9,55	8,60	9,90	10,80	5,94	8,95	8,50	10,44
$M_{Rank}B$	14,21	12,46	13,13	13,92	12,83	12,08	13,65	13,63	11,09	11,42
$p_{2tailed}$	0,027	0,439	0,210	0,052	0,271	0,662	0,002	0,340	0,741	0,096
r	0,474	0,168	0,276	0,419	0,243	0,099	0,685	0,227	0,082	0,359
Template modified unintendedly										
N	7	7	7	7	7		4	4	7	
$M_{Rank}A$	5,13	3,25	4,75	3,75	4,00		2,50	4,75	2,00	
$M_{Rank}B$	2,50	5,00	3,00	4,33	4,00		2,50	3,00	3,00	
$p_{2tailed}$	0,143	0,400	0,429	1,000	1,000		1,000	0,429	1,000	
r	-0,671	0,433	-0,507	0,154	0,000		0,000	-0,500	0,500	
Other Bugs										
N	15	15	15	15	15	12	14	13	9	15
$M_{Rank}A$	8,25	7,25	7,50	6,92	6,92	6,10	6,20	6,00	8,00	5,50
$M_{Rank}B$	7,83	8,50	8,33	8,72	8,72	6,79	8,22	9,33	6,38	4,75
$p_{2tailed}$	1,000	0,459	0,799	0,633	0,465	0,759	0,425	0,503	0,786	0,170
r	-0,066	0,196	0,104	0,255	0,207	0,095	0,237	0,205	-0,135	-0,367

$r > 0 \Leftrightarrow M_{Rank}B > M_{Rank}A$; $r < 0 \Leftrightarrow M_{Rank}B < M_{Rank}A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

5.2.3 Mistake Occurrence and Tries to Solve Gradient

Table 5.9 Mann-Whitney U-Test: Occurrence and Try Gradients of Group A against B partitioned by Topic

	Occurrence Gradient			Tries to Solve Gradient		
	$a \to b$	$a \to c$	$b \to c$	$a \to b$	$a \to c$	$b \to c$
All Exercises						
N	102	102	94	52	55	50
M_A	-0,667	-0,098	0,031	0,155	0,320	1,965
M_B	-0,513	0,082	-0,297	0,384	1,524	1,075
p_{2tail}	0,169	0,192	0,459	0,862	0,046	0,327
r	0,137	0,130	-0,077	0,025	0,269	-0,140
Reference Attribute Exercises						
N	32	32	46	12	24	26
M_A	-0,462	1,333	0,133	-0,473	0,845	1,761
M_B	-0,504	0,820	-0,171	0,198	2,044	1,263
p_{2tail}	0,947	0,929	0,531	0,268	0,083	0,639
r	-0,016	-0,017	-0,094	0,352	0,357	-0,096
Inheritance Exercises						
N	67	67	54	46	38	33
M_A	-0,522	-0,406	-0,289	0,310	0,055	1,799
M_B	-0,302	-0,243	-0,214	0,387	0,684	0,307
p_{2tail}	0,063	0,308	0,702	0,513	0,805	0,270
r	0,227	0,125	0,053	0,098	0,042	-0,196
Array Exercises						
N	19	19	19	2	3	5
M_A	-0,900	-0,967	0,667	0,500	-0,500	0,083
M_B	-0,889	-0,222	-0,385	-0,714	0,583	2,500
p_{2tail}	1,000	0,334	0,483	1,000	0,667	0,200
r	0,018	0,192	-0,169	-0,707	0,707	0,775

$r > 0 \Leftrightarrow M_B > M_A$; $r < 0 \Leftrightarrow M_B < M_A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

Table 5.10 Mann-Whitney U-Test: Occurrence and Try Gradients of Group A against B partitioned by Mistake Type (1)

	Occurrence Gradient			Tries to Solve Gradient		
	$a \to b$	$a \to c$	$b \to c$	$a \to b$	$a \to c$	$b \to c$
Mistakes when Creating Arrays						
N	5	5	2		2	
M_A	-0,50	-0,83	-1,00		-0,50	
M_B	-1,00	-0,67	-1,00		1,50	
p_{2tail}	0,400	1,000	1,000		1,000	
r	-0,548	0,000	0,000		0,707	
Mistakes relating to Array Indices						
N	7	7	7			
M_A	-1,00	-1,00	-1,00			
M_B	-0,67	-1,00	-1,00			
p_{2tail}	0,429	1,000	1,000			
r	0,436	0,000	0,000			
Mistakes relating to Iteration of Arrays						
N	2	2	6			5
M_A	-1,00	-1,00	2,33			0,08
M_B	-1,00	5,00	1,67			2,50
p_{2tail}	1,000	1,000	0,900			0,200
r	0,000	0,707	-0,183			0,775
Mistakes during Modification of Arrays						
N	3	3				
M_A	-1,00	-1,00				
M_B	-1,00	-1,00				
p_{2tail}	1,000	1,000				
r	0,000	0,000				
Mistakes relating to Attribute values, Getters, and Setters						
N	13	13			8	
M_A	-1,00	1,00			0,83	
M_B	-1,00	0,44			2,12	
p_{2tail}	1,000	0,631			0,393	
r	0,000	-0,149			0,369	
Template modified unintendedly						
N			4			
M_A			-1,00			
M_B			-1,00			
p_{2tail}			1,000			
r			0,000			
Other Bugs						
N	12	12	14	11	10	12
M_A	-0,20	-0,20	-0,20	2,10	0,49	0,26
M_B	-0,07	-0,14	-0,17	0,19	0,51	0,36
p_{2tail}	0,735	0,912	1,000	0,979	1,000	0,909
r	0,108	0,056	0,067	-0,029	0,000	0,049

$r > 0 \Leftrightarrow M_B > M_A$; $r < 0 \Leftrightarrow M_B < M_A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

Table 5.11 Mann-Whitney U-Test: Occurrence and Try Gradients of Group A against B partitioned by Mistake Type (2)

	Occurrence Gradient			Tries to Solve Gradient		
	$a \to b$	$a \to c$	$b \to c$	$a \to b$	$a \to c$	$b \to c$
Element from Class Diagram missing						
N	13	13	18	11	9	14
M_A	-0,33	0,00	-0,27	1,09	0,92	0,58
M_B	-0,18	0,11	0,08	0,84	0,17	0,12
p_{2tail}	0,755	0,685	0,502	0,152	0,667	0,510
r	0,136	0,122	0,170	-0,469	-0,173	-0,191
Element from Class Diagram implemented wrongly						
N	18	18	11	10	4	3
M_A	-0,85	-0,85	-0,83	-0,78	-0,52	-0,46
M_B	-0,47	-0,97	-1,00	3,00	-0,53	-0,56
p_{2tail}	0,111	0,418	0,564	0,700	1,000	0,667
r	0,380	-0,171	-0,235	0,145	-0,224	-0,707
Mistake in or relating to Constructor						
N			13			
M_A			-1,00			
M_B			-1,25			
p_{2tail}			0,487			
r			-0,324			
Mistake relating to Reference Attributes						
N	22	22	19	17	18	16
M_A	-0,40	0,98	0,90	3,57	-0,18	0,64
M_B	-0,31	1,17	0,68	1,17	1,33	3,11
p_{2tail}	0,629	0,230	0,386	0,063	0,132	0,071
r	0,109	0,262	-0,207	-0,455	0,363	0,463

$r > 0 \Leftrightarrow M_B > M_A$; $r < 0 \Leftrightarrow M_B < M_A$; Color-Coding: $p \leq 0.05$; $p \leq 0.10$; $p \leq 0.15$

6 Discussion

6.1 Error Quotient

Independent of how the exercises were partitioned, the error quotient did not yield a significant difference between groups A and B. Given the small sample size, besides the desired significance level of 5 percent, the level of 10 percent has been reviewed. The only exception was exercise 08c on the topic of arrays, where a significant medium effect ($r = -0,533; p = 0,033$) that the error quotient of group B having lower values than group A was observed.

These test results align with the observations of the created boxplots representing the EQ values. They show that a wide variety in the collected data exists which is reflected in large 95%-confidence-intervals. Unfortunately, these results are therefore not usable to answer the research questions. One possible reason could be that the error quotient has only one value for all types of mistakes and can not display effects that exist only for certain types of mistakes. Another reason could be that for different mistake types contrary effects may exist that cancel each other out.

Due to those unclear results of the error quotient, it will not be considered when answering the research questions.

6.2 RQ1

To answer RQ1, the results of the Mann-Whitney-U-Tests for absolute occurrence numbers and their gradients are examined.

6.2.1 Results for all Exercises and different Exercise Topics

The only exercise where a difference between groups A and B for the absolute occurrence count with $p \leq 10$ exists is exercise b in the topic *inheritance* (r=0,225; p=0.053). This value indicates a small effect strength to students in group B having more mistake instances in their submissions for the second exercise with instructive feedback.

Looking at the occurrence gradient that describes how the number of occurrences of mistake instances (of all mistake types) changed between two exercises, the only value significant at a level of 10 percent is the change between exercises a and b in the topic *inheritance* (r=0,169; p=0,063).

Although only one occurrence gradient had a significant difference between groups A and B, it can be noted that both feedback strategies reduce the occurrences of mistake instances between the first and second exercises in all topics. Looking at the total value for all topics, both strategies reduce the occurrences of mistake instances by over 50 percent between exercises a and b.

Even if most comparisons between groups A and B partitioned by topic did not lead to significant results, it is conspicuous that the values of the correlation coefficient were - with only a few exceptions - either close to zero or positive, which means that group B had more mistake occurrences when looking at total occurrences and did not decrease their mistake count in following exercises as strong as group A.

6.2.2 Results for different Mistake Types

Among the absolute numbers of mistake occurrences partitioned by mistake type, more values complied with the significance level of 5 or 10 percent.

The strongest effect in those values (although it is only significant on the level of 10 percent) is the value for the average occurrence count of *mistakes when creating arrays* in exercises a and b, which was significantly lower for group B than for group A (r=-0,894; p=0,067). Still, it is interesting that there is no more difference between the occurrences in exercise c (which has uniform non-instructional feedback for both groups) for that type of mistake (r=-0,228; p=1,000). This might point to an unstable effect of the feedback to reduce occurrences of specific mistakes instances in exercises with varied feedback that does not last to following exercises and does, therefore, not support learning to avoid a certain type of mistake in exercises where the scaffolding is reduced.

Contrary to mistakes during array creation, more mistake instances of the type *element from the class diagram implemented wrongly* occurred in the b-exercises of group B with a medium to large effect (r=0,495; p=0,043). However, that again is not a stable effect that lasts until the following exercises in which no significant difference between the groups was found.

Looking at mistakes related to *reference attributes*, students of group b made significantly more mistakes in the first exercises of a topic (r=0,474; p=0,027). In the second exercise, no such difference was present, while in the third exercise, with uniform feedback for both groups, the difference appeared again with a similar effect strength (r=0,419; p=0,052). This again points to a learning effect that lasts for exercises where instructive feedback supports students. Whereas in the following exercises, where this scaffold is reduced to the simpler feedback strategy, the effect disappears again and can, therefore, be considered unstable.

The last mistake type with significant differences between both groups is when an *element of a given class diagram was not implemented correctly* in exercise b (r=0,495; p=0,043). Here, students from group B made significantly more such mistakes, while no significant difference could be observed in exercises a and c.

As the occurrence gradient contained no significant differences between groups A and B when splitting by mistake type, no valid explanation of the effects on the absolute occurrence number can be found.

6.2.3 Conclusion

RQ1 In conclusion, no general effect of the choice of one of the given feedback strategies on the repetition of mistakes in subsequent exercises and, thereby, on the learning success could be found. When specifically looking at single exercise topics or mistake types, significant differences between the absolute occurrence count or the occurrence gradient between subsequent exercises could be found sporadically. However, these effects did not last to the following exercises especially not to the ones where the scaffold was reduced to a uniform level.
Therefore, **Hypothesis H1 could not be verified** in general, nor for certain exercise topics or mistake types. However no opposite effect could be found as well.

6.3 RQ2

As described in section 4.3.2, no valid time-based measure for the speed of solving a mistake could be found. Due to the finding that there is a high correlation between the number of change events and the session time [35] for answering RQ2 the number of submission resp. tries to solve a mistake is used to approximate the speed of a solution for a mistake. To find differences, the results of the Mann-Whitney-U-tests of tries to solve one mistake occurrence and of the respective gradient are compared.

6.3.1 Results for all Exercises and different Exercise Topics

When comparing the absolute number of tries to solve one mistake in groups A and B for *all exercises* at once, no significant results could be found. However, the gradient between exercises a and c showed a significant difference between the groups. Both groups had a positive value, meaning they needed more tries in exercises c than in exercises a. The value of group B was significantly larger than the value of A (r=0,269; p=0,046; M_A=0,320; M_B=1,524).

For exercises on the topic of *inheritance*, no significant differences between the groups were found.

The absolute values for tries to solve a mistake instance in *array exercises* contained a difference on the significance level of 10 percent between the two groups in exercises a (r=0,385; p=0,096) and c (r=0,689; p=0,071). In exercise A, a medium effect of group B needing more tries was found, and in exercise c, this effect was even large.

In the exercises on the topic *reference attributes*, significant differences between the groups were found in the number of tries to solve a mistake instance in exercises b (r=0,518; p<0,001), c (r=0,344; r=0,031), and the average tries of a and b which are the exercises with instructive feedback (r=0,250; p=0,042). The only gradient that significantly differed between the groups in this exercise topic was the one between exercises a and c (r=0,357; p=0,083). These results show that the choice of the feedback strategy has a stable effect on the speed with which mistakes in exercises on the topic of *reference attributes* are solved. In the second and third exercises on reference attributes, students in group B with instructive feedback needed more tries to solve mistakes than students in group A. For exercise c, this result aligns with the expectation in hypothesis H2.2 that students with non-instructive feedback in exercises a and b need fewer tries to solve mistakes in exercise c, whereas the difference for exercise b is contrary to the expectation from hypothesis H2.1 that students with instructive feedback solve these exercises with instructive feedback faster than the ones with non-instructive feedback.

6.3.2 Results for different Mistake Types

Like the previous results, the results for the *tries to solve a mistake instance* partitioned by mistake type suffer from bad significance values due to the small sample. This can be seen in table 5.10, where many of the U-test results between the gradients could not be calculated due to the small number of datasets.

Nevertheless, a few significant differences could still be discovered. These include a significant result that students from group B needed more tries to solve mistakes with *missing elements from the class diagram* in the project exercise (r=0,399; p=0,097) and a significant large effect at the difference between the groups at exercises b when trying to solve *mistakes that are in or related to constructors* (r=0,594; p=0,034).

Mistakes related to *reference attributes* had the most utilizable test results. In b-exercises, students from group B needed significantly more tries to solve mistakes than students from group A (r=0,685; p=0,002). This effect occurs again in the project exercise but with slightly lower effect strength and significance at 10 percent (r=0,359; p=0,096). However, in exercises a and c, no significant difference was found between the groups. Still, the gradient between the number of tries to solve a mistake between the different exercises showed significant differences. For both groups, the number of tries to solve mistakes about reference attributes increased from exercise a to b, but the one of group B grew less strong (r=-0,455; p=0,063). Looking at the gradient from exercises b to c, the opposite effect could be observed, and the number of tries to solve reference attribute mistakes of the students in group B grew stronger than those of group A (r=0,463; p=0,071). Thus, from exercises a to c no significant difference could be observed. These findings about the tries to solve reference attribute mistakes support the assumptions from hypotheses H2.1 and H2.2 that group B needs fewer tries in exercises with instructive feedback and more tries in consecutive exercises with non-instructive uniform feedback than group A.

6.3.3 Conclusion

RQ2 In conclusion, no general effect of the choice of one of the given feedback strategies on the speed of solving mistake instances could be found. However, for specific exercise topics and mistake types significant results could be found.
Hypothesis 2.1 could not be verified in general and for exercises about *reference attributes* where even significant opposing effects (r=0,250; p=0,042) were discovered. Additionally, opposing effects were found for mistakes related to constructors (r=0,594; r=0,034).
Hypothesis 2.2 could not be verified in general. However, it was verified for certain exercise topics and mistake types. The number of tries to solve mistakes in exercises about *reference attributes* supports H2.2 with a moderate effect strength (r=0,344; p=0,031) and for exercises about *arrays* with a large effect (r=0,689; p=0,071). Additionally, the evidence of tries to solve mistakes with *missing elements from class diagram* (r=0,399; p=0,097) and related to *reference attributes* (r=0,359; p=0,096) supports hypothesis 2.2 for these specific mistake types.

6.4 Limitations

Circumstances As in most cases of small studies conducted in authentic classroom settings, this one, too, suffers from several limitations and external influences.

These start with the small sample size, which leads to many results that have unacceptable significance and several values that could not be calculated at all. That the students worked in teams reduces the sample size even further. Additionally, I observed (in an unstructured manner) that some teams divided the work, and different team members worked on different exercises, which falsified the results of the team's progress. Students reported that this behavior was caused by time pressure, which aligns with a problem reported by Jenkins [3] that the learning speed in a school environment is often set by the needs of assessment and the school year. Another thread is that the results of the programming exercises were used to grade the students. This was inevitable, but students reported that it added to the pressure, which is in line with literature [3].

Another huge problem was problems caused by the IT infrastructure. Most conspicuous were network problems randomly blocking the Git protocol for cloning the exercise template and uploading submissions to the ArTEMiS server.

Evaluation For evaluating the resulting data, several problems occurred. The most significant is that no detailed data about the working behavior could be collected. This was especially problematic when answering RQ2 as no valid information about the time between submissions was available when students worked on an exercise in two sessions. Due to that, no time measure could be used for the speed of solving a problem, which is a big handicap as it is known that there is only a moderate correlation between fine- and coarse-grained time measures [31]. Besides that, questioning the students if and how intensely they read the feedback text would have been interesting to gain a better understanding of the working behavior.

Additionally, the chosen approach to calculate the error quotient was found to be too unspecific to gain information about certain mistake types. Using the other measures of mistake occurrences and number of tries to solution, showed that the differentiation between the mistake types is important.

Furthermore, no structured survey about the students' experience and my observations has been conducted. Due to the collected data having several shortcomings, structured survey results would have helped interpret the data.

To provide more specific data about the mistakes the students made, insights into the compiler errors would be crucial as compiler errors are the ones that occurred the most when learning to program [2]. Unfortunately, ArTEMiS does not provide such information for evaluation.

7 Summary and Future Work

All in all, designing automated feedback for programming exercises and the use of an autograding platform is a highly complex topic. Besides general design principles for exercises, scaffolding, and feedback, exercise topics, and mistake types have to be considered. Feedback strategies that work well for certain types of mistakes might be completely insufficient for other types of mistakes of exercise topics.

To make matters worse, not all findings for normal feedback can be transferred to automated feedback, as seen in the results of RQ2. Unfortunately, many results suffered from the small sample size and, therefore, unsatisfactory significance. Nevertheless, automated feedback using autograding systems in K-12 is an exciting topic as most research about autograders is conducted in universities. The results of this thesis can, therefore, be a good starting point for further work in the field of automated feedback in K-12 educational contexts.

In future work on the topic, it is important to ensure that samples are large enough to create significant results. Additionally, it should be ensured that (meta-)information about the working process is available as detailed as possible. Besides information on the working process, knowledge about the students' experience can be interesting in interpreting their behavior. Furthermore, analyzing the specific code changes after certain feedback items would be interesting to get an even deeper understanding of how students react to which type of feedback.

Declaration of the use of artificial intelligence

In preparing this thesis, I utilized Grammarly[1] for grammar and style correction across all chapters, ensuring clarity and coherence in my writing. I used DeepL[2] to enhance language quality and translate minor parts of the literature. I used Microsoft Copilot[3] for inspiration while creating the students' exercises. Additionally, I used GitHub Copilot [4] to speed up the coding of the exercises' templates, solutions, and test cases and generate code snippets in Python and SQL for data analysis.

[1] www.grammarly.com
[2] www.deepl.com
[3] copilot.microsoft.com
[4] github.com/features/copilot

List of Figures

List of Tables

Bibliography

[1] Stephan Krusche and Andreas Seitz. Artemis. pages 284–289. doi: 10.1145/3159450.3159602.

[2] Chin Soon Cheah. *Factors contributing to the difficulties in teaching and learning of computer programming: A literature review.* Ali Simsek, Eskisehir, 2020.

[3] Tony Jenkins. On the difficulty of learning to program. 2002.

[4] Ulf P. Lundgren. *Frame factors and the teaching process : a contribution to curriculum theory and theory on teaching.* 1972.

[5] Tilman Michaeli. *Debugging im Informatikunterricht.* 2021. doi: 10.17169/refubium-28811. URL `https://refubium.fu-berlin.de/handle/fub188/29061`.

[6] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. Stop the (autograder) insanity: Regression penalties to deter autograder overreliance. In Mark Sherriff, editor, *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ACM Digital Library, pages 1062–1068, New York,NY,United States, 2021. Association for Computing Machinery. ISBN 9781450380621. doi: 10.1145/3408877.3432430.

[7] Hieke Keuning, Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. 2018. URL `https://dl.acm.org/doi/pdf/10.1145/3231711`.

[8] Andrew Luxton-Reilly, Ewan Tempero, Nalin Arachchilage, Angela Chang, Paul Denny, Allan Fowler, Nasser Giacaman, Igor Kontorovich, Danielle Lottridge, Sathiamoorthy Manoharan, Shyamli Sindhwani, Paramvir Singh, Ulrich Speidel, Sudeep Stephen, Valerio Terragni, Jacqueline Whalley, Burkhard Wuensche, and Xinfeng Ye. Automated assessment: Experiences from the trenches. In *ACE '23: Proceedings of the 25th Australasian Computing Education Conference*, pages 1–10, 2023. doi: 10.1145/3576123.3576124.

[9] Luisa Greifenstein, Isabella Graßl, Ute Heuer, and Gordon Fraser. Common problems and effects of feedback on fun when programming ozobots in primary school. In Mareen Grillenberger, editor, *Proceedings of the 17th Workshop in Primary and Secondary Computing Education*, ACM Digital Library, pages 1–10, New York,NY,United States, 2022. Association for Computing Machinery. ISBN 9781450398534. doi: 10.1145/3556787.3556860.

[10] Luisa Greifenstein, Markus Brune, Tobias Fuchs, Ute Heuer, and Gordon Fraser. Impact of hint content on performance and learning: A study with primary school children in a scratch course. In Sue Sentance and Mareen Grillenberger, editors, *Proceedings of the 18th WiPSCE Conference on Primary and Secondary Computing Education Research*, pages 1–10, New York, NY, USA, 2023. ACM. ISBN 9798400708510. doi: 10.1145/3605468.3605498. URL `https://dl.acm.org/doi/pdf/10.1145/3605468.3605498`.

[11] Susanne Narciss. Designing and evaluating tutoring feedback strategies for digital learning. *Digital Education Review*, (23):7–26, 2013. ISSN 2013-9144. URL `https://dialnet.unirioja.es/servlet/articulo?codigo=4335567`.

[12] artemis.cit.tum.de, 2024. URL `https://artemis.cit.tum.de/`.

[13] Yorah Bosse and Marco Aurélio Gerosa. Why is programming so difficult to learn? *ACM SIGSOFT Software Engineering Notes*, 41(6):1–6, 2017. ISSN 0163-5948. doi: 10.1145/3011286.3011301.

[14] Robert Moser. A fantasy adventure game as a learning environment. pages 114–116. doi: 10.1145/268819.268853.

[15] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming. 2017. doi: 10.1145/3077618. URL https://dl.acm.org/doi/pdf/10.1145/3077618.

[16] Siti Rosminah MD Derus and Ahmad Zamzuri Mohamad Ali. Difficulties in learning programming: Views of students.

[17] Janneke van de Pol, Monique Volman, and Jos Beishuizen. Scaffolding in teacher–student interaction: A decade of research. *Educational Psychology Review*, 22(3):271–296, 2010. ISSN 1573-336X. doi: 10.1007/s10648-010-9127-6. URL https://link.springer.com/article/10.1007/s10648-010-9127-6.

[18] Nicola Yelland and Jennifer Masters. Rethinking scaffolding in the information age. *Computers & Education*, 48(3):362–382, 2007. ISSN 03601315. doi: 10.1016/j.compedu.2005.01.010. URL https://www.sciencedirect.com/science/article/pii/S0360131505000187.

[19] David Wood, Jerome S. Bruner, and Gail Ross. *THE ROLE OF TUTORING IN PROBLEM SOLVING*. 1976. URL https://elearning.auth.gr/pluginfile.php/1557342/mod_resource/content/4/wood%2c%20bruner%20%20ross%20the%20role%20of%20tutoring%20%282%29.pdf.

[20] C. A. Stone. The metaphor of scaffolding: its utility for the field of learning disabilities. *Journal of learning disabilities*, 31(4):344–364, 1998. ISSN 0022-2194. doi: 10.1177/002221949803100404.

[21] C. A. Stone. Should we salvage the scaffolding metaphor? *Journal of learning disabilities*, 31 (4):409–413, 1998. ISSN 0022-2194. doi: 10.1177/002221949803100411.

[22] Heather Leary, Brian R. Belland, Andrew E. Walker, Megan Whitney Olsen, and Heather Leary. A pilot meta-analysis of computer-based scaffolding in stem education. *Journal of Educational Technology & Society*, 18(1):183–197, 2015. ISSN 11763647. URL http://www.jstor.org/stable/jeductechsoci.18.1.183.

[23] Nam Ju Kim, Brian R. Belland, Mason Lefler, Lindi Andreasen, Andrew Walker, and Daryl Axelrod. Computer-based scaffolding targeting individual versus groups in problem-centered instruction for stem education: Meta-analysis. *Educational Psychology Review*, 32(2):415–461, 2020. ISSN 1573-336X. doi: 10.1007/s10648-019-09502-3. URL https://link.springer.com/article/10.1007/s10648-019-09502-3.

[24] Simon Rupp, René Pawlitzek, and Carlo Bach. Metriken zur messungvon lernerfolg im informatik- grundlagen-unterricht. 2017. URL https://dl.gi.de/server/api/core/bitstreams/d89146bc-c463-4fea-8ee7-678f3401616a/content.

[25] Bridgid Finn and Janet Metcalfe. Scaffolding feedback to maximize long-term error correction. *Memory & cognition*, 38(7):951–961, 2010. doi: 10.3758/MC.38.7.951. URL https://link.springer.com/article/10.3758/mc.38.7.951.

[26] Fernando Gutierrez and John Atkinson. Adaptive feedback selection for intelligent tutoring systems. *Expert Systems with Applications*, 38(5):6146–6152, 2011. ISSN 09574174. doi: 10.1016/j.eswa.2010.11.058.

[27] Alberto Simões and Ricardo Queirós. On the nature of programming exercises: Schloss dagstuhl – leibniz-zentrum für informatik. volume 81. doi: 10.4230/OASIcs.ICPEC.2020.24. URL http://arxiv.org/pdf/2006.14476.

[28] Shuchi Grover, Stephen Cooper, and Roy Pea. Assessing computational learning in k-12. pages 57–62. doi: 10.1145/2591708.2591713. URL https://dl.acm.org/doi/pdf/10.1145/2591708.2591713.

[29] Angela Carbone, John Hurst, Ian Mitchell, and Dick Gunstone. Principles for designing programming exercises to minimise poor learning behaviours in students. 2000. URL https://dl.acm.org/doi/pdf/10.1145/359369.359374.

[30] André L. Santos. Shifting programming education assessment from exercise outputs toward deeper comprehension (invited talk): Schloss dagstuhl – leibniz-zentrum für informatik. *OASIcs, Volume 112, ICPEC 2023*, 112, 2023. doi: 10.4230/OASICS.ICPEC.2023.1.

[31] Juho Leinonen, Francisco Enrique Vicente Castro, and Arto Hellas. Fine-grained versus coarse-grained data for estimating time-on-task in learning programming. In *Proceedings of The 14th International Conference on Educational Data Mining (EDM 2021)*, 2021. URL https://acris.aalto.fi/ws/portalfiles/portal/66787593/EDM2021_TimeOnTask.pdf.

[32] Morgane Chevalier, Christian Giang, Laila El-Hamamsy, Evgeniia Bonnet, Vaios Papaspyros, Jean-Philippe Pellet, Catherine Audrin, Margarida Romero, Bernard Baumberger, and Francesco Mondada. The role of feedback and guidance as intervention methods to foster computational thinking in educational robotics learning activities for primary school. *Computers & Education*, 180:104431, 2022. ISSN 03601315. doi: 10.1016/j.compedu.2022.104431.

[33] Stephan Krusche. Artemis user, contributor, and admin guide. URL https://docs.artemis.cit.tum.de/.

[34] Johan Snider. Edit, run, error, repeat: Learning analytics to identify most improved programming student. 2024. URL https://www2.it.uu.se/student/thesis_project/links2/cer_thesis.pdf.

[35] Juho Leinonen, Leo Leppänen, Petri Ihantola, and Arto Hellas. Comparison of time metrics in programming. 2017. doi: 10.1145/3105726.3106181. URL https://dl.acm.org/doi/pdf/10.1145/3105726.3106181.

[36] Matthew C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, ACM Conferences, pages 73–84. ACM, New York, NY, 2006. ISBN 1595934944. doi: 10.1145/1151588.1151600.

[37] Patrick Bassner, Eduard Frankford, and Stephan Krusche. Iris: An ai-driven virtual tutor for computer science education. 2024. URL https://ase.cit.tum.de/research/publications/bassner2024iticse.pdf.

[38] A. Devolder, J. Van Braak, and J. Tondeur. Supporting self-regulated learning in computer-based learning environments: systematic review of effects of scaffolding in the domain of science education. *Journal of Computer Assisted Learning*, 28(6):557–573, 2012. ISSN 1365-2729. doi: 10.1111/j.1365-2729.2011.00476.x. URL https://onlinelibrary.wiley.com/doi/full/10.1111/j.1365-2729.2011.00476.x.

[39] Journal of Computing Sciences in Colleges. Introduction to bluej: a java development environment: Journal of computing sciences in colleges: Vol 16, no 3, 2001.

[40] Jürgen Hedderich and Lothar Sachs. *Angewandte Statistik: Methodensammlung mit R*. Springer eBook Collection. Springer Spektrum, Berlin and Heidelberg, 17., überarbeitete und ergänzte auflage edition, 2020. ISBN 9783662622940. doi: 10.1007/978-3-662-62294-0. URL https://link.springer.com/content/pdf/10.1007/978-3-662-62294-0.pdf.

[41] Wanja A. Hemmerich. ungepaarter t-test | statistikguru.de, 9/10/2024. URL https://statistikguru.de/spss/ungepaarter-t-test.

[42] Wanja A. Hemmerich. Mann-whitney-u-test | statistikguru.de, 9/10/2024. URL https://statistikguru.de/spss/mann-whitney-u-test.

[43] Catherine O. Fritz, Peter E. Morris, and Jennifer J. Richler. Effect size estimates: current use, calculations, and interpretation. *Journal of experimental psychology. General*, 141(1):2–18, 2012. doi: 10.1037/a0024338. URL https://psycnet.apa.org/fulltext/2011-16756-001.pdf?auth_token=d66de2a4a2d8dfbe0a1d5bd618614f274124ce3e&returnUrl=https%3A%2F%2Fpsycnet.apa.org%2Frecord%2F2011-16756-001.

A Appendix

A.1 Digital Appendix (Database, Java Source Code)

All files in the digital Appendix are embedded into the PDF document. Each file can be opened by clicking on the pin left to its name. Not all PDF viewers support embedded files. They have been tested with Adobe Acrobat and Okular. As a fallback, the digital appendix is available at `https://bit.ly/VH_BachelorThesisAppendix`.

- SQLite3 Database containing the processed data (*.db)
- Java Source Code: Template, Solution and Unit Tests of all Exercises (*.zip)

A.2 Exercises (original, German)

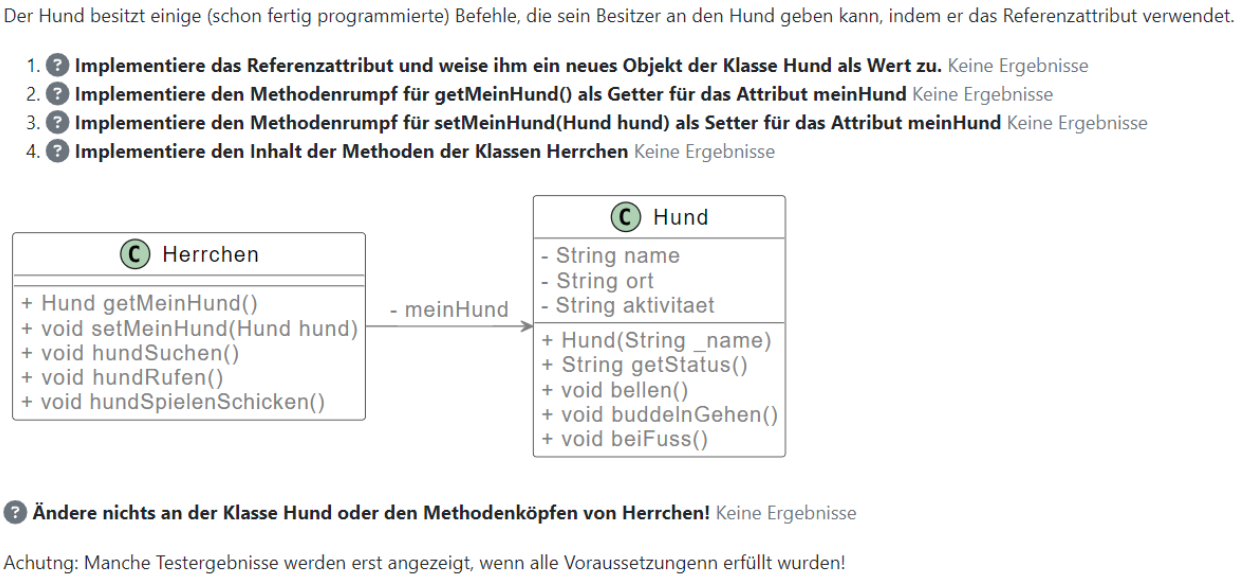


Figure A.1 Exercise Statement of Exercise 06a (original, German)

Aufgabe

1. **Klassenstruktur implementieren** Keine Ergebnisse
   Implementiere die Methoden (zunächst leer) und die Attribute von Fahrer (siehe Klassendiagrammsunten).
2. **Methoden implementieren** Keine Ergebnisse
   Im Konstruktor von Fahrer soll hierbei ein neues Objekt der Klasse Fahrzeug angelegt und dem entsprechenden Attribut zugewiesen werden. Der Konstruktor von Fahrzeug benötigt als Parameter ein Objekt der Klasse Fahrer. Das aktuelle Fahrerobjekt, kann in Methoden der Klasse mit `this` verwendet (und auch als Parameter übergeben) werden. Die Benennung der restlichen Methoden beschreibt bereits, was getan werden soll.

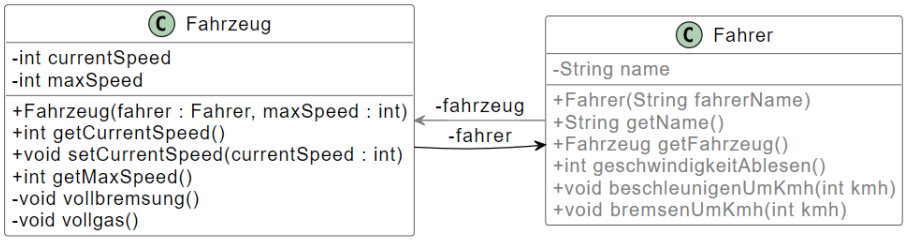
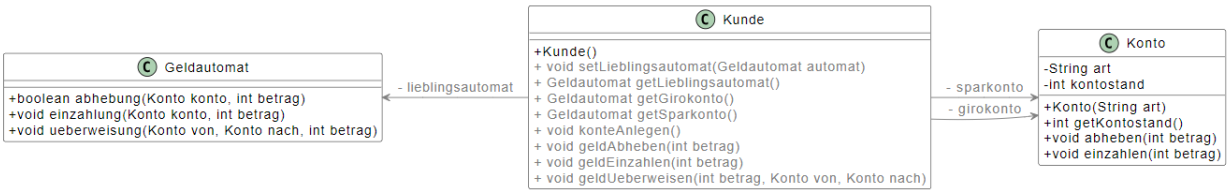


Figure A.2 Exercise Statement of Exercise 06b (original, German)

Ein `Kunde` kann bei seiner Bank zwei Konten besitzen: ein Girokonto und ein Sparkonto. Möchte er mit diesen Konten interargieren, nutzt er hierfür einen `Geldautomat`. Jeder Kunde hat einen `lieblingsautomat`, der im entsprechenden Referenzattribut gespeichert wird. Die Klassen `Geldautomat` und `Konto` sind bereits fertig. Die Klasse `Kunde` musst du noch fertigstellen. Wie die Klassen miteinander zusammenhängen, siehst du unten im Klassendiagramm.

1. **Deklariere das Referenzattribut lieblingsautomat und weise ihm ein neues Objekt der Klasse Geldautomat als Wert zu.** Keine Ergebnisse
2. **Deklariere die Referenzattribute Sparkonto und Girokonto, weise ihnen aber noch keine Werte zu** Keine Ergebnisse
3. **Weise den beiden Konto-Referenzattributen in der Methode kontenAnlegen() jeweils ein neues Objekt mit der Art 'Girokonto' bzw. 'Sparkonto' zu.** Keine Ergebnisse
4. **Implementiere den Methodenrumpf für setLieblingsautomat(Geldautomat automat) als Setter für das Attribut lieblingsautomat** Keine Ergebnisse
5. **Implementiere den Methodenrumpf für die Getter-Methoden getLieblingsautomat(), getGiroKonto() und getSparKonto()** Keine Ergebnisse
6. **Implementiere den Inhalt der Methoden der Klassen Kunde, indem du Methoden deine Lieblingsautomats aufrufst.** Keine Ergebnisse
   Tipp: Direkte Abhebungen und Einzahlungen sind hierbei nur vom Girokonto möglich. Eine Interaktion mit dem Sparkonto ist nur per Überweisung möglich.



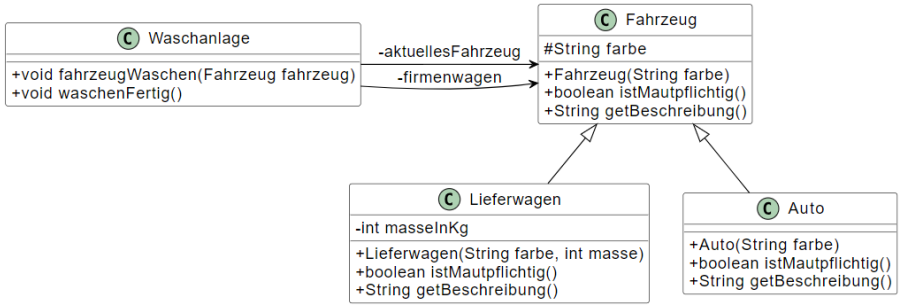
Achutng: Manche Testergebnisse werden erst angezeigt, wenn alle Voraussetzungenn erfüllt wurden!

Figure A.3 Exercise Statement of Exercise 06c (original, German)

Die Software einer Waschanlage soll in der Lage sein, verschiedene Arten von Fahrzeugen zu verwalten. Die allgemeine Fahrzeugklasse ist bereits fertig implementiert. Ebenso gibt es bereits die leeren Klassen Auto und Lieferwagen.

1. **Bearbeite Auto und Lieferwagen so, dass sie Unterklassen von Fahrzeug werden. Lege hierbei jeweils einen Konstruktor mit den gleichen Parametern wie bei Fahrzeug an und vergiss nicht den Konstruktor der Oberklasse aufzurufen.** Keine Ergebnisse
2. **Verändere 'Fahrzeug' so, dass in den Unterklassen auf alle Attribute direkt zugegriffen werden kann.** Keine Ergebnisse
3. **Deklariere in 'Lieferwagen' ein integer-Attribut 'masseInKg' und ergänze einen Konstruktor-Parameter (mit beliebigem Namen), dessen Wert zu dem Attribut zugewiesen wird.** Keine Ergebnisse
4. **Überschreibe in 'Lieferwagen' die Methode 'istMautpflichtig(), sodass true zurückgegeben wird, wenn der Lieferwagen schwerer als 3500kg ist.** Keine Ergebnisse
5. **Überschreibe 'getBeschreibung()' in beiden Unterklassen, sodass 'Fahrzeug' im jeweils zurückgegebenen Text durch 'Auto' bzw. 'Lieferwagen' ersetzt wird.** Keine Ergebnisse
6. **Weise dem Referanzattribut 'firmenwagen' ein neues Objekt der Klasse Lieferwagen mit beliebigen Werten zu** Keine Ergebnisse
7. **In der Waschanlage aktuell nur Autos gewaschn werden. Prüfe daher in 'fahrzeugWaschen(...)', ob es sich um ein Objekt der Klasse Auto handelt und weise den Paramterwert nur dann dem Attribut 'aktuellesFahrzeug' zu.** Keine Ergebnisse

Das Klassendiagramm zeigt keine Ergebnisse an, sondern dient nur zur Übersicht über die Struktur.

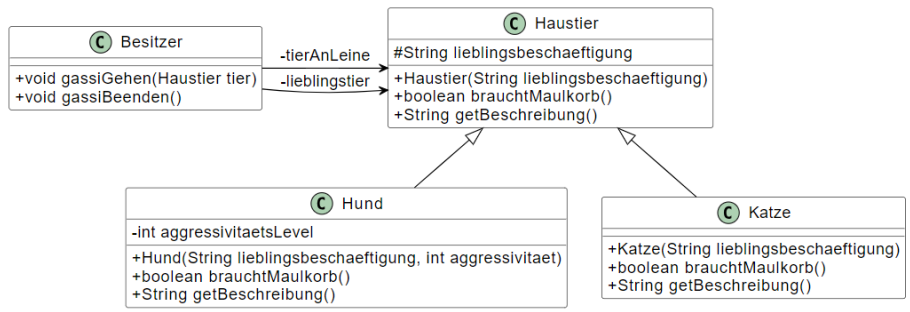


Pass auf, dass du nicht versehentlich die Klassennamen oder die Package Anweisung änderst! Keine Ergebnisse

Figure A.4 Exercise Statement of Exercise 07a (original, German)

1. **Bearbeite Katze und Hund so, dass sie Unterklassen von Haustier werden. Lege hierbei jeweils einen Konstruktor mit den gleichen Parametern wie bei Haustier an und vergiss nicht den Konstruktor der Oberklasse aufzurufen.** Keine Ergebnisse
2. **Verändere 'Fahrzeug' so, dass in den Unterklassen auf alle Attribute direkt zugegriffen werden kann.** Keine Ergebnisse
3. **Deklariere in 'Hund' ein integer-Attribut 'aggresivitaetsLevel' und ergänze einen Konstruktor-Parameter (mit beliebigem Namen), dessen Wert zu dem Attribut zugewiesen wird.** Keine Ergebnisse
4. **Überschreibe in 'Hund' die Methode 'brauchtMaulkorb(), sodass true zurückgegeben wird, wenn der Hund ein Aggresitivaets-Level größer als 5 hat.** Keine Ergebnisse
5. **Überschreibe 'getBeschreibung()' in beiden Unterklassen, sodass 'Haustier' im jeweils zurückgegebenen Text durch 'Katze' bzw. 'Hund' ersetzt wird.** Keine Ergebnisse
6. **Weise dem Referanzattribut 'lieblingstier' ein neues Objekt der Klasse Hund mit beliebigen Werten zu** Keine Ergebnisse
7. **Der Besitzer kann nur mit Hunden gassi gehen. Prüfe daher in 'gassiGehen(...)', ob es sich bei dem Parameter um ein Objekt der Klasse Hund handelt und weise den Paramterwert nur dann dem Attribut 'tierAnLeine' zu.** Keine Ergebnisse

Das Klassendiagramm zeigt keine Ergebnisse an, sondern dient nur zur Übersicht über die Struktur.

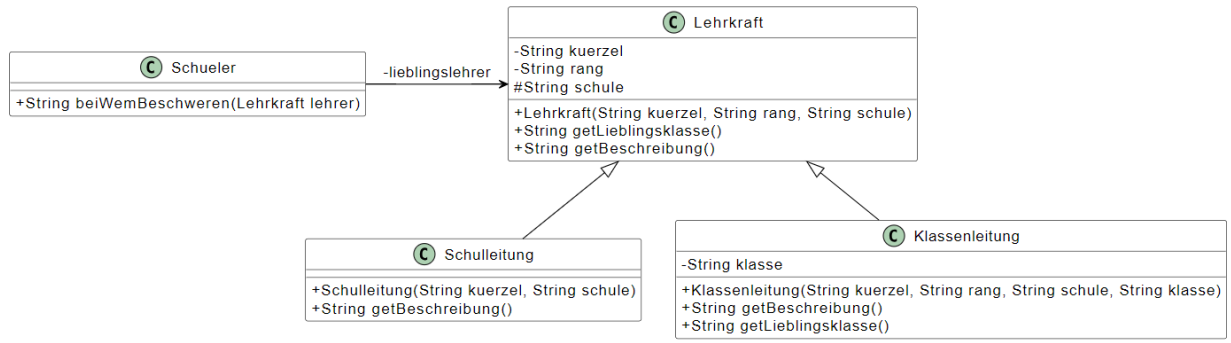


Pass auf, dass du nicht versehentlich die Klassennamen oder die Package Anweisung änderst! Keine Ergebnisse

Figure A.5 Exercise Statement of Exercise 07b (original, German)

1. **Setze unten stehendes Klassendiagramm im Programmcode um. Die Klasse 'Lehrkraft' muss dafür NICHT geändert werden! Vergiss nicht, dem Attribut 'klasse' von Klassenleitung den Wert des entsprechenden Konstruktor-Parameters zuzuweisen. Eine Schulleitung hat außerdem immer den Rang 'OStD'.** Keine Ergebnisse
2. **Überschreibe in 'Klassenleitung' die Methode 'getLieblingsklasse()' so, dass die eigene Klasse als Lieblingsklasse zurückgegeben wird.** Keine Ergebnisse
3. **Überschreibe 'getBeschreibung()' in beiden Unterklassen, sodass an den Text der Methode aus der Oberklasse entweder "und ist Klassenleitung der Klasse ..." oder "und ist Schulleitung des ..." mit einem jeweils passenden Wert statt '...' angehängt wird.** Keine Ergebnisse
4. **Weise dem Referanzattribut 'lieblingslehrer' ein neues Objekt der Klasse Klasseleitung mit beliebigen Werten zu** Keine Ergebnisse
5. **Die Methode 'beiWemBeschweren(Lehrkraft lehrer)' gibt zurück, bei wem man sich über die Lehrkraft, die Parameterwert ist, beschweren kann. Das ist bei einer Schulleitung "Ministerialbeauftragter", bei einer Klassenleitung "Schulleitung" und bei einer normalen Lehrkraft "Klassenleitung".** Keine Ergebnisse

Das Klassendiagramm zeigt keine Ergebnisse an, sondern dient nur zur Übersicht über die Struktur.

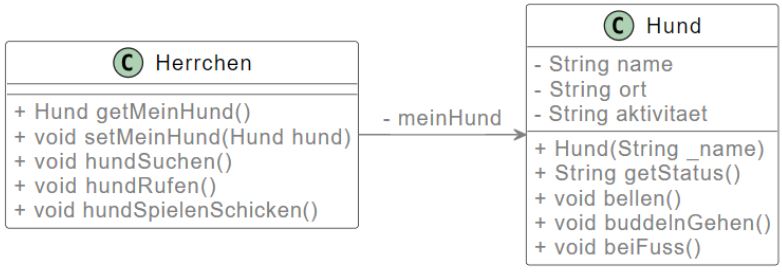


Pass auf, dass du nicht versehentlich die Klassennamen oder die Package Anweisung änderst! Keine Ergebnisse

Figure A.6 Exercise Statement of Exercise 07c (original, German)

Der Hund besitzt einige (schon fertig programmierte) Befehle, die sein Besitzer an den Hund geben kann, indem er das Referenzattribut verwendet.

1. **Implementiere das Referenzattribut und weise ihm ein neues Objekt der Klasse Hund als Wert zu.** Keine Ergebnisse
2. **Implementiere den Methodenrumpf für getMeinHund() als Getter für das Attribut meinHund** Keine Ergebnisse
3. **Implementiere den Methodenrumpf für setMeinHund(Hund hund) als Setter für das Attribut meinHund** Keine Ergebnisse
4. **Implementiere den Inhalt der Methoden der Klassen Herrchen** Keine Ergebnisse



Ändere nichts an der Klasse Hund oder den Methodenköpfen von Herrchen! Keine Ergebnisse

Achutng: Manche Testergebnisse werden erst angezeigt, wenn alle Voraussetzungenn erfüllt wurden!

Figure A.7 Exercise Statement of Exercise 08a (original, German)

Aufgabe

1. **Klassenstruktur implementieren** Keine Ergebnisse
   Implementiere die Methoden (zunächst leer) und die Attribute von Fahrer (siehe Klassendiagrammsunten).
2. **Methoden implementieren** Keine Ergebnisse
   Im Konstruktor von Fahrer soll hierbei ein neues Objekt der Klasse Fahrzeug angelegt und dem entsprechenden Attribut zugewiesen werden. Der Konstruktor von Fahrzeug benötigt als Parameter ein Objekt der Klasse Fahrer. Das aktuelle Fahrerobjekt, kann in Methoden der Klasse mit `this` verwendet (und auch als Parameter übergeben) werden. Die Benennung der restlichen Methoden beschreibt bereits, was getan werden soll.

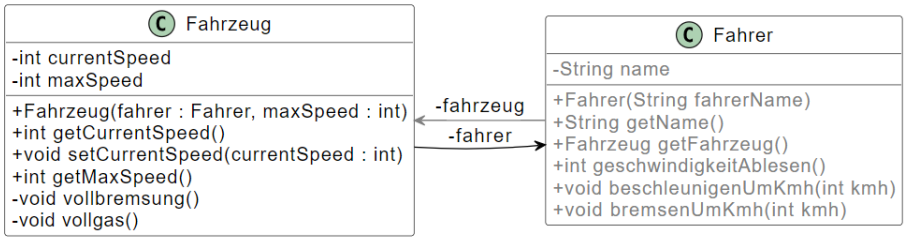
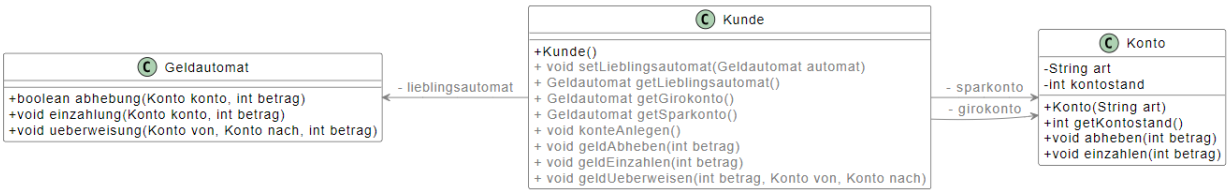


Figure A.8 Exercise Statement of Exercise 08b (original, German)

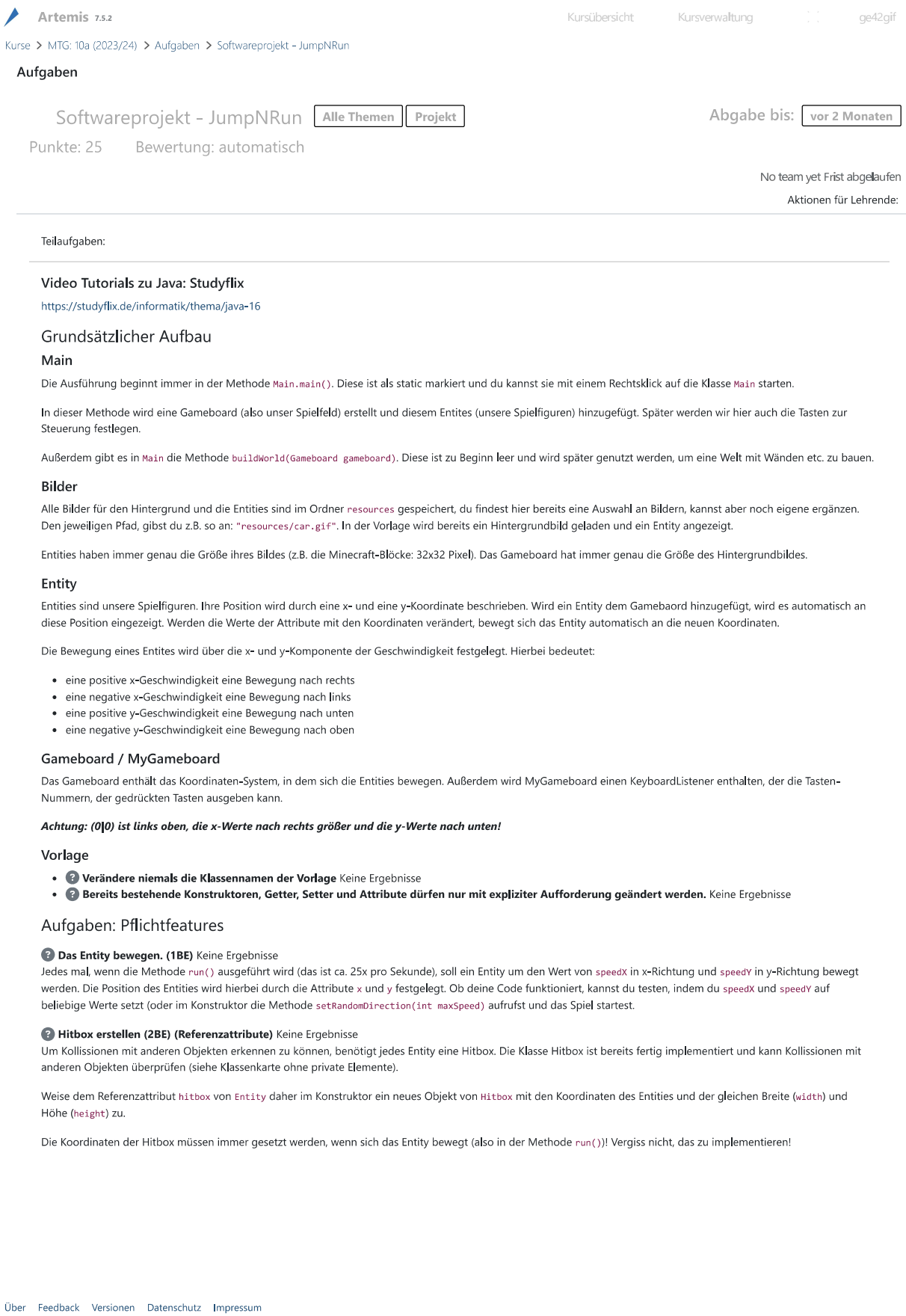
Ein `Kunde` kann bei seiner Bank zwei Konten besitzen: ein Girokonto und ein Sparkonto. Möchte er mit diesen Konten interargieren, nutzt er hierfür einen `Geldautomat`. Jeder Kunde hat einen `lieblingsautomat`, der im entsprechenden Referenzattribut gespeichert wird. Die Klassen `Geldautomat` und `Konto` sind bereits fertig. Die Klasse `Kunde` musst du noch fertigstellen. Wie die Klassen miteinander zusammenhängen, siehst du unten im Klassendiagramm.

1. **Deklariere das Referenzattribut lieblingsautomat und weise ihm ein neues Objekt der Klasse Geldautomat als Wert zu.** Keine Ergebnisse
2. **Deklariere die Referenzattribute Sparkonto und Girokonto, weise ihnen aber noch keine Werte zu** Keine Ergebnisse
3. **Weise den beiden Konto-Referenzattributen in der Methode kontenAnlegen() jeweils ein neues Objekt mit der Art 'Girokonto' bzw. 'Sparkonto' zu.** Keine Ergebnisse
4. **Implementiere den Methodenrumpf für setLieblingsautomat(Geldautomat automat) als Setter für das Attribut lieblingsautomat** Keine Ergebnisse
5. **Implementiere den Methodenrumpf für die Getter-Methoden getLieblingsautomat(), getGiroKonto() und getSparKonto()** Keine Ergebnisse
6. **Implementiere den Inhalt der Methoden der Klassen Kunde, indem du Methoden deine Lieblingsautomats aufrufst.** Keine Ergebnisse
   Tipp: Direkte Abhebungen und Einzahlungen sind hierbei nur vom Girokonto möglich. Eine Interaktion mit dem Sparkonto ist nur per Überweisung möglich.



Achutng: Manche Testergebnisse werden erst angezeigt, wenn alle Voraussetzungenn erfüllt wurden!

Figure A.9 Exercise Statement of Exercise 08c (original, German)



Artemis 7.5.2 Kursübersicht Kursverwaltung ge42gif

Kurse > MTG: 10a (2023/24) > Aufgaben > Softwareprojekt - JumpNRun

Aufgaben

Softwareprojekt - JumpNRun Alle Themen Projekt

Abgabe bis: vor 2 Monaten

Punkte: 25 Bewertung: automatisch

No team yet Frist abgelaufen

Aktionen für Lehrende:

Teilaufgaben:

Video Tutorials zu Java: Studyflix

https://studyflix.de/informatik/thema/java-16

Grundsätzlicher Aufbau

Main

Die Ausführung beginnt immer in der Methode `Main.main()`. Diese ist als static markiert und du kannst sie mit einem Rechtsklick auf die Klasse `Main` starten.

In dieser Methode wird eine Gameboard (also unser Spielfeld) erstellt und diesem Entites (unsere Spielfiguren) hinzugefügt. Später werden wir hier auch die Tasten zur Steuerung festlegen.

Außerdem gibt es in `Main` die Methode `buildWorld(Gameboard gameboard)`. Diese ist zu Beginn leer und wird später genutzt werden, um eine Welt mit Wänden etc. zu bauen.

Bilder

Alle Bilder für den Hintergrund und die Entities sind im Ordner `resources` gespeichert, du findest hier bereits eine Auswahl an Bildern, kannst aber noch eigene ergänzen. Den jeweiligen Pfad, gibst du z.B. so an: `"resources/car.gif"`. In der Vorlage wird bereits ein Hintergrundbild geladen und ein Entity angezeigt.

Entities haben immer genau die Größe ihres Bildes (z.B. die Minecraft-Blöcke: 32x32 Pixel). Das Gameboard hat immer genau die Größe des Hintergrundbildes.

Entity

Entities sind unsere Spielfiguren. Ihre Position wird durch eine x- und eine y-Koordinate beschrieben. Wird ein Entity dem Gamebaord hinzugefügt, wird es automatisch an diese Position eingezeigt. Werden die Werte der Attribute mit den Koordinaten verändert, bewegt sich das Entity automatisch an die neuen Koordinaten.

Die Bewegung eines Entites wird über die x- und y-Komponente der Geschwindigkeit festgelegt. Hierbei bedeutet:

- eine positive x-Geschwindigkeit eine Bewegung nach rechts
- eine negative x-Geschwindigkeit eine Bewegung nach links
- eine positive y-Geschwindigkeit eine Bewegung nach unten
- eine negative y-Geschwindigkeit eine Bewegung nach oben

Gameboard / MyGameboard

Das Gameboard enthält das Koordinaten-System, in dem sich die Entities bewegen. Außerdem wird MyGameboard einen KeyboardListener enthalten, der die Tasten-Nummern, der gedrückten Tasten ausgeben kann.

Achtung: (0|0) ist links oben, die x-Werte nach rechts größer und die y-Werte nach unten!

Vorlage

- **Verändere niemals die Klassennamen der Vorlage** Keine Ergebnisse
- **Bereits bestehende Konstruktoren, Getter, Setter und Attribute dürfen nur mit expliziter Aufforderung geändert werden.** Keine Ergebnisse

Aufgaben: Pflichtfeatures

Das Entity bewegen. (1BE) Keine Ergebnisse
Jedes mal, wenn die Methode `run()` ausgeführt wird (das ist ca. 25x pro Sekunde), soll ein Entity um den Wert von `speedX` in x-Richtung und `speedY` in y-Richtung bewegt werden. Die Position des Entities wird hierbei durch die Attribute `x` und `y` festgelegt. Ob deine Code funktioniert, kannst du testen, indem du `speedX` und `speedY` auf beliebige Werte setzt (oder im Konstruktor die Methode `setRandomDirection(int maxSpeed)` aufrufst und das Spiel startest.

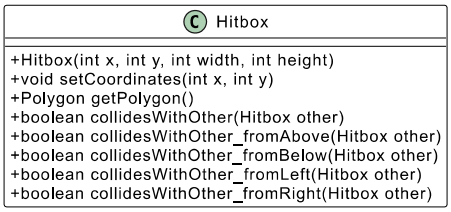
Hitbox erstellen (2BE) (Referenzattribut) Keine Ergebnisse
Um Kollissionen mit anderen Objekten erkennen zu können, benötigt jedes Entity eine Hitbox. Die Klasse Hitbox ist bereits fertig implementiert und kann Kollissionen mit anderen Objekten überprüfen (siehe Klassenkarte ohne private Elemente).

Weise dem Referenzattribut `hitbox` von `Entity` daher im Konstruktor ein neues Objekt von `Hitbox` mit den Koordinaten des Entities und der gleichen Breite (`width`) und Höhe (`height`) zu.

Die Koordinaten der Hitbox müssen immer gesetzt werden, wenn sich das Entity bewegt (also in der Methode `run()`)! Vergiss nicht, das zu implementieren!

Über Feedback Versionen Datenschutz Impressum

Figure A.10 Exercise Statement of Project Exercise [part 1] (original, German)



Verschiedene Arten von Entites (4BE, Vererbung) Keine Ergebnisse

Implementiere die Vererbung, wie im folgenden Klassendiagramm dargestellt. Die Klassen existieren bereits, passt aber noch nicht zu dieser Struktur, einige Methoden müssen auch von `Entity` zu `PlayerEntity` verschoben werden. Elemente, die bereits korrekt existieren, werden in grün dargestellt (die sagt nichts darüber aus, ob eine Methode das richtige tut!).

Kopiere den Inhalt, der bisher in `Entity.playerRun()` war, in `PlayerEntity.run()` und lösche `Entity.playerRun()` anschließend.

Achtung: Die Klassen `MyGameboard` und `Hitbox` werden ohne Methoden und Attribute angezeigt. Nimm an diesen Klassen in diesem Schritt keine Änderungen vor!

Dieses Pflichtfeature kann bisher nur teilweise überprüft werden!

Prüfen, ob Ränder der Welt erreicht wurden (Arrays)

In dieser Teilaufgabe wird bestimmt, ob ein Entity den Spielfeldrand erreicht. Bearbeite diese Aufgabe in der Klasse `PlayerEntity`.

Den ersten Schritt implementieren wir in der Methode `public boolean[] whichEdgesReached()`. Hierbei soll ein boolean-Array mit 4 Elementen zurückgegeben werden, dass true für ein erreichtes Ende der Welt enthält und false, wenn ein Ende nicht erreicht wurde.

Die 4 Werte im Array haben dabei folgende Bedeutung:

1. Wert: Linker Rand erreicht?
2. Wert: Oberer Rand erreicht?
3. Wert: Rechter Rand erreicht?
4. Wert: Unterer Rand erreicht?

Tipp: Die Breite und Höhe des Gameboards erhältst du mit `gb.getWidth()` bzw. `gb.getHeight()`. Die Breite und Höhe des Entities selbst entsprechend mit `img.getWidth(null)` bzw. `img.getHeight(null)`.

Hierbei gibt es 2 Schwierigkeitslevel (Schwierigkeit 1 ist Voraussetzung für Schwierigkeit 2):

1. **Ein Rand gilt als erreicht, wenn die Koordinaten des Entities den Rand erreicht haben. (2BE)** Keine Ergebnisse
2. **Ein Rand gilt als erreicht, wenn die Umrandung des Entities bzw. seiner Hitbox den Rand erreicht hat. (1BE)** Keine Ergebnisse

Verhalten, wenn Rand der Welt erreicht wurde (Arrays)

Diese Teilaufgabe arbeitet mit dem zurückgegebenen Array von der vorherigen Teilaufgabe. Sie wird in der Methode `public void edgeReached(boolean[] edges)` bearbeitet. Es gibt verschiedene Möglichkeiten, die für unterschiedliche Ränder unterschiedlich angewendet werden dürfen:

1. Das Entity taucht auf der gegenüberliegenden Seite wieder auf.
2. Das Spiel endet (`gb.stopGame("Gameover-Message")`).
3. Das Entity bleibt im Bild aber Spielfeld wird bewegt.
4. Das Entity wird abgestoßen:
   - Schwierigkeit 1: Die Bewegungsrichtung wird genau umgekehrt (also beide Geschwindigkeiten umgedreht).
   - Schwierigkeit 2: Die Bewegungsrichtung wird nur genau umgekehrt, wenn das Entity senkrecht auf den Rand stößt, andernfalls ändert sich die Bewegungsrichtung um genau 90-Grad Winkel abgestoßen, wo bei die Spitze des Winkel den Spielfeldrand berührt.

Wichtig: Aufgrund der vielfältigen Möglichkeiten, diese Teilaufgabe umzusetzen, und um eure Kreativität nicht unnötig einzuschränken, wird die Teilaufgabe nicht automatisch überprüft.

Implementiere den Keyboard Listener (4BE) (Arrays, Referenzattribute) Keine Ergebnisse

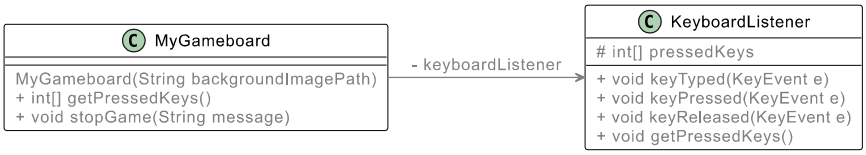
Figure A.11 Exercise Statement of Project Exercise [part 2] (original, German)

Um herauszufinden, welche Tasten gedrückt und losgelassen werden, programmieren wir die Klasse `KeyboardListener` und fügen ein Objekt davon als Referenzattribut zu `MyGameboard` hinzu. Halt dich dabei an unten stehenden Ausschnitt des Klassendiagramms. Die farbigen Markierungen zeigen dir auch direkt, ob deine Methode das richtige tut bzw. ob dem Referenzattribut zu Beginn ein richtiger Wert zugewiesen wird.

Der `KeyboardListener` ist event-gesteuert. Das bedeutet, dass die Methoden `keyTyped`, `keyPressed` und `keyReleased` automatisch aufgerufen werden, wenn ein Tastatur-Ereignis (=Event) auftritt, also eine Taste gedrückt oder losgelassen wird. Darum musst du dich also nicht kümmern, sondern nur sicherstellen, dass die Methode dann das richtige tut.

Hierbei wird jeweils ein Objekt der Klasse KeyEvent übergeben. Für uns ist dabei nur interessant, welche ID die gedrückte Taste hat. Diese bekommen wir mit der Methode `e.getKeyCode()`.

- `keyboardListener`: Soll bei Deklaration ein neues Objekt zugewiesen bekommen.
- `pressedKeys`: Enthält ein 20 Elemente langes int-Array mit allen aktuell gedrückten Tasten-IDs. Achte darauf, dass jede Tasten-ID nur einmal eingetragen wird. Eine leere Position soll mit dem Wert `-1` dargestellt werden. Zu Beginn ist das Array leer.
- `keyTyped(KeyEvent e)`: Diese Methode soll nichts tun. Verändere hieran nichts.
- `keyPressed(KeyEvent e)`: Die ID der gedrückten Taste soll an einer leeren (`-1`) Position in das Array `pressedKeys` eingefügt werden.
- `keyReleased(KeyEvent e)`: Die ID der gedrückten Taste soll aus dem Array `pressedKeys` entfernt werden (natürlich nur, wenn sie darin enthalten war) und die entsprechende Stelle mit `-1` ersetzt werden.
- `getPressedKeys()`: Standard-Getter für das Attribut `pressedKeys` (in KeyboardListener) bzw. in MyGameboard Weitergabe des Rückgabewert des KeyboardListeners.
- Konstruktor `MyGameboard`: zusätzlich zum Aufruf von super() soll mit Aufruf der Methode `addKeyListener(keyboardListener)` der Keyboard Listener am Computer 'angemeldet' werden, sodass die Events ausgelöst werden können.
- `stopGame(String message)`: Schon fertig :) Hält das Spiel an und zeigt die übergebene Nachricht an.



Tastatureingabe auf die Bewegung des PlayerEntity anwenden.

- `PlayerEntity.applyKeyboardControls`: Wird in der run()-Methode vor der Ausführung der Bewegung aufgerufen und bekommt als Parameter-Wert ein Array mit den aktuell gedrückten Tasten übergeben. Diese werden dann auf `speedX` und `speedY` angewendet. Welche Tasten-ID welche Bewegungsrichtung bedeutet, wird in den entsprechenden Attributen von `PlayerEntity` gespeichert.
  Schwierigkeit 1: Bei Drücken einer Bewegungsrichtung bewegt sich das Entity in die entsprechende Richtung. (2BE) Keine Ergebnisse
  Schwierigkeit 2: Bei Drücken einer Bewegungsrichtung wird das Entity in die entsprechende Richtung beschleunigt, bei Drücken der Sprung-Taste, wird die y-Geschwindigkeit auf einen Wert kleinergleich -10 gesetzt. (2BE) Keine Ergebnisse
  Für das Gamedesign sollte sichergestellt werden, dass das PlayerEntity nicht irgendwann unendlich schnell bewegt wird. Gleiches gilt für die Sprung-Taste. Hier kann man entweder überprüfen, ob das Entity auf dem Boden steht (`standingOnGround()`) oder eine 'Abkühlzeit' festgelegt werden, sodass z.B. nur bei allen 25 Durchläufen von `run()` gesprungen werden kann. Das Springen funktioniert nur in Kombination der Gravitation sinnvoll! **Diese Empfehlungen werden nicht automatisch überprüft.**

Stößt das PlayerEntity mit einem CollectableEntity zusammen, soll es dieses einsammeln (2BE, Referenzattrbiute, Arrays, Vererbung) Keine Ergebnisse
Hierbei wird der `score` des `PlayerEntity` erhöht und (optional) das `CollectableEntity` vom Spielfeld entfernt. Bearbeite diese Aufgabe in `PlayerEntity.collect()`.

Überprüfe hierzu, für alle Entites (außer `this`) im zurückgegebenen Array von `gb.getEntities()`, ob es ein `CollectableEntity` ist und ob die eigene Hitbox mit der des jeweils anderen Entities kollidiert.

Verwende auch hierfür wieder die Methoden der Hitbox.

Ein Entity, das keinen Boden unter den Füßen hat, soll herunterfallen (Referenzattribute, Arrays, Vererbung)

Voraussetzung für alle Schwierigkeiten: Überprüfen, ob Player auf einem Block steht (3BE) Keine Ergebnisse
1. Schwierigkeitsgrad: Das Entity soll mit konstanter Geschwindigkeit (beliebig schnell) nach unten (positive y-Richtung) fallen. (1BE) Keine Ergebnisse
2. Schwierigkeitsgrad: Das Entity soll konstant (beliebig stark) nach unten beschleunigt werden, solange es keinen Boden unter den Füßen hat. (1BE) Keine Ergebnisse

Kein Boden unter den Füßen bedeutet hierbei, dass es nicht von oben (die Hitbox hat hierfür eine passende Methode!) mit einem `BlockEntity` kollidiert.

Die Aufgabe beginnst du in der Methode `standingOnGround()` von `PlayerEntity` (zur Überprüfung, auf welchem BlockEntity der Spieler steht). Das geht sehr ähnlich wie in der vorherigen Aufgabe.

In der Methode `gravitation()` verwendest du `standingOnGround()` um zu überprüfen, ob dein PlayerEntity auf einem Block steht und lässt es ggf. herunterfallen oder hältst den Fall an (siehe oben).

Aufgabendetails

Tipp: Wenn du die Überprüfung des dynmaischen Typs so erweiterst, erhältst du eine Variable `other`, die Datentyp `BlockEntity` hat, deren Wert du bei Bedarf zurückgeben kannst. Falls der Spieler nicht von oben mit einem `BlockEntity` kollidiert, soll null zurückgegeben werden.

Veröffentlichungsdatum:	15. Apr. 2024 08:00
Abgabe bis:	1. Juli 2024 23:59
Besonders möglich:	Nein

```
if (e instanceof BlockEntity other) {
    ...
}
```

Juhu, fertig mit den Pflichtfeatures. Jetzt könnt ihr euch kreativ austoben. Ich freue mich auf eure Spiele!

Figure A.12 Exercise Statement of Project Exercise [part 3] (original, German)

A.3 Mistake List

Type	Mistake
array creation	Array creation: First element missing
array creation	Array creation: Index out of bounds
array creation	Array creation: Last element missing
array creation	Array creation: Wrong element
array creation	Array creation: Wrong object at index
array creation	Array with wrong length created
array index	Array access: Illegal index not caught
array index	Array access: Wrong elemented at shifted index exercise
array index	Array access: Wrong index for first element
array index	Array access: Wrong index for last element
array index	Array: Index out of bounds
array index	Array: Index out of bounds when searching last element
array iteration	Array count: Too high
array iteration	Array count: Too low
array iteration	Array interation: Wrong sum of elements
array iteration	Array iteration: Condition inverted
array iteration	Array iteration: Condition missing
array iteration	Array iteration: no action
array iteration	Array search: False negative
array iteration	Array search: False positive
array iteration	Move by keyboard
array modification	Array insert: Duplicate elements
array modification	Array insert: First position instead of first free position
array modification	Array insert: Not inserted
array modification	Array modification: Wrong element
array modification	Array modification: Wrong length
array modification	Array remove: Element not in array, but array modified
array modification	Array remove: Element not removed
attributesValue-getter-setter	Getter not behaving correctly
attributesValue-getter-setter	No value stored in attribute
attributesValue-getter-setter	Setter not behaving correctly
constructor problems	Attributes in constructor not set correctly
constructor problems	Init value stored to attribute at wrong point
constructor problems	Unknown constructor error
constructor problems	Unknown constructor error in child class
element from cd missing	Class diagram: Attribute missing
element from cd missing	Class diagram: Constructor missing
element from cd missing	Class diagram: Method missing
element from cd missing	Class diagram: Reference attribute missing
element from cd missing	Method from class diagram
element from cd missing	Missing inheritance
element from cd wrong	Class diagram: Attribute wrong
element from cd wrong	Class diagram: Constructor wrong
element from cd wrong	Class diagram: Method wrong
element from cd wrong	Class diagram: Reference attribute wrong
element from cd wrong	Inheritance: Move to subclass (by cd)
game semantics	Constant speed used instead of acceleration
game semantics	Movement on gameboard
game semantics	Position on gameboard
game semantics	Sync movement
general bug	Accidential recursion
general bug	Cast exception between primitive types
general bug	Cast exception due to inheritance
general bug	Method/Ctr call with wrong values
general bug	Nullpointer
general bug	Wrong String
reference attributes	Method of referenced Object not called
reference attributes	Method of referenced Object not called correctly
reference attributes	Value not stored in reference attribute
reference attributes	Wrong instanceof-check or wrong ref attribute assignment
Unwanted template modification	Class or package broken (correct in template)
Unwanted template modification	Template: Attribute modified
Unwanted template modification	Template: Method modified

Table A.1 List of Mistakes with Mistake Types

A.4 Results of Shapiro-Wilk-Tests for normal distribution of the EQ

Table A.2 Results of the Shapiro-Wilk-Test for normal distribution of EQ partitioned by exercise topic

Exercise Topic	Group A			Group B		
	Statistic	df	Sig.	Statistic	df	Sig.
ref. Attributes	0,941	29	0,106	0,911	32	0,012
inheritance	0,916	27	0.031	0,875	33	0,001
arrays	0,811	23	<0.001	0,954	32	0,183
Proj.	0,965	11	0,837	0,913	12	0,236

Color-Coding: $Sig. > 0.05$; $Sig. > 0.05$ for values of Group A and B

Table A.3 Results of the Shapiro-Wilk-Test for normal distribution of EQ partitioned by number in topic

Exercise Number	Group A			Group B		
	Statistic	df	Sig.	Statistic	df	Sig.
a	0.940	27	0.123	0.872	32	0.001
b	0.891	29	0.006	0.970	31	0.527
c	0.896	23	0.021	0.918	34	0.015
Proj.	0.965	11	0.837	0.913	12	0.236

Color-Coding: $Sig. > 0.05$; $Sig. > 0.05$ for values of Group A and B

Table A.4 Results of the Shapiro-Wilk-Test for normal distribution of EQ partitioned by exercise

Exercise	Group A			Group B		
	Statistic	df	Sig.	Statistic	df	Sig.
06a	0,958	10	0,760	0,867	11	0,071
06b	0,891	9	0,206	0,925	10	0,401
06c	0,826	10	0,030	0,789	11	0,007
07a	0,940	10	0,552	0,850	11	0,043
07b	0,857	10	0,070	0,927	10	0,419
07c	0,973	7	0,917	0,818	12	0,015
08a	0,888	7	0,264	0,815	10	0,022
08b	0,793	10	0,012	0,951	11	0,658
08c	0,813	6	0,076	0,944	10	0,597
Proj.	0,965	11	0,837	0,913	12	0,236

Color-Coding: $Sig. > 0.05$; $Sig. > 0.05$ for values of Group A and B

Table A.5 Results of the Shapiro-Wilk-Test for normal distribution of EQ partitioned by feedback type

Exercise	Group A			Group B		
	Statistic	df	Sig.	Statistic	df	Sig.
different	0,918	56	<0,001	0,928	63	0,001
uniform	0,942	34	0,069	0,918	46	0,003

Color-Coding: *Sig.* > 0.05; *Sig.* > 0.05 for values of Group A and B

Table A.6 Results of the Shapiro-Wilk-Test for normal distribution of EQ partitioned by feedback type and topic

Feedback Types	Exercise Topic	Group A			Group B		
		Statistic	df	Sig.	Statistic	df	Sig.
different	ref. Attributes	0,941	19	0,278	0,894	21	0,026
	inheritance	0,893	20	0,031	0,882	21	0,016
	arrays	0,829	17	0,005	0,931	21	0,145
uniform	ref. Attributes	0,826	10	0,030	0,789	11	0,007
	inheritance	0,973	7	0,917	0,818	12	0,015
	arrays	0,813	6	0,076	0,952	11	0,670
	project	0,965	11	0,837	0,913	12	0,236

Color-Coding: *Sig.* > 0.05; *Sig.* > 0.05 for values of Group A and B