Teaching Software Testing Using Automated Grading

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

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

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

1 Introduction

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

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

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

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

doi:10.18420/abp2021-3

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

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

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

2 Didactic Goals - What the students should learn

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

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

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

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

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

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

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

3 Task Design and Implementation

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

3.1 Design

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

We decided to split the exercise into two parts to prevent students from skipping reviewing the given code and then having more trouble with the test programming. The first part is essentially like a usual assignment, without any testing being by the students involved. Then, the second part is concerned with writing tests for the code of the first half. Thus, by the time the students reach the point where they have to write the tests, they are very much familiar with what their test code is interacting with—there is no code one understands better than one's own code. In PABS, the online grading system, tests are available, which examine the implementations of the first part-code, as well as the tests of the second part. A more detailed explanation of how this is achieved follows in 3.2. This way, the students can be sure that their first part-implementation is probably correct and can utilize it for local debugging when implementing the tests in the second half. We decided on this approach instead of a full TDD scenario to ensure that students are not faced with the challenge of having to write correct code and correct tests at the same time. Furthermore, the PABS test results are later used for grading.

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

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

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

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

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

3.2 Implementation

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

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

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

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

We developed a minimal-invasive solution, which allows switching the class under test while keeping the test programming for the students as authentic as possible. First off, the first part of the exercise was altered from the original exam. Originally, classes with specified names had to be implemented, and the exam tests were directly using the constructors to create objects. Now, an interface is given for each class. The students write classes that implement this interface and rewrite a static factory method in each interface to return an object of their implementation. This already allows the tests to handle different instances of the interfaces, however, the selection of the instance to test is still not possible.

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

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

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

4 Student Evaluation

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

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

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

5 Related Work

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

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

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

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

6 Conclusion

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

Bibliography

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

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

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

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

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

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

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

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

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