AuDoscore: Automatic Grading of Java or Scala Homework

Norbert Oster,[1] Marius Kamp,[1] and Michael Philippsen[1]

Abstract: Fully automated test-based grading is crucial to cope with large numbers of student homework code. *AuDoscore* extends JUnit and keeps the task of creating exercises and corresponding grading tests simple. Students have a set of public smoke tests available. Grading also uses additional secret tests that check the submission more intensely. *AuDoscore* ensures that submissions cannot call libraries if the lecturer explicitly forbids them. Grading is not susceptible to the problem of consecutive faults by partially replacing student code with cleanroom code provided by a lecturer. *AuDoscore* can be run as a stand-alone application or integrated into our *Exercise Submission Tool (EST)*. This paper briefly describes how both tools interact, depicts *AuDoscore* from the point of view of the lecturer, and describes some key technical aspects of its implementation.

Keywords: grading, student code submission, Java, Scala

1 Introduction

Up to 750 freshmen from about 23 degree programs take our annual course on “Algorithms and Data Structures” (AuD² for short) that also teaches them to program in Java. The course has a workload of 300 h (10 ECTS), 25% of which are devoted to writing Java programs of different sizes as graded weekly homework. In some semesters, the students submit up to 17.000 Java source code files. Up to 390 other students submit more than 600 Scala files as homework for our annual course on “Parallel and Functional Programming”, PFP.[2] As this is clearly too much code to be manually checked and graded, we apply an entirely automatic grading process for both Java and Scala. Our lightweight grading system *AuDoscore* works both stand-alone and as part of our “Exercise Submission Tool” (EST), our electronic student and exercise management platform.

Below we first describe how students use our system. Then we present the lecturer’s interface and share some technical details about *AuDoscore*. Finally, we outline our experience and discuss some related work.

2 How Students Experience AuDoscore

When submitting solutions for programming exercises, students do not directly interact with *AuDoscore*. Instead, they upload their source code files using our web-based lecture

[1] Friedrich-Alexander University Erlangen-Nürnberg (FAU), Programming Systems Group,
Martensstr. 3, 91058 Erlangen, Germany, email: [norbert.oster|marius.kamp|michael.philippsen]@fau.de
[2] https://www2.cs.fau.de/aud and https://www2.cs.fau.de/pfp

management system called “Exercise Submission Tool” (EST), which then passes their files to *AuDoscore*. *AuDoscore* is an extension of JUnit and basically uses two test sets: a public and a secret test set. The public tests are provided as a smoke test to ensure that the student solutions adhere to the expected interfaces. Both the public and the secret tests contribute to the grading, but the secret tests usually check the submission in more detail.

In conjunction with EST, the *AuDoscore* process comprises several stages: Students first submit their solutions (one or more Java or Scala source code files) via the web-based EST. Immediately after having uploaded a new submission, EST shows that work is in progress (⌛) and forwards the source files to *AuDoscore*, which then performs the following four stages before providing the respective feedback to the student.

Stage 0: The source code is *compiled* together with additional interfaces or helper classes provided to the students, but without any test cases yet. If this stage fails, running and grading the submission is impossible. Therefore, the student is immediately shown a skull (☠) and warned that this submission is worth 0 points.

Stage 1 checks if the submission *compiles* against the public test set and does *not use forbidden* API functionality (see Section 4 for details). If it fails at least one check, the student sees an ✘ symbol, indicating that the submission is worth 0 points.

Stage 2 *executes* the public test set. An exclamation mark (❗) indicates that the submission fails on at least one public test case. Otherwise a ✓ concludes the preliminary feedback.

Stage 3 is the real *grading* step that executes the submission with both the public and secret test sets. The latter tests are run against the original student source code as well as code modified with replacements in order to reduce subsequent faults (see Section 4 for details). Students can re-submit modified and improved versions as often as they like before the end of the submission deadline. A lecturer has to confirm the grading result before students can see their test results and the graded points on their EST web-page.

The feedback in stages 0–2 and the option to re-submit mimics a continuous integration. It provides (partial) feedback as early as possible and even before the deadline, enabling the students to continuously revise and learn. However, as *AuDoscore* must run all the above stages upon each individual submission, the system load on the computers running *AuDoscore* is high. We therefore use several distributed instances of DOMjudge.[3]

3 Lecturer’s User Interface

From the point of view of a lecturer, *AuDoscore* is just a set of additional annotations on top of the JUnit framework. In order to apply automatic grading on a practical exercise, the lecturer has to write a cleanroom solution and two JUnit test classes. For activation (Listing 1),

[3] https://www.domjudge.org/

both the public and the secret test classes must declare a `@Rule` that tells *AuDoscore* how to collect the result of each individual test case (test method). The mandatory `@ClassRule` creates a summary of the grading results and the feedback to the student.

List. 1: Rules used to activate *AuDoscore*.

```
@Rule
public final PointsLogger pl = new PointsLogger();
@ClassRule
public final static PointsSummary ps = new PointsSummary();
```

A secret test class is simply marked with a `@SecretClass` annotation. The public test class requires an `@Exercises` annotation, which holds a nested array of `@Ex` annotations, i.e., a tuple of an arbitrary but unique name and its grading points. Listing 2 shows an example.

List. 2: Declaration of tasks and grading points.

```
@Exercises({@Ex(exID = "T1a", points = 5), @Ex(exID = "T1b", points = 7),
            @Ex(exID = "T2",  points = 11) })
```

Each test case (test method) must be annotated with the JUnit annotation `@Test` and an annotation `@Points` with `bonus` or `malus` values. To ensure that faults in student submissions do not exceed given runtime limits (e.g., by means of infinite loops or recursion), all `@Tests` must have a positive `timeout` (in milliseconds).

List. 3: Declaration of grading test cases.

```
@Points(exID = "T2", bonus = 8) @Test(timeout = 500)
public void test1() { ... }
@Points(exID = "T2", malus = 6) @Test(timeout = 1000)
public void test2() { ... }
```

The `bonus` attribute in Listing 3 specifies that a student gains the given number of points if the corresponding test succeeds. A failing bonus test does not yield any points. In contrast, a successful test case annotated with a `malus` attribute has no impact on the final score whereas a failing malus test reduces the number of points by the specified amount. Thus, malus tests allow a lecturer to cancel the effects of a passed bonus test. We outline the relevance of malus tests in Section 5 where we describe how we use such tests to detect possible cheating.

To facilitate the addition or removal of test cases, the sum of the `bonus` values does not have to match the maximal number of points specified in the `@Ex` annotation. Instead, *AuDoscore* computes a relative score for a task τ as follows:

$$grade_\tau = \left\lfloor \max\left(0, \frac{\sum_{TC_\tau}^{passed} |bonus| - \sum_{TC_\tau}^{failed} |malus|}{\sum_{TC_\tau} |bonus|}\right) \cdot |Ex_\tau.points| \right\rfloor_{(0.5)} \tag{0.5}$$

Example: Let us assume we have five test cases for τ = "T2": four with a *bonus* = *4, 8, 10, 12*, resp., and one with a *malus* = *6*. As declared in Listing 2, T2 is worth a total of 11 points. If a submission fails on the first ($b = 4$) and the last ($m = 6$) test cases, then it is graded $\left\lfloor \max\left(0, \frac{(8+10+12)-(6)}{4+8+10+12}\right) \cdot 11 \right\rfloor_{(0.5)} = 7.5$ points. If a lecturer adds another test case without changing the total number of points, (s)he does not have to adjust the existing tests because the *bonus* and *malus* values are normalized wrt. the total number of points.

In order to learn and practice basic algorithms and data structures, students are typically requested to implement for instance sorting procedures or fundamental data structures like linked lists or heaps. As the Java API comes with several pre-defined classes and methods that would undermine the intention of the homework, lecturers can limit the parts of the Java/Scala API that students may use. To do so, lecturers use the `@Forbidden` and `@NotForbidden` annotations. Both accept an array of strings, each one denoting a (possibly wild-carded part of the) fully qualified name of packages, classes, methods, or fields. The example in Listing 4 is taken from a homework in which students had to implement a hash table. The annotations disallow the use of any class from `java.util` and restrict the usable methods from `java.lang.Math` to `round` and `pow` only. If a student uses at least one of the forbidden API features, *AuDoscore* issues a notification in **Stage 1** (see Section 2).

List. 4: Restriction of available Java API functionality.

```
@Forbidden({"java.util", "java.lang.Math"})
@NotForbidden({ "java.lang.Math.round", "java.lang.Math.pow" })
```

Good object-oriented programmers break down the functionality of their applications into several softly coupled but highly cohesive methods. In order to train this skill, lecturers can provide interfaces of the classes to be implemented, declaring and describing the different methods and their cooperation. Consider an exercise requiring a `QuickSort` implementation with the methods `sort`, `partition`, `choosePivot`, and `swap`. If a student implemented a faulty `swap` and calls this method from a correct `partition`, individual grading test cases for both `swap` and `partition` fail due to the common cause. In order to cope with those cascading consecutive failures without punishing students twice, *AuDoscore* provides the `@Replace` annotation as shown in Listing 5 for the `QuickSort` example.

List. 5: Replacement of student code with cleanroom code.

```
@Replace({ "QuickSort.swap", "QuickSort.choosePivot" })
```

Test cases annotated this way are first executed with the original code as submitted by the student. Then, these tests are run again, but beforehand, the methods declared in the `@Replace`ment are exchanged with the cleanroom solution implemented by the lecturer (see Section 4 for technical details). The final grading is computed as the maximal points of both test runs—this way, incompatible implementations of the cleanroom codes never lower the results that the unmodified submission achieves. In the example above, a test case for `partition` annotated with a `@Replace` as shown in Listing 5 no longer fails due to a faulty

implementation of `swap`. In contrast, a faulty `partition` method still causes a failure of this test case.

4 Technical Design of AuDoscore

This section explains some technical details of the implementation available on GitHub.[4] One of the core components of our grading system is the set of annotations sketched in Section 3. Since those annotations are merely meant to provide information at runtime, the main functionality is implemented in the `PointsLogger` resp. `PointsSummary` classes. `PointsLogger` subclasses the JUnit API class `TestWatcher` and is used to intercept the test set execution. This way, JUnit notifies *AuDoscore* about skipping resp. starting and completing individual test cases—including the verdict of each test run. For each test execution, `PointsLogger` extracts the task name and test case weight from the `@Points` annotation and stores them together with the verdict and execution time in a `ReportEntry`. `PointsSummary` extends the JUnit class `ExternalResource` and gives *AuDoscore* control over the test execution. Upon startup, this class installs a custom `SecurityManager` that prevents students from interfering with *AuDoscore* and from injecting unwanted code that, e.g., opens arbitrary files, executes processes, or stops the VM. `PointsSummary` also replaces `System.out` and `System.err` in order to prevent flooding the grading output and to exclude the runtime of student debugging output from the runtime measurements. Additionally, it validates the annotations of the public and secret test classes and methods (e.g., to enforce `timeouts` and `@Points` for all test cases), collects the information from the `@Exercises` annotation and prepares for report acquisition. After having executed all tests, JUnit notifies `PointsSummary` to then generate reports for both the student and the lecturer.

The technically most sophisticated component of *AuDoscore* is the `@Replace`ment support. The class `ReplaceMixer` exchanges individual methods in the student submission with their cleanroom counterparts. To achieve this goal, we use the compiler functionality built into the JDK to manipulate the abstract syntax tree (AST) of the student source code. Thus, the `@Replace`ment mechanism is the only component that needs modification when porting *AuDoscore* to another JVM language.

We steer the features `@Forbidden`, `@NotForbidden`, and `@Replace` from outside Java using simple Linux shell scripting. This script runs the stages described in Section 2 and checks the restrictions of `@Forbidden` and `@NotForbidden` by searching the output of the `javap` disassembler for matches of the forbidden prefixes. This way, lecturers may also forbid individual bytecode instructions. In **Stage 3**, the script runs all test sets (public and secret) with and without all required `@Replace`ments and collects the results. A similar shell script applies *AuDoscore* in a local stand-alone environment without EST integration, e.g., during the development of the cleanroom and test sets or when lecturers resp. teaching assistants need to inspect the results of individual student code in more depth.

[4] https://github.com/FAU-Inf2/AuDoscore

Limitations. *AuDoscore* still has some limitations that we plan to tackle in the future. First, it considers only test runs for grading and does not yet take into account coding style or other non-functional aspects of the submitted code. Second, *AuDoscore* only works for programs written in Java or Scala, although it may be easily extended to any JVM-based language. Third, *AuDoscore* lacks specialized features to test parallel code. Fourth, *AuDoscore* provides no means to integrate external libraries. Fifth, *AuDoscore* cannot be configured to search for `@Forbidden` code only in parts of the submissions (e.g., individual methods or specific code lines).

5 Experience

The statistics in Table 1 summarize the productive *AuDoscore* use for the AuD and PFP courses from winter term 2014/15 to 2016/17.

Table 1: Statistics on 5 semesters of *AuDoscore* use.

	Winter/Summer Term	WT2014/15	ST2015	WT2015/16	ST2016	WT2016/17
AuD	# registered students	668	198	716	172	733
	# programming tasks	37	30	38	32	34
	# Java submissions	10478	1263	12526	856	11189
	# submitted Java files	12685	1647	17274	1167	13846
	# LOC (total)	1230394	94069	1528133	80859	1215662
	# public/secret test cases	160/562	147/737	336/810	143/635	224/804
	# post-deadline revisions	26	24	24	27	21
	# manually changed grades	102	9	723	2	13
PFP	# registered students	–	–	24	390	51
	# programming tasks	–	–	12	7	7
	# Scala submissions	–	–	1	699	21
	# submitted Scala files	–	–	1	699	21
	# LOC (total)	–	–	14	18144	487
	# public/secret test cases	–	–	24/21	15/54	15/54
	# post-deadline revisions	–	–	0	6	0
	# manually changed grades	–	–	0	76	0

AuDoscore worked reliably and used the given number of test cases to grade the submitted homework. Only in rare cases we had to revise some secret tests after the deadline, as the row named "post-deadline revisions" shows. Due to the large diversity of submissions, we sometimes did not anticipate a certain way of misbehavior. We also had mistakes where the homework compiled with the public test but not with the secret test. The latter typically happened when the public test did not exercise the interface of the student submission to the same extent as the secret test. Since the preliminary feedback shown to the student ends with **Stage 2**, mistakes of this kind are immediately shown as an *internal error* to the lecturer only. This way, lecturers can revise the test cases early and even during the submission period. Table 1 also shows that in general the students considered the grading to

be fair. Only rarely they could convince a teaching assistant or lecturer to manually change grades. Other manual interventions were necessary to cope with an *AuDoscore* limitation, e.g., when something was `@Forbidden` in a certain segment of the students' code only.

Advice on Test Case Construction. When students are asked to explicitly write a recursive solution, the annotations of Section 3 provide no straightforward way to test whether they really did so. Hence, task specifications should require that students call a specific method in the base cases of their recursion. *AuDoscore* can then inspect the stack trace during test execution to grade recursive programs. Since this feature was missing, we had to manually check and down-grade an unusual number of AuD submissions in WT2015/16.

Mimicking continuous integration also opens up the door for a vector of attacks. Instead of implementing the expected algorithm, we have seen submissions that just contain a cascade of `if`-cases, each of which returning exactly the expected value visibly declared in the public test. To detect such submissions, the lecturer can use anti-cheat tests. For example, such tests may call the method to be graded with many different arguments. From the diversity of the results one can judge whether the implementation is (almost) univariate and thus most probably cheating. These anti-cheat tests may be implemented as secret malus tests to (partially) cancel the points granted by the public tests.

6 Related Work

There already exist several systems for automatically grading programming assignments. According to recent literature reviews [CAR13, Ih10], many of them serve a special purpose (e.g., are developed solely for a specific lecture) whereas *AuDoscore* and only some other tools are geared towards widespread adoption.

The methods for automatically grading programming assignments differ regarding how students use them. Systems like Marmoset [Sp06] require that students install custom tools on their workstations to submit solutions. Others (e.g., PABS [If15]) work in concert with a version control system. In contrast, the system by Amelung et al. [AFR08] or *AuDoscore* are purely web-based and thus allow students to work with any environment.

Like *AuDoscore*, many systems rely on testing to evaluate submissions [KJH16]. Some tools (e.g., ProgTest [dSMB11] and Web-CAT [EP08]) even use test cases written by students during the grading process. *AuDoscore* currently only supports test sets written by the lecturer. AutoLEP [Wa11], among others, combines testing with static analysis to determine a grade. We plan to investigate how an external static analysis tool can easily be integrated into *AuDoscore* using a lightweight set of Java annotations.

Beside testing, JACK [GSB08] uses graph transformation rules on a graph generated from the submitted source code. This allows a lecturer to specify checks that a valid solution

must pass. This approach is more powerful than the annotations provided by *AuDoscore*. However, we argue that annotations are easier to write than graph transformation rules.

Graja [Ga15] extends JUnit by providing helper classes for common grading tasks. It also uses annotations to improve the feedback shown to the students. In contrast, *AuDoscore* is designed to smoothly integrate into usual JUnit testing by introducing lightweight annotations that configure the grading system.

The recent trend to teach mobile application development in introductory courses has led to an emergence of tools like RoboLIFT [AE12] or the system by Heimann et al. [He15] for the automated assessment of Android applications. Furthermore, there are specialized solutions for assessing concurrent exercises [OB07]. Some systems, like GATE [MS13], support randomized exercises. Praktomat [KSZ02] also supports multiple variants of an exercise and additionally allows mutual feedback among students. Currently, *AuDoscore* lacks these advanced features.

7 Conclusion

This paper presents *AuDoscore*, a system for the fully automated test-based grading of student homework code submissions. We described the feedback that *AuDoscore* presents to students and how a lecturer can quickly enhance JUnit tests to use them for automatic assessment. This notably reduces the time required to grade the submissions. *AuDoscore* also provides features like forbidden API calls and code replacements to avoid the problem of consecutive faults. Our experience shows that AuDoscore scales well even for courses taken by 750 students. However, our approach seems to allure students to try to outsmart the system. A lecturer has to take this into account when designing tests.

Acknowledgments

We thank all contributors to *AuDoscore*, in particular Tobias Werth.

References

[AE12] Allevato, Anthony; Edwards, Stephen H.: RoboLIFT: Engaging CS2 Students with Testable, Automatically Evaluated Android Applications. In: SIGCSE'12: Technical Symp. Computer Science Education. Raleigh, NC, pp. 547–552, February–March 2012.

[AFR08] Amelung, Mario; Forbrig, Peter; Rösner, Dietmar F.: Towards Generic and Flexible Web Services for E-Assessment. In: ITiCSE'08: Conf. Innovation and Technology in Computer Science Education. Madrid, Spain, June–July 2008.

[CAR13] Caiza, Julio C.; Álamo Ramiro, José M. del: Programming Assignments Automatic Grading: Review of Tools and Implementations. In: INTED'13: Intl. Technology, Education and Development Conf. Valencia, Spain, pp. 5691–5700, March 2013.

[dSMB11] de Souza, Draylson M.; Maldonado, José C.; Barbosa, Ellen F.: ProgTest: An Environment for the Submission and Evaluation of Programming Assignments based on Testing Activities. In: CSEE&T'11: Conf. Softw. Eng. Education and Training. Waikiki, Honolulu, HI, pp. 1–10, May 2011.

[EP08] Edwards, Stephen H.; Pérez-Quiñones, Manuel A.: Web-CAT: Automatically Grading Programming Assignments. In: ITiCSE'08: Conf. Innovation and Technology in Computer Science Education. Madrid, Spain, p. 328, June–July 2008.

[Ga15] Garmann, Robert: E-Assessment mit Graja – ein Vergleich zu Anforderungen an Softwaretestwerkzeuge. In: ABP'15: Automatische Bewertung von Programmieraufgaben. Wolfenbüttel, Germany, November 2015.

[GSB08] Goedicke, Michael; Striewe, Michael; Balz, Moritz: , Computer Aided Assessment and Programming Exercises with JACK. ICB-Research Report No 28, 2008.

[He15] Heimann, Mathis; Fries, Patrick; Herres, Britta; Oechsle, Rainer; Schmal, Christian: Automatische Bewertung von Android-Apps. In: ABP'15: Automatische Bewertung von Programmieraufgaben. Wolfenbüttel, Germany, November 2015.

[If15] Iffländer, Lukas; Dallmann, Alexander; Beck, Philip-Daniel; Ifland, Marianus: PABS - a Programming Assignment Feedback System. In: ABP'15: Automatische Bewertung von Programmieraufgaben. Wolfenbüttel, Germany, November 2015.

[Ih10] Ihantola, Petri; Ahoniemi, Tuukka; Karavirta, Ville; Seppälä, Otto: Review of Recent Systems for Automatic Assessment of Programming Assignments. In: Koli Calling'10: Koli Calling Intl. Conf. Computing Education Research. Koli, Finland, pp. 86–93, October 2010.

[KJH16] Keuning, Hieke; Jeuring, Johan; Heeren, Bastiaan: Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In: ITiCSE'16: Conf. Innovation and Technology in Computer Science Education. Arequipa, Peru, pp. 41–46, July 2016.

[KSZ02] Krinke, Jens; Störzer, Maximilian; Zeller, Andreas: Web-basierte Programmierpraktika mit Praktomat. Softwaretechnik-Trends, 22(3), 2002.

[MS13] Müller, Oliver; Strickroth, Sven: GATE - Ein System zur Verbesserung der Programmierausbildung und zur Unterstützung von Tutoren. In: ABP'13: Automatische Bewertung von Programmieraufgaben. Hannover, Germany, October 2013.

[OB07] Oechsle, Rainer; Barzen, Kay: Checking Automatically the Output of Concurrent Threads. In: ITiCSE'07: Conf. Innovation and Technology in Computer Science Education. Dundee, Scotland, pp. 43–47, June 2007.

[Sp06] Spacco, Jaime; Hovemeyer, David; Pugh, William; Emad, Fawzi; Hollingsworth, Jeffrey K.; Padua-Perez, Nelson: Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In: ITiCSE'06: Conf. Innovation and Technology in Computer Science Education. Bologna, Italy, pp. 13–17, June 2006.

[Wa11] Wang, Tiantian; Su, Xiaohong; Ma, Peijun; Wang, Yuying; Wang, Kuanquan: Ability-training-oriented automated assessment in introductory programming course. Computers & Education, 56(1):220–226, 2011.