STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance

Elisa Baniassad
ebani@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada

Lucas Zamprogno
lucasaz@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada

Braxton Hall
braxtonh@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada

Reid Holmes
rtholmes@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada

ABSTRACT

Autograders are an invaluable tool for deploying assessments in large classes. However students sometimes rely on the autograder in place of careful thought for ways to improve to their solution. We sought to naturally encourage students to check their own solutions more, and hammer the grader less. To do this, we imposed a penalty each time a student's grade went down: we called these regression penalties. We assessed whether the introduction of these penalties resulted in less reliance on the autograder without hurting student performance.

Encouragingly, the number of autograder submissions was reduced by roughly half while only slightly decreasing the median final grade. Students reported feeling nervous about their submissions, but noted that they checked their own solutions by testing their code far more than they would have without the penalty. Students also expressed positivity about the regression model.

CCS CONCEPTS

• **Social and professional topics** → **Computing education** ; **Computer science education**.

KEYWORDS

assessment, software engineering, autograder

ACM Reference Format:
Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *The 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432430

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432430

1 INTRODUCTION

Autograders are an invaluable tool for deploying assessments in large classes. They allow us to pose challenging problems that would be intractable to grade manually. We rely heavily on autograders as educators at a large, publicly funded institution, with a typical class size of roughly 320, split over multiple sections.

However, educators who have used autograders in their courses know that students use autograders in place of their own careful reflection. Some autograders can be configured to pose artificial barriers such as wait times between submissions, or limits on the number of times they can submit (e.g., [5, 14]) as a means to encourage self-reliance. But even with these baffles, the students tweak and submit as often as they can and predominantly rely on autograder feedback to direct their work. Ultimately, *they are learning the grader, not learning the concepts.* [1]

We characterize scores moving up and down as *turbulence*, and we saw significant turbulence in our students' grades; we observed 8.5% of submissions induced regressions for our 3rd year software engineering course in Fall 2019 (326 students, 163 pairs). Nearly half of teams regressed once, and one in five regressed twice or more. This turbulence is visible in Figure 1 where each line depicts a team's grade across submissions. Grades go up, but they also go down.

One of the goals of most general software engineering classes is to teach students the value of testing and also to instill them with a sense of software hygiene, including the proper use of development and production branches. Commits made to development branches can fail tests: the code can get worse before it gets better, and the branch can serve as a sandbox for developers to tweak and refine their projects. Production branches, however, are the code that is used by clients. This code should not get worse and better – it should only improve. Production branches should not have failing tests, and changes must be well thought-out and validated. Examining how students were using our autograder, we realised they were using their production branch and the autograder in

[1] In a conversation entitled **The things you learn the hard way - beware return scores instantly** on the forum for the grading tool *Formative*, an educator laments: *"Today, I witnessed a student simply clicking until green without even reading the question!"*. The suggested solutions by the tool developer was limiting the number of tries, or hiding scores until final submission. [10]

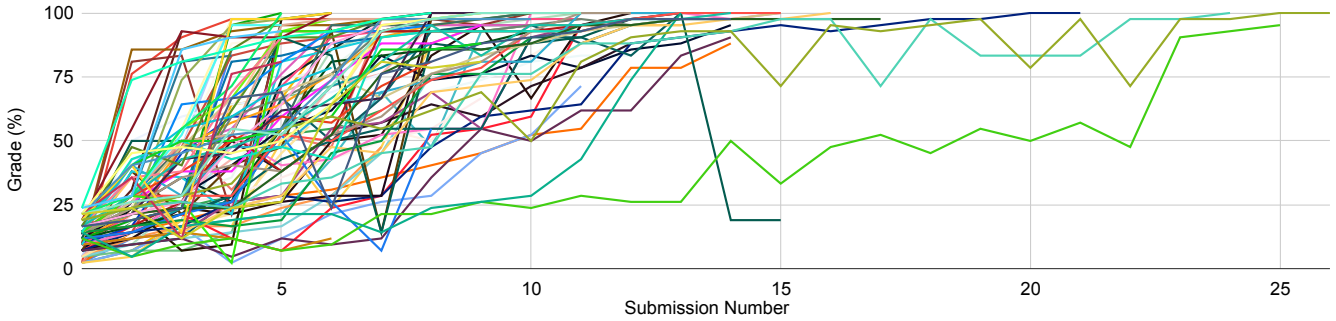

Figure 1: Student submissions without regression penalties. 8.5% submissions caused a regression, showing considerable turbulence in grade changes. Each subsequent submission by a team adds an additional point along the X-axis.

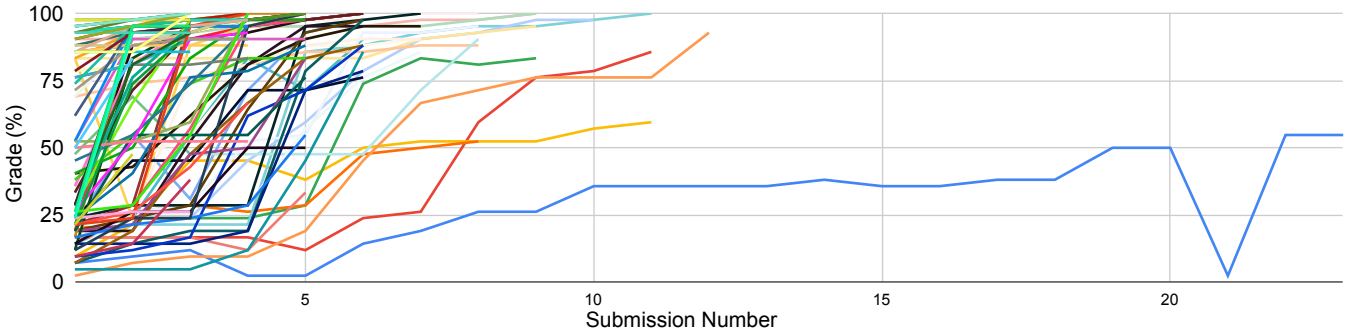

Figure 2: Student submissions with regression penalties. Autograder submissions decreased by 48%. 2% of submissions caused a regression, representing a significant drop in turbulence as compared to the prior semester.

place of their own tests: essentially, they were treating it like a development environment to gain feedback from the autograder for direction.

Although we have encouraged testing with test coverage grades, test coverage often lagged behind grades, suggesting that students were achieving good coverage *after* submitting code, as opposed to before, and were relying on the autograder to tell them if their project had improved, not relying on their own testing.

To reduce over-reliance on the autograder, we imposed a penalty each time a student's grade went down: we called these **regression penalties**. Regression penalties are designed to organically inspire students to write their own tests. A regression penalty is a permanent penalty on a student's grade, corresponding to the size of the regression. In our case, we set the penalty at half the value of their regression as a permanent penalty on their final grade. For example: if their grade regressed from 89 down to 87 in a subsequent submission, their final grade was now capped at 99 (2 percentage points regression becomes a 1 point final penalty).

To pair regression penalties with a sense of natural, real-world consequence, we wove the new grading scheme into a stronger Software Engineering narrative:

> *When you submit to the autograder, you are releasing to your client. You cannot remove functionality from the client. You must have confidence that when you push to production, you will be improving the client's life, not making it worse.*

Research Question

Can regression penalties reduce student reliance on the autograder as tester? And if so, to what extent?

2 RELATED WORK

There is extensive work and associated literature on the provision of automatic grading for programming assignments (e.g., [2, 5, 12, 14]). Techniques involve running test cases against submissions and using success combined potentially with other factors such as test coverage [5, 12], style checking [11], or structural checking [4]. Other approaches involve assessing students tests themselves against reference implementations, each with their own mutations, to identify how many of the mutants student tests identify [13]. Another allows educators to validate the fairness of their assessment framework through mutation testing [3].

Prior work has found that there are benefits moving from single submissions to allowing resubmissions for formative feedback. However, students can become reliant on the automated feedback over their own tests [1], or resubmit rapidly without taking the time to think about their submission [8]. In an attempt to address these downsides, some automated assessment tools employ features such as limiting submission number or frequency, limiting feedback, and randomizing exercises [7]. Web-CAT is a representative example of such a system, in which test cases are used to establish a grade for a

submission, and a maximum number of submissions could be set [5]. Other tools employ industry-standard tools for assessment [6].

We are only currently aware of one approach that used over-use of a grader as a motivation for their technique: The AutoGrader system by Liu et al. [9]. Their approach uses semantic analysis of programming submissions to determine a program's correctness, and synthesise counter examples to assess students' adherence to a correct solution. They cite the risk of autograder overuse as a motivation for this style of analysis, synthesis, feedback, and assessments, suggesting that with this approach, tinkering is not as likely.

These techniques are complementary with our approach and results. Our regression-penalties approach can be applied as an addition to any autograding system that allows multiple submissions.

3 METHODOLOGY

To answer our research question, we applied our approach to one instance of a third year software engineering course and used the prior instance of the same course as a point of comparison.

3.1 Measures

We tracked several per-team quantitative indicators of success for both the Fall 2019 semester (prior to regressions) and the Winter 2020 semester (with regressions):

- The number of submissions. A submission reduction could point to a reduced reliance on the autograder.
- The grades at each submission. Grades that steadily increased might indicate that students are taking increased care.
- The average grade across all submissions. An improved average grade across all submissions might indicate that students are submitting higher quality code to the autograder.
- The final grade. It was not our wish to hurt students' grades – we only wanted to reduce using the autograder as a crutch. If average final grades were similar this would suggest that students obtained the same level of learning.
- The number of times grades fell. By comparing the two semesters we would be able to see if teams' grades fell fewer times, suggesting that they were being more careful in their submissions, and were tweaking their submissions and using the autograder as a crutch, less than without penalties.
- The final penalty amount (for Fall 2019 this is the penalty that would have been incurred, had regressions been in place). By comparing the two semesters, we would be able to see if students naturally incurred fewer penalty points, allowing us to compare the magnitude of reduction in code-tweaking.

At the end of the Winter 2020 semester we polled all students asking: *Did having regression penalties change the way you did the project, and if so, how?* We designed this question in this way to mitigate response bias: had we asked a leading question such as *Did you think regression penalties helped you learn?* or *Do you think regression penalties made you test more?*, we would have run the risk of biased results for which we could not correct. We also did not want to give students a specific checklist of things to consider, for the same reason. Instead, we asked this open ended question, and then coded the responses for specific concepts.

We coded responses for:

- Specific mentions of increased personal testing.
- Specific mentions of stress (positive or negative).
- Specific mentions of perceived adverse effects on learning, process, or grades.
- Specific mentions of attaining the learning outcome of only pushing well-tested code to production.
- General statements related to increased caution when pushing to production.

3.2 Course demographics

We evaluated our approach on the Winter 2020 (January–April) instance of our third year software engineering course; this instance had 274 students who performed the project in pairs. We then compared our results to the Fall 2019 (September–December) instance of the same course; this instance had 326 students who performed the project in pairs.

The student demographics were equivalent between the two instances. The project deliverable specifications were identical, students had the same amount of time to complete the work. The student-to-TA ratio was the same across both semesters. The student backgrounds were similar since all students must follow a preordained sequence of courses. The same ratio of students (roughly an 8th of students) had co-op work experience between the two semesters.

3.3 Baseline Course Design

The Fall 2019 course instance served as the baseline for our evaluation, in which there were 163 pairs of students. We focus on the first (of three) major deliverables for the project in this course. The subsequent deliverables changed slightly between semesters, which meant that student behaviour would be less comparable across semesters, while the first major deliverable was stable. This deliverable had 42 unit tests that evaluated the student submissions. Final student solutions for this deliverable averaged 1056 source lines of code in the latest instance of the course.

Students were allowed to check their grade every 12 hours, but could submit their code any time. The grading rule was that we would take the maximum grade from all submissions, regardless of whether they requested a grade or not. Due to a limitation in our setup, we are not able to distinguish between times when students requested a grade versus when they were committing incremental work. However, students were aware that they were being graded in the background on every submission (e.g., on every `git push`), and knew there there was no harm in committing since we always took their maximum grade. Additionally, the grader returned linting errors on the most recent commit from each submission, so submitting even without requesting a grade, did have grade-relevant value to the students. Hence, in our comparison to the Winter 2020 semester we believe it is a fair comparison to count all commits as grader submissions.

Since a portion of the grade was allocated for test coverage, some submissions improved coverage, but did not affect core functionality. To isolate the submissions that were core functionality related we culled the submissions that improved only test coverage, so that we would be able to compare similar submission types across the two

semesters. We then considered only the core functionality scores when comparing to the Winter 2020 semester.

Students were able to submit their code to their repository any time, as this was the way they shared code within the team. Every submission triggered a background grading process (on the most recent commit in the submission) which gave them basic information about their submission, such as whether the submission would build on the test infrastructure, whether it timed out when run, and whether it passed the code style check. It did not report on the core functionality. For that, students had to request a grade from the grader by placing a comment on their commit in GitHub.

3.4 Semester With Regression Penalties

In Winter of 2020, we introduced regression penalties in a section with 137 pairs of students. The regression penalties represented the only meaningful change between the two course instances we evaluated. We made other minor changes to facilitate the introduction of the penalties:

- Because students were new to the penalty approach, we softened the regression rule slightly by saying that their worst regression would be forgiven.
- We removed all formerly applied artificial baffles for them using the autograder. Specifically, we removed the 12 hour wait time between autograder submissions along with the portion of their grade that corresponded to test coverage.
- We developed a set of smoke tests so that students could deploy their development code on the production architecture - this did not give them access to the production-level (client-level) tests. This mimics how a real development environment would work: it would be the same architecture as the production environment, or as close as possible to it, to allow thorough testing but were only sufficient to indicate whether their code would run on the production environment.

4 RESULTS

In this section we describe the quantitative and qualitative impacts of the implementation of regression penalties on autograder usage.

	Fall'19 n=163	Winter'20 n=137	% Change
Regression penalties applied	X	✓	—
Teams regressing >= 1 times	46%	6.5%	-39.5%
Teams regressing >= 2 times	18%	0.7%	-17.3%
Average calls to grader	8.96	4.62	-48.4%
Submissions that regressed	8.50%	2.00%	-78.6%
Average # of regressions	0.76	0.08	-89.5%
Average penalty points	8.48%	1.45%	-82.9%
Average assignment grade	92%	82%	-10.9%
Median assignment grade	97%	96%	-1.0%

Table 1: Quantitative regression results.

4.1 Quantitative Comparisons

The broad quantitative differences between the course instances can be found in Table 1. In the Fall 2019 instance, without regression penalties students, invoked the autograder an average 9.0 times. 46% of teams regressed at least once, and the average regression penalty would have been 8.5%, were a penalty in place. The median grade for the deliverable was 97%.

In the Winter 2020 instance, while subject to regression penalties, students invoked the autograder an average 4.6 times. 6.5% of teams regressed at least once, and the average regression penalty was 1.5%, which represents a significant drop in turbulence. The median grade for the deliverable was 96%.

The largest impact of the regression penalty seemed to be on the average number of regressions per team, which fell by almost 90%. This drop can be linked to the percent of teams regressing at least once: In the Fall 2019 semester, nearly half the teams regressed, whereas with regression penalties, this fell to 6.5%. The number of submissions to the grader also fell by nearly 50%, as did the percentage of submissions that resulted in a regression (from 8.54% to 1.7%), meaning that students were submitting to the grader only half as much, and regressed far less than that.

8.5% of submissions caused a regression before the penalties were applied but only 2% caused a regression afterwards, which is a significant drop in turbulence. These differences can be seen comparing the autograder performance before (Figure 1) and after (Figure 2) the regression penalties were added.

The average grade for the submission also fell by roughly 10%. We will discuss this in more detail in the Analysis section on Student Performance (Section 5.2).

4.2 Student Perceptions

241 out of 247 students (97%) responded to the survey in which we asked them to report on changes to their process that resulted from the introduction of regression penalties. 209 (86%) indicated that they felt their process was influenced by the presence of regression penalties.

4.2.1 Testing. 83 students (~33% of those who reported influence) responded that they perceived an increase in their own testing practices as a way to prepare submissions to production.

> *In previous classes with autograders I would just make changes and test using the autograder rather than write my own exhaustive test suite. The regression penalty discouraged me from doing that and forced me to write my own tests.*

4.2.2 Caution. 163 students (~68%) reported an increase in their level of caution that aligned with our goals for the approach:

> *It was the first time I really double checked and triple checked my code before merging into master for a school assignment*

Some students reported feeling hesitant to submit a new version if their score was already high, and that this may have hurt their overall score:

> *Regressions were definitely on my mind throughout the project. Those combined with my partner usually being ready to commit a day or two after me lead me to accept the first mark we received when merging to master more often than not. I think I may have approached it differently with more time or in a work setting, but on a time crunch it lead me to adopt a "better safe than sorry" mindset.*

4.3 Stress & Risk Avoidance

50 students (~20%) explicitly indicated they felt stress with the penalties. 36 students (~15%) mentioned feeling that the penalties brought undue risk, and 13 students reported that they believe their final grade was hurt by not taking more risk with their submissions. One group indicated they failed the deliverable because of waiting so long to merge that they missed the deadline. The other 12 groups reported not attempting a perfect score when reaching 90-95% because they perceived the risk of a penalty to outweigh the benefit of a perfect score. 19 students (~8%) reported feeling stressed but also feeling positive about the penalties. They provided comments similar to this:

> *Yes, it is both good and bad. Good thing is that it motivates me to write more tests and check if my code satisfies all the specs before submitting to autobot. However, it does add a lot of stress at the same time as we might spend additional unnecessary time checking on trivial details on the project before where we could just submit and get feedback right away. Overall, I do think regression penalties is somewhat useful as it simulates a real world working experience and forces us to be more responsible on our code.*

4.4 Positive Assessment

Many students (76/241, ~31%) reported having a positive regard for the regression penalties, making statements similar to this:

> *Yes, I was very careful I didn't break anything before committing to master. I think if it weren't for this policy, I would have been tempted to use the test suite used to grade our projects as my primary measure of progress. Having to avoid regressions though, forced me to ensure that my own tests were sufficient. This definitely helped me in the long run, because I actually knew what I was testing, as opposed to the hidden test suite. Overall, it definitely got me into the habit or writing better test suites.*

Of those, 26 students (~10%) specifically mentioned the product-release quality learning outcome we were trying to instill as a benefit for the policy:

> *The regression penalties forced me and my partner to be more methodical when debugging and brainstorming solutions to our problems. I think even though it brings on some pressure, it's an adequate reflection of pushing a working prototype in a real workplace, so at least now I sort of have an idea of what to expect. Overall I like them.*

5 DISCUSSION

In this section we synthesise various quantitative and qualitative results to analyse the efficacy of the regression penalties technique with respect to the two main components of our research question: decreasing autograder abuse and maintaining student performance.

5.1 Reduction in Autograder Over-reliance

To assess whether students' reliance on the autograder was reduced, we examined both the turbulence of the autograder submissions and the student's perception of their own process changes. We observed a reduction in turbulence in student submissions when regression penalties were in place. This was observed quantitatively using several measures:

(1) The number of submissions in the regression penalty instance was substantially lower than in the prior course instance, pointing to a more measured development practice.
(2) Examining the proportion of submissions that caused a regression. In the regression penalty instance, the proportion of submissions causing regressions was dramatically lower than in the prior course instance.
(3) With respect to the *magnitude* of the regressions, the regressions tended to be much larger prior to the penalties as well, suggesting students were able to submit smaller less risky changes once the penalties were in place.

Reduced autograder reliance was also supported by the subjective responses from students. 163/247 (~65%) of the students reported increased care prior to submitting. Of those, 82 specified that this specifically involved increased testing using their own test suites.

5.2 Student Performance

To assess student performance we examined both the average and median grades for the deliverable and the students' perception of their own performance. Although the average grade differed substantially between the two instances, the median grade is quite similar. To look at this further, we plotted the grade histograms for the two course instances in Figure 3. This shows that the the distribution of grades between the instances was similar, although the peak was lower and the tail was longer in with the regression penalties.

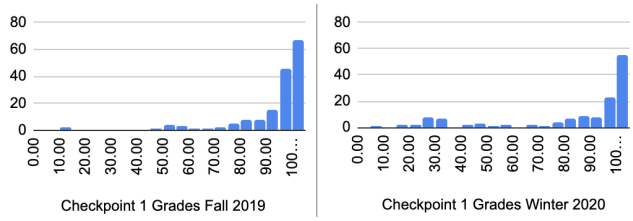

Figure 3: Comparison of grades between instances.

We can look more deeply at grade impacts by examining student comments. 13 students' perception that grades were somewhat compromised was borne out. All but one of these reports were students saying that once they hit 95% they stopped because the risk of a penalty outweighed the reward of achieving a perfect grade. However one otherwise strongly performing team reported that they achieved a poor grade because they had been hesitant to submit to the grader, and hence missed the deadline. Those

comments may explain the slightly lower peak at the top of the histogram, and also increased activity in the lower grades.

But is this an unequivocally bad thing? Increased caution, at the expense of risk-taking, may not be a negative within a Software Engineering context. Some students reported fear at pushing their changes, which is actually quite representative of how it feels to publish code to real clients. When combined with the reduction in submissions, and self-reported increase in testing, this suggests that students were thinking more, and considering the problem more. As educators in software engineering, we actually find it encouraging that students are reporting choosing to be meticulous about their code before release. It was our suspicion that in the Fall of 2019 and prior to that, students were receiving inflated grades because of their ability to submit risk-free to the grader, effectively gaming the grader as opposed to considering the problem. It is conjecture, but we believe that the grades that were reported in the Winter of 2020 were more accurately reflective of students' comprehension of the problem and a more realistic assessment of their level of skill.

In future semesters, we may need to tune it so that students still feel a sense of caution, but do not feel paralysed. Or, we might employ increased coaching to remind students about how the penalties work, and that they get one free penalty (the 12 teams that reported not trying for perfect grades had no prior regressions, so they would actually have been able to try risk free).

6 STUDY LIMITATIONS

There are a number of limitations to our *in vivo* experimental design for evaluating regression penalties.

6.0.1 Unable to Compare Coverage. A drawback of this study is that we are unable to compare coverage performance, and so cannot accurately gauge the *quality* of the tests written by students during the Winter 2020 semester. Because there were no grades allocated to coverage, many students chose not to commit their tests to their repositories.

6.0.2 Max Grade as Confounding Influence. The grading rule that we would take the maximum grade from all submissions may have exacerbated the thrashing that appears at the end of the Fall 2019 deliverable submission period. We believe the influence of this rule is mitigated by two factors:

(1) In prior semesters we had a rule that it was the last submission to the autograder that counted. This resulted in students reverting their submission to their own best commit right before the deadline, and then submitting that commit.
(2) The same grading rule is in place for the semester with the regression penalties.

6.0.3 Smoke Tests as Confounding Influence. In Winter 2020 we provided the students a set of smoke tests against which they could run their development code. These were tests that allowed students to see whether the code would run properly on the architecture, and did a basic feature check. Smoke tests were not sufficient to determine whether the code would achieve a satisfactory grade when pushed to production. This is supported by the fact that only 14 of the 209 students who indicated an effect on their process mentioned using the smoke tests. While we did not have tests specifically called *smoke tests* in the Fall 2019 semester, we did give students automatic feedback on the status of their code style, build status, and timeout.

6.0.4 Potential Response Bias on Perception of Penalties. Students knew that we were introducing regression penalties to better tie practice to their learning outcomes. Thus, there may have been some wish to tell us what we wanted to hear when reporting on the experience with the penalties. We cannot rule this out completely, however we believe this effect was muted because:

(1) Students knew their responses were entirely anonymous and would not be viewed until after the course was over.
(2) We did not directly ask whether students liked or disliked the penalties. We specifically asked them to indicate how it influenced their process. We did not tell them that we were hoping they would reflect on their own tests.

6.0.5 Generalisability to Other Courses. We employed regression penalties in a software engineering course that was explicitly teaching a client-centric view of releasing code. We believe our results show that students benefited from the introduction of regression penalties. But would courses in other fields of computing, or even more broadly, garner the same benefits? The fact that a large number of students specifically noted being more careful, regardless of whether they appreciated the policy, or were aware of the intended learning outcomes, suggests that the increased care is separate from the broader context of the introduction. That said, students in other courses may feel that the rule is more arbitrarily introduced. If this were to be included in a different field, it may be necessary to weave it into a sensible, learning-driven outcome that students would appreciate beyond suggesting that they should learn to consider their submissions more carefully within their own understanding of the problem and solution, rather than repeatedly using the autograder as a crutch.

7 CONCLUSIONS

Based on our analysis, the regression penalties performed well in terms of reducing over-reliance on the autograder as a feedback crutch. We saw a large reduction in grade drops when penalties were introduced, which was supported by over half the students' perception that they used more care, and in roughly a third of the cases specifically mentioned more reliance on their own test cases to bolster their confidence prior to submitting to the grader. We observed a small dampening effect in terms of risk taking, however given the nature of the domain, risk aversion is a positive learning outcome. We believe that these results could translate to any course employing an autograder, however we found that it was valuable to have an area-relevant narrative to justify the penalties and would advise anyone employing this technique to insert a similar mechanism. We still believe the penalties would reduce turbulence, but the positive and situated mindset associated with the penalties may have played a non-trivial part in their success, and the mood of the class with respect to their inclusion.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, our colleagues Meghan Allen and Margo Seltzer for their substantive input, and Andrew Stec for his help with data wrangling.

REFERENCES

[1] Kirsti M. Ala-Mutka. 2007. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* (Feb. 2007). https://doi.org/10.1080/08993400500150747

[2] Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. 2018. The Effect of a Web-Based Coding Tool with Automatic Feedback on Students' Performance and Perceptions. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*. 2–7. https://doi.org/10.1145/3159450.3159579

[3] Benjamin Clegg, Siobhán North, Phil McMinn, and Gordon Fraser. 2019. Simulating Student Mistakes to Evaluate the Fairness of Automated Grading. In *Proceedings of the International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 121–125. https://doi.org/10.1109/ICSE-SEET.2019.00021

[4] Anton Dil and Joseph Osunde. 2018. Evaluation of a Tool for Java Structural Specification Checking. In *Proceedings of the International Conference on Education Technology and Computers (ICETC)*. 99–104. https://doi.org/10.1145/3290511.3290528

[5] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: Automatically Grading Programming Assignments. *SIGCSE Bulletin* 40, 3 (June 2008), 328. https://doi.org/10.1145/1597849.1384371

[6] Sarah Heckman and Jason King. 2018. Developing Software Engineering Skills Using Real Tools for Automated Grading. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*. 794–799. https://doi.org/10.1145/3159450.3159595

[7] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the International Conference on Computing Education Research (KOLI)*. 86–93. https://doi.org/10.1145/1930464.1930480

[8] Ville Karavirta, Ari Korhonen, and Lauri Malmi. 2007. On the Use of Resubmissions in Automatic Assessment Systems. *Computer Science Education* (Feb. 2007). https://doi.org/10.1080/08993400600912426

[9] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics. In *Proceedings of the International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 126–137. https://doi.org/10.1109/ICSE-SEET.2019.00022

[10] Formative Authorised Educator MGarcia. 2018. The things you learn the hard way - Beware "return scores instantly". (2018). https://discuss.goformative.com/t/the-things-you-learn-the-hard-way-beware-return-scores-instantly/1880/16

[11] Károly Nehéz, Sándor Király, and Oliver Hornyak. 2017. Some aspects of grading Java code submissions in MOOCs. *Research in Learning Technology (RLT)* 27 (07 2017). https://doi.org/10.25304/rlt.v25.1945

[12] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. *ACM SIGCSE Bulletin* 38, 3 (June 2006), 13–17. https://doi.org/10.1145/1140123.1140131

[13] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the International Conference on International Computing Education Research (ICER)*. 131—-139. https://doi.org/10.1145/3291279.3339416

[14] Lucas Zamprogno, Reid Holmes, and Elisa Baniassad. 2020. Nudging Student Learning Strategies Using Formative Feedback in Automatically Graded Assessments. In *Proceedings of the International Conference on Systems, Programming, Languages, and Applications: Software for Humanity - Education Track (SPASH-E)*. 1–11. https://doi.org/10.1145/3426431.3428654