This is a preprint. Please cite as:
Matthew Pike, Boon Giin Lee, and Dave Towey, (2021). "The Robots will Rule: Improving Coursework Marking and Feedback through an Automated System", in Proceedings of the International Conference on Open and Innovative Education (ICOIE 2021), pp. 268-277.

The Robots will Rule: Improving Coursework Marking and Feedback through an Automated System

[1]Matthew Pike, [2]Boon Giin Lee, and [3]Dave Towey

[1,2,3] School of Computer Science, University of Nottingham Ningbo China, Ningbo, Zhejiang, People's Republic of China
[1] matthew.pike@nottingham.edu.cn, [2] boon-giin.lee@nottingham.edu.cn, [3]dave.towey@nottingham.edu.cn

Abstract

Purpose – Delivering high-quality, timely and formative feedback for students' code-based coursework submissions is a problem faced by Computer Science (CS) educators. Automated Feedback Systems (AFSs) can provide immediate feedback on students' work, without requiring students to be physically present in the classroom — an increasingly important consideration for education in the context of COVID-19 lockdowns. There are concerns, however, surrounding the quality of the feedback provided by existing AFSs, with many systems simply presenting a score, a binary classification (pass/fail), or a basic error identification ("The program could not run"). Such feedback, with little guidance for how to rectify the problem, raises doubts as to whether these systems can stimulate deep engagement with the related knowledge or learning activities. This paper presents an experience developing and deploying a new AFS that attempts to address the identified current deficiencies.

Design/methodology/approach – We developed an AFS to mark and provide feedback to 160 CS students studying an introductory Databases class. The experience of designing, deploying, and evolving the AFS is examined through reflective practice, and focus-group (involving peer teachers) analysis. The student experience of the AFS is explored through formal University-level feedback systems, and follow-up survey and interviews.

Findings – In contrast to most introductory-level coursework feedback and marking, which typically generate significant student reaction and change requests, our AFS deployment resulted in zero grade-challenges. There were also no identified marking errors, or suggested inconsistencies or unfairness. Student feedback on the AFS was universally positive, with comments indicating an AFS-related increase in student motivation. Interesting perspectives that emerged from our reflections and analysis included the issues of how much impact did our own Software Engineering training and approach to building and deploying the AFS impact on this success.

Originality/value/implications – Our successful experience of building and using an AFS will be of interest to the entire teaching community, not just CS/SE educators. The associated increases in marking and feedback reproducibility, accountability, and automation represent an important advance in AFS technology. In collaboration with our students, we are currently evolving the AFS into an autonomous learning object that they will be able to use independently of regular classes. Eventually, we hope to release the AFS to a wider audience as an open educational resource (OER).

Keywords: automated feedback, student engagement, technology-enabled student advising, open education resource

1 Introduction

The impact of the global COVID-19 pandemic caught a number of higher education (HE) institutions (HEIs) unprepared, requiring them to rapidly rethink and revise their approach to teaching and learning (T&L) practices, particularly regarding assessment and the provision of feedback to students (Gill et al., 2020). This is especially true for Computer Science (CS) educators, who rely on in-person interactions during practical laboratory sessions to provide guidance to novice programmers who embark on the challenging adventure of learning to program.

Programming is a complex activity, requiring learners to abide by strict syntax rules, decipher complex (and often unintuitive) error reports, and develop correct logical statements and structures to solve a given problem or task (Jenkins, 2002). Learning to program is a practice-led activity (Tan et al, 2009). Students cannot learn to program by reading a textbook or attending lectures, alone. Practice is a critical factor in the acquisition of programming skills (Robins et al., 2003). Laboratory sessions are not only an opportunity for CS students to practice their programming, also allow them to obtain immediate, contextual feedback on their code from teaching staff. Often, this feedback is delivered in a multimodal form, including spoken explanations, referencing to code on-screen through finger pointing and sometimes the direct editing of the student's code to demonstrate the necessary remedial action. The provision of feedback is known to be a critical factor in reinforcing student's learning (Shute, 2008). It is also the source of great dissatisfaction amongst students, who are consistently less satisfied with feedback than with any other feature of their courses (Nicol et al., 2014). In the aftermath of COVID-19 the increasingly distributed and hybrid nature of HE learning environments will challenge traditional forms of feedback provision. Alternative approaches are therefore required to address this challenge.

Many CS educators have explored the use of Automated Feedback Systems (AFSs) for the provision of feedback to novice programmers (Towey & Zhang, 2017; Singh et al., 2013; Odekirk-Hash & Zachary, 2001; Ala-Mutka, 2005). AFSs have many favorable properties — including timeliness and consistency — and can be an always-available resource, allowing students to receive formative feedback outside of timetabled sessions. AFSs are not a silver bullet, however, as there are many shortcomings associated with their use. Students often remark that AFS feedback is not sufficiently detailed or specific, often providing only a "pass or fail" indicator of performance, with little or no guidance to novice programmers about how to proceed to improve or fix their code (Keuning et al., 2016). CS educators have also highlighted issues, including the adaptability of these systems for use in particular assessments, which often required additional programming to overcome technical constraints imposed by the AFS itself (Keuning et al., 2016).

In this paper we explore an alternative approach to the development and deployment of an AFS. Specifically, our approach only generates and builds the feedback that students will receive *after* they have submitted their solutions. This is in contrast to conventional AFS approaches, where the marking logic and feedback are typically prepared in parallel with the task specification — before students submit their solutions. Similarly, in our application, students do not interact directly with the AFS, in-fact, they may not even be aware of the existence of the AFS. Students

complete their assignments *as usual*, without needing to conform to the technical or logistical constraints imposed by the AFS, thus preserving the ecology in which they write their code. Additionally, we explore the impact of applying standard software methodologies, specifically Agile and Test-Driven Development in an educational context. We posit that the application of these methodologies may improve the quality of feedback provision in the context of developing programming skills.

2 Context

The University of Nottingham delivers the same degree content across all of its campuses, including at the University of Nottingham Ningbo China (UNNC), where the undergraduate degree programmes related to CS are fully accredited by relevant professional organisations. During the first year of the CS programme, a strong focus is placed on developing students' programming skills. The "Databases and Interfaces" (DBI) class is one of the programming-based classes that all CS undergraduates complete. DBI provides students with a general introduction to the theory and practice of database systems, with students learning to create and interact with databases using a structured query language, SQL (Eisenberg & Melton, 1999). Additionally, students learn to design and implement graphical user interfaces (GUIs) in a web-based context, interacting with standard web technologies such as HTML (Hickson & Hyatt, 2011) and CSS (Bos et al., 2005). The DBI class has run, under various names and formations, for over a decade at UNNC and is a well-established part of the CS programme.

In the aftermath of the COVID-19 pandemic, the DBI teachers revised and adapted the class for a hybrid delivery, which included providing for approximately 30 remote-learning students and 130 in-person students. Prior to COVID-19, DBI relied heavily on a centralized technical infrastructure, situated on the university campus. This infrastructure provided students with access to an Apache[1] webserver and a MariaDB[2] database server, allowing DBI students to complete weekly laboratory tasks and coursework. This infrastructure worked well with on-campus students, providing a reliable and convenient resource. Other classes within the school also used this infrastructure. However, a number of off-campus students, especially those outside of Mainland China, faced great difficulties obtaining a reliable connection to the campus network. Some of the technical limitations were insurmountable, introducing a significant challenge to the university and great frustration to students. The class instructors decided, for 2020-2021, to transition to a local technical infrastructure, meaning the software required for the participating in DBI would be run directly on the students' personal computing devices. In doing so, the instructors were keen to use freely available and lightweight (small memory and hard-disk requirements) software solutions, being aware that some students were working on constrained computing devices.

SQLite[3] was used as the database server software. SQLite is a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is freely available on all major operating systems and is a small download (<1MB), which was

[1] Apache Web Server - https://httpd.apache.org/
[2] MariaDB Database Server- https://mariadb.org/
[3] SQLite Database Engine - https://www.sqlite.org/index.html

an important consideration for students with poor network connectivity. Apache was replaced by Flask[4], a lightweight web framework written in Python[5]. Flask provides functionality to serve static (HTML, CSS) and dynamic webpages using data stored in SQLite (and other) databases. Again, Flask was chosen for its lightweight installation process and its availability on all major operating systems.

The first week of laboratory sessions were dedicated to ensuring that students' machines were correctly configured, and capable of using the software. This process was relatively smooth, with few technical challenges. Once the set-up was confirmed, we began to specify and develop the coursework around these software technologies.

DBI is assessed through both a written examination and coursework. The coursework is split into two activities, with Coursework-1 consisting of multiple quizzes assessing knowledge and understanding, while Coursework-2 (CW2) focusses on application. CW2 will be the focus of the remaining sections. For CW2, students were presented with a fictional, but realistic, scenario, in which they were to complete a partially-implemented inventory tracking system. Students were given a partial codebase and an SQLite database, upon which their solutions would be developed. The database was structurally complete, with all the necessary tables in place to support the application's data requirements. The data itself were incomplete, an issue students would be required to address (Task 1, below). All students received an identical initial (partially-implemented) codebase, to which they were expected to add the functionality specified below. The coursework involved standard web-technologies that students had become familiar with through the DBI course (HTML, CSS, SQL and Python). Students were given approximately three-weeks to complete CW2, which contributed 25% to their overall DBI mark. In addition to a task description, students were also provided with a detailed marking scheme. Students were required to complete the following three tasks for CW2:

Task 1. Database Initialization – Students were provided with inventory data stored in a Comma Separated (CSV) file. Task 1 required students to implement a code-based procedure capable of initializing the database using the data stored in the CSV file. This required students to parse the contents of the CSV and specify the necessary SQL to insert the data into the tables provided with the coursework.

Task 2. Read from the Database – The partially implemented codebase had the necessary information to display the details of the inventory, through a web-based user interface. Task 2 required students to develop the SQL necessary to extract and collate the details of inventory items, for display in the web application; and to integrate the code with the existing code base.

Task 3. Insert into the Database – Students were required to obtain input from users, through a web-based form, and insert it as a new item in the inventory collection. Task 3 also required students to perform the necessary validation and checking of the user provided input to ensure it met the specified data specifications.

---

[4] Flask Micro Web-Framework - https://flask.palletsprojects.com/en/1.1.x/

[5] The Python Programming Language - https://www.python.org/

3 Automated Feedback Systems

Students' solutions were submitted through Moodle[6], the learning management system used at UNNC. Solutions were then downloaded by the coursework marker and prepared, through an automated process, for use with the AFS. Preparation consisted of renaming submissions according to student's ID and separating each task's solution files into sub-directories, isolating individual components of the assessment process. It was decided that students would not interact directly with the AFS: Instead, they would develop their solutions in a typical development environment, unconstrained by any technical or operational limitations of the AFS (Keuning et al., 2016). This approach had the benefit of preserving the ecology of the task, mirroring the development students will encounter when completing industrial work. The feedback itself would be delivered in a feedback-file on Moodle.

Once preparation was complete, the CW2 marker then initiated the first stage of the marking procedure. The overall process was divided into two phases: 1) running the application; and 2) parsing the output, allocating a mark and generating the feedback. The two phases were separated to avoid potential interference between student solutions and the operation of the AFS. Security was a practical concern for the first phase, something the marker considered carefully. All student solutions were run in a sandboxed Docker[7] container, a lightweight virtualization technology that allows code to run in consistent, isolated operating environment. Changes to a container's files or settings by an application are discarded once the container is powered down. This approach was taken to ensure that the environment where students' solutions were run was consistent; it also follows best security practices regarding the running of untrusted code (Špaček et al., 2015). The output of each student solution was captured, and stored to disk, ready for stage two, where the actual assessment was performed. The orchestration of this process was entirely automated.

Stage two saw the output of the previous stage being analysed according to the marking criteria. It was here that the CW2 marker began to implement the marking criteria, in code. This approach differs from the conventional, with the majority of existing AFSs being used as the framework around which assessment tasks are built. Typically, the marking logic used in an AFS will be developed simultaneously with the specification of the particular coursework. In this case, however, the AFS was developed *after* students had submitted their solutions. By following this approach, the CW2 marker had a corpus of *real solutions* upon which feedback appearing in the solutions. In addition to providing specific, tailored feedback, this approach enables a high level of adaptability, something that is a limiting factor in many existing AFSs (Keuning et al., 2016).

When marking, the CW2 marker first wrote a test to identify the correctness of a particular solution. Correct solutions are straightforward to mark, as they meet a predictable, pre-specified criterion: The marker can check whether or not the criterion was correctly realized in the student solution, and allocate a mark, accordingly. However, as is common with developing programmatic solutions, there may be

[6] Moodle learning management system - https://moodle.org/
[7] Docker Containerization Platform - https://www.docker.com/

multiple correct solutions that are equally valid. Certain solutions may, however, possess properties that are more desirable — for example: faster execution, lower memory utilization, or a solution that follows best practices. Therefore, in addition to identifying the correctness of a given solution, where appropriate, the feedback was tailored according to the particular implementation approach taken.

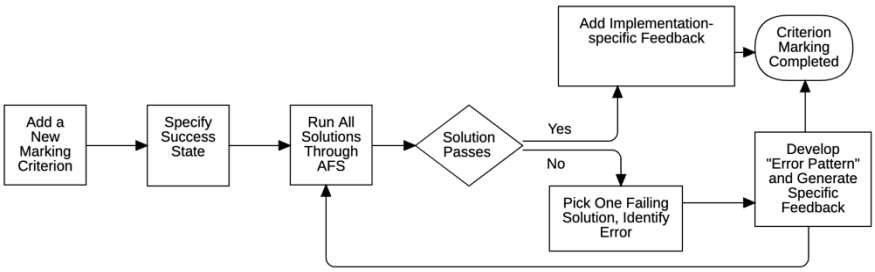

Fig. 1. The implementation workflow for a single marking criterion within the AFS

Having identified, marked and developed feedback for correct solutions, the CW2 marker then looked at solutions that failed to meet the specified marking. This process began with the developer performing a manual inspection of one of the incorrect solutions. Having identified the particular error, an "error pattern" was developed (in code) to identify the particular fault, and added to the AFS. Specific feedback was also generated to highlight the error and provide advice on how to attempt to rectify it. The AFS was then run again, on all solutions, to identify others that shared the same "error pattern". This process was repeated until all solutions were processed. This approach borrows from the software development methodology, Test-driven development (TDD) (Janzen & Saiedian, 2005). TDD requires a developer to first write a test for a particular unit of functionality, before writing the code to fulfill that functionality. The test can then be used to validate that implementation of the functionality meets the functional requirements specified in the test. However, in this application of TDD, the CW2 marker specified tests to interrogate the correctness of student solutions and provide feedback accordingly. Students' solutions were therefore viewed as the 'units' under test. Test coverage is a metric used in TDD to measures the amount of testing performed by a set of tests. Test coverage, in the context of this application, was the coverage of feedback provided to the set of student solutions. We required 100% coverage, meaning that every student submission had a mark and at least one item of feedback for each marking criterion.

The DBI CW2 marking was successfully completed, and submitted on time. Students received a detailed report containing their individual feedback and score. In contrast to most introductory-level coursework feedback and marking, which typically generate significant student reaction and change requests, our AFS deployment resulted in no grade-challenges. There were also no identified marking errors, or suggested inconsistencies or unfairness. Student feedback on the AFS was universally positive, with comments indicating an AFS-related increase in student motivation.

Task 1 [4.0/5]

Task 1 was marked using the following procedure:

- The original `iMusic.db` database (containing 50 tracks) was copied into the working directory, which also included `TracksToAdd.tsv` and the `TSV_Import.py` that you submitted.
- The command `python TSV_Import.py` was executed in the working directory containing your solution.
- An automated process was used to identify differences between the `iMusic.db` database generated by your solution (`TSV_Import.py`) and the validated (correct) version. **Note** - TrackId was ignored when performing this comparison.

ID	Requirement	Feedback	Marks
1A	Your solution executes, without error or modification, using one of the commands `python TSV_Import.py` or `python3 TSV_Import.py`	The submitted solution ran without modification or error.	1/1
1B	Your solution correctly parses the TSV using the `csv` module.	The solution makes correct use of the CSV library.	1/1
1C	Your solution inserts all entries specified in the TSV file correctly into the database using the `sqlite3` module.	A good solution, which correctly handles Tab and UTF-8 characters and does not insert the header in the Track table. However your solution not handle quotes characters correctly.	1.5/2
1D	Your solution should have suitable checks to ensure that the operation completed with raising an `Exception`.	A single try/catch statement is used to capture exceptions across the entire solution. Be warned - this is generally considered bad practice, specific exceptions should be handled.	0.5/1

Fig. 2. Example of AFS-generated feedback for a single task

We suggest that our AFS addressed a category of errors that are commonly encountered when marking a large number of student assignments requiring detailed technical analysis. Specifically, our AFS promotes:

- **Consistency** – The marking is deterministic: The feedback and grading of coursework is the same for all submissions that follow a particular implementation strategy. Consistency here refers to both the provision of marks and feedback.
- **Correctness** - Simply automating the process of marking does not ensure the correctness of the output. However, when combined with the incremental and iterative development methodology detailed here, a number of the shortcomings typically associated with AFSs were successfully avoided.
- **Tailored Feedback** – Feedback provided to students is specific, relevant, and detailed; and not simply determined on a pass or fail basis.

4 Discussion

Good software engineering practice includes performing regular retrospectives, reflecting and learning from experiences (Dybå et al., 2014). The innovation described in this paper was in response to a growing number of first-year CS students participating in the DBI class. The increase in the number of students was predictable, the impact of COVID-19, however, was less so. It is hoped that the introduction of an AFS has helped to address both challenges. The introduction of the AFS in the 2020-2021 run of DBI helped address the issue of a larger class size, with coursework

feedback scores and feedback being returned, on-time, with only a single marker being involved in the marking process. There was a significant initial time investment required in developing the AFS, crucially however, the time required was comparable to the time required to manually mark 160 student submissions. The net benefit of consistent and correct marking, and detailed and tailored individual feedback, as well as a reusable education resource, struck a positive tone with the DBI teachers. Equally, future runs of DBI will benefit from the AFS. The system will still require tailoring for the specific task students are expected to complete, but much of the current infrastructure is reusable.

Going forward, it is hoped that the AFS can be repurposed as an Open Education Resource (OER) (Towey et al., 2019). This would provide an effective solution for the DBI instructors, who benefit from current students receiving high-quality, consistent and timely feedback on their coursework. Additionally, in the form of an OER, students would be able to interact with and obtain formative feedback on solution attempts – without explicitly interacting with the instructors. The AFS OER could also be deployed during the laboratory sessions, reducing the demands on staff, allowing them to provide more detailed and insightful feedback to students encountering situations or questions that are not addressed by the AFS.

The repurposing of the AFS as an OER will address another issue that students raise in their evaluation of DBI: the amount of content delivered in the class. Students report being overwhelmed by the number of technologies introduced. This an unfortunate (but necessary) reality of web-based development, which is dependent on the interaction of several technologies, that, in isolation, serve little purpose. The availability of an OER would allow students to prepare for DBI outside of timetabled hours, when feedback from the teacher is not available. Similarly, the analytics available to the teacher, from students' interactions with the AFS, may help highlight to the teacher the particular problem areas that students are experiencing.

Finally, this work also exemplifies the increasingly prevalent role automation is playing in HE (Beck et al., 1996; Andriessen & Sandberg, 1999). As we saw, removing the human from the process of manually marking CW2 led to more consistent and uniform feedback, while preserving specific, tailored feedback to individual students. Although in our AFS, there remains a human "in the loop" who is responsible for specifying the "error patterns" and generating responses, this may be viewed as a bootstrapping process: We could argue that the data necessary for a fully autonomous, generalizable AFS is being gathered to support a future, self-sustaining artificial intelligence (AI) AFS.

5 Conclusion

The provision of feedback is an important, but challenging, aspect of developing programming skills in novice programmers. The work presented in this paper outlines a novel approach in the specification and development of an AFS. Additionally, this work demonstrates the successful application of two Software Engineering methodologies in a T&L context, specifically in generating summative feedback of students' programming solutions. The current AFS will continue to be used, and there are plans to evolve it, to better support student learning in future DBI coursework and

laboratory sessions. Further, we are planning to repurpose the AFS as an OER, and make it available to the community, enabling novice programmers to develop their programming skills independent of classes or institution.

References

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer science education*, *15*(2), 83-102.

Andriessen, J., & Sandberg, J. (1999). Where is education heading and how about AI. *International Journal of Artificial Intelligence in Education*, *10*(2), 130-150.

Beck, J., Stern, M., & Haugsjaa, E. (1996). Applications of AI in Education. *XRDS: Crossroads, The ACM Magazine for Students*, *3*(1), 11-15.

Bos, B., Çelik, T., Hickson, I., & Lie, H. W. (2005). Cascading style sheets level 2 revision 1 (css 2.1) specification. *W3C working draft, W3C, June*.

Dybå, T., Maiden, N., & Glass, R. (2014). The reflective software engineer: reflective practice. *IEEE Software*, *31*(4), 32-36.

Eisenberg, A., & Melton, J. (1999). SQL: 1999, formerly known as SQL3. *ACM Sigmod record*, *28*(1), 131-138.

Gill, A., Irwin, D., Towey, D., Walker, J., & Zhang, Y. (2020). Reacting to the Coronavirus: A Case Study of Science and Engineering Education switching to Online Learning in a Sino-foreign Higher Education Institution. In *Proceedings of the 2020 International Conference on Open and Innovative Education (ICOIE 2020)*, pp. 385-404.

Hickson, I., & Hyatt, D. (2011). HTML 5. *W3C Working Draft WD-html5-20110525*, 53.

Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. Computer, 38(9), 43-50.

Jenkins, T. (2002, August). On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences* (Vol. 4, No. 2002, pp. 53-58).

Keuning, H., Jeuring, J., & Heeren, B. (2016, July). Towards a systematic review of automated feedback generation for programming exercises. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (pp. 41-46).

Nicol, D., Thomson, A., & Breslin, C. (2014). Rethinking feedback practices in higher education: a peer review perspective. *Assessment & Evaluation in Higher Education*, *39*(1), 102-122.

Odekirk-Hash, E., & Zachary, J. L. (2001, February). Automated feedback on programs means students need less help from teachers. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education* (pp. 55-59).

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education*, *13*(2), 137-172.

Shute, V. J. (2008). Focus on formative feedback. *Review of educational research*, *78*(1), 153-189.

Singh, R., Gulwani, S., & Solar-Lezama, A. (2013, June). Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation* (pp. 15-26).

Špaček, F., Sohlich, R., & Dulík, T. (2015). Docker as platform for assignments evaluation. *Procedia Engineering*, *100*, 1665-1671.

Tan, P. H., Ting, C. Y., & Ling, S. W. (2009, November). Learning difficulties in programming courses: undergraduates' perspective and perception. In *2009 International Conference on Computer Technology and Development* (Vol. 1, pp. 42-46). IEEE.

Towey, D., Reisman, S., Chan, H., Demartini, C., Tovar, E., & Margaria, T. (2019, December). OER: Six perspectives on global misconceptions and challenges. In *2019 IEEE International Conference on Engineering, Technology and Education (TALE)* (pp. 889-895). IEEE.

Towey, D. & Zhao, K (2017 , December). Developing an Automated Coding Tutorial OER. In *Proceedings of the IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE 2017)*, pp. 233-238.