EMPIRICAL STUDY

Experiences with Test-Driven Elaborated Feedback for Teaching Introductory Programming

Antonio Garcia-Dominguez[a] and Tony Beaumont[b]

[a]Department of Computer Science, University of York, Heslington, York, UK;
[b]College of Engineering and Physical Sciences, Aston University, Birmingham, UK

ARTICLE HISTORY
Compiled October 23, 2025

ABSTRACT

Background and Context: Computer Science students learning how to code need timely and effective feedback. Delivering such feedback can be challenging in large cohorts with diverse starting points and personal circumstances, given the available resources.

Objective: automated test-driven feedback specific to each assignment can provide rapid feedback on common mistakes, leaving more time for instructors to provide individualized assistance.

Method: we developed AutoFeedback, an open-source system combining instructor-designed test suites and feedback templates to deliver immediate and task-specific elaborated feedback on interim work during practical laboratories. AutoFeedback was used in two editions of the first-year programming module at Aston University. Data on student engagement, performance, and reception was collected and analysed through descriptive statistics, Mann-Whitney / Kendall tests, and bottom-up thematic analysis.

Findings: students engaged more consistently, liked receiving immediate feedback and a clear indication of progress, and requested its integration in other modules. Statistical tests showed that interim work improved with further submission attempts in response to feedback. Some students did not know how to respond to occasional terse feedback, or had tests fail due to irrelevant differences, or pass despite of mistakes.

Implications: Automated test suites can provide interim feedback, in addition to grading, by driving test results and program outputs through feedback templates tailored by instructors. To be effective, instructors must monitor the student experience, refining tests and feedback templates to address common mistakes and clarify explanations. These refinements must be communicated to students, so they feel supported.

KEYWORDS
Unit testing; test-driven development; containerization; static analysis; testing education

1. Introduction

The increasing demand for software developers makes computing courses increasingly attractive to prospective students: only in the UK, HESA reported 153,825 enrolments in computing-related courses in the 2020–21 academic year, a 16.55% increase over the 131,985 enrolments in 2019–20 (Higher Education Statistics Agency, 2022). Although

CONTACT Antonio Garcia-Dominguez. Email: a.garcia-dominguez@york.ac.uk

this is a positive trend in general, it has resulted in larger cohorts, identified by Kara, Tonin, and Vlassopoulos (2021) to be correlated with lower grades, especially in STEM subjects where teaching and learning are more faculty-driven (compared to the more student-driven activities in non-STEM subjects). One cause of this drop may be the challenge of providing effective feedback with the available resources: Poulos and Mahony (2008) identified that students preferred consistent and transparent assessment according to clear criteria with early feedback, and that the feedback contributed to a successful transition to university for students in their first year. Delivering timely feedback to large cohorts can be difficult, and first-year students will need more detailed feedback to build a solid foundation and understand what it is like to study at university.

In the case of a Computer Science course, the first hurdle students need to cross is learning how to "think in code" to write programs. In the first-year CS1OOP programming module at Aston University, students achieve this by completing weekly programming labs to practice what they have learned from lectures and seminars. The students spend most of the first term with an introductory Java-based programming tool (Processing, Reas & Fry, 2006), before moving on term 2 to a full-fledged integrated Java development environment (the Eclipse IDE[1]). This regular programming practice is consistent with the proposal by Kolb (1984) that learning is experiential: learners should actively do something to form ideas which are then modified through their experiences. The process only works with timely feedback, and given the large number of students, the process had to be automated. Given the challenge of delivering timely and detailed feedback, in the summer of 2020 it was decided to revamp term 2 to use test-driven development (TDD) as an additional source of feedback that students could obtain at any time.

The CS1OOP teaching staff reviewed the tools available at the time to provide test-driven automated feedback for students, finding them difficult to use for first-year students without prior knowledge of Git and continuous integration, and lacking the ability to provide customised feedback to students. Faced with this gap, we developed AutoFeedback (AF), a system that reuses existing open source testing frameworks and build tools, and uses containerisation to isolate student submissions. AF is available as open source under the Apache Public License from its GitLab repository[2].

This paper contributes the first description of the design of the AutoFeedback system, as well as the requirements that motivated it, and a report of the experiences and student feedback obtained during its use with large cohorts (with over 300 students) in the 2020–21 and 2021–22 academic years. The discussion covers the design and refinement of the course materials for automated test-driven feedback, and the student engagement and performance over the experience. Thematic analyses have been conducted on two surveys during the 2020–21 academic year, and one survey during 2021–22: both report positive results, with respectively 88.9% and 87.5% of the responses agreeing that AF helped them identify issues. At the same time, the survey points to improvement areas for AF and the general practice of automated test-driven feedback.

The rest of this paper is structured as follows. Section 2 presents the key pedagogic concepts behind AutoFeedback and the provision of automated feedback for programming tasks. Section 3 covers the design of AutoFeedback in more detail. Section 4 describes how AF was piloted in 2020–21 and 2021–22. Section 5 details the research

[1] `https://www.eclipse.org/ide/`

[2] `https://gitlab.com/autofeedback`

methods used to judge the effectiveness of AutoFeedback, and Section 6 presents our results. Section 7 discusses the meaning of these results and the threats to their validity. Section 8 places the results in context of related work. Section 9 offers our conclusions and outlines several areas for further research and development.

2. Background

The nature and adequate provision of feedback has been a focus of significant research over history: in this paper, we specifically focus on *formative feedback*, defined by Shute (2008) as "information communicated to the learner that is intended to modify his or her thinking or behavior for the purpose of improving learning". Concluding their review, Shute noted that useful feedback depended on three things: the student needing that feedback, the student receiving the feedback in time to use it, and the student being able and willing to use that feedback. While Shute found several meta-analyses that found feedback to improve learning between .4 and .8 standard deviations (compared to control conditions), they also observed that further research was needed on interactions between task and student characteristics, and instructional contexts.

Wong and Beaumont (2012) surveyed students across several Computer Science-related courses on their perception of the quality of the feedback they obtained about their work, and noted that high-scoring tutors produced feedback in a feed-forward style, stating what was done inappropriately, its consequence, and how to improve it. They noted a number of helpful and unhelpful styles of feedback. Helpful feedback was clear, easy to understand, used simple language and a friendly tone, and gave praise where it was due[3]. Formative feedback which simply stated how to fix an error in the submitted work was considered unhelpful to students.

The rest of this section describes key works that further investigate *how* to produce and provide feedback, and how to *close the loop* to see if that feedback was effective. We discuss the integrated model of the feedback process by Narciss (2013), and then outline the findings from Keuning, Jeuring, and Heeren (2018) on automated feedback generation for programming, which build upon Narciss' earlier works.

2.1. Pedagogical foundations: the Interactive Tutoring Feedback model

The Interactive Tutoring Feedback (ITF) model by Narciss (2013) synthesizes various theoretical and empirical insights of feedback frameworks and research. ITF considers that feedback helps regulate a learning process so that the learners acquire the knowledge and competencies needed to master learning tasks. In this paper, the knowledge and competencies would be understanding a specific programming language and developing programming skills, and the learning tasks would be the programming assignments.

The main components of the ITF model are outlined in Figure 1, which revolve around two feedback loops: an *internal* feedback loop for the learner, and *external* feedback loop involving the external source of feedback. The *controlled process* is the acquisition of the competences in order to master the learning task at hand. The feedback processes require reference levels: the *internal reference level* is the learner's own understanding of what is required of them, and the *external reference level* is what the teacher and instructional medium specify as the requirements of the task.

[3]Note that the use of praise is considered unhelpful in some of the studies reviewed by Shute (2008).

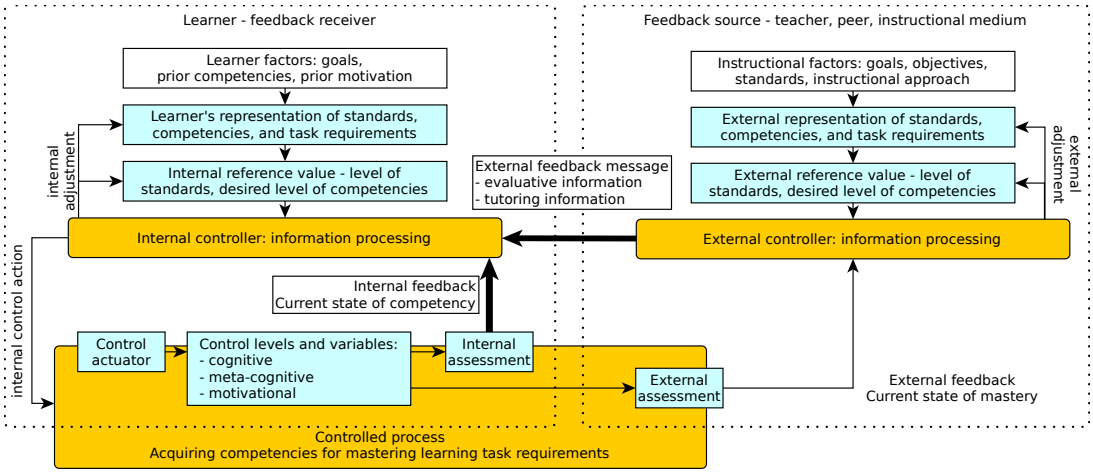


Figure 1. Components of the Interactive Tutoring Feedback model by Narciss: adapted and simplified from (Narciss, 2013)

In the internal feedback loop, the learner uses their *control actuator* to attempt the task, as regulated by a number of cognitive, meta-cognitive, and motivational variables. These respectively cover their knowledge, their strategy to approach the task, and their level of engagement. Their attempt at the task is then assessed, both internally (by the learner themselves) and by the external feedback source (a teacher, a peer, or an instructional medium such as the system proposed in Section 3). These assessments result in both *internal and external feedback*, which are fed back to the learner's *internal controller* to decide on a *control action* to improve their performance at the task, until they reach their reference value.

In the case of a programming assignment, the learner may have their own understanding of whether their code meets the requirements set out in the programming assignment (their internal assessment). This internal assessment may not be entirely accurate, e.g. if the learner misunderstood or missed a requirement, or misunderstood one of the concepts in the language. The external feedback should aim to provide a clear picture to the learner, so they can adjust their approach to the task.

Narciss considered feedback content to be divided into *simple* and *elaborated* components. The simple components only inform the student about their performance or outright tell them the solution, and are categorised into knowledge of performance (KP, e.g. "15 of 20 correct"), knowledge of result/response (KR, e.g. "correct/incorrect"), and knowledge of the correct results (KCR, e.g. a description of a correct solution). In contrast, elaborated feedback (EF) components intend to inform the learner so they can improve their own work (which matches the above definition from Shute of formative feedback). Narciss identified five EF categories: knowledge about task constraints (KTC), about concepts (KC), about mistakes (KM), about how to proceed with the task (KH), and about meta-cognition (KMC). This elaborated feedback is based on the teacher's cognitive task analyses (i.e. knowing where learners typically struggle).

The external loop has its own adjustment process, where the external representations of task requirements, standards, and desired levels of competency may change. The feedback generation of an automated system will need adjusting as the teachers identify typical mistakes and misconceptions from students. Likewise, the task requirements may need clarification if they are found to be ambiguous or incomplete. In summary, it is not only the learners that need to operate in a feedback loop, but also the teachers

Component	Hint subtype
Task Constraints (KTC)	Task Requirements (TR) Task-processing rules (TPR)
Concepts (KC)	Explanations on subject matter (EXP) Examples illustrating concepts (EXA)
Mistakes (KM)	Test failures (TF) Compiler errors (CE) Solution errors (SE) Style issues (SI) Performance issues (PI)
How to Proceed (KH)	Error correction (EC) Task-processing steps (TPS) Improvements (IM)
Meta-cognition (KMC)	(no subtypes)

Table 1. Programming-focused subtypes for each of the Narciss (2013) elaborated feedback ("Knowledge About...") components, according to Keuning et al. (2018)

and the way in which they configure and use the automated feedback system.

2.2. Automated feedback generation for programming tasks

Keuning et al. (2018) reviewed 101 tools that automatically generated feedback from programming exercises. To label the feedback produced by the tools, the above categories of elaborated feedback from Narciss (2013) were specialised for this domain, adding further subtypes: these are listed in Table 1. It is worth noting how there is a specific subtype for compiler errors: these are known to be problematic to students, due to their lack of readability for novice developers (Becker et al., 2019).

The review found that 96% of the tools provided knowledge of mistakes: the most common subtypes were test failures, solution errors, and compiler errors. There were also many tools that provided hints on how to proceed (44.6%). However, few tools provided knowledge on task constraints (14.9%) or concepts (16.8%), and meta-cognition was only addressed by one tool.

Keuning et al. also took into account the educational problem classification from Le, Loll, and Pinkwart (2013), who considered five classes of problems based on the range of possible solution strategies and their automated verifiability. They focused on tools that supported problems of Class 2 (one solution strategy with multiple implementation variants, such as filling in a code template), and of Class 3 (multiple known solution strategies that can be anticipated in advance). They found that 76.2% of the tools supported Class 3 exercises, and 23.8% only supported Class 2 exercises. It was observed that Class 2 tools gave more feedback on fixing mistakes and taking the next step, at the cost of not recognising alternative strategies.

In terms of techniques, the review noted that Class 3 tools were heavily reliant on

automated testing (67.5%), program transformations (39%), and basic static analysis (39%). Class 2 tools were less reliant on automated testing (only being used in 29.2% of them), and used model tracing far more often (25%) to provide concrete hints on how to proceed based on the known solution strategy.

The authors also considered the types of inputs normally used by the tools. Class 3 tools were more reliant on model solutions than Class 2 tools (55.8% vs 33.3%), and also needed more test data (57.1% vs 16.7%). They noted that sophisticated techniques such as model tracing and intention-based diagnosis made it harder to add new exercises and adjust the tool. They also found that very few papers addressed the role the teacher could take in making these adjustments. They concluded that except for test-based automated systems, teachers could not easily adapt tools to their own needs.

3. The AutoFeedback system

Before developing AutoFeedback in the summer of 2020, a number of existing options were considered. There are mature solutions for continuous integration which can automatically run tests when a new version of a project is committed to its source control system, such as GitHub Actions (GitHub Inc., 2022) or GitLab (GitLab Inc., 2022). These are the ideal solutions for teams familiar with version control systems and automated build systems, and in fact they are commonly used by second-year and final-year students at Aston University. On the other hand, first-year students going through their first programming experiences can significantly struggle with their complexity: our aim was not to compete with these mature solutions, but rather lower the barriers to entry for automated feedback as much as possible.

On the other side of the spectrum, there were web-based platforms that provide *code challenges* with small predesigned test suites, such as Codewars (Qualified, 2022) or HackerRank (HackerRank, 2022). These incorporate significant gamification components to keep participants motivated, and are very accessible to students. Unfortunately, they focus on small single-file programs, rather than multi-file projects, with tests only on inputs and outputs. We wanted automated feedback over programs that spanned multiple files, involved external libraries, and included checks on the design of the program and the quality of its code.

We considered two academic proposals as well: specifically, Gradeer and PABS. We discuss these options, as well as other options which we found after developing AutoFeedback, in Section 8 (related work). The proposals we found at the time did not find an option meeting these requirements:

- Be usable by first-year students without knowledge about version control systems, build systems, or continuous integration platforms.
- Integrate with traditional IDEs (e.g. Eclipse or IntelliJ), with only a few clicks needed to make a well-formed submission.
- Provide elaborated feedback for every failed test, hinting at common misconceptions and mistakes to assist the internal adjustment of the ITF model.
- Report on individual and cohort-wide student performance, to help identify difficult/brittle tests and refine them and their feedback, enabling the external adjustment of the ITF model.
- Allow for adapting the production and provision of feedback for a wide range of Class 2/3 problems, without requiring any changes to the tool.

For this reason, we decided to develop AutoFeedback (AF from now on), a web-based

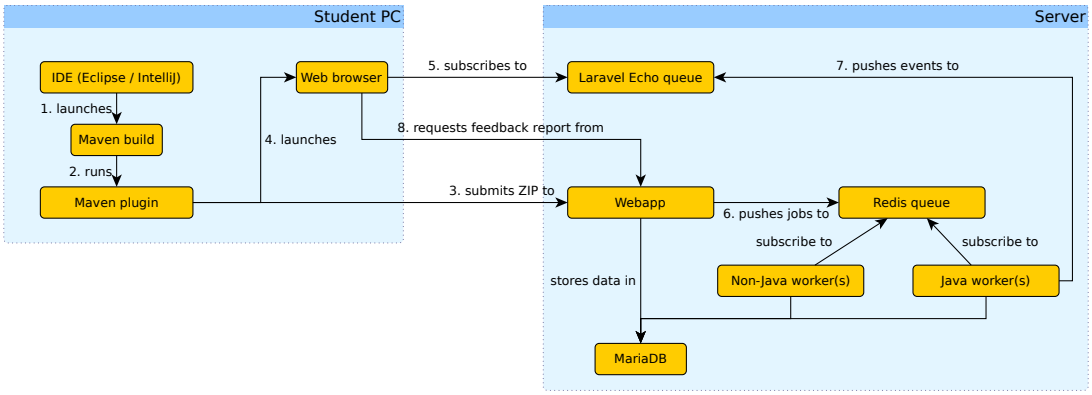


Figure 2. Overall architecture of AutoFeedback, and major steps in assessing a submission. Nodes are components, numbered edges describe assessment steps, and non-numbered edges describe persistent relationships between components.

platform for automated feedback. This subsection will discuss its overall architecture, the underlying data model, how assessments are designed, how students use the platform, and how teachers can inspect individual and cohort-wide student performance.

3.1. Architecture

Figure 2 describes the high-level architecture of AutoFeedback, and the major steps involved in the submission and assessment of code from a student.

AutoFeedback required a way of controlling the process of building and analyzing student code, so we provide students with Maven-based template projects. Maven was chosen because it is a mature system to automate a project's build, reporting, and documentation, which can be customized via plugins. We have built a Maven plugin for AF to control the process of providing feedback on student code.

A student wanting feedback on their progress will initiate a Maven build via a `.launch` file which runs the AF Maven plugin. The student's project is packaged into a ZIP file and submitted to the AF server. The student's web browser will then be automatically directed to a page that will display their feedback when it is ready.

On the server side, submissions are queued to allow AF to handle large numbers of submissions. When a submission reaches the front of the queue, AF builds the project and runs the tests, collecting all test output. The test output is analyzed using tutor-designed templates that generate the feedback to students. That feedback is sent to the waiting web page and displayed to the student.

In order to protect against poorly written code from students, the Java workers run under a number of security, time, and resource constraints, using Docker containerization. Java worker containers are single-use, being recreated from a clean image on every build. The process ensures both ease of use for students and security that limits the effect of incorrect or malicious code running in the server.

3.2. Data model

The AutoFeedback data model follows a typical Virtual Learning Environment (VLE) structure, where administrators can set up a number of modules containing assessments organized in a folder hierarchy. Teachers can control the availability period of

an assessment, set an indicative submission date, and provide a Markdown-based rich description to display to students (with support for syntax highlighting).

Teachers populate assessments by uploading a ZIP with the *model solution* to the webapp: currently, this must contain a Maven project. The uploaded ZIP is run in a Java worker (Figure 2): the standard error, standard output, and test results (e.g. JUnit-style XML files) are stored as build results. The XML files list every test class and test method in the tests accompanying the model solution. These test identifiers are used to populate the feedback template. Teachers can set via the webapp how many marks each test is worth when run against student-submitted code.

Teachers can define a Markdown feedback template for each test, with specific rules on what to show depending on whether the test passed, failed, or produced output matching certain conditions (more details in Section 3.3). Tests can be optionally grouped by assigning them to "tasks" (e.g. "Section 1" of the worksheet), rather than by test class.

Students are given starting code based on the model solution, also based on Maven but with a reduced test suite, as some tests may reveal details about the task to be performed. Teachers will create *file overrides* for the files that should be extracted from the model solution, overwriting any student-submitted files at the same relative path. Anything related to the project configuration and any code that the students should not modify is usually overridden. A student submission will be run against the full set of tests: both those submitted by the student, and those extracted from the model solution via file overrides.

Model solutions can be updated by instructors while the assessment is available, e.g. to make tests less brittle, offer assertions with better feedback, or add new tests that highlight common mistakes from students. This enables the external adjustment of the ITF model. All versions are stored and identified with a sequential counter, and student submissions track the model solution version they were evaluated against, to ensure reproducible results.

3.3. Feedback templates

The generation of feedback needs to model the process a tutor might follow in analysing which parts of a student's code is correct, and which are not. This process will look at the following components from Table 1:

- KTC: put the feedback within the context of the assignment constraints, and informing the student about which part of the task is being tested.
- KC: relate the feedback to the concepts taught in the course.
- KM: provide an easier-to-understand version of typical error messages that can be produced during this task, possibly with examples of what causes them.
- KH: give the student hints on how to proceed.
- KMC: provide strategies for learning, thinking, and problem-solving within the context of the task.

Simply drafting some text to be output when a given test fails will not be sufficient to address all of the above, or to do it in a way that will be helpful to the student.

The approach in AutoFeedback is to consider that for Class 2/3 problems, the available strategies are known in advance, and therefore common mistakes and problems can also be known in advance. This means that if the tests are appropriately designed to detect these situations, the test outputs (including failure messages, such as com-

````
```af_when_passed
Good job!
```
`````af_when_failed
````af_when_substring NullPointerException
You ran into a case of a null pointer exception.
This happens when accessing a property in an
object variable set to null, as in this example:
```java
Object o = null;
o.toString(); // will cause exception
```
````
`````
````

**Listing 1** Example of AutoFeedback conditional Markdown blocks for feedback templates

pilation/execution errors and identifiers for failed assertions) can be used as inputs to a *feedback template* that goes beyond basic KM information: as it is specific to one aspect of one task, it can cover KTC/KC/KH/KMC aspects as well.

In AF, feedback templates can be attached to each test case in the JUnit XML output of the Maven build of the model solution. This feedback template is written in Markdown, which provides its own formatting facilities (bold/italics font, syntax highlighting, hyperlinks, images, videos, etc.). In order to cover the many possible mistakes we can anticipate in a Class 3 problem without overwhelming the student, AF extends Markdown with fenced blocks that selectively display feedback based on whether the test passed, failed, or contained a line matching a specific substring or regular expression. Listing 1 shows a simple example where some encouragement will be given to the student if the test is passed, and if the test is failed, AF will check if `NullPointerException` appeared in the output: if it did, it will then explain how these exceptions can happen.

Markdown was chosen as a base notation to address the issue identified by Keuning et al. (2018) around the lack of adaptability of existing tools, and how they did not involve the teacher. The teacher can refine these templates at any time from the AF web interface, clarifying the feedback provided around a typical mistake across an entire cohort with minimal effort. Many teachers in Computer Science are already familiar with Markdown, and it is well supported by existing tooling.

### *3.4. Student feedback reports*

After a submission, the AF Maven plugin uses the system's default browser to open the initially "pending" feedback report, which automatically reloads once the build completes, showing a report like the one in Figure 3. Students can retrieve their uploaded file, view previous attempts, and know the model solution version that their code was tested against.

Regarding the simple feedback components from Narciss (Section 2.1), KR is provided through the "Status" field (Completed/Failed), and KP is provided in two ways: as an "Overall mark" (where each test can be worth a certain amount of marks), and as separate counts of tests that passed, failed, crashed, were skipped, or were missing from the JUnit XML (usually due to tests not compiling as the student missed some of the lab requirements). KCR is not provided by AutoFeedback: it is up to the

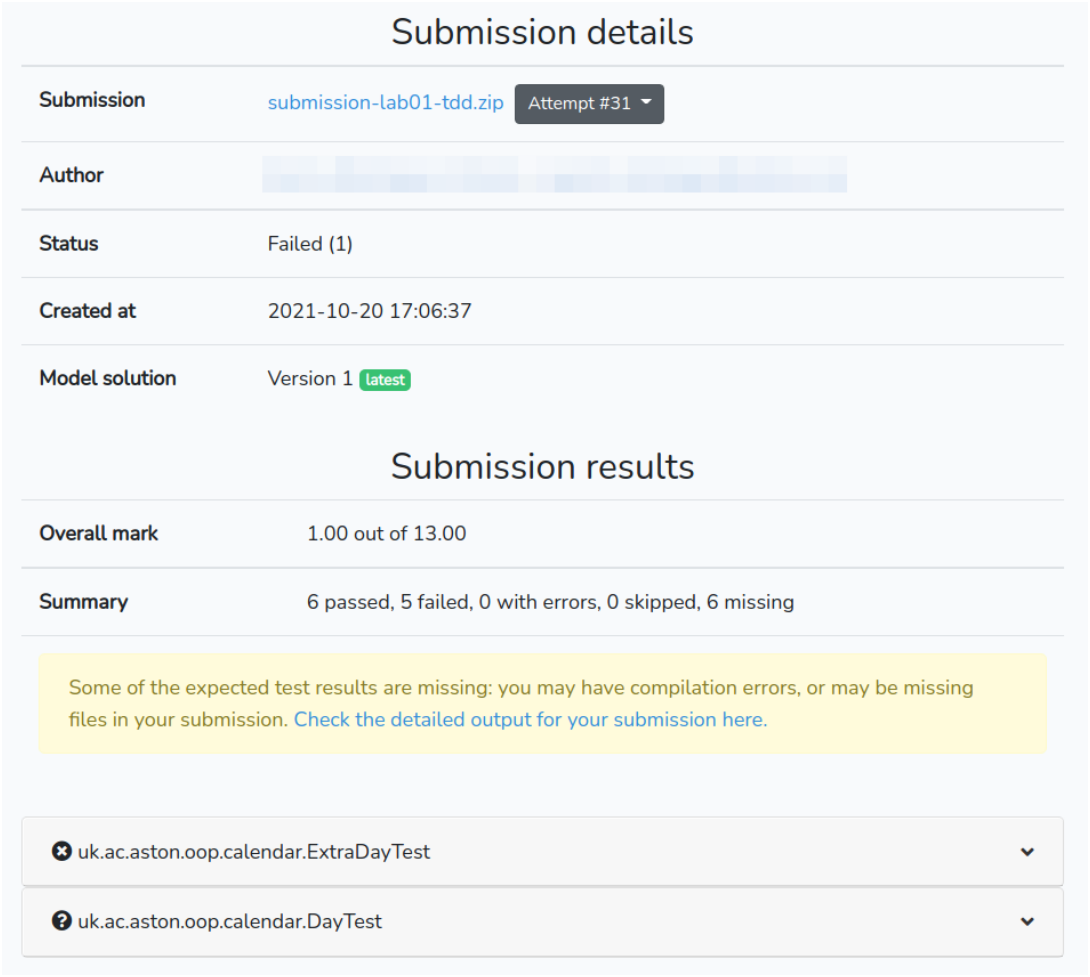



**Figure 3.** Screenshot of a test report, with both test suites collapsed

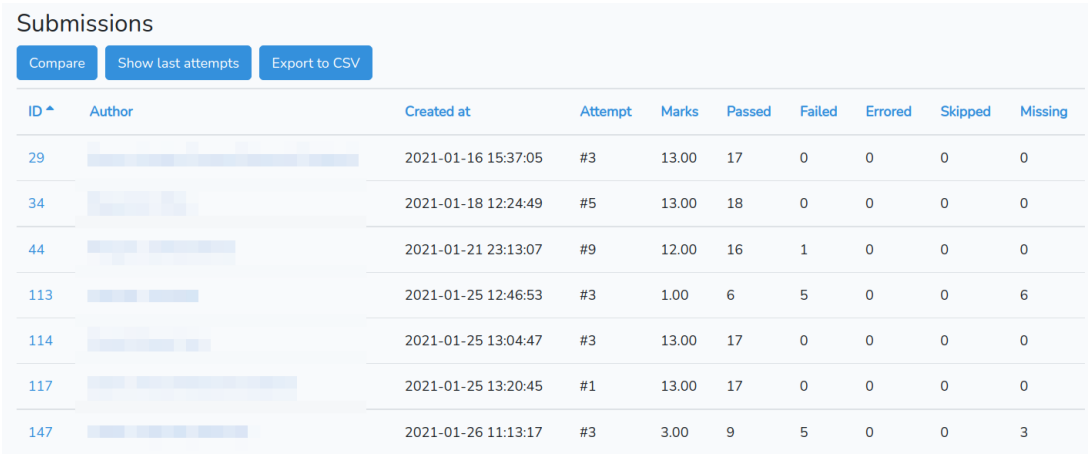



**Figure 4.** Screenshot of an AutoFeedback submissions table, with attempt and test result counts. User names have been distorted for privacy.

teacher to provide a model solution through other means (e.g. via their virtual learning environment), if they desire.

The rest of the report is dedicated to elaborated feedback. Looking at Table 1, the relevant components are distributed as follows:

- KTC can be provided in two ways: by associating tests to specific tasks in the worksheet (in which case the test will be in a collapsible drawer named after the task, instead of being organised by test class), and by reminding students of what the test requires them to do in the feedback template.
- KC is provided through the feedback template, by including examples of relevant code (as in the simple example of Listing 1), or links to the appropriate course material (e.g. when reminding students of how to compare strings in Java).
- KM is provided at a basic level by AutoFeedback itself, which indicates which tests passed and failed (KM-TF), and includes test outputs including error messages (KM-SE, KM-PI). KM-CE (compiler errors) can be collected per-test if the Maven build is set up to use the Eclipse Java compiler (c.f. Section 4.2). The teacher can use the conditional fenced blocks in the feedback templates to expand upon this basic level of KM (e.g. explain a specific failed assertion in more detail, perhaps with a supporting diagram).
- KH is entirely reliant on the conditional fenced blocks in the feedback templates. The teacher may show a typical mistake that causes a given test assertion to fail, and a strategy to correct that mistake. The teacher may also structure a test to have multiple assertions that represent the various steps of a task, and use a conditional fenced block to clarify what the next task should be after passing the first few assertions.
- KMC is also reliant on the feedback templates. In this case, the teacher can suggest a particular way to think about the problem if a test which is observed to be challenging to pass, or potential sources of information.

### *3.5. Teacher reports*

Collecting the individual test successes and failures of every student allows instructors to identify common mistakes and challenging tasks, and conduct the external


**Figure 5.** Screenshot of AutoFeedback progress treemap, with ratios of passed (green tick) / failed (red cross) / errored (yellow exclamation) / missing (black question mark) runs per test (using only latest attempts).

adjustment of the external feedback loop in the ITF model. AF provides a table of submissions with attempt and test result counts (Figure 4), and a dashboard with the relative ratios of passed, failed, errored, and missing results across all test cases (Figure 5). Instructors can check this information periodically to refine test suites and feedback templates based on the student experience.

The submissions table allows all columns to be sorted in ascending and descending order, allowing for quickly finding the latest feedback report for a particular student (e.g. when clarifying feedback to that student), students failing tests despite many attempts, or students passing all tests except for one (which may be brittle, or where the worksheet may be unclear). Sorting by time gives an idea of student progress over the week. It is possible to only show the latest attempt from each student, and the table can be exported to a CSV file for *ad hoc* analysis.

The progress dashboard gives a general idea about which tests are being particularly difficult to pass for the cohort, using green for passing executions, red for failed, yellow for errored, and black for missing ones. Each color has a matching icon, to help users with color perception disabilities. Instructors can click an area to list its submissions. In Figure 5, the large proportion of failed and errored runs suggests that the `makeAppointmentsOverlap` test in the ExtraDayTest test suite was particularly difficult, and could benefit from further clarification. The sizable ratio of "missing" tests in black was due to AF being optional in its pilot year, with a number of students deciding not to use it.

## 4. Piloting AutoFeedback

AF was piloted in term 2 of the first-year object-oriented programming module of the Aston University Computer Science course during the 2020–21 academic year. After positive reception by students and staff, it was made compulsory for 2021–22. This section describes the module in detail, and the integration of AF into it.

### *4.1. Module description and evolution*

The CS1OOP module at Aston University spans the whole academic year, and is designed to introduce students to object-oriented programming, without expecting any prior programming experience. Most of the material for the module (lecture notes, slides, quizzes, and lab instructions) is hosted in the University's centrally-managed virtual learning environment (VLE), Blackboard.

In 2020–21, the students were assessed through a final exam and a set of multiple-choice quizzes: quiz marks only counted with a valid attempt at the corresponding lab in their portfolio. In term 1 (October to December), students learn basic programming by using the Processing software sketchbook system (Reas & Fry, 2006). In term 2 (February to April), students migrate to the Java programming language, and use a dedicated Java IDE (Eclipse) to write code. Term 2 covers inheritance, interfaces, UML, object-oriented design patterns, the Java Collections Framework, the JavaFX graphical user interface toolkit, exceptions, and file input/output.

During the 2020–21 year, the module was changed to introduce the Eclipse IDE before the term 1 holidays (allowing more time for transitioning from the beginner-friendly Processing environment to a full IDE), and use week 1 of term 2 to introduce automated unit testing and test-driven development. Lab 1 of term 2 demonstrated these concepts through JUnit and AutoFeedback, with step-by-step guidance on submitting code and interpreting feedback. Being a pilot year for AF, it was made optional in case student reception was poor, or there were performance/stability issues. Students were encouraged to use AF to obtain rapid feedback before doing their final submission to Blackboard (which would be their official submission to the assessment), and were told that submissions that passed all the AF checks would be considered valid in regard to quiz marks (without requiring manual review).

In 2021–22, given the initial success of AF, term 2 quizzes were replaced by using AF to compute weekly portfolio scores, perceived by students as a more transparent approach. This made AF compulsory during term 2, although students still had to separately do their official submission to Blackboard (due to organisational constraints, it was not possible to have AF directly interface with Blackboard).

Due to the COVID-19 pandemic, students completed laboratories from home during 2020–21: students joined a timetabled online session where lab tutors explained the task, and students could ask questions and receive one-to-one support. Given the increased friction in obtaining help from teaching staff in an online setting (requiring resolving connectivity, screensharing, and audio/video issues before a simple question could be asked), AF provided a convenient way to get rapid feedback at any time. From 2021–22 onwards, laboratories were conducted on campus again.

### *4.2. Designing and refining materials for AutoFeedback*

Each lab was redesigned in 2020–21 around AF, turning it into a pair of Maven projects: a model solution with all tests to be run, and a stripped-down version to be used as a starting point by students. Over 2020–21 and 2021–22, model solutions were refined in several ways to deal with issues found by students or through the reporting facilities in Section 3.5. The labs are listed in Table 2, along with an overview of their changes over the years: further detail is provided below on the most impactful changes.

***Feedback in the presence of test compilation errors.*** The first lab worksheet had students use a test-driven approach to develop a calendar appointment booking

| Name | Changes during 2020–21 | Changes during 2021–22 |
|---|---|---|
| Lab 1: test-driven development | Use ECJ for per-test compilation error feedback. | Separate tests by worksheet step, and refine tests for autograding. |
| Lab 2: inheritance | Use JavaParser to check Javadoc quality. | Use ECJ as well, refine tests. |
| Lab 3: interfaces 1 | Refine model solution to be more flexible. | Separate tests by worksheet step, use ECJ, refine tests. |
| Lab 4: interfaces 2 | Clarify worksheet based on student feedback. | Same as above. |
| Lab 5: UML | PlantUML-specific assertion library to specify expected classes. | Same as above. |
| Lab 6: design patterns | Use Mockito for interaction-based testing. | Refine tests to require full functionality for marks. |
| Lab 7: collections | Use JavaParser to ensure JMH is correctly used for micro-benchmarking. | Add new test and make other tests less brittle against irrelevant case differences. |
| Lab 8: JavaFX preparation | Use of TestFX for flexible auto-grading of GUIs. | Use ECJ, add support for JavaFX 17. |
| Lab 9: JavaFX assessment | Same as above. | Same as above. |
| Lab 10: I/O and exceptions | Use JavaParser to flag common mistakes. | Refine tests and worksheet. |

**Table 2.** Laboratory session plan for CS1OOP, with changes over 2020–21 (the migration to AF), and 2021–22 (iterative refinement).

routine. Students created their own tests (in addition to those in the model solution), and AF ran both solution and student tests.

This initial lab showed a weakness in the approach at the time: Maven builds are usually designed to "fail fast", e.g. as soon as any line of code fails to compile. This is problematic when running tests on student code that may not follow all the indications in the worksheet (e.g. exact function names), meaning that some of the instructor-provided tests may not compile. Without test results, AF would only present the raw build errors, without any generated feedback. The model solutions were refined to use the Eclipse Java compiler[4], which can run Java code even with some compilation errors: upon reaching a line that failed to compile, an exception is thrown and shown to the student as the cause of failing that test.

***Looking inside the code.*** When discussing object-oriented design, code documentation, or the use of certain APIs, it is necessary to inspect the code rather than simply run it and compare the results with an expected output. As an example, Mockito[5] was used to test that students implemented object interactions appropriately when applying object-oriented design patterns. JavaParser was used to evaluate the quality of their Javadoc comments, and to find known common mistakes (e.g. the incorrect use of == instead of `Object.equals()`), by writing visitors which performed custom lab-specific static analysis on the students' code. These uses of JavaParser served to conditionally display elaborated feedback around *knowledge of mistakes*, detecting com-

[4] `https://www.eclipse.org/jdt/core/`
[5] `https://site.mockito.org/`

mon anti-patterns in student submissions and explaining why they were problematic to the students.

***Evaluating UML diagrams.*** The UML lab used AF to check that students are creating valid UML class diagrams for a problem domain, using the PlantUML textual notation[6]. The AF model solution lab used the PlantUML export to the XMI format (based on XML), and the tests checked that the expected elements were present in the XML document through an in-house assertion library for UML class diagrams. The assertions automatically generate human-readable test failure messages if the student's diagram does not meet the expectations (e.g. a certain type was not found, a given association end does not have the expected type of aggregation, or an operation parameter is missing or has the wrong type).

Compared to the approaches mentioned in Section 8.2, the assertion library is only intended for a subset of Class 2 problems (according to Le et al.'s classification), as it only works with a specific model solution (but allows some variability in naming). This is a good match with the purpose of this first-year UML lab, which was only to familiarise students with the UML notation as a blueprint for writing code, and how to use PlantUML to draw those diagrams: performing object-oriented design from natural language requirements is taught in the second year. Improving the assertions to allow more variability (e.g. by using semantic similarity and/or by allowing multiple solutions) to allow a broader range of Class 2/3 problems is part of our future work.

***Evaluating JavaFX GUIs.*** The module teaches the JavaFX graphical user interface toolkit, using an approach where the GUI is designed visually and saved as an XML-based document, which is then integrated with a Java "controller" class following certain conventions. JavaFX is taught over two labs: a preparatory one, and an assessed one worth 40% of the lab marks for term 2 of CS1OOP.

The AF model solutions for both labs included two types of tests: some checked the XML document (e.g. presence of expected GUI components), and others ran the GUI and interacted with it automatically by using TestFX[7]. The TestFX-based tests are robust to minor differences between submissions, by following high-level instructions such as "click on the Create button, ignoring case".

## 5. Methods

Having redesigned the CS1OOP module to use AutoFeedback, several research questions were raised about its impact on the students:

- (RQ1) How did AutoFeedback impact their engagement with the course, and how much did students engage with AutoFeedback across both years?
- (RQ2) How well did students perform when supported by AutoFeedback?
  - (RQ2a) How did using AutoFeedback and the change in course policies affect student results?
  - (RQ2b) How did their marks improve through resubmissions?
- (RQ3) How did students perceive the provision of automated elaborated feedback by AutoFeedback? Was it perceived negatively, neutrally, or positively?

[6] `https://plantuml.com/`
[7] `http://testfx.github.io/TestFX/`

| Feature | 2019–20 | 2020–21 | 2021–22 |
|---|---|---|---|
| **Delivery style** | On-campus | Online | On-campus |
| **Students enrolled** | 257 | 321 | 341 |
| **# w/Blackboard submissions** | 192 | 245 | 262 |
| **# w/AF submissions** | N/A | 240 | 269 |
| **Use of AF** | N/A | Optional | Mandatory (encouraged interim submissions for early feedback) |
| **Recorded submissions** | N/A | Last one | Last 30 + best before deadline + best overall |
| **# AF submissions** | N/A | 18,900 | 39,100 |
| **Use of quizzes** | Practice W1, then W3, W4, W6, W10, W14 | Every week | None |
| **Marks source for labs** | Quizzes | Quizzes (if tests passed) | Tests |

**Table 3.** Comparison table of CS1OOP editions under study

The following subsections will discuss the methods used to answer each of these research questions. The anonymized and aggregated data used for these analyses, as well as the supporting scripts and tooling, is available from the Aston University institutional data repository (Garcia-Dominguez, 2022).

### *5.1. RQ1: student engagement*

For RQ1, which dealt with the level of engagement of the students, we used the automated data collection and reporting of the enrolment systems, Blackboard and AutoFeedback. We focused on three editions of the module (summarised in Table 3):

- 2019–20 was the last edition before AutoFeedback was introduced. There were 257 students enrolled (88.6% male, 11.4% female). The average age was 19, with 64.9% aged under 20, 32.1% aged 20–24, and 3.0% aged 25+.
- 2020–21 was the year AF was piloted, with 321 students enrolled on CS1OOP (85.7% male, 14.3% female). The average age was 19, with 79.0% aged under 20, 18.6% aged 20–24, and 2.4% aged 25+.
- 2021–22 was the year AF was made compulsory, with 341 students enrolled (87.4% male, 12.6% female). The average age was 19, with 77.5% aged under 20, 21.7% aged 20–24, and 0.8% aged 25+.

We collected the number of students that made submissions to each of the submission points in Blackboard across the three years. In 2019–20, students went through a practice quiz in week 1, took 5 quizzes in weeks 3/4/6/10/14 (week 14 being right

after the Easter holidays), and took the JavaFX assessed lab in week 9. 2020–21 used 10 weekly labs and quizzes (to be done after completing the lab), and 2021–22 only used 10 weekly labs. We excluded the optional preparatory lab for JavaFX on week 8: it did not have a submission point in 2019–20 and 2020–21, and its optional nature means that many students may not have used it in 2021–22.

To compare the engagement profiles across the three years, we computed the proportion of students that made submissions over the various weeks of the course, both against the total number of students enrolled (listed above), and against the maximum weekly engagement seen that year. In 2019–20, the maximum weekly engagement (measured in number of students that submitted to a Blackboard submission point in one of its weeks) was of 192 students. This was 245 students in 2020–21, and 262 in 2021–22.

From AutoFeedback, we obtained the number of attempts that each student had made for each lab, spanning around 18,900 submissions for 2020–21, and 39,100 for 2021–22. The significantly higher number of submissions in 2021–22 is likely due to several factors: the larger cohort, the fact AF was compulsory, and the refinement of the worksheets to encourage students to submit at certain points to detect issues earlier. Students could submit as many attempts as desired, as we did not want to create undue stress on students around running out of attempts, and the queue-based approach ensured every submission would be evaluated at some point. In total, AF was used by 240 students in 2020–21, and 269 students in 2021–22[8].

To evaluate the impact of these changes in the use of AF on the number of attempts made for each lab, we used a Mann-Whitney U test with the null hypothesis that the students had made similar numbers of attempts in 2020–21 and 2021–22, and computed $A$ effect sizes (Vargha & Delaney, 2000) to evaluate the intensity of the change. Given the definition of the Vargha and Delaney $A$ effect size as $A_{12} = P(X_1 > X_2) + .5P(X_1 = X_2)$, where $X_1$ would be 2020–21 attempt counts, and $X_2$ the 2021–22 ones. A value of 0.5 would mean $X_1$ and $X_2$ were stochastically equal, whereas 0.2 would mean that the values of $X_1$ were consistently smaller than those of $X_2$.

### *5.2. RQ2: student performance*

For RQ2, which addresses how well students did at the various labs in the presence of AutoFeedback, we took into account the changes in the labs and the use of AF from 2020–21 to 2021–22. In 2020–21, a student needed to obtain full marks in AF to have their quiz mark count[9], whereas in 2021–22 the quizzes were dropped and the lab marks were directly obtained from AF.

This change in policy means that the question can be further subdivided into two: i) whether the change in policy resulted in different distributions of marks across years, and ii) whether the students improved their results in later submission attempts. Each subquestion is discussed separately below.

#### *5.2.1. RQ2a: effects of AF and change in policy*

To compare the results across both years, we used the proportion of achievable marks that were obtained, as a number between 0 and 1. We conducted Mann-Whitney U

[8]The fact that there were more students submitting to AF than to Blackboard in 2021–22 can be attributed to some students dropping out of the course early on, without submitting to Blackboard: there were at most 260 unique AF users from Lab 4 onwards.

[9]Since AutoFeedback was optional in 2020–21, if a student did not use AF, the module staff put the student's code from the virtual learning environment through the same tests as in the AF model solution.

tests across the 2020–21 and 2021–22 results of each lab with the null hypothesis that the results had been similar, and computed $A$ effect sizes.

It is important to note that there are some other factors besides this change in policy, which may interfere with the comparison. First, since marks were going to be computed directly from AF from 2021–22 onwards, the automated tests had to be refined together with their mark values to ensure some basic rules were adhered to: i) submitting from the initial code should not grant any marks, and ii) marks should increase gradually as the student completes sections of the worksheet. Secondly, the situation around the ongoing pandemic changed how labs were delivered (remote in 2020–21, and back to being on-campus in 2021–22).

#### *5.2.2. RQ2b: improvement over resubmissions*

We were only able to study the evolution of the students' results over successive attempts for 2021–22, as AF only stored the latest attempt of each student in 2020–21. We considered the last 30 attempts made by each student, which always included their highest-scoring attempt ever, and their highest-scoring attempt before the assessment deadline. We had to set this limit due to disk space constraints: the value was computed based on the median number of attempts in 2020–21.

To check if students improved their marks over successive attempts, Kendall association tests were run between the attempt numbers and the obtained mark ratios (as numbers from 0 to 1) over all submissions stored for each lab in 2021–22. Kendall tests were chosen as they were non-parametric, dealing better with potential skews in the achieved marks which would not meet the assumptions of Pearson correlation tests. In light of how Kendall's $\tau$ measures the concordance of rankings, labs with higher $\tau$ values would have marks improve more consistently on successive attempts across students.

We complemented this Kendall test with a visual analysis of the distribution of the relative changes in the ratio of achieved marks, using box plots: intuitively, if students were improving over attempts, we should see the boxes centered around positive values.

### *5.3. RQ3: student reception*

For RQ3, dedicated to studying how students received the new AF system, we used the existing surveying facilities in Blackboard to obtain feedback from the students in 2020–21 and 2021–22. This was done by sending announcements to the entire class, inviting them to voluntarily take part in the survey. Two such anonymous surveys were conducted in 2020–21, and one in 2021–22:

- The first survey was on week 4 of term 2 of 2020–21, shortly after AutoFeedback had been introduced. It covered the whole module, only asking students two questions: what they liked so far, and what could be improved. Most responses focused on the content and method of delivery, but some mentioned AF directly.
- The second survey was on week 10 of term 2, near the end of the term. This was specific to AutoFeedback, containing the following questions:
  (1) What did you like about AutoFeedback?
  (2) Did you have any issues using AutoFeedback?
  (3) How much do you agree with this statement? "AutoFeedback helped identify the issues in my lab code."
  (4) If you disagreed, could you tell us why?

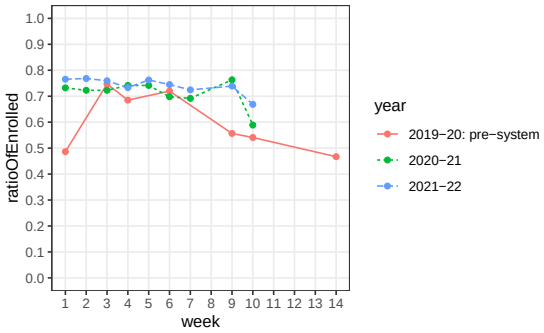



**Figure 6.** Participation ratios over the weeks for 2019–20 (before AF was introduced), 2020–21 (pilot with optional use of AF), and 2021–22 (mandatory use of AF). Measured as ratio of students who made a submission to Blackboard over the 10 weeks, over those enrolled in the course.

(5) How much do you agree with this statement? "The feedback from AutoFeedback was easy to understand."
(6) If you disagreed, could you tell us why?
(7) What is your preference regarding adding short videos to the AutoFeedback feedback? (No strong preference / I would like both / I prefer only text / I prefer only videos)
(8) Would you like to see AF used in other modules?
(9) Would you prefer the feedback to be organised in some other way?
(10) Where else do you think AF (or our use of AF) could be improved?

- The final survey was done on week 5 of term 2 of 2021–22, using the same questions as in the above survey to compare reception of AF after implementing most of the improvements requested in 2020–21.

The surveys contained a mix of free-form and multiple-choice questions (MCQs). For the free-form questions, in order to obtain a unified set of findings across all responses, thematic analysis was conducted according to the recommendations of Braun and Clarke (2006), using a data-driven approach to generate the initial codes, grouped later into themes. A single coder (the first author of this paper): after a pass, all responses were revisited in case they met one or more of the codes discovered in this round. This process was repeated until no more changes were required for the encoding. The second author compared the encodings against the responses, validating that they were accurate and correct.

## 6. Results

This section presents the results of the research questions in Section 5, based on the methods described for each research question.

### *6.1. RQ1: student engagement*

#### *6.1.1. Participation ratios*

Figure 6 shows the student participation ratios for all three years. Comparing the shapes of the various lines, it can be seen that before AF was introduced, the maximum engagement happened in week 3 (the week of the first quiz worth marks), being

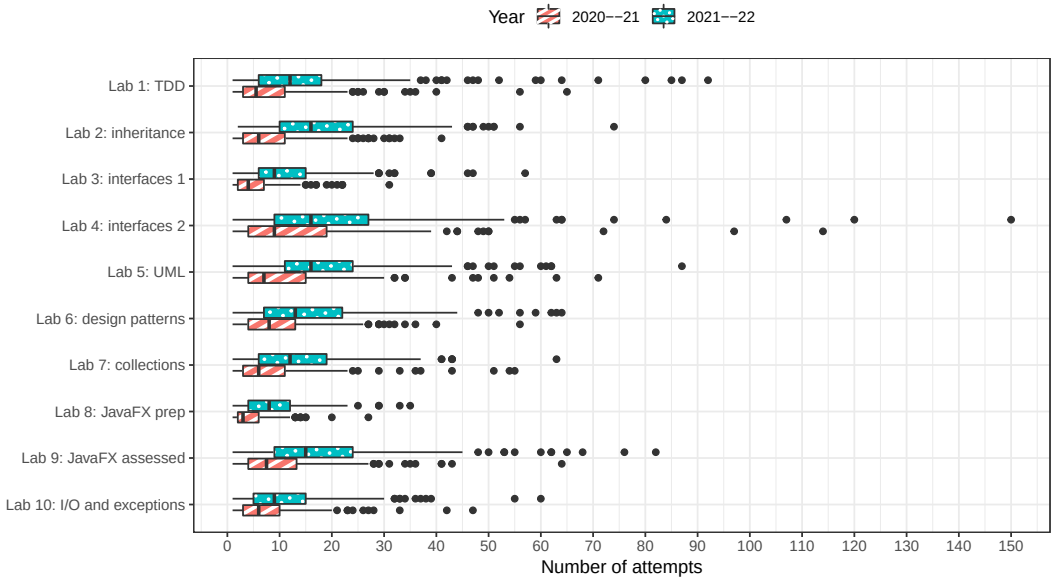



**Figure 7.** Box plots for submission attempt counts, by lab and year: outlier dots represent data points over 1.5 times outside the interquartile range.

similar to the maximum engagement with AF. Engagement was significantly lower in week 9 (the week of the assessed JavaFX lab) before AF was introduced, with only 62.5% of those students making submissions. In comparison, in 2020–21 the maximum engagement takes place in the assessed JavaFX lab week, remaining steady except for the last week. In 2021–22, the maximum engagement takes place in week 2, and then remains steady for the most part, with 96.2% of those students taking on the assessed JavaFX lab.

It can be seen that in general, the student engagement has been higher in 2021–22 than in 2020–21 and before AF was introduced. Comparing 2019–20 and 2020–21, it appears that while engagement was higher in weeks 1/4/9/10, there were still some points where it was slightly lower (weeks 3 and 6). One likely reason is that students were able to freely take the quizzes in 2019–20 without having to submit code, whereas doing the 2020–21 quizzes required writing code which would pass the AF tests. This means that some students who were not otherwise engaging could simply try their luck in the 2019–20 quizzes, whereas this was not possible from 2020–21 onwards.

### *6.1.2. Attempt statistics*

Figure 7 shows the distributions of the numbers of attempts per lab and year. Attempts increased noticeably in 2021–22, with all box plots showing right-shifted quartiles and medians in comparison to 2020–21. The Mann-Whitney U tests rejected for each lab the null hypotheses that the populations of 2020–21 and 2021–22 had identical distributions, with all $p$-values less than 0.001, supporting the alternative hypotheses that they followed different distributions. $A$ effect sizes are large for Lab 2 (0.20) and Lab 3 (0.22), small for Labs 4/10 (0.34 and 0.36), and medium for the other labs (ranging between 0.27 and 0.33). The third quartile was around 15 attempts in 2020–21, whereas it was over 20 in 2021–22. This confirms that the students did take into account the encouragement from 2021–22 of using interim submissions for early feedback, and that

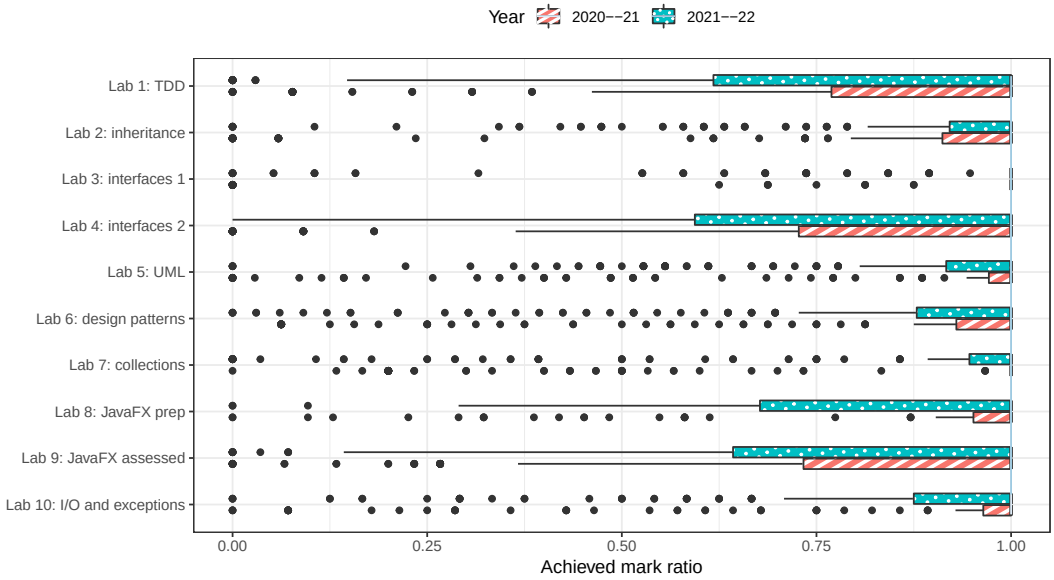



**Figure 8.** Box plots for achieved AutoFeedback mark ratios, by lab and year: outlier dots represent data points over 1.5 times outside the interquartile range. Medians are connected with a line across each year. According to the Mann-Whitney U tests on Table 4, only Labs 1/8/9 showed statistically significant differences between 2020–21 and 2021–22.

they continued to do so across the course.

There are noticeable outliers: for example, Lab 4 showed students with 107, 120, and 150 attempts in 2021–22. The number of outliers[10] per lab varied between 10 and 15 in 2020–21, and between 5 and 18 in 2021–22. A Wilcoxon signed rank test between the ratio of outliers per lab in 2020–21 (among its 321 enrolled students) and in 2021–22 (among its 341 enrolled students) could not reject the null hypothesis that they followed identical distributions ($p$-value = 0.13): there was no statistically significant difference between the two years in this regard. These outliers may point to students acting on an unproductive trial-and-error approach that would require additional support, and suggest that the teacher reports from Section 3.5 should be consulted periodically to detect these cases and intervene.

### *6.2. RQ2a: effects of AF and change in policy*

Figure 8 shows box plots of the ratios of achieved AutoFeedback marks for each lab in 2020–21 and 2021–22, where a ratio of 1 means that full marks were achieved. 2021–22 shows a broader range of marks, although many of these differences turned out not to be statistically significant: Table 4 shows that only the Lab 1/8/9 results were significantly different across 2020–21 and 2021–22 according to Mann-Whitney U tests (with $p$-values below 0.05), and even for those, the $A$ effect sizes are negligible as they are very close to 0.5.

The Lab 1 differences may be due to the tests being refined to be stricter in several locations, as the marks would be computed directly from AF rather than from the quiz: these may also explain the lower Q2 marks in other labs (e.g. Labs 4, 5, 6, 7, and 10),

[10]We consider a student to be an outlier if their number of attempts exceeded the third quartile plus 1.5 times the interquartile range.

**Table 4.** Mann-Whitney U tests with "achieved AutoFeedback mark ratios follow the same distribution in 2020–21 and 2021–22" as null hypotheses for each lab: rows with $p$-value $< 0.05$ reject these null hypotheses. $A$ effect sizes close to 0.5 are negligible.

| Lab | $p$-value | $A$ effect size |
|---|---|---|
| Lab 1: TDD | 0.02 | 0.55 |
| Lab 2: inheritance | 0.69 | 0.51 |
| Lab 3: interfaces 1 | 0.71 | 0.51 |
| Lab 4: interfaces 2 | 0.38 | 0.52 |
| Lab 5: UML | 0.87 | 0.50 |
| Lab 6: design patterns | 0.67 | 0.51 |
| Lab 7: collections | 0.68 | 0.51 |
| Lab 8: JavaFX prep | 0.04 | 0.56 |
| Lab 9: JavaFX assessed | 0.02 | 0.55 |
| Lab 10: I/O and exceptions | 0.26 | 0.53 |

**Table 5.** Kendall rank-based association test results between attempt number and overall mark, over the submissions stored in AutoFeedback for 2021–22, by lab.

| Lab | p-value | Kendall $\tau$ |
|---|---|---|
| Lab 1: TDD | <0.01 | 0.30 |
| Lab 2: inheritance | <0.01 | 0.44 |
| Lab 3: interfaces 1 | <0.01 | 0.41 |
| Lab 4: interfaces 2 | <0.01 | 0.42 |
| Lab 5: UML | <0.01 | 0.55 |
| Lab 6: design patterns | <0.01 | 0.38 |
| Lab 7: collections | <0.01 | 0.34 |
| Lab 8: JavaFX prep | <0.01 | 0.45 |
| Lab 9: JavaFX assessed | <0.01 | 0.34 |
| Lab 10: I/O and exceptions | <0.01 | 0.40 |

although as noted above, these differences were not statistically significant. The Lab 8 JavaFX preparatory materials were the same in 2021–22, so these differences may be due to the student cohorts themselves, or the change in the staff that supervised the laboratories in 2020–21 and 2021–22. Lab 9 had the same starting code in both years, but expanded upon it in a different way each year.

The median was 1 for all labs on both years: this can be seen in Figure 8 as medians are connected with lines, and the lines for each year are a straight vertical line across the 1.0 mark ratio. This means that at least half of the students achieved full marks, managing to pass all tests. The first quartile (Q1) differs between labs: for 2021–22, while Q1 was over 0.9 in Labs 2/3/5/7, it was only around 0.6 in Labs 1/4/8/9. Lab 1 is a test-driven development lab, requiring students to design their own tests for the first time in their careers, making it more challenging. Q1 for Lab 4 confirms the impressions from Section 6.1 that its use of many OO concepts (abstract classes and interfaces) makes it more difficult. Some students may not have engaged as much with Lab 8 (being optional), and Lab 9 (the assessed JavaFX lab) is done in exam-like conditions, where support is limited to technical issues with the lab computers.

### *6.3. RQ2b: improvement over resubmissions*

The Kendall association test results are shown on Table 5: all $p$-values are below 0.01, rejecting the null hypothesis that the two variables are independent, and supporting

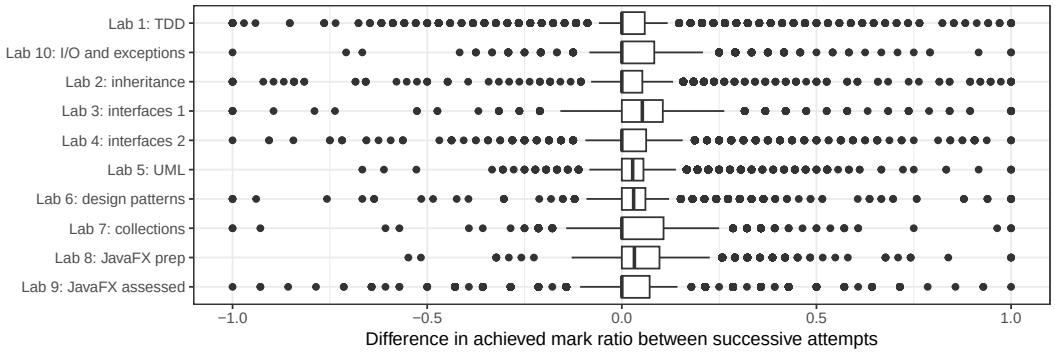



**Figure 9.** Box plots for differences in ratio of achieved/achievable marks across successive submissions from the same student for each lab in 2021–22, using the attempts stored by AF: the last 30 attempts, the best attempt before the deadline, and the best overall attempt. Outlier dots represent data points over 1.5 times outside the interquartile range.

the alternative hypothesis that a positive association existed between attempt numbers and obtained marks.

The actual $\tau$ values differ by lab, however. Lab 5 has the highest Kendall $\tau$, being mostly centered on teaching basic UML modeling via the PlantUML textual notation, and therefore not requiring a large amount of problem solving: students will typically improve their marks at a consistent rate across attempts. On the other hand, Lab 9 (the assessed JavaFX lab) has one of the lowest $\tau$ values, likely because it only specifies the requirements of the new GUI to be developed and does not provide step-by-step instructions: some students will pass most tests on their first few attempts, while others will require more attempts before they reach their final mark.

Figure 9 shows box plots of the relative changes in the ratio of achieved marks across successive attempts for each lab in 2021–22. In all labs, the first quartile is zero, meaning that for 75% of successive attempts, marks do not decrease. In Labs 3/5/6/8, the median is higher than zero: most successive attempts result in an improvement in the score. Lab 9 is one of the more difficult labs, with a zero median. There is a number of outliers, spanning from -1.0 to +1.0. A -1.0 delta usually indicates that the student went from a fully-working solution to one that failed all tests (e.g. due to a refactoring or submitting the wrong Eclipse project): these were rare, however, with only 27 cases out of 33094 data points. Conversely, many of the 50 instances of a +1.0 delta will be from the students restoring their old submission.

### *6.4. RQ3: student reception*

Tables 6–8 summarise the results of the surveys in Section 5.3, which will be discussed below. MCQs were analysed in a quantitative manner, by comparing the number of responses for the various options: Figure 10 shows the results.

#### *6.4.1. 2020–21 — Week 4 survey*

Table 6 shows 5 out of 20 participants mentioned AF. P11 (participant 11) and P20 were entirely positive, only mentioning how AF had helped point out issues in their code. P4 acknowledged its usefulness, but mentioned that the feedback could be unclear at times and that it needed to be revised to provide more clear guidance. P12 also considered AF useful, but mentioned a number of problems with the tests: they could

**Table 6.** AutoFeedback-related themes mentioned by participants in the 2020–21 term 2 week 4 module survey

| Theme | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 | P11 | P12 | P13 | P14 | P15 | P16 | P17 | P18 | P19 | P20 | Count |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **AutoFeedback** | | | | | | | | | | | | | | | | | | | | | |
| Feedback is unclear at times | | | | • | | | | | | | | | | | | | | | | | 1 |
| Useful to detect issues | | | | • | | | | | | | • | • | | | | | | | | • | 4 |
| Tests can fail even if the code is working | | | | | | | | | | | | • | | | | | | | | | 1 |
| Does not allow for all approaches to solve a problem | | | | | | | | | | | | • | | | • | | | | | | 2 |
| Passing all tests should not be mandatory to gain quiz marks | | | | | | | | | | | | • | | | | | | | | | 1 |

be brittle (failing even if the code met the requirements) or inflexible (requiring a specific approach/naming convention in order to pass). P12 considered that asking for all tests to pass before the quiz marks would count was too harsh, and that a more gradual approach was needed which allowed for some tests to fail (at the expense of some marks). Finally, P15 did not take well to AF, considering AF an impediment to their creativity in tackling problems however they wanted.

This initial response motivated a number of changes in strategy for 2020–21. Student submissions were regularly monitored to look for tests that proved to be particularly challenging: feedback templates were revised accordingly to guide students on the most common issues, brittle tests were improved to allow for a greater variety of solutions, and lab worksheets received clarifications in their steps. Several times, additional "code smell" tests would be introduced in the middle of the week, which used static analysis to explicitly detect and highlight recurring difficult-to-debug mistakes students were making (e.g. using == to compare strings in Java). Students were kept informed of these refinements and any other improvements in AF through regular announcements, to let them know that the staff was monitoring and improving the experience.

The above response also partly motivated the removal in 2021–22 of the "harsh" requirement to require all tests to pass to take the quiz: instead, the AF mark would be used as-is, meaning that students would still be granted partial marks if they managed to pass some but not all the tests. It was also one of the reasons for introducing in 2021–22 the conditional feeedback blocks of Section 3.3.

It is worth noting that there was some self-organisation from the students: a few weeks after this survey, a study group decided to tackle the labs as soon as they were released, and send feedback to instructors on the clarity of the worksheets and the brittleness of the tests, so the rest of the cohort would benefit. For instance, they identified a few cases where string comparisons were too strict in a number of cases in Lab 7 (dedicated to Java collections).

#### *6.4.2. 2020–21 — Week 10 survey*

Table 7 shows the themes identified across the 18 responses. Every participant had something positive to say, and 16 out of 18 participants stated that AF had helped point out issues in their code. 4 participants were happy that they did not have to wait for teaching staff to receive feedback. P17 praised the ease of use of AF, and P2 liked the concise feedback in the form of passed/failed tests. P17 recognised the efforts of the teaching staff in making tests more flexible in response to student feedback.

Students mentioned several issues in their negative feedback. 7 participants mentioned that tests would sometimes fail without providing a clear reason. In the first

**Table 7.** Themes mentioned by participants in the 2020–21 term 2 week 10 survey on AutoFeedback

| Theme | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 | P11 | P12 | P13 | P14 | P15 | P16 | P17 | P18 | Count |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Feature requests** | | | | | | | | | | | | | | | | | | | |
| Allow for providing feedback on the feedback | | | | | | | | | | • | | | | | | | | | 1 |
| Collapsible drawers for test classes | | | | | | | | | | | | | | | | | • | | 1 |
| Dark mode | | | | | | • | | | • | | | | | | | | | | 2 |
| Help locate the faulty lines of code | | | | | | | • | | | | | | | | | | • | | 2 |
| Only show the tests that failed | | | | | | | | | | | | | | • | | | • | | 2 |
| Reference worksheet step from feedback | | | | | | • | | | | | | | | | | | | | 1 |
| **Negative feedback** | | | | | | | | | | | | | | | | | | | |
| AF can have spikes in load | | | | • | | | • | | | | | • | | | | • | | | 4 |
| AF did not work for the student some weeks | | | | | • | | | | | | | | • | | | | | | 2 |
| Sometimes the student does not know what to do in response to the feedback | | | | | • | • | • | | | | | | • | • | | | | | 5 |
| Sometimes the student needed to reload the page manually | | | | | | • | | | | | | | | • | | | | | 2 |
| Teachers should not entirely rely on AF for assessment | | | | | | | | | | • | | | | | | | | | 1 |
| Tests need to be more flexible | | • | • | | | • | | | | • | | | | | • | | | • | 6 |
| Tests sometimes fail without providing a clear reason | • | | • | | • | • | | | | • | • | • | | | | | | | 7 |
| Waiting times for feedback can be long sometimes | | • | | • | | | | | | | | • | • | • | | • | | | 6 |
| **Positive feedback** | | | | | | | | | | | | | | | | | | | |
| AF helped point out issues | • | • | • | | • | • | • | • | • | • | • | • | • | • | • | • | | • | 16 |
| AF was easy to use | | | | | | | | | | | | | | | | | • | | 1 |
| Minimal and concise feedback is good | | • | | | | | | | | | | | | | | | | | 1 |
| Tests were made more flexible in response to student feedback | | | | | | | | | | | | | | | | | | • | 1 |
| Not having to wait for teaching staff | | • | | • | | | • | | | | | | • | | | | | | 4 |

weeks of AF, this could happen when the test failed to compile. P6 also mentioned the case of several "failsafe" tests in Lab 4 (an object-oriented simulation) which were only testing that the student had not inadvertently introduced a regression in the starting code, and simply told students in the feedback template that they should be passing the test from the start of the lab, unless they had made a mistake in a previous step: the feedback templates for these failsafe tests did not mention what behaviour they were expecting, which did not help students figure out what they had broken.

The next most prevalent issue was still the brittleness of the tests: 6 participants mentioned how tests needed to be more flexible, as they could fail because of minor spelling mistakes, slightly different text strings being produced by their programs (e.g. case or whitespace differences), or differences in method/variable names.

Six participants mentioned long waiting times, and four mentioned how AF could have spikes in workload (especially on weekends, as the submission deadline drew near). This can sometimes result in build queues that are longer than desirable. In 2020–21, AF had 2 build nodes allocated to it: this was raised to 4 build nodes in 2021–22.

Five participants touched on how sometimes they did not know what to do in response to the feedback. They considered that in some cases, the wording of the feedback was very vague or was worded in terms they found hard to understand as first-year students. They asked for the feedback to be made a bit more specific, without entirely giving out the answer. Students often emailed module staff to ask for clarification on the feedback, which resulted in refinements of the feedback templates, but the friction involved in writing an email may have discouraged some students.

The next two items of negative feedback discussed more rare issues with AF, with two students mentioning how it did not work for them in some weeks, or that they needed to reload the feedback page manually if their connection was unstable. Regarding the

first issue, all reported problems prior to this survey had been resolved: these may have been issues that went unreported to the teaching staff until this survey. The second issue was related to issues in a library.

Finally, one student commented that teachers should not rely entirely on AF for assessment, with a specific focus on the assessed GUI lab. To mitigate the risk of the automated tests producing unfair marks on the GUI lab, the teaching staff sampled the submissions that had not achieved full marks, and moderated the AF-produced mark by inspecting the submitted code manually. Any issues detected on the automated tests were resolved, and the automated marking was rerun on all submissions that had not achieved full marks: students were given the highest grade across their attempts.

The third and last theme consisted of feature requests. Several were implemented in the summer between the 2020–21 and 2021–22 academic years, including: i) a "dark mode" easier on the eyes if working late at night (requested by two students), ii) collapsible drawers for test class results (requested by one student), and iii) a way to reference the worksheet step from the feedback template (requested by one student). One student asked for the ability to reply to a specific item of feedback from AF (rather than having to write an email): this is part of our future work, as it may help aggregate information on which feedback templates need the most refinement. Two students asked for a view where only failed tests were shown: this has been partially implemented, by automatically showing test classes that have passed all tests in "collapsed" form. Finally, two students asked for assistance on locating the problematic lines of code that caused a test to fail: this can be very challenging to achieve generally in an automated manner, but there are promising results in the area of mutation-based fault localization that could be reused, such as the work of Papadakis and Le Traon (2015) in their Metallaxis tool. Automated fault localisation could reduce the need for manually crafted conditional feedback blocks in response to typical error messages (which can take significant amounts of effort).

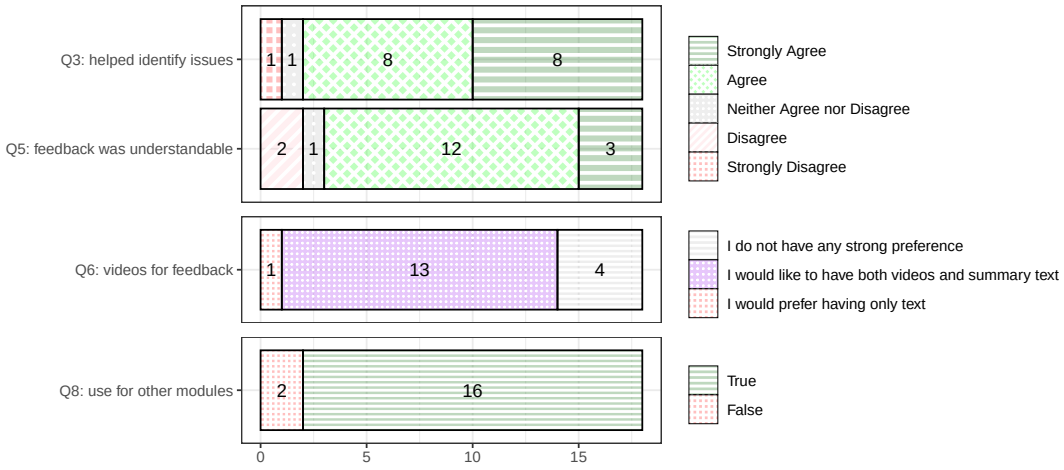
The answers to Questions 3, 5, 7 and 8 have been summarised in Figure 10a. In general, most students agree or strongly agree that AutoFeedback helped identify issues (16 out of 18) and that the feedback was understandable (15). Students generally would prefer to have both videos and summary text for their feedback in future editions of the module (13), and no students said that they would want only videos. Most students (16) wanted to see AutoFeedback used in other modules.

#### *6.4.3. 2021–22 — Week 5 survey*

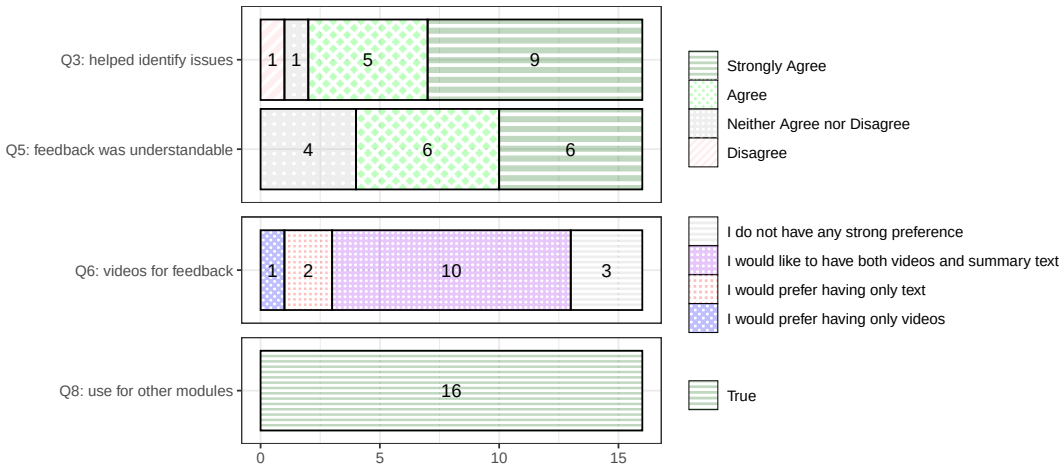
Table 8 summarises the themes identified in the 16 responses. To allow for easier comparison, the thematic analysis started from the themes identified in the 2020–21 week 10 survey, and new themes have been annotated with "(new)".

In regard to positive feedback, 12 participants mentioned how AF helped them point out issues in their code, 24 liked not having to wait for teaching staff to receive feedback on their code, and 2 considered AF was easy to use. Three participants liked how tests were clearly linked to worksheet sections, which is a new feature introduced in 2021–22: this is the grouping by "tasks" mentioned in Section 3.2. Two participants mentioned how passing the tests felt rewarding, and that AF was giving them a push in the right direction while completing the exercises.

Several of the negative feedback themes from the previous survey are not mentioned anymore, such as the workload spikes, or seeing tests fail for no apparent reason. The most frequent negative theme is still not knowing what to do in response to the feedback, as some feedback templates are still considered to be worded in overly




(a) 2020–21 week 10 survey




(b) 2021–22 week 5 survey

**Figure 10.** Responses to multiple-choice questions in the 2020–21 and 2021–22 student surveys

**Table 8.** Themes mentioned by participants in the 2021–22 term 2 week 5 survey on AutoFeedback

| Theme | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 | P11 | P12 | P13 | P14 | P15 | P16 | Count |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Feature requests** | | | | | | | | | | | | | | | | | |
| Help locate the faulty lines of code | | | • | | | | | | | | | | | | | | 1 |
| Option for only showing failed tests | | | | | | | | | | | • | | | | | | 1 |
| (new) Improve color scheme for readability | | | | | | | | | | | • | | | | • | | 2 |
| (new) Link from Blackboard to AutoFeedback | | | | | | | | | • | | | | | | | | 1 |
| (new) Student-facing performance dashboard | | | | | • | | | | | | | | | | | | 1 |
| (new) Web development module should use AF too | | | | | | | | | | | • | | | | | | 1 |
| **Negative feedback** | | | | | | | | | | | | | | | | | |
| Sometimes the student does not know what to do in response to the feedback | • | • | • | | • | | | | • | | | | | | • | | 6 |
| Sometimes the student needed to reload the page manually | | | | | | | | | | | • | | | | | | 1 |
| Tests need to be more flexible | | | | • | | | | | | | | | | • | | | 2 |
| (new) Sometimes tests do not notice intentionally introduced faults | | • | | | | | | | | | | | | | | | 1 |
| **Positive feedback** | | | | | | | | | | | | | | | | | |
| AF helped point out issues | • | • | • | • | | • | • | • | • | • | • | | • | | • | | 12 |
| AF was easy to use | | | | | • | | | | | | | | | | • | | 2 |
| Not having to wait for teaching staff to obtain feedback | • | | | • | | | | | | | | | | | • | • | 4 |
| (new) Passing tests feels rewarding | | | | | | | | | | • | • | | | | | | 2 |
| (new) Tests are clearly linked to worksheet sections | • | | | | | | | | | • | | • | | | | | 3 |

technical or vague ways, or lack concrete guidance on what needs to be done. This reinforces the need for students being able to reply to feedback and raise the instructors' attention to the feedback templates needing improvement. Being novice programmers, there is also the chance that they did not understand the new topic that the test depends upon: providing links to learning resources to address this gap in knowledge may be more productive than directly helping them with the task. A student may also try to skip a test and make later tests pass, which may lead to confusion over the feedback on later failing tests might not be helpful until the earlier test passes.

There was only one student asking for tests to be more flexible, which suggests year-to-year improvements were effective, but that further refinement is still needed. The specific cases given by the students were related to tests needing to be designed to fail independently of each other, or tests that suffered from JavaParser syntax trees drastically changing in the presence of minor textual variations (e.g. an empty line between a method and its Javadoc comment). Another example was a string comparison which was unnecessarily sensitive to whitespace. It is interesting to note that one student mentioned experimenting with AF by intentionally introducing faults and seeing if this resulted in AF failing a test, which did not always happen: this suggests that mutation analysis (recently surveyed by Papadakis et al. (2019)) could be used to refine the instructor-written AF tests to improve their sensitivity.

Only two feature requests from 2020–21 are still outstanding: automated assistance to locate the faulty code (a research topic of its own), and an option for only showing failed tests. Students have made a number of additional feature requests: i) improving the color scheme (better contrast in dark mode, and more accessible colours for users with color perception impairments), ii) providing a link from Blackboard to AF (which was quickly implemented), iii) providing students with a dashboard of the Java topics they need to reinforce based on their performance, and iv) having the first-year web development module use AF as well.

The MCQ responses are shown in Figure 10b. 14 students agreed or strongly agreed

that AF helped identify issues, and 12 students said that feedback was understandable (with the remaining 4 being neutral). Students still majorly preferred to have a mix of both videos and text for feedback (10 out of 16). Most importantly, all responses said they would like to see AF used in other modules.

## 7. Discussion

After describing the results in Section 4, this section will outline the lessons learned during these first two years of developing and using AF, and identify the threats to the validity of these observations.

### *7.1. Automated test-driven feedback for programming labs*

In general, student reception of automated test-driven feedback was positive, with some initial pushback due to the perception that it would severely constrain their freedom to solve assignments as desired. In this regard, it was important to ensure the students felt they were listened to when they raised an issue with the tests or the feedback templates. Modules incorporating automated test-driven feedback should ensure staff have the time needed to monitor student performance and feedback, and revise tests and feedback templates accordingly. Various forms of learner analytics should be provided to enable such monitoring, both at the individual level (e.g. comparing tests passed across attempts for a student) and at the cohort level (e.g. visualizing which tests are proving to be the most difficult). Ideally, there should be ways for students to initiate discussions with instructors on the feedback via the platform, avoiding the friction of having to start a separate email conversation.

First-year students are often getting to grips with producing syntactically valid code, and will react negatively to receiving no feedback at all if their submission is simply rejected with no output. If the automated feedback is test-driven, and not all tests are shared with the students (to prevent giving away parts of the solution), it should at least include the syntactic errors relevant to each test. Ideally, it should also attempt to run as many tests as possible, even if some other tests do not compile: the core principle is to give as much feedback as possible even if the submission is of low quality at the moment.

The proposed approach requires writing effective tests and feedback templates. Tests should be flexible enough to allow all reasonable solution approaches, while being clearly isolated from other tests (i.e. not failing because another test failed to run or compile), and setting out time limits for completion within a certain order of magnitude (the test designer will be usually able to set a more accurate limit than a platform's default timeout: sorting 10 list elements should take much less than the 10 minutes given by AF to submissions by default). Test designers should consider a variety of methods to check their expectations, such as comparing initial and final program states, verifying that the expected object interactions have happened, or performing test-specific inspections on the students' code (e.g. to detect common mistakes). Test designers may also benefit from automated approaches to assess the robustness of a test suite against common mistakes, e.g. by adopting approaches such as *mutation analysis* which measure how well the test suite detects small code variations or *mutations*.

As for the feedback templates, students preferred explanations in lay language, with reminders of the new terminology introduced in the module. Some of the raised issues were about the overly terse explanations limited to a short summary of the assertions

checked by the test in question. If a specialist testing framework is being used, the feedback templates may need to help students interpret the error message: this was often the case with failing Mockito tests, where students need to understand how their code did not result in the intended interactions. On the other hand, the amount of feedback being given must be kept within reasonable limits: to this effect, the conditional Markdown blocks helped show only the feedback that was relevant to the student.

### *7.2. Impact on student engagement and performance*

The participation ratios in 2020–21 and 2021–22 (after AF was introduced) were shown to be more consistent than in 2019–20, even across the policy change between 2020–21 and 2021–22. This suggests that automated test-driven feedback can motivate students to engage with the programming tasks more consistently than assuming they will perform any such tasks before a typical multiple-choice question quiz. Students mentioned multiple times liking how they could obtain feedback at any time.

The Kendall rank-based association tests showed that marks generally improved from attempt to attempt, though some labs proved more difficult. Most students were able to achieve more than 50% of the lab marks (Figure 8), and according to Figure 9 the median change in marks was never lower than 0, and was often greater than 0.

### *7.3. Design of the platform*

Students generally considered AF easy to use: there were no major issues raised during 2020–21 and 2021–22 beyond Lab 1. AF only used the existing facilities in their pre-installed development environments and browsers, did not require learning a version control system, and used the same credentials as Blackboard. This allowed students to focus on learning coding, leaving learning Git to the second year team project.

During 2020–21, the students complained about a lack of processing power during peak times, which was remediated for 2021–22. The worker-based design of AF would make it simple to add further processing power as needed should cohorts grow further, but due to its reliance on a simpler deployment technology (Docker Compose), it would not automatically scale up and down to the same extent as a cluster-oriented container orchestration platform like Kubernetes.

Students wanted to manage the amount of information shown at once (grouping tests by task and viewing only test failures), and tweak the colour scheme to suit their preferences (dark mode). A number of features will require further research, including support for automated fault localisation, creating links between the failed tests and the underlying theory to review, or using the feedback for exam revision.

### *7.4. Threats to validity*

The above findings are subject to several threats to validity, following the classification from Wohlin et al. (2012).

#### *7.4.1. Conclusion validity*

This type of threat is related to whether the insights gained follow from the obtained results (e.g. the observed results are statistically significant). A large part of the analysis has been based on descriptive statistics, especially quartiles and medians (due to their

robustness against outliers), although a number of statistical tests have been conducted. The comparison of the consistency of the participation ratios across 2019–20, 2020–21 and 2021–22 was visual in nature. For student performance, the initial comparison based on boxplots between the two years was supplemented with a Mann-Whitney U test to check if the differences were statistically significant, $A$ effect sizes to check if the differences were large enough, and a Kendall rank-based statistical test to validate if marks improved on later attempts.

To compare the achieved mark ratios between the two years, a Mann-Whitney U test highlighted the labs with significant differences (although $A$ effect sizes were negligible in those cases), and visual inspection of boxplots was used to identify trends. It is worth noting that many of the data points in Figure 8 are at the 1.00 maximum (students achieved all marks), which may impact the reliability of the Mann-Whitney U test and result in an underestimation of the $A$ effect size: there is a possibility that the effect was larger than measured by $A$. We originally expected to see larger differences from a change in policy, given that the purpose of the AF marks changed (from unlocking a quiz to becoming their performance indicators), and also how some of them were computed (tests were refined for this new purpose, as discussed in Section 5.2).

The analysis of the distribution of attempts needed to reach various mark thresholds was not considered to require statistical tests, as the analysis was on a lab-by-lab basis and no year-to-year comparisons were needed.

In regard to the student feedback surveys, the conclusions are largely dependent on how the responses to the students were coded, which may vary from researcher to researcher: this is a known issue with thematic analysis. Specifically, in this case the first author did the coding, and the second author validated it.

#### *7.4.2. Internal validity*

Internal validity considers the causal relationship between the treatment (e.g. the use and refinement of AutoFeedback and its materials) and the results (e.g. more consistent engagement, mark ratios, higher attempt counts in 2021–22). There may be unknown differences in the composition of the three student cohorts (2019–20, 2020–21, and 2021–22) that may have caused some of the differences across years: while entry requirements to the CS course have been essentially the same, the number of students has grown from year to year. In this regard, some consistency from year to year has been observed, e.g. the participation ratios were very similar between 2020–21 and 2021–22. Another possible internal threat is that the test suites could have defects that negatively or positively impacted the marks achieved by the students: to manage this threat, all issues reported by students on the test suites were promptly fixed, and the fix was confirmed with the students. Additionally, AutoFeedback includes an extensive automated test suite of its own to ensure that the results are correctly computed, recorded and reported.

In relation to the student feedback surveys, the main limitation is that due to participation being voluntary, response rates were lower than desirable. In 2020–21, out of 240 unique AutoFeedback users, the mid-module survey received 20 responses and the end-of-module responses received 18 responses. In 2021–22, out of 269 unique AF users, the mid-module survey received 16 responses. There is the risk that the responses could be skewed towards the more engaged and higher-performing students, and that the results could be more positive than if all students had responded. Due to the anonymous nature of the survey, it is not possible to verify whether the respondents were skewed in this manner, or whether they followed a similar distribution of results as the entire

cohort.

#### *7.4.3. Construct validity*

This aspect is concerned with the relation between the theory (e.g. the ideas of test-driven development and test-driven feedback) and observation (e.g. how they appear to students via the teaching materials and AutoFeedback). The students receive a lecture on TDD before being exposed to AF, and the lab worksheets regularly remind students to test their work so far and ensure certain expected tests pass before continuing. In order to make the experience as realistic as possible for students, AF reuses existing mature testing frameworks (JUnit, Mockito, TestFX) which have been widely adopted by TDD practitioners, and allows students to use any IDE of their choice that supports Maven. In regard to the implementation of automated test-driven feedback, the templates are limited to manually-written blocks which are conditionally shown to students based on their test results: alternative approaches which generate text automatically in some other way (e.g. via automated fault localisation) could perhaps have produced reports that were more insightful for an individual student (although they could also be more difficult to understand).

#### *7.4.4. External validity*

This last class of threat is concerned with the generalization of the results. AutoFeedback has only been used for one module of one CS course at a single university: evaluating its effectiveness in other cohorts (ideally at other institutions) is part of our future work. In addition, it has only been used in earnest with Java programs, although most of AF only requires that the test framework produces a JUnit XML file: a final-year project ran successful experiments integrating PHP and JavaScript support by leveraging this fact, and these improvements are waiting to be merged into the main AF codebase.

## 8. Related work

Automated provision of feedback for programming assignments has received significant attention over time. In this section, we discuss several tools similar to AutoFeedback, techniques for designing and evaluating tests, as well as other experience reports around automated assessment tools.

### *8.1. Autograding systems*

Our initial search before starting the development of AutoFeedback found the Gradeer and PABS systems. We identified several other systems after developing the first versions of AutoFeedback, including WebTA, Web-CAT, and Test My Code.

Gradeer (B. Clegg, Villa-Uriol, McMinn, & Fraser, 2021) is a recently published system for automated grading of student submissions, combining guided manual checks with automated JUnit test suites and linting tools like Checkstyle (2021) and PMD (2021): it is focused on grading rather than on providing rapid feedback.

PABS (Ifflander, Dallmann, Beck, & Ifland, 2015) is another automated grading system, which uses Subversion repositories to manage tests and student submissions. Submissions are assessed by Akka agents running Gradle builds, and the approach has

been successfully used since 2010 over 19 courses with almost 1300 students. On the other hand, PABS required students to commit their code to SVN, and the feedback was limited to test results: module staff could not add custom hints on typical mistakes that fail tests. PABS is not publicly available as open source software.

WebTA (Ureel & Wallace, 2015) is a web-based system for automated critiquing of student programs in introductory student courses. Similarly to our intended approach, WebTA supported a feedback loop where students receive feedback from predefined "critiques", which help instructors focus on providing feedback at a deeper level beyond common issues. Students submit Java source code through a web interface, and this code is checked against instructor-defined tests (implemented in JUnit) and critiques (implemented as regular expressions). Instructor-defined tests could provide hints to students upon failure. In addition, WebTA integrated with the authors' institution's Learning Management System (Canvas), providing grades. The authors reported higher mean and median grades after WebTA was introduced in a first-year introductory programming course, and a second-year data structures course. While the idea of having a large predefined library of code antipatterns is attractive, our experience using regular expressions for automated coding quizzes suggests that they could be brittle against minor syntactic variations in student code: ideally, they should be implemented as pattern matching over abstract syntax trees.

Web-CAT (Edwards, 2003) is one of the most feature-rich alternatives to AutoFeedback that is available as open-source software. Web-CAT is a web-based platform where students can submit code (whether from their browser or their IDE), and one or more grader "plugins" can process that code to generate grades and feedback. Web-CAT graders measure three scores: test validity (how many student-provided tests are passed by the reference solution), test completeness (how well the student-provided tests cover the student-provided solution), and code correctness (how many student-provided tests are passed by the student-provided solution). This allows for designing assessments where students are responsible for creating tests that verify the requirements, instead of only relying on instructor-provided tests. Web-CAT includes a TDD-driven Java grader plugin, which in its current version automatically tests and analyses programs using a predefined Ant script, using (Checkstyle, 2021) and (PMD, 2021) to perform static analysis. While Web-CAT also runs tests and produces feedback, similar to AutoFeedback, there are a number of key differences:

- We have not found any features for fine-grained cohort-level analytics (e.g. finding out which instructor-provided tests were giving students more trouble). It appears that plugins only generate reports for individual submissions.
- The Java TDD plugin only uses a predefined build script. Teachers would typically want to provide custom build scripts for each assignment (with the ability to automatically fetch dependencies from public software repositories), and add an extra layer of isolation by running student code in unprivileged and resource-limited Docker containers.
- Hints provided to students from the Java TDD plugin are directly extracted from the test assertions in the instructor-provided tests, limiting the level of detail that could be provided in them, and the available formatting capabilities (as these are typically short Java strings). AutoFeedback can use arbitrary Markdown-formatted text to provide feedback, which may include embedded images, videos, and links (e.g. to past course material), which is kept decoupled from the test definition and can be easily updated from its web interface.

Test My Code (TMC) (Vihavainen, Luukkainen, & Pärtel, 2013; Vihavainen, Vikberg, Luukkainen, & Pärtel, 2013) has some similarities with AutoFeedback. It provides timely scaffolding to students who are working through exercises, allowing them to work more independently. It also allows marks to be awarded for steps completed, and makes testing of code visible to students, providing feedback from the results of testing. From the instructors' viewpoint, it allows exercises to be updated and modified over time, and it gathers data from student activity. It is also based on a client-server architecture, with server-side tests, use of Maven for fetching libraries, and IDE integration via plugins (TMC uses a NetBeans-specific plugin, whereas AF uses an IDE-agnostic Maven plugin). We have noted several other differences between TMC and AF:

- One feature in Test My Code that AutoFeedback does not have is analysis of algorithms, providing some complexity analysis from algorithm execution times.
- The automated feedback aspect of Test My Code does not allow for conditional feedback, which is provided in AF: we have found this conditional feedback to be essential in providing more helpful and directed guidance that allows students to understand why their code is failing the tests. Instead, TMC allows instructors to manually write code reviews for submissions.
- TMC uses Java reflection to see inside students' code. In AF, besides reflection, we have used static analysis tools and also *mocks* which can substitute certain program components and check that they implement specific object interactions.
- The data gathering aspect of Test My Code seems to be for future detection of plagiarism: AF only has some very rudimentary features to detect plagiarism, but its data collection currently allows mapping student performance over the provided exercises, which provides instructors detailed information about which aspects of the exercises were done well by the students, and which were not.

It is important to note a recent 2022 state-of-the-art survey by Paiva, Leal, and Figueira (2022) on automated assessment in CS education, which was published several years after the development of AutoFeedback started. The survey examined 30 different tools present at the time, noting the prevalence of unit testing for finer-grained information compared to output comparison. It noted a trend towards the use of containerisation for improved security (with over 25% of the tools adopting it). It also found that while nearly all tools included failure information and failed tests in their feedback, only some of these tools provided evaluation logs, gave hints, produced a structured report, or allowed instructors to add manual hints (and no tools provided all elements at once). The survey also showed that while it was common for tools to collect submission history, include a code editor, and produce simple statistics, more advanced analyses that enabled deeper insights about student behaviour were rare.

Messer, Brown, Kölling, and Shi (2024) conducted a similar survey of automated grading and feedback tools, covering 121 papers from 2017 to 2021. They made similar findings, where tools primarily used a combination of unit testing and static analysis approaches. They noted that the feedback from the tools was usually limited to tests passing or failing, the expected and actual output, or how they differed from the reference solution. In this respect, AF is among that majority of tools based on tests and static analysis, but it stands out in terms of the flexibility that feedback templates offer. The survey noted that few tools assessed other aspects besides correctness, such as readability, maintainability, or documentation. In our use of AF, we had some checks of Javadocs (Section 4.2), but these were simple checks, such as ensuring parameters were documented, instead of using readability metrics like the Flesch reading ease score

used by Eleyan, Othman, and Eleyan (2020). The survey noted that in many cases, AATs cannot award partial grades for incomplete or uncompilable programs, and that some AATs had started using program repair techniques to solve this issue: in this regard, the use of the Eclipse Java compiler by AF is within this line, although it uses an already mature approach in wide use for years from the Eclipse IDE.

It is worth noting that some of the above tools predate AutoFeedback, with Test My Code and Web-CAT already appearing in the Keuning et al. (2018) survey, which covered 101 tools from 1960 to 2015, and whose major findings were discussed in Section 2.2. JACK is another tool discussed in Keuning's survey and revisited in later surveys, which we discuss later in Section 8.4. Web-CAT also appeared in a prior survey by Ihantola, Ahoniemi, Karavirta, and Seppälä (2010), which mentioned the lack of open-source tools as a reason for the constant development of new systems: we had a similar perception, which is why we decided to open-source AF. Even earlier, the surveye from Douce, Livingstone, and Orwell (2005) is notable in how it divided tools across generations, with the third generation being web-oriented (like AF).

### *8.2. Automated grading of UML models*

Many software engineering courses require students to create UML diagrams when designing object-oriented systems (whether it is their data model, their high-level architecture, or their detailed design). Often, students are given a description of the requirements in natural language, and are asked to come up with a UML model that meets these requirements. Grading these UML models manually can be very labour-intensive, as there are many details to check (i.e. expected types, attributes, operations, and relationships). To make matters more difficult, often the requirements can be met in multiple ways (from slight variations in naming, to completely different ways to break down responsibilities across objects). Le et al. (2013) considered UML as an "ill-defined" problem: within their classification of educational problems, they examined various automated grading systems that handled Class 2 and Class 3 UML problems.

Moritz and Blank (2008) presented an early approach which generated a "solution template" from an English problem description which could be customised by the teacher and used to automatically mark submissions, and which allowed for some variability by using semantics-aware string matching. Striewe and Goedicke (2011) suggested using graph queries to specify the teacher's expectations in a more flexible way than through direct comparison against a model solution. More recently, Bian, Alam, and Kienzle (2020) proposed an approach that allowed for multiple model solutions, giving students marks based on the solution that was closest to their submission using a combination of syntactic, structural, and semantic matching criteria. Bian et al. reported that after refining their tool's configuration, the average difference between the automated and manual marks was within 9%, and identified 37 cases where the automated grading was more consistent than manual grading.

### *8.3. Evaluation of tests written by instructors and students*

The quality of the test suite written by the instructors is crucial to obtaining accurate assessments of the students' achievements, and providing insightful feedback to the students. We already observed in AF that some students tried intentionally introducing defects and seeing if the instructor tests could catch them, and noted that some defects were not detected, and we suggested using mutation analysis to refine

our tests. B. S. Clegg, McMinn, and Fraser (2021) experimented with this same idea across 190 student solutions to first-year programming assignments, and confirmed that mutation scores were positively correlated to the detection rate of faulty students' solutions. They recommended to first reach 100% coverage on a model solution, and then generate mutants using an off-the-shelf mutation testing tool.

Being a first-year course, only the first unit in CS1OOP had students write their own tests: this was more oriented towards understanding the TDD process behind AutoFeedback, rather than giving them an in-depth introduction to test design (which is covered in a later module). AF would run the students' own tests, but did not assess them itself. That said, if we were to add test design in future editions of the module, or adapt AF to a software testing module, it would require integrating an approach to evaluate those tests. One of the most common approaches is to measure the student tests is to measure how thoroughly they cover the implementation (whether the student's, or the model solution's): for instance, this is used by Web-CAT (Edwards, 2003). A later work by Edwards and Shams (2014) compared code coverage, mutation testing and all-pairs testing (i.e. running a student's test suite against the other students' submissions), and found that only all-pairs testing was strongly correlated with the test suite's bug-revealing capability. They also noted that student-written test suites tended to have very low bug-revealing capabilities, suggesting students tended to write only the same basic checks, and missing many of the corner cases.

Other works disagree with this push for all-pairs testing. Smith, Tang, Warren, and Rixner (2017) argue that all-pairs testing requires students to submit a final test suite "blindly" without the ability to interact with the implementations against which they would be evaluated, and instead propose an interactive tool where students propose tests against an instructor-provided program and receive feedback in terms of the next faulty version of the program their tests cannot catch. Kazerouni et al. (2021) argue that all-pairs testing is slow and not amenable to incremental feedback (which would be an issue if integrated in AF), and have a similar concern about requiring several completed and compatible solutions to the assessment: instead, they propose reducing the high cost of mutation analysis by selecting a reduced set of mutation operators that can evaluate test suites at minimal cost, finding that 1 or 2 deletion operators can be enough for most educational settings, and leaving the full set of mutation operators only for small submissions (less than 341 lines of code). Hall and Baniassad (2022) reported a successful experience using hand-written mutants to evaluate student-written tests, with a small set of 18 mutants having strongly correlated student scores against a larger set of 73 mutants, and students engaging much more often with the testing section of their forums compared to when code coverage was the target metric.

### *8.4. Automated code feedback tools*

Various projects have looked into static analysis for automated feedback generation. Pedal (Gusukuma, Bart, & Kafura, 2020) is an open-source framework for static analysis of student code written in Python, which can automatically produce feedback in a more relatable way than typical Python stack traces. Pedal provides a set of functions that instructors combine into a script for a given assignment. These functions can perform pattern matching against the student code (e.g. to detect infinite recursion), conduct static analysis for typical mistakes (e.g. overwriting a function before it is read), and run black-box tests (e.g. function should return a certain value given some inputs). It is worth noting that while AutoFeedback has no specialised features

for static analysis, its Maven-based design allows teachers to use existing parsing and static analysis libraries in their tests. In fact, the basic static analysis described in Section 4.2 reused an existing library for parsing Java code.

PyTA[11] is another static analysis tool for Python student code. In addition to predefined checks (e.g. using a variable before its assignment), it allows for defining contracts: pre-/post-conditions for functions (checked before and after each call), and class invariants (checked after methods calls and when attributes are reassigned). Similarly to our discussion of Pedal above, AutoFeedback does not have any such specialised functionality built-in, and it would have to be written into the tests: one option could be to port the idea to Java and package it as a reusable library.

Jeuring et al. (2022) selected several datasets that included the steps students took to solve a programming problem, and annotated it with the various points at which experts would typically intervene and how they would intervene. They used that experience to compare the feedback given by experts to that given by learning environments, considering tools such as CodeWars, Codecademy, or CodingBat, among others. Several of the tools mentioned allowed students to ask for predefined hints, although it is repeatedly noted that these hints did not take the current code status (unlike our conditional feedback templates). The authors noted that while learning environments typically wait for a user action (e.g. a submission) and cannot deal with hint-avoiding behavior, experts would step in themselves at certain points: for instance, when seeing tinkering behavior or continuous try-and-error submissions. In fact, we did observe such try-and-error behavior in some students, resulting in very high attempt counts for some labs (c.f. Section 6.1.2) that would necessitate human intervention. They observed that experts would typically combine correct unit test results with hints and enhance compiler error messages (both of which AF conditional templates could do).

Among the tools reviewed in (Jeuring et al., 2022), the flexibility of the JACK system (Striewe, 2014) was highlighted, as it was possible to reconfigure it to match the expert feedback for one of the dataset's tasks. JACK can perform teacher-defined static and dynamic checks on the student submissions, with a dedicated query language (GReQL) for defining patterns on the code, and has recently moved to a worker-based architecture similar to that of AF (Striewe, 2023). GReQL patterns can have specific pieces of feedback attached to them upon the existence or absence of a match, which is close to the conditional templates in AF. AF decouples the check from the feedback text, however: the check could be implemented using any Maven library as a test assertion, and a Markdown template would be written to produce certain feedback based on the result of that assertion. They noted an issue in JACK when teaching novice programmers, in that it could not produce any feedback if there were syntax errors, and considered it better for novice programmers to ignore those errors in a first step and address a critical logical error with valuable feedback. While the reason AF attempts to run tests in the presence of syntax errors is different (for dealing with cases where teacher-written tests do not compile due to students ignoring some lab requirements, c.f. Section 4.2), it could also be useful for this situation.

iSnap (Price, Dong, & Lipovac, 2017) is an extension to the Snap! programming environment[12]. iSnap has extended Snap! to allow logging of student actions and on-demand hints designed to give help to students. While this is helpful for absolute novices, the iSnap developers recognised that an over-reliance on help was not conducive to helping students learn: we prefer to leave some of the problem solving to the

[11] https://github.com/pyta-uoft/pyta

[12] https://snap.berkeley.edu/

students. Whereas iSnap suggests *how* the student should change their program, our intended approach is to show the student *why* the program needs to be changed.

### *8.5. Experiences using automated assessment tools*

Several other works have reported the impact of automated feedback on students' performance, and the attitudes from students towards that automated feedback. Pettit, Homer, Holcomb, Simone, and Mengel (2015) reviewed in 2015 data from 24 different automated assessment tools (AATs) around four dimensions, and found that while there was evidence that AATs were helpful for student learning, supporting the instructors, and assessment accuracy, there was no conclusive answer on how students generally perceived AATs. They mentioned the case study from Rubio-Sánchez, Kinnunen, Pareja-Flores, and Velázquez-Iturbide (2014), where Mooshak (a tool normally used for programming contests) was used as an AAT: while the students considered the use of Mooshak a good idea, they thought the feedback produced by Mooshak was poor, and that it increased their workload. Many of the errors students were getting from Mooshak were due to its use of a different compiler from the one in their machines, which meant sometimes their programs would work on their machines but not on Mooshak's server. Rubio-Sánchez et al. mentioned that the Mooshak developers were refining the tool to show hints and the number of passed tests. In contrast, AutoFeedback was designed to use the same compiler that students would use on their machines (the Eclipse Java compiler), and has specific functionality for giving hints and showing per-test outputs and results from the start. In its second year, AF only had 4 neutral responses on whether the feedback was understandable, with the other 12 responses agreeing that it was understandable.

Kyrilov and Noelle (2016) noted a prior negative experience with simple correct / incorrect (KR) feedback, where half of the students needed large amounts of time to reach a correct solution, and a small number of students either gave up or resorted to dishonest practices. They looked at whether it was possible to cluster student submissions based on their feedback, based on the same intuition we had that many students tend to make the same mistakes. They manually clustered incorrect student submissions over 5 different exercises in a module that introduces object-oriented programming, and found that there were between 4 and 11 clusters per exercise, with roughly 10 on average, meaning that it could be possible for an instructor to provide richer feedback for each cluster if there were a way to automatically classify submissions based on errors. This would be an interesting avenue for future work, as AF already collects per-test error output that could be used for clustering.

The developers of Test My Code noted that on the first year of its introduction, only 58% of the feedback was positive (Vihavainen, Luukkainen, & Pärtel, 2013). It was only after significant iterative refinement on the second year that feedback became largely positive (80% during their Spring 2012 CS1 course). They also observed an increase in pass rates from 55.49% before introducing their Extreme Apprenticeship (XA) method into their modules, to 73.45% after introducing XA, and then to 75.76% after introducing TMC into their XA-based modules. Likewise, AF went through an intense period of refinement on its first year, and stabilized to good reception on its second year, to the point that students wanted to have it in other modules.

More recently, Leite and Blanco (2020) compared the effects of automated and human-written feedback on students on an intermediate Artificial Intelligence course, by dividing students into two groups: the first group received feedback only from a

purpose-specific in-house grading tool (which checked the logic of the algorithm, rather than just running black-box tests), and the second group also received feedback from teaching assistants who reviewed the output of the grading tool, and added two types of feedback difficult to automate: specific references to what caused students' code to fail a criterion, and feedback on the clarity, efficiency, and syntax usage of the code. They found that receiving human feedback was very likely to improve students' conceptual understanding, but that the improvement was limited to certain undetermined areas, and they noted that human feedback had little effect on the students' final projects. Where they saw noticeable impact, however, was in the distribution of overall course grades, where the students in the middle two quartiles (the "average" students) obtained better grades when receiving human-written feedback. Our experience with AF is that automated feedback is complementary to human feedback: what the automation does is to free up the time of the teaching assistants from clarifying common mistakes, and letting them discuss higher-level concerns and helping with less common mistakes.

Leite and Blanco also noted that human assistants tended to provide more partial marks than their AAT, as they understood better when the students had gotten closer to a correct solution, so they had to review the automatically-graded submissions to award partial credit in the same way. We had to do something similar in the JavaFX auto-assessed lab (Section 4.2): after the students had made their submissions, we manually reviewed all submissions who were close to achieving full marks and revised the test suites in case they were being too strict — once the test suite was refined, we reran all student submissions who did not have full marks, and gave the student the highest mark across the old and refined test suite.

We also found a few mentions in the literature about the concern that students may "game" the automated grader if they are given unlimited submission attempts, doing trial-and-error submissions without considering the feedback. Leite and Blanco (2020) mentioned that one student tested their code on an assignment 200 times, the developers of the Kattis tool (Enström, Kreitz, Niemelä, Söderman, & Kann, 2011) mentioned some tests had over 100 solutions to a problem (although they did not see that as an issue), and we have seen similar outliers with 150 attempts for a given assessment (Section 6.1.2). Besides directly limiting the attempts by setting a maximum number or imposing a delay between submissions, several works have studied approaches to discourage such behavior, and we are considering whether to implement them in AF:

- Spacco et al. (2006) showed how their Marmoset tool encouraged starting work early and not resubmitting too often by limiting the number of times the students could run their code against the instructor's confidential *release tests*, simulating the cost of a release to production by having students spend *release tokens*. Instructors generally saw the concept of release tests as being pedagogically sound, and students considered the feedback from release tests to be useful and a motivator to start work early. It is also interesting to note that one of their instructor testimonies recognized the same need to review the test cases after the projects had been made available to students (which AF has specific provisions for, including a dedicated model solution versioning system).
- Baniassad, Zamprogno, Hall, and Holmes (2021) experimented with the concept of *regression penalties*, where a penalty was imposed each time a student's grade went down. These were explained to the students as part of a larger Software Engineering narrative (i.e. submitting to the autograder was equivalent to releasing code to a client). Introducing these penalties significantly reduced the number of submissions and how often the students' code went through regressions, at nearly

no impact to the median overall mark. Students reported having to change their working practices due to these penalties, having to write their own tests more often: they appreciated how they built better habits, but there were students who felt stressed due to the penalties.

Finally, we note the case study from Barra et al. (2020) on transitioning the assessment of a programming course to an online format during the COVID-19 pandemic, by using an AAT. They mentioned similar concerns to ours in terms of the high number of students and the need for frequent and timely feedback for first-time programmers. Moving the face-to-face lessons, the programming assignments, and the final exam into an online format meant the AAT went from being a complementary tool to the face-to-face sessions and tutorials, to being the primary tool. Their in-house AAT (autoCOREctor) ran tests locally (using hashes to ensure students did not manipulate the tests) and submitted the results to the LMS, and gave unlimited attempts to the students. Across the 85 responses to their survey, they found that students were divided on whether the feedback from the tool was useful: the authors recognize that the tool only pointed out the errors but did not tell them how to fix them, as they considered it would have been detrimental to their learning. They also note that students often had syntax errors which caused the AAT to produce cryptic errors (as we also observed during the first few weeks AF was deployed, c.f. Section 4.2). The students overwhelmingly preferred having the AAT feedback over no feedback at all, and would have used it even if it had been optional. These positive results also match ours (Figure 10), although the response rate to our survey was noticeably lower.

### *8.6. Tool comparison matrix*

From the discussion so far, it is possible to identify a feature set that an ideal tool for automated feedback over programming assignments would need to have, and perform comparisons against the tools mentioned in Section 8.1. Unfortunately, we could only find open-source repositories for Gradeer, Web-CAT, and TMC: we could not find the repositories for PABS, WebTA, or JACK (discussed in Section 8.4). Table 9 summarizes the results based on prior publications from the various tools, our study of their open-source code-bases and public documentation. It is important to note that due to gaps in the available documentation, we may have missed some of the features in the above alternatives.

The table has been divided into several groups of features. We focused first on the types of feedback that can be given from each tool. All tools support feedback based on test results, and most tools (notably AF being excluded) also allow for running a code style checking tool such as Checkstyle or PMD. AF did not include this feature as code style was not a major focus for first-year students (this was usually left to the second year), but it is a gap that would need to be covered. AF has partial support for static analysis and documentation checks, due to its ability to use custom build scripts that allow for having additional libraries for non-standard tests, but it cannot be said to support these natively. Only TMC tackled performance-oriented tests (e.g. checking whether a sorting algorithm has been implemented in $O(nlogn)$ time).

The next group focuses on assessing student-written tests: this is one aspect where we could only find explicit support in Web-CAT, using coverage measurement. We did not find any explicit support for mutation-based test assessment (as mentioned in Section 8.3.

The third group covers a number of features for helping organise the feedback. We

| **Feature** | AF | Gradeer | Web-CAT | TMC |
|---|---|---|---|---|
| Supported feedback types | | | | |
| Test result feedback | ✓ | ✓ | ✓ | ✓ |
| Performance feedback | × | × | × | ✓ |
| Static analysis feedback | ∼ | × | × | × |
| Documentation feedback | ∼ | × | × | × |
| Code quality feedback | × | ✓ | ✓ | ✓ |
| Assessment of student tests | | | | |
| Coverage-based test assessment | × | × | ✓ | × |
| Mutation-based test assessment | × | × | × | × |
| Feedback-oriented features | | | | |
| Solution versioning | ✓ | × | × | ∼ |
| Test-specific hints | ✓ | ✓ | ✓ | ✓ |
| Template-driven feedback | ✓ | × | × | × |
| Manual regrading | × | ✓ | ✓ | ✓ |
| Collecting student feedback | × | × | × | ✓ |
| Repair for partial marks | ∼ | × | × | × |
| Teacher support features | | | | |
| Cohort-level analytics | ✓ | × | × | ∼ |
| Plagiarism detection | × | × | × | × |
| Discouraging "gaming" | × | × | × | × |
| Deployment features | | | | |
| IDE-based submission | ✓ | × | ✓ | ✓ |
| Web-based user interface | ✓ | × | ✓ | ✓ |
| Custom build scripts | ✓ | × | × | × |
| Containerised execution | ✓ | × | × | ✓ |

**Table 9.** Feature comparison matrix for AF and selected open-source automated assessment tools within Section 8: ✓ indicates there is support, ∼ indicates partial support, × indicates there is no support for the feature.

could only find two tools that had some explicit handling of the evolution of the model solution over time: AF explicitly stores every model solution in its database and ties assessment results to (model solution, submission) version pairs, and TMC can refresh itself by reloading the assessment description from its source Git repository. All tools support giving some hints when a test fails (JSON file in Gradeer, test assertions in Web-CAT / TMC), but only AF uses a template-driven approach with conditional feedback based on a combination of test result and test output lines. Most tools support manual regrading, except AF: while we consider this feature to be useful, we considered it a higher priority to work on solution versioning so that every student would be assessed consistently — if there was an issue in the test, this issue should be corrected there so that all students would benefit. While reviewing TMC's documentation, we found that it supports *feedback questions* in assessments, where students can provide their own feedback: this could be useful to adopt in AF for knowing what the students think about the automated feedback a specific lab. Finally, we could not find any evidence of automated program repair of any kind in the other tools besides AF: in our case, the support is partial as we do not allow for any configurability — it is up to the Eclipse Java Compiler to decide how to repair a program that does not fully compile.

The "teacher support features" group includes features that we would appreciate as instructors, whether to protect the integrity of the assessment (plagiarism detection, discouraging "gaming" via repeated nearly-identical submissions), or to understand better the current state of the cohort. TMC has some organisation-level reports, according to its documentation, and AF has assessment-level reports: we could not find such reports in Gradeer or Web-CAT. We could not find evidence of the other desirable features in this group in any of the systems: this suggests that there is space for integrating the research results of Section 8.5 into the widely available web-based automated assessment tools.

The last group is more technical in nature, and considers a number of desirable features to improve the student, teaching, and server administrator experience. Out of the four systems, only Gradeer did not support direct submission from the student's integrated development environment, reducing friction and the likelihood of an incorrectly packaged submission. 3 of the 4 tools had web-based user interfaces (whereas Gradeer is only usable from the command-line by the teacher). Most tools did not allow for custom build scripts that would make it possible to change the dependencies / compilation options used for each assessment: this was only explicitly considered in AF. Finally, only AF and TMC executed their code in a containerised environment: AF used Docker containers, and TMC used a specialised Linux-based "sandbox".

## 9. Conclusion and future work

In this paper, we have provided the first description of AutoFeedback, an open-source system for automated feedback over Java-based programming assignments which is based on standard testing frameworks (JUnit, Mockito, TestFx), can be deployed entirely on university premises, and it is specifically designed to support the concurrent iterative refinement of the test suites, the feedback templates, and the students' work. AutoFeedback is motivated by the dual feedback loop of the Interactive Tutoring Feedback model by Narciss (2013), using feedback templates to deliver elaborated feedback for the learner's internal feedback loop, and providing various reports to enable the teacher's external adjustment of the external feedback loop. A single-node deployment

of AF has processed over 58,000 submissions over the 2020–21 and 2021–22 academic years of the first-year object-oriented programming module at Aston University.

The paper has discussed the original design and refinements of the teaching materials used with AF, and how AF was integrated into the student experience. Ensuring students felt they were shaping AF was crucial to its success: tests were refined if found brittle, feedback templates were clarified if a question was raised, and various features were introduced at the students' request. Some students self-organized to provide early feedback on the tests to instructors, to improve their robustness against the most common approaches to solve the exercises. Based on our experience, we have given a number of recommendations on how to write templates and design test suites for test-driven feedback.

Overall student experience has been positive: students liked obtaining feedback at any time, and noted it helped find the underlying causes for the problems in their code. More consistent participation was observed, and lab marks improved with further submission attempts. Students found AF easy to use, as it only required two clicks from their IDE and did not expect any knowledge of version control tools. However, students also reported multiple areas for improvement. Some tests did not notice intentionally introduced faults or were too brittle, discouraging some valid approaches to solve assignments: these are recognised problems in test design, and mutation analysis (which can automatically evaluate if the test can catch these mistakes) could help design better test suites for automated feedback. In other cases, the feedback template was too terse or jargon-heavy, leaving the student without a clear idea of what to do in response: this suggests that a conscious design of "mental models" of the students at each point of the course may be useful.

When compared to other existing automated assessment tools, AutoFeedback is among the majority of the observed tools in its use of a combination of automated unit testing and static analysis. Its major standing out features are its use of a Markdown dialect for feedback that is driven by the test result and text output, its versioning system to relate evolving model solutions to student submissions, and its provision of cohort-wide metrics of progress against specific tests. It also demonstrates some ongoing trends, such as the use of program repair to grant partial marks: in its case, it uses the Eclipse Java compiler, which has already been broadly in use before AF was created. We note that AF does not yet have features to evaluate the quality of the tests, as these were not part of the learning outcomes for this first-year programming module, and that it could still be expanded to measure other attributes besides correctness, such as readability, maintainability, and the quality of the documentation.

This research can be expanded in several directions. The highest priority is to repeat the experience in other contexts (e.g. cohorts from other institutions and/or other programming courses), to evaluate the generalizability of the results. This may require adding support for other programming languages besides Java, for which a migration of AF from Docker Compose to the Kubernetes containerization platform is planned. Some of the analyses in this paper could be integrated into the AF learner analytics, not only to study the individual learning journey of each student but also to find the most common challenges faced by the students and ensure that the feedback templates provide suggestions to tackle them. Finally, it is planned to introduce features for discouraging trial-and-error submissions, and for students to provide feedback on the feedback they receive by starting conversations that may be explicitly linked to clarifications and improvements in the feedback templates (such as adding links to learning resources that reinforce the underlying programming topics being targeted by the tests).

## Acknowledgments

The authors would like to thank the rest of the teaching staff of CS1OOP at Aston University for their cooperation with introducing automated feedback into the student experience (especially Dr. Nick Powell, Vangelis Fafoutis, and Renato Barros Arantes), as well as the students who provided feedback to shape AutoFeedback. The authors were employed at Aston University during these experiments. Access and use of the anonymized student participation, performance and feedback data from CS1OOP 2019–20, CS1OOP 2020–21 and CS1OOP 2021–22 for the purposes of pedagogical research and publication (including the dataset) was authorized by the Aston University Computer Science Programme Committee and EPS Research Ethics Committee on the proviso that the data is extracted anonymously, no direct quotes are published, and only anonymized, analyzed and aggregated data is made public.

## Declaration of interest statement

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

Baniassad, E., Zamprogno, L., Hall, B., & Holmes, R. (2021, March). STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (pp. 1062–1068). New York, NY, USA: Association for Computing Machinery.

Barra, E., López-Pernas, S., Alonso, A., Sánchez-Rada, J. F., Gordillo, A., & Quemada, J. (2020, January). Automated Assessment in Programming Courses: A Case Study during the COVID-19 Era. *Sustainability*, *12*(18), 7451.

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., ... Prather, J. (2019, December). Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (pp. 177–210). New York, NY, USA: Association for Computing Machinery.

Bian, W., Alam, O., & Kienzle, J. (2020, October). Is automated grading of models effective?: assessing automated grading of class diagrams. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 365–376). Virtual Event, Canada: ACM.

Braun, V., & Clarke, V. (2006, January). Using thematic analysis in psychology. *Qualitative Research in Psychology*, *3*(2), 77–101.

Checkstyle. (2021). *Checkstyle homepage.* `https://checkstyle.sourceforge.io/`. (Date of last access: 2021-09-19)

Clegg, B., Villa-Uriol, M.-C., McMinn, P., & Fraser, G. (2021, May). Gradeer: An Open-Source Modular Hybrid Grader. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (pp. 60–65).

Clegg, B. S., McMinn, P., & Fraser, G. (2021, March). An Empirical Study to Determine if Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (pp. 1055–1061). New York, NY, USA: Association for Computing Machinery.

Douce, C., Livingstone, D., & Orwell, J. (2005, September). Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, *5*(3).

Edwards, S. H. (2003, September). Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, *3*(3).

Edwards, S. H., & Shams, Z. (2014, May). Comparing test quality measures for assessing student-written tests. In *Companion Proceedings of the 36th International Conference on Software Engineering* (pp. 354–363). New York, NY, USA: Association for Computing Machinery.

Eleyan, D., Othman, A., & Eleyan, A. (2020, September). Enhancing Software Comments Readability Using Flesch Reading Ease Score. *Information*, *11*(9).

Enström, E., Kreitz, G., Niemelä, F., Söderman, P., & Kann, V. (2011, October). Five years with kattis — Using an automated assessment system in teaching. In *2011 Frontiers in Education Conference (FIE)* (pp. T3J–1–T3J–6). (ISSN: 2377-634X)

Garcia-Dominguez, A. (2022, September). *Aggregated results from use of the AutoFeedback system in the 2020-21 and 2021-22 editions of CS1OOP (Object-Oriented Programming)* (Data collection). Birmingham, UK: Aston University. (doi:10.17036/researchdata.aston.ac.uk.00000578)

GitHub Inc. (2022). *Github actions — features.* `https://github.com/features/actions`. (Date of last access: 2022-09-26)

GitLab Inc. (2022). *GitLab CI/CD.* `https://docs.gitlab.com/ee/ci/`. (Date of last access: 2022-09-26)

Gusukuma, L., Bart, A. C., & Kafura, D. (2020, February). Pedal: An Infrastructure for Automated Feedback Systems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (pp. 1061–1067). New York, NY, USA: Association for Computing Machinery.

HackerRank. (2022). *HackerRank homepage.* `https://www.hackerrank.com/`. (Date of last access: 2022-09-26)

Hall, B., & Baniassad, E. (2022, December). Evaluating the Quality of Student-Written Software Tests with Curated Mutation Analysis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E* (pp. 24–34). New York, NY, USA: Association for Computing Machinery.

Higher Education Statistics Agency. (2022, February). *What do HE students study?* (Tech. Rep.). Retrieved 2022-08-29, from `https://www.hesa.ac.uk/data-and-analysis/students/what-study#changes`

Ifflander, L., Dallmann, A., Beck, P.-D., & Ifland, M. (2015, November). PABS - a Programming Assignment Feedback System. In *Proceedings of the Second Workshop "Automatische Bewertung von Programmieraufgaben"* (Vol. 1496). Wolfenbüttel, Germany: CEUR-WS.

Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010, October). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 86–93). Koli Finland: ACM.

Jeuring, J., Keuning, H., Marwan, S., Bouvier, D., Izu, C., Kiesler, N., ... Sarsa, S. (2022, December). Towards Giving Timely Formative Feedback and Hints to Novice Programmers. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (pp. 95–115). New York, NY, USA: Association for Computing Machinery.

Kara, E., Tonin, M., & Vlassopoulos, M. (2021, June). Class size effects in higher education: Differences across STEM and non-STEM fields. *Economics of Education Review*, *82*.

Kazerouni, A. M., Davis, J. C., Basak, A., Shaffer, C. A., Servant, F., & Edwards, S. H. (2021, May). Fast and accurate incremental feedback for students' software tests using selective mutation analysis. *Journal of Systems and Software*, *175*.

Keuning, H., Jeuring, J., & Heeren, B. (2018, September). A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education*, *19*(1), 3:1–3:43.

Kolb, D. (1984). *Experiential learning: experience as the source of learning and development.* Englewood Cliffs, NJ: Prentice Hall. Retrieved from http://www.learningfromexperience.com/images/uploads/process-of-experiential-learning.pdf(dateofdownload:31.05.2006)

Kyrilov, A., & Noelle, D. C. (2016, April). Do students need detailed feedback on programming exercises and can automated assessment systems provide it? *J. Comput. Sci. Coll.*, *31*(4), 115–121.

Le, N.-T., Loll, F., & Pinkwart, N. (2013, July). Operationalizing the Continuum between Well-Defined and Ill-Defined Problems for Educational Technology. *IEEE Transactions on Learning Technologies*, *6*(3), 258–270. (Conference Name: IEEE Transactions on Learning Technologies)

Leite, A., & Blanco, S. A. (2020, February). Effects of Human vs. Automatic Feedback on Students' Understanding of AI Concepts and Programming Style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (pp. 44–50). New York, NY, USA: Association for Computing Machinery.

Messer, M., Brown, N. C. C., Kölling, M., & Shi, M. (2024, February). Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Trans. Comput. Educ.*, *24*(1).

Moritz, S., & Blank, G. (2008, June). Generating and Evaluating Object-Oriented Designs for Instructors and Novice Students. In *Intelligent Tutoring Systems for Ill-Defined Domains: Assessment and Feedback in Ill-Defined Domains.* (pp. 35–43). Montreal, Canada.

Narciss, S. (2013). Designing and evaluating tutoring Feedback Strategies for Digital Learning. *Digital Education Review*(23), 7–26.

Paiva, J. C., Leal, J. P., & Figueira, A. (2022, June). Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education*, *22*(3), 34:1–34:40.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. L., & Harman, M. (2019, January). Chapter Six - Mutation Testing Advances: An Analysis and Survey. In A. M. Memon (Ed.), *Advances in Computers* (Vol. 112, pp. 275–378). Elsevier.

Papadakis, M., & Le Traon, Y. (2015). Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability*, *25*(5-7), 605–628.

Pettit, R., Homer, J., Holcomb, K., Simone, N., & Mengel, S. (2015, June). Are Automated Assessment Tools Helpful in Programming Courses? In *2015 ASEE Annual Conference and Exposition Proceedings* (pp. 26.230.1–26.230.20). Seattle, Washington: ASEE Conferences.

PMD. (2021). *PMD.* https://pmd.github.io/. (Date of last access: 2021-09-19)

Poulos, A., & Mahony, M. J. (2008, April). Effectiveness of feedback: the students' perspective. *Assessment & Evaluation in Higher Education*, *33*(2), 143–154.

Price, T. W., Dong, Y., & Lipovac, D. (2017, March). iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 483–488). Seattle Washington USA: ACM.

Qualified. (2022). *Codewars homepage.* https://www.codewars.com/. (Date of last access: 2022-09-26)

Reas, C., & Fry, B. (2006, September). Processing: programming for the media arts. *AI & Society*, *20*(4), 526–538.

Rubio-Sánchez, M., Kinnunen, P., Pareja-Flores, C., & Velázquez-Iturbide, A. (2014, February). Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, *31*, 453–460.

Shute, V. J. (2008, March). Focus on Formative Feedback. *Review of Educational Research*, *78*(1), 153–189.

Smith, R., Tang, T., Warren, J., & Rixner, S. (2017, June). An Automated System for Interactively Learning Software Testing. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 98–103). New York, NY, USA: Association for Computing Machinery.

Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. K., & Padua-Perez, N. (2006, September). Experiences with Marmoset: designing and using an advanced submission and testing system for programming courses. *ACM SIGCSE Bulletin*, *38*(3), 13–17.

Striewe, M. (2014). *Automated Analysis of Software Artefacts – A Use Case in E-Assessment* (PhD thesis, Universität Duisburg-Essen, Duisburg, Germany). Retrieved 2024-11-15, from `https://core.ac.uk/download/pdf/33797255.pdf`

Striewe, M. (2023). Architectural Revision of the E-Assessment System JACK. In T. Batista, T. Bureš, C. Raibulet, & H. Muccini (Eds.), *Software Architecture. ECSA 2022 Tracks and Workshops* (pp. 19–26). Cham: Springer International Publishing.

Striewe, M., & Goedicke, M. (2011, June). Automated checks on UML diagrams. In *Proceedings of the 16th annual joint conference on Innovation and Technology in Computer Science Education* (pp. 38–42). Darmstadt Germany: ACM.

Ureel, L. C., & Wallace, C. (2015, October). WebTA: Automated iterative critique of student programming assignments. In *2015 IEEE Frontiers in Education Conference (FIE)*.

Vargha, A., & Delaney, H. D. (2000). A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, *25*(2), 101–132.

Vihavainen, A., Luukkainen, M., & Pärtel, M. (2013, January). Test my code: An automatic assessment service for the extreme apprenticeship method. In P. Vittorini, R. Gennari, I. Marenzi, T. D. Mascio, & F. D. l. Prieta (Eds.), *2nd international workshop on evidence-based technology enhanced learning* (Vol. 218, pp. 109–116). Heidelberg: Springer International Publishing.

Vihavainen, A., Vikberg, T., Luukkainen, M., & Pärtel, M. (2013). Scaffolding students' learning using Test My Code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (pp. 117–122). New York, NY, USA: Association for Computing Machinery.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg. (doi:10.1007/978-3-642-29044-2)

Wong, S. H. S., & Beaumont, A. J. (2012). A quest for helpful feedback to programming coursework. *Engineering Education*, *7*(2), 51–62. (doi:10.11120/ened.2012.07020051)