The Application of Mutation Testing to Enhance the Automated Assessment of Introductory Programming Assignments

Benjamin Simon Clegg

Supervised by Prof. Phil McMinn & Prof. Gordon Fraser

A thesis submitted in partial fulfilment of the
requirements for the degree of Doctor of Philosophy

Department of Computer Science
Faculty of Engineering
The University of Sheffield

September 2021

Contents

Abstract

Growing cohorts of students enrolled in introductory programming courses reveal a challenge in manual assessment; it is impractical for a tutor to manually evaluate hundreds or even thousands of programs written by students in a timely manner. Furthermore, manual assessment is not always fair; tutors can make mistakes in their assessment. Automated assessment provides a solution to these problems; a computer can evaluate the correctness and style of students' programs, and generate feedback accordingly, in much less time, and with a high degree of consistency. A particularly widespread approach to do this is test-based automated assessment, in which a tutor writes a test suite to evaluate the correctness of students' programs, which is automatically executed by a computer to generate a grade and applicable feedback according to the results of these tests.

Such assessment test suites are not necessarily flawless, however. For example, a test suite may not detect some faults present in students' programs; they may receive inaccurate grades and feedback where their mistakes are missed. In the software engineering industry, adequacy metrics and test goals are often employed to ensure that test suites can detect faults; by achieving such test goals and high adequacy metrics, a test suite should be able to detect faults more reliably. One approach is to measure coverage; which elements of a program are executed by a test suite, and which are not. Naturally, a test suite which exercises more of a program should be more capable of detecting faults. However, executing a program element does not guarantee that a fault within it is detected, for example, some faults only manifest for particular states of the program. Mutation testing offers a different approach to evaluating

the adequacy of a test suite. Mutation testing involves generating artificial faulty variants of the program, called mutants, and executing the test suite on each of them. A test suite which detects more of these mutants should be more capable of detecting faults. Furthermore, the undetected mutants can be used to inform the creation of new tests to improve adequacy.

Accordingly, in this thesis I investigate how mutation testing can be used to improve grading test suites. First, I consider how different test suites can generate varying grades for students' solution programs; is there a risk of inadequate test suites generating unfair grades? I also investigate how different observable properties, including coverage and the detection of mutants, impact such changes in grades. Finally, I evaluate how applicable mutation testing is to improving grading test suites; do the fundamental assumptions of mutation testing hold for students' programs, and does improving a test suite's ability to detect artificial faults also improve its ability to detect students' faults?

Acknowledgements

First, I would like to thank my supervisors, Phil and Gordon, for their excellent guidance and advice throughout my research. This work truly would not have been possible for me to accomplish without them. Thanks to Siobhán North and Mari-Cruz Villa-Uriol for their assistance in data collection. I would also like to thank my labmates for their support throughout my work, and for the great lunchtime banter in the VT lab. Thanks to Abdullah for his teachings in R, data analysis would have been a lot tougher without them. Thanks to all of my friends and family for their unrelenting moral support in my endeavours. Finally, thanks to my parents for their constant support and faith in my work.

To all of you, you have my eternal appreciation; I could not have achieved this alone.

Publications Resulting From This Research

As a result of my research, I have written five papers, which I have presented at various international conferences:

- Benjamin Clegg, Siobhán North, Phil McMinn, and Gordon Fraser – *"Simulating Student Mistakes to Evaluate the Fairness of Automated Grading"* – International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET), 2019 [1] (Track acceptance rate: 30%)

- Benjamin Clegg, Phil McMinn, and Gordon Fraser – *"The Influence of Test Suite Properties on Automated Grading of Programming Exercises"* – Conference on Software Engineering Education and Training (CSEET), 2020 [2] (Conference acceptance rate: 37%)

- Benjamin Clegg, Phil McMinn, and Gordon Fraser – *"An Empirical Study to Determine if Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding"* – ACM Technical Symposium on Computer Science Education (SIGCSE), 2021 [3] (Track acceptance rate: 31%)

- Benjamin Clegg, Maria-Cruz Villa-Uriol, Phil McMinn, and Gordon Fraser – *"Gradeer: An Open-Source Modular Hybrid Grader"* – International Conference on Software Engineering: Software Engineering Education and Training Joint Track with CSEET (ICSE-JSEET), 2021 [4] (Track acceptance rate: 33%)

- Benjamin Clegg, Phil McMinn, and Gordon Fraser – *"Diagnosability, Adequacy & Size: How Test Suites Impact Autograding"* – Conference on Software Engineering Education and Training (CSEET): Collaborative Special Track at the Hawaii International Conference on System Sciences (HICSS), 2022 [5] (Track acceptance rate: 47%) (Winner of Best Paper Award)

Chapter 1

Introduction

With an ever increasing demand for computer science and software engineering education [6], it has become more important than ever to increase the supply of education in the field, using teaching resources (particularly the time of tutors) in as efficient a manner as possible. Alongside demand for traditional education, the userbase of Massive Open Online Courses (MOOCs) is also growing, as online education is particularly beneficial for those with the desire to study part-time, with the goal of changing careers, or improving their employability [7]. *Automated assessment* is essential to evaluate the performance of large numbers of students; it is infeasible to manually grade thousands of students' programs [8]. Without the ability to evaluate students' programs, tutors are limited in their teaching; assessment is important to provide students with feedback [9] and accreditation [10], and to provide tutors with the pedagogical knowledge of where students require additional instruction [11].

While there are multiple approaches to the automated assessment of students' programs, including verification [12], static analysis [13, 14], clustering [15, 16], and machine learning [17], test-based approaches are perhaps the most prevalent [18–21]. This involves running an automated test suite on a student's program, and generating a grade from the tests' results, such as the proportion of tests that pass. Similarly the results of individual tests can be used

to generate feedback for students, by assigning appropriate messages to particular test results [22]. Such test suites are unique to the programming task under assessment, and are therefore typically written by the tutor of a given programming course.

This test based approach to automated grading is not without its limitations, however; test suites can vary in quality, with poorer test suites not effectively detecting students' faults. Such low quality test suites pose a unique problem for automated grading; students may not receive appropriate feedback if their faults are not detected. In addition, students may receive grades that are too high or low depending on how many tests detect (or fail to detect) their mistakes, creating a potential source of unfairness. This issue can be exacerbated if a tutor does not hold the necessary knowledge to construct effective grading test suites. For example, tutors can lack awareness of the programming mistakes that students make, and the prevalence of these mistakes [23]. Fortunately, it is possible that by evaluating the quality of their grading test suite, a tutor can understand how it should be improved. This approach is already commonplace in software testing; a variety of test adequacy metrics aim to directly evaluate a test suite's quality, and guide a tester on how it can be improved.

1.1 The Application of Adequacy Metrics in Test-Based Automated Assessment

A particularly simple test adequacy metric is *code coverage*, which measures the proportion of program elements (such as lines of code) that are executed by a test suite [24]. Any program elements that are not executed by any test may contain a fault which no test could ever detect; suites that cover more of the program should detect more potential faults. In the context of automated assessment, assuming that a tutor has a correct reference program that perfectly implements a task's specification, any uncovered program elements (e.g. lines of code) represent aspects of a task's specification that

a test suite cannot evaluate. By using the uncovered lines as *test goals*, a tutor has a clear target on where their test suite requires improvement. One key benefit of coverage is that it introduces little computational overhead over executing a test suite alone; it can be evaluated in little time, allowing a tutor to quickly understand how their test suite can be improved. Coverage is by no means perfect, however; some faults are subtle, and are not detected by merely being executed [25]. For example, some faults may only manifest under particular program states. Consequently, achieving full coverage of a reference program does not guarantee that a test suite can identify any subtle faults that students introduce.

Fortunately, an alternative approach does address this weakness of coverage; *mutation testing* [26]. This involves automatically generating a large number of faulty variants of a program (such as a reference solution), called *mutants*. These mutants are generated using a mutation tool, which implements mutation operators; rules for how a mutant should be generated from an input program. Mutants directly introduce faults; by detecting every mutant, a test suite should be capable of also detecting real, even subtle faults. This offers a clear advantage over coverage, since a test suite must detect these artificial faults, rather than simply execute parts of a program. Mutation testing relies on the assumption that mutants are capable of emulating real faults. Existing work has shown that this assumption holds for real faults in open-source and commercial software [27, 28], but no work has investigated that this also holds for students' faults in their solutions to programming tasks. In this thesis, I aim to address this, by investigating if such assumptions also hold for students' faults, supporting the use of mutation testing to evaluate and guide the improvement of grading test suites.

1.2 Goals

I aim to accomplish two primary goals in this thesis:

- To identify how the quality of test suites affects the grades that they generate.
- To determine the suitability of using mutation testing to evaluate the quality of a grading test suite.

1.3 Thesis Structure & Scientific Contributions

Chapter 2: Literature Review

This chapter includes a review of existing work, first providing a summary of introductory programming education, followed by relevant pedagogical frameworks. I next consider the assessment of students' programs, and approaches to conduct this assessment automatically. Following this, I explore existing work that examines the mistakes that students make. Next, I review software testing techniques, code coverage, and mutation testing. I also consider test generation techniques, fault localisation, and diagnosability metrics since these may also be relevant to evaluating and improving grading test suites. Finally, I primarily conclude that there is a clear scope to evaluate how mutants relate to students' faults, and how effective mutation testing is in comparison to code coverage in evaluating grading test suites. I also conclude that diagnosability metrics may provide some insight into how test suites can impact generated grades.

Chapter 3: Datasets

In order to realise the goals of my research, I require students' solutions to programming assignments to perform empirical studies on. I collected two sets of solutions from several cohorts of the introductory Java programming module at the University of Sheffield. This chapter describes these datasets; the assignments that they are for, the reference solutions of these assignments, mutants that I generate using these reference solutions, the students' solutions themselves, and the tests that evaluate their correctness.

Chapter 4: Gradeer

This chapter outlines my automated grading tool, Gradeer, which I developed while conducting my research. I used Gradeer to execute tests on the students' solutions and mutants within my dataset for my empirical research. Alongside the lead of the University of Sheffield's introductory Java programming module, I successfully deployed Gradeer to grade the module's assignments, and to provide students with feedback accordingly.

Contribution 4.1: An open-source modular automated assessment tool that also enhances manual assessment.

Chapter 5: Investigating the Influence of Test Suite Properties on Automated Grading

This chapter presents the results of my empirical study on how different measurable properties of test suites can impact the grades that they generate. I find that different test suites can yield drastically different grades, and that various measurable properties have a considerable impact on these grades.

Contribution 5.1: Empirical evidence that different test suites can yield significantly varying grades for students' solution programs.

In particular, I find that the number of tests in a suite, the proportion of other faulty solutions it detects faults within, its mutation score, and its uniqueness (a diagnosability metric) have a significant impact on grades. This supports my hypothesis that using mutation testing can improve test suites, as by achieving a maximum mutation score, a tutor can reduce the corresponding impact that test suite inadequacy would have on grades; mutation testing would improve the consistency of automated grading.

Contribution 5.2: A statistical comparison of how various observable properties of test suites influence the grades that they generate.

Chapter 6: What Programming Mistakes Do Students Make?

This chapter describes my qualitative analysis of students' programming mistakes. I used a coding approach, categorising each new mistake that I encountered as I manually examined the students' source code. I present the frequencies of these mistake categories for each subject class, as well as descriptions of the categories themselves.

Contribution 6.1: A qualitative analysis of students' programming mistakes.

Chapter 7: Deriving Mutation Operators From Students' Mistakes

In this chapter, I consider which mutation operators implemented by existing tools would simulate the programming mistakes that I identified in the previous chapter. For the remaining mistake categories, I propose new mutation operators that would appropriately simulate their addition to a reference solution.

Contribution 7.1: Definitions of new mutation operators to simulate students' programming mistakes.

Chapter 8: MutaGen: Implementing Mutation Operators

This chapter describes how I implemented my newly derived mutation operators in my mutation tool, MutaGen. I use abstract syntax tree manipulation to introduce these complex mutants to a program, since this allows the structure of the program's source code to be modified. I also detail the mutants that MutaGen generates from the reference solutions of my dataset's subject classes.

Contribution 8.1: A prototype mutation tool that implements my new mutation operators.

Chapter 9: Evaluating the Suitability of Mutation Operators to Simulate Students' Mistakes

This chapter presents the results of my empirical study to evaluate the performance of artificial mutants to simulate students' mistakes for the purpose of evaluating grading test suites. I use a variety of methods to evaluate the applicability of the coupling effect for mutants and students' solution programs; a traditional coupling evaluation, probabilistic coupling, and my new bidirectional coupling evaluation. I find that mutants are typically coupled to students' faulty programs.

Contribution 9.1: Empirical evidence that the coupling effect holds for mutants and students' solution programs.

I also perform an analysis of sampled test suites to compare the performance of using mutation testing to improve test suites against using coverage, and to evaluate which types of mutants are the most effective. I find evidence that simple mutants are the most effective for guiding test suite construction, but that their use alone does not offer a great benefit over using only a code coverage metric. However, I also find evidence that using both code coverage and mutation testing to guide the development of a test suite is especially effective in improving its ability to detect students' faults.

Contribution 9.2: An empirical comparison of the effectiveness of code coverage and mutants generated by different tools in evaluating the adequacy of a grading test suite.

Chapter 10: Conclusions & Future Work

In this chapter, I summarise the findings of my research. I also use my findings to consider potential avenues for future research, including possible techniques to inform tutors of how they can improve the fairness of their grading test suites.

Chapter 2

Literature Review

2.1 Overview

In this chapter, I consider existing literature that is relevant to the automated assessment of introductory programming courses, and techniques for improving test suites which could be applied in this field.

I first consider introductory programming education (Section 2.2). Following this, I investigate how pedagogical frameworks can inform practices in teaching introductory programming courses (Section 2.3), then how these frameworks relate to programming assessments (Section 2.4). Next, I review various approaches to automated assessment (Section 2.5), with a particular focus on approaches that use software testing. Following this, I examine the mistakes made by programming students that automated assessment should reveal (Section 2.6). Next, with an objective to identify potential means to improve automated assessment with existing techniques, I review software testing (Section 2.7), and techniques to improve its effectiveness in revealing faults; particularly code coverage (Section 2.8), mutation analysis (Section 2.9), and test generation (Section 2.10). I also consider fault localisation (Section 2.11), since its techniques provide some insight into improving a test suite's ability to isolate individual faults, since this may offer a means to make automated

assessment tests only evaluate individual learning outcomes. Finally, I use the observations of this literature review to identify an avenue for my research (Section 2.12).

2.2 Introductory Programming Education

Programming is a required skill in both software engineering and other technical disciplines, both in formal education and in practice within industry [29]. However, becoming a proficient programmer is notoriously difficult, with novices facing many challenges on their path [30]. This provides a clear mandate for effective education in programming; with sufficient guidance, novice programmers can navigate these challenges.

2.2.1 Novice Programmers

Naturally, students enrolled in introductory programming courses lack the knowledge and intuition of experienced programmers, who have often honed their skills over years of practice. Despite this, different novice programmers exhibit vastly different skill levels [31]. There are several possible causes for this. For example, some students may have prior experience; Lahtinen et al. found that 58.8% of students in their study had some experience in programming prior to attending university, and that 40.6% of these self reported as having intermediate programming skill [30]. Kanaparan et al. found a positive correlation between students' enjoyment in programming and their skill [32]; such an emotional connection to programming is also a factor. The personal experiences of Dawson et al. indicate that non-major students are likely to enjoy programming courses less [33], so a novice's field of study may also have an impact on their programming skill.

2.2.2 Challenges in Learning Programming

It is important for educators to understand the problems that students face in order to facilitate their improvement [23,34]. However, these challenges are multifaceted, complex, and even interdependent to some extent; tutors may find it difficult to fully model students' problems. For example, Brown et al. found that students make some programming mistakes at a frequency that tutors cannot accurately estimate [23]. In addition, tutors and students do not reach a consensus on how difficult students find some aspects of programming to learn [30].

Lahtinen et al. conducted a survey of students and tutors to determine which aspects of introductory programming courses are the most challenging [30]. They found that students find some programming concepts difficult to learn, particularly recursion, pointers and references, error handling (i.e. exceptions), standard libraries, and abstract data types. In comparison, students tended to find variables, conditionals, and loops easier to learn. Their study also revealed that students found some aspects of writing programs challenging, including designing a program to solve a particular problem, and splitting a program into a series of procedures. More significantly than these, however, is that students reported finding bugs in their programs especially challenging. This notion is further supported by the work of Lister et al., who found that some students faced a particular challenge in reading and *understanding* programs [31]. They postulate that this may be a prerequisite to effectively writing code; students who cannot understand what the source code of a program means will naturally struggle to write a similar program themselves. This effect could be further explained by students having unviable mental models of programming concepts; their internal understanding of how particular aspects of programs work is simply incorrect [35]. In order to correct students' mental models, tutors must first reveal their flaws, and construct correct models in their place. This requires tutors to be able to understand this aspect of a student's cognition, and to hold knowledge of effective teaching strategies to guide a student towards a correct mental model. Pedagogical frameworks offer a means to enable tutors to identify and understand such challenges

that students face, and to develop effective strategies to assist students in overcoming these challenges.

2.3 Pedagogical Frameworks

Pedagogical frameworks define particular educational philosophies, providing tutors with a means to form teaching and assessment strategies [36]. In this section, I will focus on three particular pedagogical frameworks, two of which seek to model students' behaviour, and another which focuses on the knowledge that a tutor holds.

2.3.1 Bloom's Taxonomy

Perhaps the most influential pedagogical framework is Bloom's Taxonomy; originally intended to enable assessment artefacts to be shared between American educational institutions, it has since been translated into several languages, and has seen multiple variants and extensions [37].

The original taxonomy defines a hierarchical model of cognition, split across six levels of increasing complexity; knowledge, comprehension, application, analysis, synthesis, and evaluation. These levels are clarified and redefined in the revised taxonomy of Anderson et al., which renames each hierarchical group for clarification, and adds a second level of dimensionality to separate knowledge and cognitive processes, giving broad examples of each dimensional pairing [38]. Thompson et al. provide a series of examples for each of the revised model's cognitive levels, geared towards computer science [39]:

- *Knowledge (Remember):* Recalling relevant information, such as a design pattern that has been previously defined and taught in a course.
- *Comprehension (Understand):* Yielding meaning from a given statement, for example by converting an algorithm from one form (e.g. pseudocode)

Table 2.1: The two dimensions of the revised Bloom's taxonomy, with example actions for each. (Original source is Oregon State University, but the link is now defunct; a copy is in Forehand's work [37].)

		Cognitive Process					
Knowledge		*Remember*	*Understand*	*Apply*	*Analyse*	*Evaluate*	*Create*
	Factual	List	Summarise	Classify	Order	Rank	Combine
	Conceptual	Describe	Interpret	Experiment	Explain	Assess	Plan
	Procedural	Tabulate	Predict	Calculate	Differentiate	Conclude	Compose
	Meta-Cognitive	Appropriate Use	Execute	Construct	Achieve	Action	Actualise

to another (e.g. a verbal description).

- *Application (Apply):* Performing a known procedure, such as applying a known algorithm to solve a new problem.

- *Analysis (Analyse):* Decomposing learning material into subcomponents, then determining their relations to one another. One example of this would be to split a programming problem into classes and methods.

- *Evaluation (Evaluate):* Make a judgement based on previously taught standards. For example, writing a test suite which properly exercises a solution program to ensure that it is correct according to a specification.

- *Synthesis (Create):* Combining concepts to produce functionality, such as formulating a solution to a complex problem using a series of algorithms and design patterns. This level has been swapped with "Evaluate" to be the highest level in the revised taxonomy. This follows the notion that independent creation is the ultimate end goal of any educational journey.

Anderson's revised model converts the original "knowledge" cognitive level into its own dimension, and replaces the original level with "remember", since knowledge itself takes several forms of complexity [38]. These new knowledge levels are factual, conceptual, procedural, and meta-cognitive [37]. These two dimensions can be combined to individual actions, which can be used to directly formulate learning outcomes, as shown in Table 2.1. In summary, beginners can remember specific facts; experts have a full understanding of their cognition, and use it to create from nothing.

Johnson and Fuller further examined the application of Bloom's taxonomy in computer science education via a human study, using 54 first-year computer science assessments [40]. This study tasked a group of academics with evaluating which aspects of Bloom's taxonomy each assessment was focused on. These evaluations were directly compared to the opinions of a group of lecturers who delivered each assessment. They found that the beliefs of each group tended to diverge; the lecturers who presented the assessment intended for it to focus on one aspect of the taxonomy, but determining which aspect is unclear from examining the assessment externally. Johnson and Fuller note that this could be due to a module's lecturer simply having greater understanding of the context in which the assessment is applied, or that alternatively there is a fundamental disagreement as to what each level of the taxonomy really means. Furthermore, in this study, most of those tasked with evaluating the assessments had experience with using the taxonomy in other aspects of their profession, suggesting that it is perhaps more likely for tutors to misunderstand the true complexity levels of the assessments' learning outcomes. This may have severe implications for students' learning; assessments may not truly exercise, evaluate, and provide them with experience of relevant learning outcomes, impeding their learning. In addition, Johnson and Fuller found that tutors believed that synthesis and evaluation were not relevant to their modules, instead focusing on application, from a belief that practice is integral to computer science. Johnson and Fuller propose an additional cognitive level beyond synthesis and evaluation to remedy this; higher application, representing a critical approach to application. Pair programming assessments could perhaps satisfy such an aspect.

Fuller et al. propose a further modification to Bloom's taxonomy, with the goal of specifically accommodating computer science [41]. This specifically entails splitting the cognitive dimension in two: *interpreting* (remember, understand, analyse, and evaluate), and *producing* (apply, and create). This loosens the hierarchy of the original model, allowing for theoretical foundations to be paired with practical tasks to support students' learning. This approach is particularly beneficial for computer science and introductory programming, since "learning by doing" is an effective means for students to learn [42–44].

2.3.2 SOLO

The SOLO taxonomy—the *Structure of the Observed Learning Outcome*—was first introduced by Biggs and Collis [45]. Unlike Bloom's taxonomy, which provides a structure for what is required for students to achieve a given learning outcome, SOLO instead focuses on the nature of their response itself. Bloom's taxonomy can be used to define assessment material; SOLO assists with the assessment itself.

SOLO defines several classes for the complexity of students' responses, for which Lister et al. provide examples in the context of computer science, based on a think aloud study of students' responses to multiple choice questions [46]:

- *Prestructural:* the student's response is heavily influenced by misconceptions, unrelated preconceptions, or a clear lack of knowledge. One example of this is mistaking an array's index for its contents.
- *Unistructural:* the student understands part, but not all, of the problem. This can be considered as an "educated guess."
- *Multistructural:* the student understands every part of the problem, but not how those parts relate. For example, a student may trace a method, and come to the correct return value, but not grasp exactly what the method does.
- *Relational:* the student is able to combine each part of the problem into a single whole. Unlike the multistructural response above, a student would be able to understand what the program is doing. Their final answer may still be incorrect if they come to the incorrect answer, such as by skipping over the details after finding a pattern in what a program is doing.
- *Extended abstract:* The student's response goes a level beyond the problem itself, connecting it to some wider context. For example, the student makes a comment on the limitations of an example program's functionality.

Lister et al. note that while SOLO is an effective means of understanding students' responses, it can be hard to apply it where their thought processes cannot be examined [46]. For example, in a multiple choice question, the answers alone do not reveal a student's understanding. Instead, open response questions or comments in a program can reveal this understanding. However, it is important to remember that higher level SOLO responses do not necessarily guarantee correctness; a student can make a relational response that is still incorrect.

SOLO has seen a considerable degree of research and application in computer science education. Sheard et al. performed a study on students' exam scripts, by evaluating their answers to questions in terms of the SOLO taxonomy [47]. They found that there is a correlation between students' SOLO complexity and exam scores. They also identified a correlation between some questions with respect to the SOLO levels of students' responses; for such questions students were likely to give responses of similar complexity. Izu et al. conducted a similar study using programming students' exam responses [48]. Their results corroborate those of Sheard et al., with positive correlations between SOLO levels and exam scores. They also use their results to provide real examples of response levels in programming. Unistructural responses include not understanding loop semantics, while multistructural responses include the incorrect use of loop ranges. These results reveal that SOLO can effectively model students' understanding of a learning outcome.

SOLO may still have some limitations in computer science. For example, Clear et al. note that while different assessors evaluate the SOLO levels of responses with a reasonable degree of consistency, there is still room to improve the ease of categorisation for assessors [49]. Consequently, they suggest additional categories to improve this process, with a focus on establishing that a response is erroneous or omits important details; relational error, multistructural omission, and multistructural error.

2.3.3 Pedagogical Content Knowledge

Pedagogical Content Knowledge (PCK) is a framework which considers a tutor's knowledge in relation to teaching [50]. Specifically, it not only considers that tutors must have a thorough understanding of the content, but that tutors should also hold knowledge of how to teach it effectively [51]. This includes means of representing the content in an easily understood manner, such as particularly effective examples, visualisations, or analogies. In addition, it includes understanding what makes a subject difficult to learn for students; common pitfalls or mistakes, and how they can be avoided.

PCK has seen some use in computer science education. Buchholz et al. developed an approach to train university teachers, with a focus on increasing their pedagogical content knowledge [52]. Similarly, Saeli investigated the PCK of secondary school programming teachers [53]. This included identifying some concepts and topics that such teachers widely found students to struggle with, such as establishing problem solving skills, method parameters, loops, and arrays.

PCK is extended by TPACK, *Technological Pedagogical Content Knowledge*, which includes a component of also understanding available technologies [54]. TPACK combines the applicability of technology to not only the content and pedagogy independently, but also a combination of the two. Therefore, this framework would be relevant anywhere that technology can benefit students' learning, such as automated assessment; the primary focus of my work.

Doukakis et al. conducted a survey of computer science tutors to identify trends in their TPACK [55]. They found that while computer science tutors typically have a good grasp of technology, they often require further training on applying technology in their teaching, with some tutors not using technology to accommodate students' learning at all.

2.3.4 Implications for Learning and Teaching

All three of these frameworks provide some insight into effective teaching of computer science. Bloom's taxonomy allows tutors to construct learning outcomes that target specific knowledge of a topic, while exercising and employing different aspects of students' cognition. SOLO provides insight into how students have engaged with material through assessment, beyond the simple binary of correct and incorrect. TPACK provides a means to consider how the knowledge of tutors themselves impacts their abilities to accommodate students' learning, specifically with the symbiosis of content knowledge, pedagogical concepts, and available technologies.

These frameworks can also relate to one another to some extent. For example, Atun and Usta investigated how TPACK relates to the learning outcomes achieved by students [56]. They found that applying TPACK to construct a learning exercise results in improved uptake of learning outcomes, and improved problem solving. Consequently, if tutors focus on TPACK when constructing learning materials and assessments, their students will achieve learning outcomes with higher levels in Bloom's taxonomy, and provide more complex SOLO responses.

With such relations in mind, one aspect of learning and teaching in which each framework can be considered is assessment. Bloom's taxonomy informs which concepts assessments should target, informing tutors on how to assess learning outcomes of different concepts, while guiding students to develop their cognitive ability. SOLO allows tutors to evaluate *how* students respond to these assessments and learning outcomes, and where they may need additional work, or an adaptation in strategy. Finally, TPACK allows tutors to develop an effective assessment strategy to maximise this information gain, while also creating learning opportunities for students.

2.4 Assessment

In order to evaluate students' understanding of learning outcomes, they must be assessed. There is considerable evidence that students respond well to a "learning by doing" approach to programming [30]. Similarly, a practical approach to assessment is widely deemed to be effective, with tutors often employing programming assignments to assess their students [57]. Other forms of assessment are also effective for gauging students' grasp of learning objectives and providing feedback, such as multiple choice questions [31], and Parson's problems [58]. These types of assessment can be especially effective if used in variety with programming assignments [9]. However, these types of assessment are out of scope for my research, since, in my view, they do not hold the same breadth of research challenges as practical programming assignments.

Such a programming assignment can have two primary goals. First, to serve as a *formative* learning exercise in itself, by providing students with feedback and practical experience, effectively increasing their level of understanding, as modelled by Bloom's taxonomy and SOLO. Second, to provide a *summative* evaluation of each student's understanding of a set of learning outcomes.

2.4.1 Formative Assessment

Formative assessment is only concerned with directly accommodating students' learning, and has several properties, as described by Grover [9]. The primary focus of formative assessment is to provide students with high quality feedback. This feedback can be delivered continuously across different assessments, where tutors first determine students' understanding, consider how it can be improved, and present students with the means to improve their level of understanding. Such feedback is most effective when it provides a detailed explanation on how to improve, rather than a simple declaration on the correctness of a solution [59]. There is also evidence that simpler learning outcomes are better understood by students if they receive this feedback im-

mediately, though it is better to provide some delay for complex outcomes [59], perhaps to allow students time to digest the learning exercise. How feedback is presented is also important, with one on one dialogue between students and educators being particularly effective compared to merely disseminating knowledge with no interaction [60]. Bloom notes that this feedback may be more effective if it is decoupled from grading students' abilities [9, 61], but this does not necessarily mean that feedback should not also be provided in a graded, summative assessments [10, Chapter 10, p. 185].

Formative assessment also integrates some aspects of pedagogical frameworks into assessment. For example, students can understand the goals set by the learning outcomes through the provision of feedback, specifications, or assessment criteria [9]. Furthermore, tutors can build upon their PCK by understanding how students engage with the assessment, and by observing the mistakes that they make. Analysing the SOLO levels of students' responses—for example, by examining their source code—may provide a means to develop this PCK, since it allows tutors to clearly evaluate students' understanding [11]. This PCK prepares tutors for future assessments, which may have higher stakes, such as end-of-module summative assessments, or to restructure the course in future revisions. In addition, a graded assessment can serve as a formative learning opportunity for students; its requirements can guide students towards a particular set of learning outcomes, as students tend to focus on the aspects that have the greatest impact on grades [62].

2.4.2 Summative Assessment

Summative assessment is performed after a set of learning activities, and aims to quantify what target learning outcomes a student has achieved [10, Chapter 10, p. 184]. The results of such assessments are used to produce a final grade for a particular course. By making passing grades compulsory to attain accreditation, summative assessment can be used to ensure that students possess a minimum level of technical competence [10, Chapter 10, p. 197]. In turn, this provides third parties with assurance that students have

gained the knowledge taught in the course [62].

Pedagogical frameworks also provide some insight into effective summative assessment. Bloom's taxonomy can be used to inform the construction of assessment tasks that focus on particular learning outcomes and cognitive levels. For example, Shuhidan et al. found that multiple choice questions tend to only exercise the lower cognitive levels of the taxonomy [11]; perhaps programming assignments would fulfil the more creative levels. In addition, the SOLO levels of students' responses are also often correlated to their grades [48].

2.4.3 Challenges

Several challenges face educators in delivering assessments with quality feedback and fair grading. The first is that tutors must construct an assessment task, and establish a means of providing feedback based on students' responses. This would be significantly hindered if tutors lack the PCK to do this correctly. This may be a prevalent issue in reality, since educators do not always have a firm understanding of the mistakes that students make [63], or which components of a course students find particularly challenging [30]. Another significant challenge is that manually assessing students' responses—such as solution programs—is incredibly time consuming [18, 20]. This is especially problematic, as cohorts of computer science students continue to grow year after year [6, 64]. One approach to manage this time cost is to elicit the help of additional educators, such as teaching assistants. However, this manifests another critical concern; grading inconsistency. Rubrics aim to address this by providing guidelines to grade solutions based on various qualities, but these can be interpreted differently by individual educators [65].

It is clear that while assessment is incredibly important for students' learning, it is by no means a simple process. However, as the concept of TPACK suggests, the correct use of available technologies can enhance an educator's capabilities, and assessment makes no exception.

2.5 Automated Assessment

Automation of the assessment process offers a potential solution to many of the challenges associated with manual assessment, particularly by reducing the amount of labour required by a tutor [57]. Automated assessment offers a good return on investment; it only incurs a single time cost in configuring the automation to assess a programming assignment, regardless of the number of students, whereas manual assessment will take longer as more students are assessed [66]. With computer science and software engineering courses growing ever larger, and the emerging popularity of massive open online courses (MOOCs) with cohorts of thousands of students, automated assessment is required not only to save time, but to make assessment feasible at all [64,67,68].

Existing automated assessment systems are able to generate grades [69] and feedback [70]. Combining these aspects of automated assessment offers multiple benefits beyond saving time. If applied properly, with limitations on solution submission frequency, automated assessment can help students to understand how to improve their solution programs while employing their own testing [71]. In addition, automated assessment provides an even playing field between students; automated graders evaluate solutions consistently, removing the bias that could emerge from assessment by multiple different human examiners [57, 65].

2.5.1 Approaches

There are several different approaches to automated assessment and feedback:

Modelling Approaches

Singh et al. applied modelling to generate feedback for incorrect programming exercise solutions [67]. Their technique assumes that the complete specification for the exercise is known (with a correct reference solution), and that potential

errors for the exercise are predictable. Their technique uses an error model; a set of pairs of potential errors and associated corrections. The technique then searches this error model, in order to apply the appropriate corrections for an erroneous solution; these corrections can be provided to students as feedback. They found that the technique was able to produce feedback for over 64% of the incorrect solution programs. Verifix aims to provide similar feedback of repairs to students' programs [72]. This tool differs, in that it models correct reference solutions and a student's incorrect solution as control flow graphs (CFGs), and uses a verification process to determine the minimum possible fix to make the student's solution equivalent to a reference solution.

Martin and Mitrovic created a constraint-based modelling approach to generate feedback for students' SQL database queries [73]. A tutor defines a set of constraints that students' queries must satisfy, by defining a series of pattern matching rules that the queries are exercised against. Should a student's solution violate one of these constraints, the student is provided with the constraint's associated feedback that the tutor has predefined.

These approaches do have some drawbacks. First, they ideally require a model specific to each individual programming task, as evidenced by Singh et al., who found that their approach was considerably less effective in correcting students' faults when using a general model compared to task-specific models [67]. Second, such approaches require particular PCK of the tutors to create these models. For example, a tutor must be aware of all of the mistakes that students frequently make to create an error model, or to create constraints that would identify them. As I have previously discussed, such knowledge is not necessarily held in an accurate and complete manner [23]. Finally, and perhaps most critically, such approaches require a high level of TPACK; tutors must not only know how to derive a model to identify students' mistakes, but they must also hold the knowledge to represent this model using a language specific to a modelling tool. For example, Singh et al.'s approach requires error models to be written in EML (Error Model Language) [67]. Due to these limitations, I will explore other approaches in my work.

Machine Learning & Data-Driven Approaches

Machine learning, clustering, and other data-driven techniques are particularly en vogue across computer science, and the field of automated assessment is no exception. One tool which applies machine learning is *sk_p*, which aims to automatically correct students' Python programs to provide them with feedback on how to improve their programs [17, 74]. The tool trains a neural model on fragments of tokenized solution programs, and uses this model to predict potential correction fragments for a new solution program for the same programming task. The tool then evaluates the fitness of the correction candidates using a set of traditional software tests. After training the model, sk_p can produce corrections in ∼5.6 seconds, which serve as near instant feedback for students.

Combéfis and Schils use unsupervised clustering to aid the assessment of large quantities of solution programs [15]. Their technique uses abstract syntax trees (ASTs) to evaluate the similarity between two solutions. The technique uses these differences to apply a traditional clustering technique, yielding a representative solution for each cluster. A tutor can then classify and provide feedback for each of these representative solutions; this feedback can be supplied to the students who wrote the solutions in the according clusters. In addition, if another student submits a new solution that matches an existing cluster, they can be given the appropriate existing feedback with no additional effort from a tutor.

These types of approaches present unique challenges in their application. First, there is a widely considered limitation in understandability for complex data-driven approaches, such as deep neural networks; it is hard to understand exactly *what* a neural model is doing [75]. This makes using such approaches for grading difficult; there is no means to ensure that students are being graded fairly, and there is no guarantee that students perform adequately well to attain accreditation. This does not necessarily prevent such systems from being applied to generate feedback, but should a machine learning system generate inaccurate feedback without oversight, students' learning will be

hindered; they would be confused as to why they are receiving feedback that is irrelevant to their solution, and tutors would not know why this erroneous feedback is generated. Second, these approaches all require large corpora of students' solutions; they can only be applied in situations where many students' solutions are available, such as MOOCs and very large courses. This also presents a "chicken or the egg" scenario in some cases; existing solutions are required to provide feedback for new solutions. For established programming assignments this may be feasible, but supervised machine learning cannot be used for new programming assignments. Unsupervised approaches, such as clustering, would still be applicable in such situations, however.

While these data-driven approaches present promising solutions to—and interesting challenges for—the problems of automated assessment, I consider them out of scope due to the dataset that is available to me. The students' solutions that I am able to collect for this work are numerous, and plentiful enough where automated assessment would provide a clear benefit, but not of a sufficient quantity for a data-driven approach to be truly effective. Instead, I consider other approaches.

Dynamic Analysis & Software Testing Approaches

A common approach to automated assessment is to dynamically evaluate students' solution programs by running a set of tutor-defined test cases against them [64, 76]. In their simplest form, test cases check that a program (or a component of it) returns the expected output for a given input. Automated assessment systems that use this approach have been deployed for decades; Hollingsworth implemented his assessment tool that checked the output of punchcard programs in 1960 [77, 78].

The approach has since evolved, with modern automated assessment tools often using conventional software testing frameworks to automatically assess students' programs [79]. Some systems also support some specialised features in programming languages; for example, Insa and Silva's *JavAssess* library uses abstract syntax trees and reflection to manipulate students' code to

assess and correct students' Java programs [18]. Software testing is able to both grade solutions and provide students with feedback. Grading can be performed by observing the results of tests, and using the proportion of tests that pass as a grade [18]. Tests can provide feedback by separating the tests into public and private sets; the private tests are kept secret and are used to generate grades, but the public tests are given to the students to execute on their own machine, allowing them to understand where their programs are deficient [57]. Furthermore, tutors can use automated assessment tools to run students' tests against their own code, and the code of other students [80]. Requiring students to write these tests acts as a learning exercise; students understand the value of testing their own code [76]. Reusing students' tests for other students' programs could help tutors to save time in assessment.

Modern test-based automated assessment systems are often also deployed as web applications, or integrated with networked infrastructure [57, 64, 81, 82]. This benefits students, since it provides them with a simple means to both submit and get timely feedback for their code. Web-based assessment systems can also benefit tutors, by offering features that save time, and provide insight into students' progress. For example, CodeAbility provides tutors with analytics tools for students' learning, as well as a system to facilitate the exchange of learning resources between different tutors and institutions [83]. Such systems can also easily incorporate dynamic analysis techniques other than software testing. For example, Combéfis and Paques developed *Pythia*, a grading system that is inspired by coding competition systems [8]. This system can impose limits on the memory and execution time of a solution program. Combéfis and Paques also propose that the complexity of a program can be evaluated by measuring its execution time for inputs of different sizes.

Cheon and Leavens note that while software testing is effective for improving the correctness of a program, the process of creating unit tests is both time consuming and challenging [84]. Similarly, this also applies to test-based automated grading, with educators requiring the appropriate TPACK to develop an effective grading test suite. As previously discussed, lacking knowledge of students' mistakes produces a challenge [23]; lacking awareness

of students' mistakes means that tutors may not write enough suitable tests. Furthermore, tutors must know how to use the appropriate testing framework to write their tests. Lacking this TPACK may cause tutors to write a test suite which does not detect students' mistakes, or even worse, misclassify correct solutions as being faulty. An incomplete test suite would not detect mistakes made by students, potentially reinforcing poor programming practices by awarding them with high grades. Inaccurate test suites could unfairly recognise correct solutions as incorrect, confusing and demotivating students.

Despite these issues, I still believe that dynamic analysis and software testing approaches to automated grading are effective. First, while tutors must hold the technical knowledge to write tests, software testing frameworks are widely used in industry [85]; they have an abundance of documentation and learning material for tutors to use. Second, by requiring that students write their own tests, automated assessment tools can help students to understand why testing is important [76, 86, 87]. Finally, the requirement of PCK, such as the mistakes that students make, is required for effective assessment regardless of the methodology that is employed. One potential benefit of a software testing approach is that many techniques have been applied to improve the detection of faults in industry. In my work, I explore how such techniques can also be used by tutors to improve their tests for automated assessment.

Static Analysis Approaches

While software testing and other dynamic analysis approaches to automated assessment are effective in evaluating the correctness and performance of students' code, there are some aspects of programming that they cannot assess, such as code style. Some automated assessment systems augment their functionality by integrating static analysis [57, 79, 88, 89]. For example, the *Praktomat* automated assessment system uses *Checkstyle* to evaluate the style of students' Java code [57, 90]. Checkstyle works by evaluating a series of rules against a program's source code. These rules check for adherence to a programming language's style guidelines, such as checking

that a variable's name is in camel case. The tool reports any rule violations, which can be used directly as feedback for students. These violations can also contribute to grade generation. In order to prevent the usage of unfair means, Praktomat also uses *JPlag*, a plagiarism and collusion detection system, which checks for similarities between students' code [91]. Without the assistance of static analysis tools, such activities would have to be manually performed by educators, incurring a huge time cost.

Static analysis still requires appropriate TPACK of tutors, particularly requiring the knowledge of which static analysis rules should be selected in an assessment. However, this is likely less challenging than writing software tests, since the rules have already been written and possibly integrated with an automated assessment tool; selection is much simpler than implementation.

2.6 Students' Mistakes

In order to effectively assess students' programs, either automatically or manually, tutors must first be able to identify the mistakes which they make [92, Chapter 10, pp. 195-196]. By revealing such mistakes, an assessment can evaluate gaps in students' understanding, and consequently, their grasp of the required learning outcomes [10, Chapter 10, p. 211]. This ultimately forms a key PCK requirement; tutors must understand the mistakes that students are likely to make in order to construct an assessment that reveals them. However, this PCK is not necessarily universally held [23]; tutors must ensure they hold it, or acquire it. Existing work has explored which mistakes students commonly makes when completing programming tasks. Such mistakes either directly impact the functionality of the program, or the style and quality of its source code.

2.6.1 Functionality Mistakes

Brown et al. conducted a study investigating the faults that introductory programming students make in software [23]. This involved analysing the Blackbox dataset, which consists of the source files of programs collected at every compilation of consenting introductory programming students using the BlueJ IDE over a two year period. Since entries in this dataset are collected at compile time, rather than when a student chooses to submit a completed program, many of the collected sources include uncompilable code. This may provide some indication of students' behaviours and mistakes while they write programs, rather than implementations that students deem to be correct. Their analysis targeted 18 fault classes previously identified in interviews with educators [93], including their frequencies, and the time taken by students to repair them, as shown in Tables 2.2 & 2.3.

Some of the mistakes identified by Brown et al. were made more frequently than others. For example, the use of *incorrect brackets* is particularly common, perhaps due to students making unintentional typos. Albrecht and Grabowski attribute such errors to "sloppiness", and also found that they are fairly common, accounting for 17% of their students' incorrect solutions [94]. Another common fault is calling methods with the *incorrect argument types*; students may not remember or understand that they must convert between types, indicating a failure to grasp this learning outcome, as modelled by Bloom's taxonomy. For example, students may use a pure string that contains a numeric value, rather than parsing it first. By contrast, students are unlikely to make mistakes of using completely incorrect symbols throughout their programs, such as using *curly brackets around a condition*.

Students also appear to find some mistakes easier to fix than others. Simple syntax errors, such as using the incorrect symbol ordering for comparators, appear to be particularly easy to fix; they often take less than a minute to fix. This is likely due to the Java compiler reporting exactly where an unexpected symbol is, so students can easily identify—then resolve—the problem. Brown et al. identified three mistake classes that exceeded the repair limit of 1000

Table 2.2: Functionality mistakes in students' Java programs, as identified by Brown et al. [23] There is a time to fix limit of 1000 seconds for each mistake.

Mistake	Description	Freq.	Median Time to Fix (seconds)
Syntax Issues			
Bad Equality Comparator	Using and assignment (`=`) instead of equality comparator (`==`)	405748	113
Unbalanced or Incorrect Brackets	Issues with parentheses, including being unbalanced, mixing types, or using incorrect types, e.g. `(x == 1]`	1861627	17
Bad Logical Operator	Using simple logical operators (`|` or `&`) instead of short-circuit operators (`||` or `&&`)	61965	1000
Unexpected Semicolon (Conditional)	Semicolon after a condition, e.g. `if (a == b); {...}`	108717	387
Unexpected Semicolon (Method)	Semicolon after a method's header	86606	50
Bad For Separators	Not using semicolons as separators in a for statement, e.g. commas	6424	36
Curly Brackets Around Condition	Using curly brackets instead of parentheses for an if statement's condition, e.g. `if { x == 0}`	284	24
Reserved Word	Using a reserved word as an identifier's name, e.g. `int new;`	2568	22
No Method Parentheses	No parenthesises after a method call, e.g. `method;`	43165	34
Bad Comparator	Incorrect symbol ordering for a comparator, e.g. `=<` instead of `<=`	9381	13
Types in Arguments	Including arguments' types when calling a method, e.g. `method(int x);`	117295	23

seconds; *bad logical operators*, *string equality*, and *ignored returns*. Since these mistakes typically exceeded the fix time limit, it is likely that students often did not repair them at all. However, while these mistakes certainly can cause major issues in a program's execution, they do not always result in a clear fault. For example, a student may be using a compound conditional statement, with the first clause checking whether an object is null, and the second evaluating the object's properties. A correct short circuit logical operator would prevent the second statement from being evaluated if the object is null, but an incorrect regular logical operator would evaluate the second statement, throwing an exception. If the student makes the mistake

Table 2.3: (Table 2.2 continued.) Functionality mistakes, as identified by Brown et al. [23]

Mistake	Description	Freq.	Median Time to Fix (seconds)
Type Errors			
Incorrect Argument Type	Calling a method with an argument of the incorrect type	1034788	59
Incorrect Return Type	Assigning a variable with a method of a different return type	32435	71
Other Semantic Errors			
String Equality	Using `==` instead of `.equals()` for string comparisons	274387	1000
Bad Static Call	Calling a non-static method from a static context	202017	48
Ignored Return	Ignoring a method's return value	274963	1000
Missing Return	A non-void method has a branch with no return statement	817140	38
Incomplete Implementer	A class should implement an interface, but is missing necessary methods	186643	107

of using the regular operator, it may not always cause a fault; it could be the case that the object is rarely null, so an obvious error is unlikely to occur for the student to directly observe. This is the case for all three of these mistakes that exceed the time budget for repair; they are arguably more related to the overall quality of a program, and a student's adherence to good programming practices.

Some mistakes do not apply universally to all programming languages. For example, the mistake of using `==` to check for *string equality* in Java programs does not apply to other languages; in JavaScript, one can use the conventional strict equality operator (`===`) without any issues. Similarly, errors related to object-oriented programming do not occur for procedural programming languages, such as C [94].

Brown et al.'s identified mistakes each fall into one of three broad categories: syntax errors, caused by using the incorrect syntax; type errors, caused by

students' misunderstanding of variables' types, and other semantic errors [23]. McCall and Kölling also studied the BlackBox dataset; they observed the mistakes made by students, in order to form a hierarchical categorisation of students' mistakes, with 80 individual error categories [95, 96]. This hierarchy forms broader groups of mistakes. For example, the "method call" group contains individual categories, such as a *parameter number mismatch* (a method is called using too few or too many parameters), and *parameter types included* (a parameter's type is included when it is passed to a method call). Since mistakes can represent a student's failure to fully grasp a learning outcome, a tutor can evaluate a student's understanding of such learning outcomes by designing tests to detect groups of mistakes that are associated with them.

2.6.2 Style & Quality Mistakes

Style guides offer a set of rules for writing code, and are often defined by software vendors or institutions, with a particular focus on consistency between code written by different programmers [97]. Such style guides provide a means to ensure the readability and maintainability of software. Similarly, students should be taught and encouraged to write code that meets a satisfactory level of readability and maintainability [98]. However, style guides are not perfect; they can contradict one another, often lack a theoretical or empirical foundation, and provide different definitions of what programming style truly refers to [99]. Oman and Cook constructed a general programming style taxonomy to remedy this, through the examination of style guides and static analysis tools [99]. Their taxonomy groups style principles into four broad categories: *general practices*, e.g. testing and debugging; *typographic*, e.g. naming conventions; *control structure*, e.g. control flow; and *information structure*, e.g. data structures. The latter three of these categories are split into two groups: *macro*, representing higher level concepts, such as modularisation; and *micro*, focusing on specifics, such as rules regarding whitespace. This taxonomy effectively provides a general approach to good program style. While individual style guides are effective for specific programming languages, it may

be worthwhile for tutors to focus on the generalisable aspects presented by this taxonomy, since students will be able to apply them across any programming languages that they use in the future.

De Ruvo et al. investigated students' style mistakes, by examining 19000 students' code samples [100]. They refer to the mistakes that they identify as "semantic style" mistakes; mistakes which do not directly impact functionality, nor outright violate simple style rules (e.g. variable naming). Instead, such mistakes reveal a lack of understanding of some programming concepts, manifesting as code snippets that experienced programmers would avoid writing, such as the mistake in Figure 2.1. Some of their identified mistake classes include the use of *unnecessary if / else-if* statements (an if condition that always evaluates to true), *empty if bodies*, and *useless declarations* (a variable is defined in one statement, and only assigned a value once).

```
if(a == true)
  return true;
else
  return false;
```

Figure 2.1: A semantic style issue, which could be more eloquently written as `return a;`, displaying a lack of understanding of boolean types and variable returns [100].

Keuning et al. also focused on mistakes that impact the quality of a students' source code, but do not necessarily violate simple style rules [101]. They conducted a study on the Blackbox dataset, by examining the solutions' rule violations that are reported by PMD [102, 103], a static analysis tool. For each rule, they determined the number of violating solutions, and their time to fix, similar to the work of Brown et al. [23]. They found that such quality issues were both common and rarely repaired by students, even if conventional static analysis tools were provided. As a result, they suggested that tutors should provide students with automatically generated feedback using static analysis tools, like PMD, to serve as feedback to correct such mistakes.

2.6.3 Implications

This existing work reveals that it is imperative for tutors to understand the diverse programming mistakes that students can make, since they reveal which learning outcomes students struggle to grasp. Concerningly, the existing work also reveals that tutors do not always accurately understand which mistakes students make, and how often [23], posing a challenge; how do tutors ensure that they can identify students' mistakes? For style mistakes, static analysis tools offer a potential solution, but it is not so simple for functionality mistakes; tutors can use testing approaches to detect students' faults, but they must have the sufficient TPACK to write tests that are effective. In my research, I will consider possible techniques to help tutors to better understand the faults that students can make, and how to identify them, to ensure that tutors can deliver accurate automated grading and feedback.

2.7 Unit Testing

Unit testing involves the definition of individual tests to evaluate the correctness of individual components of a software system, and reveal faults that are present within them [104]. A unit test executes such a component with a predefined input. The test then compares the component's output or internal state to that of the test's expected value. These comparisons are referred to as assertions. If the component's output or state matches those defined by a test's assertions, the test passes, but fails otherwise. Test-based automated assessment systems use such unit tests to evaluate the correctness of individual components of students' programs.

2.7.1 Regression Testing

Regression testing is a technique which aims to improve the effectiveness of testing in software development, by running a test suite whenever a system

under test is modified, in order to detect faults that are introduced by new changes to the program [105]. Since the test suite is executed whenever a significant change is implemented, every defect that it can potentially identify is caught, thus improving the correctness of a system while it is under development. This practice sees considerable use in the software engineering industry, and is observed to improve software quality. Automated grading could potentially be represented as regression testing, since a student's solution is essentially a set of changes made to a correct and optimal reference solution. Hence, tests created that pass on a reference solution can be used to validate potentially faulty solutions that are written by students.

While regularly executing regression tests is known to be time consuming in software development [106], this issue may not effect grading in the same manner. This is due to the fact that even if automatic grading does require a high amount of computational time, it would still reduce the amount of time required by a tutor to mark assignments manually. Additionally, while running regression tests may cause a block in software development, a tutor would not necessarily have to wait for an autograder to finish. Tutors can make more efficient use of their time and resources to prepare lecture materials, have direct contact time with students, or complete any other tasks while an autograder is running. Furthermore, if more throughput of testing is required, investment can be made into more computational hardware to run automated assessment tests at a higher speed.

2.7.2 Test Goals & Adequacy Metrics

Since suites of unit tests may vary in their ability to detect faults, it is necessary to evaluate their quality. However, this is impossible to determine directly, since real faults are usually not known without being revealed through testing; a program may contain unknown faults that are not revealed by a deficient test suite [28]. Instead, test goals provide targets for suites to achieve; by satisfying more test goals, a test suite should also be more capable of reliably detecting real faults [24, 27]. The number of test goals that are achieved

can be used to form a test adequacy metric; a numeric estimation of a test suite's fault detection capability. Software engineers use adequacy metrics as a proxy for real fault detection when developing test suites, without requiring knowledge of real faults. Similarly, tutors may be able to use test goals and adequacy metrics to evaluate their automated assessment test suites. By considering unachieved test goals as analogous to students' mistakes or failure to implement some aspects of a task's specification (i.e. unachieved learning outcomes), tutors could use test goals to predict their suite's ability to detect such mistakes or deficiencies. Several different test goals and adequacy metrics could be used for such purposes.

2.8 Code Coverage

Code coverage aims to capture test adequacy by evaluating which parts of a program a test suite has executed [107, 108]. This is performed by using individual parts of a program's source code as test goals. These test goals are achieved once they are executed by a test, or in other words, are covered. These coverage goals can also be summarised as a single coverage ratio; the proportion of goals that have been achieved. Coverage can reveal clear deficiencies in a test suite. For example, consider an empty test suite; it never executes the program under test, and cannot reveal any faults. If a test is added to the suite, it will execute more of the program under test, and now has the possibility to reveal faults that are present in the executed code. However, some parts of the program may still not be covered by the suite; faults in these locations cannot be revealed by the suite. Only by fully covering every location where faults could appear can every potential fault possibly be revealed.

There are several variants of code coverage [24, 107], as demonstrated in Figure 2.2, including:

- *Statement Coverage (Figures 2.2d & 2.2e):* Each individual statement of a program is used as a coverage goal.

```
int abs(int x) {
  if(x > 0) { return x; }
  return -x;
}
```

(a) Example method; returns the absolute values of an integer.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
}
```

(b) Test suite α.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
}
@Test void testNegative() {
  assertEquals(2, abs(-2));
}
```

(c) Test suite β.

```
int abs(int x) {
  if(x > 0) { return x; }
  return -x;
}
```

(d) Statement coverage; suite α: 67%

```
int abs(int x) {
  if(x > 0) { return x; }
  return -x;
}
```

(e) Statement coverage; suite β: 100%

```
int abs(int x) {
  if(x > 0) { return x; }
  return -x;
}
```

(f) Line coverage; suite α: 50%

```
int abs(int x) {
  if(x > 0) { return x; }
  return -x;
}
```

(g) Line coverage; suite β: 100%

```
int abs(int x) {
  if(x > 0) { return x; }
  return -x;
}
```

(h) Branch coverage; suite α: 50%

```
int abs(int x) {
  if(x > 0) { return x; }
  return -x;
}
```

(i) Branch coverage; suite β: 100%

Figure 2.2: Example test suites and their coverage measurements for a method which returns the absolute value of an integer.

- *Line Coverage (Figures 2.2f & 2.2g):* Each line of a program is used as a coverage goal. Most lines typically contain a single statement, with the exception of inline conditionals or initialising multiple variables, so this is often similar to statement coverage in practice.

- *Branch Coverage (Figures 2.2h & 2.2i):* Every possible control flow divergence is used as a coverage goal, such as the true and false outcome of an if statement. If every branch is covered, then it follows that every reachable statement is covered, since each statement must be within a branch, or is executed before a branch has been reached [24].

- *Path Coverage:* Considering a program as a control flow graph, branch coverage measures consider the proportion of *edges* of the graph that are executed by tests. By contrast, path coverage measures the proportion of possible *combinations of edges* that are executed by tests, including the order in which the edges are explored [109]. This coverage metric is especially challenging to evaluate, due to the complexity of programs' control flow graphs, especially where programs use loops. For this reason, I consider it out of scope for my research, since its complexity makes it less viable for tutors to use to inform the design of automated assessment tests.

Code coverage has been shown to be correlated with the improved detection of real faults [108]. Therefore, coverage should assist tutors in developing their test suites for automated assessment, since it can reveal the aspects of a task's specification which a test suite does not evaluate. Coverage is also fairly computationally inexpensive to evaluate in comparison to some other test adequacy metrics [107]; it will not introduce too much of a time cost to use in the development of grading test suites.

However, achieving high code coverage does not guarantee that all faults are detected; it is possible to attain 100% coverage without actually testing for a program's correctness [25]. This is further corroborated by Zhang and Mesbah's investigation of the role of assertions, where they find that both the number of assertions and the assertion coverage (the proportion of statements

that are directly covered by assertions) of a suite is strongly correlated to the suite's effectiveness [25].

Despite this limitation, I will consider line coverage in my research, primarily as a baseline adequacy metric to compare other techniques against. This will allow me to evaluate how well other techniques can help tutors to understand how their suites reveal students' faults, in comparison to this simple existing coverage metric.

2.9 Mutation Analysis & Mutation Testing

An ideal approach to evaluating the adequacy of a test suite is to exercise it against a set of faults. However, it is impossible to evaluate a test suite's ability to detect unknown faults directly. Instead, testers can use artificial faults, called mutants, to evaluate test adequacy [28]. A tool is used to generate a set of mutated variants of a program, which each contain one artificially seeded fault. These faults are created via a set of mutation operators, each of which serves as a rule to introduce a particular type of fault to a given correct program. For every location that a mutation operator can be applied to a target program, a mutant program is produced [110]. Mutation operators cover a broad range of potential changes, including the replacement of binary operators and changing scalar values.

Each mutant functions as a test goal, which is achieved if it is *killed* (i.e. detected) by any test in a test suite. *Mutation testing* provides a single test adequacy metric from the proportion of mutants that are detected by a test suite, called the mutation score [26]. *Mutation analysis* extends upon this, a process that involves evaluating the properties of individual unkilled mutants, such as their locations, the operator applied, whether they are executed by tests, or even the exact change that a mutant has made. This information can be used by a developer to better understand how their test suite is deficient, and therefore, directly enhance it by creating new tests with the explicit goal of killing these mutants.

Mutation testing subsumes code coverage metrics; to detect a mutant, a test suite must first execute the mutated code, and also make robust assertions to reveal the change that the mutant has made [107]. Furthermore, several studies have shown that mutation testing is at least as good as coverage, if not better, in estimating the effectiveness of a test suite [27, 111]. This effect is illustrated in Figure 2.3. In this example, there is a simple mutant on the method's second return statement (Figure 2.3a). Test suite α (Figure 2.3b) does not cover the mutant; it cannot reveal it. By contrast, test suite β (Figure 2.3c) both covers the mutant, and makes an assertion that is able to detect the fault. Finally, test suites γ and δ (Figures 2.3d and 2.3e) both achieve 100% coverage, but do not reveal the mutant. Suite γ performs a valid assertion, but zero is equal to "negative" zero, so the test will pass; the mutant is not detected. Suite δ does not include an assertion for the faulty branch; the mutant is executed, but the test never checks the correctness of the method's return value.

2.9.1 Core Hypotheses

Since the number of possible faults for any given program is near-infinite, mutation only focuses on a subset of the possible faults, in an effort to provide an approximation of the whole set [26]. This approximation relies on two hypotheses:

Competent Programmer Hypothesis

The Competent Programmer Hypothesis (CPH) dictates that programmers tend to have a high degree of competence [112]. Therefore, faulty programs should typically contain only a few simple faults, which can be corrected with minimal changes to the program. The simple nature of mutants reflects this behaviour, making single changes to a program [26].

It is possible that the CPH does not hold for programs written by students enrolled in introductory programming courses, as they are inexperienced

```
int abs(int x) {
  if(x > 0) { return x; }
- return -x;
+ return x;
}
```

(a) Example mutation of a method, displayed with diff notation.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
}
```

(b) Test suite α; 50% line coverage, does not detect the mutant.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
}
@Test void testNegative() {
  assertEquals(2, abs(-2));
}
```

(c) Test suite β; 100% line coverage, detects the mutant.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
}
@Test void testZero() {
  assertEquals(0, abs(0));
}
```

(d) Test suite γ; 100% line coverage, does not detect the mutant.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
}
@Test void testNoAssert() {
  abs(-4);
}
```

(e) Test suite δ; 100% line coverage, does not detect the mutant.

Figure 2.3: Example mutation of the method from Figure 2.2, and how different test suites evaluate it. The mutant is represented in diff notation; the red line starting with '`-`' is the original line, which has been replaced with the green line below it, which begins with '`+`'.

programmers. These students' programs may contain many faults; single mutants would not necessarily simulate their programs' faultiness. However, it may be reasonable to model students' faulty solutions as a series of individual faults; mutants would still represent such individual faults.

The Coupling Effect

The other core hypothesis of mutation is the coupling effect, which dictates that if a test suite is sensitive enough to detect simple faults, it must also be capable of detecting more complex faults [113]. A test suite that kills simple mutants should therefore detect complex real faults. A real fault is *coupled* to a set of mutants if the tests that detect the real fault also kill the mutants. Just et al. have shown that the majority of real faults from five open source programs couple to mutants, and that there is a correlation between real fault detection and mutation score [28].

Since faults introduced by inexperienced students may differ from those introduced by experienced programmers, it is important to demonstrate that the coupling effect holds for students' faults before using mutation to enhance automated grading techniques. Additionally, if every student's fault is coupled to at least one mutant, mutants may still reveal inadequacies in grading test suites. Accordingly, the CPH may not necessarily need to hold for mutation to have some benefit to automated grading.

Chen et al. have proposed an alternate coupling measure, called *probabilistic coupling* [27], which derives an estimated probability, p, that a real fault, f, is detected given that a test goal, g_i, is satisfied (e.g. a mutant is detected), or $p = \mathbb{P}(\text{detect } f \mid g_i \text{ is achieved})$. Probabilistic coupling offers some insight into how well a mutant captures a test suite's adequacy in revealing a real fault. Using the maximum probability for a set of test goals for the real fault allows this insight to be gained without knowledge of every possible real fault, and without an impact from irrelevant test goals (e.g. mutants that are not covered by a test). While conventional coupling shows that a test suite that detects real faults also detects mutants, probabilistic coupling essentially

reveals the inverse; the detection of mutants is associated with the detection of a real fault. In the context of automated grading, probabilistic coupling would reveal that a test suite that can detect mutants also identifies students' faults, and can therefore help to provide sufficient feedback, and grades that are not too lenient.

Potential Weaknesses

Gopinath et al. conducted a study to determine the reliability of the CPH, using real faults collected from projects hosted on Github that were written in Java, C, Python, or Haskell [114]. They found that real faults typically modify three to four tokens, introducing doubt to the CPH, which suggests that bugs introduced by competent programmers should only modify one token. This is perhaps supported by Just et al., who found that some real faults are not coupled to mutants, particularly those that are due to algorithmic errors [28], which may affect multiple tokens. However, Gopinath et al. also note that these hypotheses do not necessarily have to hold in order for mutation testing to be an effective means of evaluating test adequacy. Instead, it is better to determine if test suites that kill mutants are also able to detect real faults in realistic testing scenarios.

2.9.2 Effectiveness

Existing work has evaluated the effectiveness of mutation testing for evaluating test suites' adequacy. However, the results of various studies using different techniques have often produced drastically different outcomes [27].

Andrews et al. conducted one of the earliest comparisons between mutants and real faults [115]. In their study, they compared the detection rates of mutants to those of known faults in eight C programs, using 5000 test suites that were constructed by randomly sampling individual tests. However, their study is limited by their subject programs; only one of the eight subjects includes real faults, with the remaining seven only including manually seeded

faults. For the subject with real faults, Andrews et al. found that the proportion of killed mutants was analogous to the proportion of real faults detected. They also found that mutants are harder to detect than real faults, providing supporting evidence for the coupling effect. For manually seeded faults, they found that mutants differ significantly, and that mutants are easier to detect than such faults. Namin et al. repeated this study using a different mutation tool, Proteum [116], and only found a weak correlation between mutation score and real fault detection for the subject with real faults [117]. They also found that other elements can have an impact on test adequacy studies, such as the mutation operators used, the number of tests in each suite, and the language that subject programs are implemented in. For example, it is much less likely for a developer to introduce a memory leak in a language with a garbage collector, such as Java, than a language where memory must be managed manually, such as C. Mutation operators that disturb memory management, such as those proposed by Wu et al. [118], would not necessarily be relevant to a language like Java.

Just et al. also evaluated the use of mutants to simulate real faults with respect to software testing, using real faults and test suites from five open source Java programs [28]. In their study, they find that the majority (73%) of real faults are coupled to mutants, and that there is a positive correlation between mutation score and the detection rate of real faults. For some of the uncoupled real faults, it would be possible to implement new or enhanced mutation operators to create mutants that they couple to. For example, some faulty programs fail to call a transforming method where necessary, such as a method that sanitises strings. This could be simulated by a mutant that replaces a method call with its parameter. However, for the majority of real faults, it would be extremely difficult or impossible to create a mutation operator that allows them to be coupled, due to the requirement of context-specific knowledge. Such uncoupled real faults include incorrect algorithms, the inclusion of additional incorrect code, and invariant violations. Just et al. also investigated how the effectiveness of mutation testing's adequacy evaluation compares to that of code coverage. Their analysis involved comparing the correlations of mutation score and statement

coverage to the real fault detection rate. They found that the correlations for mutation score are positive, and typically slightly higher than those of statement coverage. They also found that a positive correlation for mutation score remains even if the test suites' coverage is controlled. These results indicate that mutation testing can be a more effective measure of test adequacy than code coverage.

Papadakis et al. conducted another study to evaluate the correlation between mutation scores and the detection of real faults, using two datasets consisting of Java and C programs, respectively [119]. Their results identify a positive correlation between mutation score and the real fault detection rate, corroborating those of Just et al. However, they also found that when the size of each test suite is controlled there is only a weak correlation. This suggests that the size of a test suite has a significant impact on the real fault detection rate; suites with more tests kill more mutants and detect more real faults.

Chen et al. note that the results of these existing studies often contradict one another, and suggest that this is due to the differences, and deficiencies, in their methodologies [27]. One of their critiques is that it is unrealistic to evaluate test adequacy with test suites that are generated by randomly sampling a target number of tests from a wider set of existing tests. Instead developers would write tests targeting a particular bug or new feature, or to increase an adequacy metric, such as by increasing code coverage. In addition, not all tests are created equal; they may exercise parts of a program completely distinct from a real fault, use different numbers of assertions, or simply be ineffective at identifying faults. They consider that it may be preferable to construct suites by using a realistic test selection process, such as those employed by the search algorithms of automated test generators.

Chen et al. also reason that the existing work may include some statistical analysis issues that arise due to the use of correlations. One of the problems that a correlation analysis presents is that explanatory variables, such as mutation score and test suite size (e.g. Papadakis et al. [119]), may be highly correlated to one another. In such an event, it is impossible to conclude which of the variables causes a relation from the correlation alone. Controlling one

of the variables, for example suite size, would also cause the correlation for the other variable alone to be weaker. Instead, Chen et al. suggest that it is preferable to determine the impact of each variable, such as by performing multiple linear regression and comparing the coefficients of the explanatory variables, or by performing an analysis that explicitly handles correlated variables, such as by examining how adequacy metrics improve a model's predictive power after the size of a test suite has already been accounted for.

In addition, Chen et al. suggest that the use of point biserial correlation to determine the relationship between mutation score and a real fault being detected or undetected (e.g. Just et al. [28]) is flawed. This is due to the maximum correlation being dependent on the probability that the real fault is detected; the maximum possible correlation is 0.8, and depending on the detection probability this may be even lower. If the results of such correlations are interpreted by the same guidelines as conventional correlations, with 1.0 indicating a perfect positive correlation, the relationship may be underestimated; two truly perfectly correlated variables would be assumed to not be perfectly correlated. This effect means that where detection probabilities are extremely low or high, the correlation will be severely underestimated. Another problem that they identified for correlation analysis is that if a random test selection process is used, the resulting correlation is heavily influenced by the probability that a randomly constructed test suite contains a test that detects a real fault. This probability is itself influenced by the proportion of such fault revealing tests in the complete test set.

Considering the limitations of the existing studies, Chen et al. suggest an alternate approach to evaluating the effectiveness of mutation testing, while taking the size of a test suite into account. This approach involves growing test suites according to one of the adequacy criteria being evaluated. Given a target adequacy criterion, a_t, (e.g. mutation score), a test suite is grown by randomly selecting an additional test that increases a_t. As the suite is grown, the suite size, real fault detection probability, and adequacy (a_t) is measured and recorded. Once a_t is fully satisfied, the growth process for the suite is halted. This can be repeated using different adequacy criteria for a_t, allowing

for the different adequacy criteria to be compared directly, with respect to suite size. Due to the use of random selection, multiple suites for each target criterion should be generated to improve validity. In addition, fully random test selection can be used to provide a baseline to compare each adequacy criterion against.

There is a challenge in comparing a fully satisfied adequacy criterion against an unsatisfied one, however. For example, a test suite using coverage selection may achieve full coverage after 15 tests, but a suite using mutation selection of the same size may only kill half of the mutants. To handle such situations, Chen et al. suggest two strategies to continue growing a suite after its selection process reaches full adequacy. One of these strategies is to "stack" the test suite, by resetting the target adequacy criterion to zero while retaining the tests, and to continue the growth process with the remaining unselected tests, with the goal of increasing the same target criterion. This is effectively equivalent to appending an additional test suite that is constructed using the same selection process. For example, if a test suite achieved 100% coverage, the stacking growth process would effectively recover coverage goals that were already achieved. The other strategy is to change the target adequacy criterion once the original criterion is satisfied. For example, a suite that has achieved full coverage could continue to be grown by selecting tests that increase its mutation score. If such a suite continues to improve the fault detection probability, it would demonstrate that mutation testing can reveal additional test inadequacy for a program even after a test suite achieves full code coverage.

Chen et al. used their approach to compare the adequacy estimation effectiveness of mutation testing and code coverage, using real faults from the Defects4J [28] dataset of Java programs. They found that suites constructed by selecting tests based on coverage reveal more real faults than those that were guided by mutation score, until full coverage was obtained. However, they also identified that it was possible to continue growing suites beyond this size, and that the larger test suites would reveal more real faults. In particular, they found that continuing to grow suites that achieve full coverage

by changing the adequacy criterion to mutation score was the most effective approach; mutants reveal test suite inadequacy even after full coverage is achieved.

2.9.3 Equivalent Mutants

It is possible for a mutant to be functionally indistinguishable from the original, correct program, where no possible test suite would be able to differentiate between the two [120, 121]. Such mutants are called equivalent mutants, and present a unique challenge to mutation testing; a set of mutants that contains equivalent mutants would present an underestimate of adequacy, and executing these useless mutants would needlessly increase computation time. In addition, equivalent mutants impact mutation analysis, as a test developer would expend effort to determine if a mutant is equivalent or not. Similarly, tutors using mutation analysis to develop automated assessment test suites would also have to spend time to manually determine if undetected mutants are indeed equivalent. Fortunately, the severity of this issue can be limited, via the use of various techniques to identify and remove these equivalent mutants [120–130].

2.9.4 Mutation Tools

Artificial mutants are generated using programs called mutation tools. These tools generate each mutant by applying a mutation operator to a source program. Each of a tool's mutation operators can be applied several times to different locations of the program, generating a variety of mutants. Early mutation tools were focused on Fortran [131–133], with focus later shifting to Ada [134], C [110], and Java [135–137].

Modern mutation tools are available for a variety of popular programming languages and application domains [138–141]. Since my dataset only includes students' solutions to Java programming assignments, I will primarily discuss

Table 2.4: Mutation operators implemented by Major [142, 143].

Operator	Description	Example
Arithmetic Operator Replacement	Replace an arithmetic operator	`x + y` → `x * y`
Logical Operator Replacement	Replace a logical operator	`x ^ y` → `x | y`
Conditional Operator Replacement	Replace a conditional operator	`x && y` → `x || y`
Relational Operator Replacement	Replace a relational operator	`x < y` → `x >= y`
Shift Operator Replacement	Replace a bitwise shift operator	`x << y` → `x >> y`
Operator Replacement Unary	Replace a unary operator	`-x` → `++x`
Expression Value Replacement	Replace an expression with a default value	`x = y` → `x = 1`
Literal Value Replacement	Replace a literal value with a default	`"Word"` → `""`
Statement Deletion	Delete a statement (e.g. `return`, `break`, method call, etc.)	`put(k)` →

Java mutation tools. Major is one such Java mutation tool, introduced by Just [142]. Major implements nine mutation operators, as shown in Table 2.4 [142, 143]. These operators are used to generate mutants using a modified Java compiler. This approach offers two main advantages over directly manipulating source code.

- Uncompilable mutants can be avoided; the compiler has contextual information that a simple text manipulator would lack.
- Individual mutants do not need to be compiled; the compilation process is only executed on the original source file, and generates every possible mutant in this execution.

Just evaluated Major's generation overhead by generating mutants for 12 open source Java projects, and comparing the projects' compile times with the Major compiler, which generates and compiles the original program and mutants, against those for the traditional Java compiler [142]. They found that Major generated a total of 539,966 mutants in 222 seconds. The same projects took 149 seconds to compile with the conventional Java compiler; Major has an overhead of only ∼49% to generate this large quantity of mutants.

Pit is another modern Java mutation tool [146, 147]. Pit implements 29 mutation operators in total, though only 11 are enabled by default [144]. Table 2.5 shows Pit's default operators, and Table 2.6 shows its extra, non-default operators. Some of the operators are not enabled by default because they are more prone to generating equivalent mutants, while others are effectively deprecated; they were reimplemented by some the default operators.

Table 2.5: Default mutation operators implemented by Pit [144].

Operator	Description	Example
Conditionals Boundary	Replace a relational operator, but only by its boundary.	`x <= y` → `x < y`
Increments	Inverse increments and decrements.	`x++` → `x--`
Invert Negatives	Remove negation operators.	`-x` → `x`
Math	Replace an arithmetic operator with a predefined alternative.	`x % y` → `x * y`
Negate Conditionals	Inverse a conditional operator.	`x <= y` → `x > y`
Void Method Calls	Remove a call to a method without a return type.	`run()` →
Empty Returns	Replace return values with an empty default.	`return 5` → `return 0`
False Returns	Replace boolean returns with `false`.	`return x` → `return false`
True Returns	Replace boolean returns with `true`.	`return x` → `return true`
Null Returns	Replace return values with `null`.	`return x` → `return null`
Primitive Returns	Replace primitive returns with `0`.	`return x` → `return 0`

Unlike Major, Pit directly mutates existing bytecode. This alternative approach is often simpler and faster, but has one significant limitation over Major's compile time approach; compiled bytecode lacks the source code's syntactic sugar, it is replaced with semantically equivalent lower level instructions for the Java virtual machine. For example, an enhanced for loop over an array's elements would be replaced with a traditional in for loop over the array's indices in the compiled bytecode. This limitation has two implications for mutation:

- Mutants that would be impossible to add to the original source code can be generated. In the example of enhanced for loops, where attempting to read data outside of the array is impossible, a bytecode mutant may modify the bounds of the for loop, resulting in an `ArrayOutOfBoundsException`.

- Generated mutants cannot be directly converted back to easily readable source code; it is more difficult to interpret the change that a mutant has made, so performing mutation analysis is more challenging. By contrast, Major enables the generation of such mutated source files.

Since Pit does not output mutated source code, it instead provides the user with a report in the form of a HTML page, which displays the original source code with annotations of the mutants' operators applied to each line of code.

Table 2.6: Additional (non-default) mutation operators implemented by Pit [144].

Operator	Description
Return Values	Replace a return value with a default. Deprecated by individual return operators.
Remove Conditionals	Remove a conditional (e.g. from an `if` statement), so that a block will always be executed (e.g. `if (x) ...` → `if (true) ...`).
Experimental Switch	Swap the first case of a switch statement with the default case.
Inline Constant	Replace literal values with a default, or increment some numeric values (e.g. `boolean a = true;` → `boolean a = false;`).
Constructor Calls	Replace object constructor calls with `null`.
Non Void Method Calls	Replace a method call that has a return value with a default value of the same type (e.g. `int a = sum(b, c);` → `int a = 0;`).
Remove Increments	Remove increment operators.
Experimental Argument Propagation	Replace a method call with one of its parameters that has the same type (e.g. `int a = sum(b, c);` → `int a = b;`).
Experimental Big Integer	Replace methods of `BigInteger` objects.
Experimental Member Variable	Remove assignments to member variables. Initial values are replaced with defaults appropriate to their types.
Experimental Naked Receiver	Replace a method call with the object that the method is called on (e.g. `s = s.trim();` → `s = s;`) [145].
Negation	Negate any numeric variable (e.g. `sum(a, b);` → `sum(-a, b);`).
Arithmetic Operator Replacement	Replace an arithmetic operator. Effectively deprecated by the *Math* operator.
Arithmetic Operator Deletion	Replace an arithmetic operation with one of its two values (e.g. `a = b + c` → `a = b`).
Constant Replacement	Replace a numeric inline constant; similar to the *Inline Constant* operator.
Bitwise Operator	Replace a bitwise operator (e.g. `&`) with its alternative (e.g. `|`) or one of the values in the operation.
Relational Operator Replacement	Replace a relational operator with one of its alternatives (e.g. `>` → `==`).
Unary Operator Insertion	Add a unary operator to a variable reference (e.g. `a` → `a++`).

Both mutation tools also feature integrated mutant analysers, which execute the user's test suite on the generated mutants, allowing the tool to directly evaluate the adequacy of test suites. Major's analyser uses several optimisations to minimise otherwise lengthy mutant execution times [142]. The first is to pre-process mutants by performing weak mutation analysis; to determine which mutants cause state infection when executed by a test. A mutant achieves state infection if its state differs from that of the original program after the mutated statement is executed. Consequently, mutants that are not covered by any tests will not achieve state infection, nor will those that are equivalent. This weak mutation analysis only requires a single execution of the test suite, and therefore adds little overhead ($\sim$57.5%) over simply testing the original program alone. Only mutants that achieve state infection are used for further strong mutation analysis, in which tests are executed directly

on mutants. This strong mutation analysis has its own optimisations:

- Mutants are not re-tested after they are killed by a test.
- Major uses test suite prioritisation by runtime; tests that take less time to run are executed first, with the aim to kill mutants sooner.

These optimisations do significantly improve Major's execution time; in Just's experiment, running all 14 programs' test suites (one project included three programs) on the original code took 1080 seconds (∼77 seconds/program). In comparison, executing all 539,966 mutants took a total of 1828 minutes (∼0.2 seconds/mutant); avoiding the unnecessary execution of some mutants significantly reduces the average execution time.

The results of Major's strong mutation analysis are reported to the user, including the number of generated, covered, state-infecting, and killed mutants, alongside the reason for a mutant being killed, such as an assertion violation, exception, or timeout (perhaps due to a mutant introducing an endless loop). Similarly, Pit's HTML report includes the mutants that are killed, and the reason why they are killed, directly annotated in its source code view. These reports inform the user of specific test deficiencies; Major lists the unkilled mutants, and Pit includes unkilled mutants in its annotations. These reported unkilled mutants can be manually inspected by the user, to better understand why the mutants are not killed by any tests, and improve the test suite accordingly. Similarly, a tutor could use mutants to understand how they should enhance their automated assessment test suites.

Pit is also designed to be easily integrated with Java projects, with direct support for Java build systems such as Maven and Gradle; users can generate and analyse mutants with a single command, or automate the process entirely with continuous integration. Such a feature could be of benefit for software testing tasks in MOOCs; new or updated tasks would only be provided to students if their test suites pass and kill every generated mutant.

2.9.5 Higher Order Mutants

Higher order mutants (HOMs), first introduced by Jia & Harman, combine the changes from multiple first-order mutants (FOMs), i.e. single statement mutants, into one mutant [148]. It is also possible to generate HOMs that are subsuming; the test cases that kill a subsuming HOM also kill every FOM that it is generated from. Consequently, using HOMs also allows for the execution time of mutation testing and analysis to be reduced, since if a subsuming HOM is killed, each of its constituent FOMs are as well; only one execution of a test suite is required, instead of multiple. Furthermore, some subsuming HOMs may not be killed by a test set which kills every constituent FOM; such HOMs are strongly subsuming. These strongly subsuming HOMs are more subtle than their constituent FOMs, presenting a stronger measure of adequacy. Figures 2.4 & 2.5 demonstrate how a strongly subsuming HOM is more subtle than its constituent FOMs.

Jia & Harman found that a genetic algorithm was the best approach to generate subsuming HOMs, and that many subsuming HOMs could be generated from 10 subject programs [148]. They also found that a search based approach could generate strongly subsuming HOMs for each subject program. Omar et al. have also shown that local search based techniques can generate strongly subsuming HOMs [149]. Nguyen & Pham found that around 50% of generated HOMs are harder to kill than their constituent HOMs [150].

HOMs could provide a means to resolve the issue of students' solution programs breaking the Competent Programmer Hypothesis; generated HOMs may be more analogous to students' programs that contain multiple faults.

2.9.6 Existing Applications in Automated Assessment

Mutation testing and analysis has seen some application in Computer Science education, particularly in the education of software testing.

Several studies have investigated the use of mutation testing and analysis to

```
Triangle getType(int a, int b, int c) {
  int t = 0;
  if (a <= 0 || b <= 0 || c <= 0)
    return INVALID;

  if (a == b) { t = t + 1; }
  if (a == c) { t = t + 2; }
  if (b == c) { t = t + 3; }

  if (t == 0) {
    if ((a + b < c) || (a + c < b) || (b + c < a))
      return INVALID;
    return SCALENE;
  }
  if (t > 3)
    return EQUILATERAL;

  if ((t == 1) && (a + b > c))
    return ISOSCELES;
  if ((t == 2) && (a + c > b))
    return ISOSCELES;
  if ((t == 3) && (b + c > a))
    return ISOSCELES;

  return INVALID;
}
```

Figure 2.4: Correct implementation of a method to determine the type of a triangle, given the length of its sides. Adapted from Jia & Harman's work [148].

assess students' software tests, with varying results. Aaltonen et al. found that mutation analysis allows for students' tests to be evaluated, irrespective of the students' original programs, even if they implement behaviour that is not in an assignment's specification [151]. This would allow for testing in more open ended creative assignments to be assessed. They also note that mutants may be able to provide students with feedback on their tests' inadequacies. However, Shams et al. warn against using students' solutions to generate mutants, since a student's solution may already be incorrect, and a mutant may actually fix the program [152]. Furthermore, the study conducted by

```
- if ((t == 1) && (a + b > c))
+ if ((t > 1) && (a + b > c))
```

(a) A first-order mutant, *FOM-a*, represented in diff notation.

```
- if ((t == 1) && (a + b > c))
+ if ((t == 1) && (a + b <= c))
```

(b) Another first-order mutant, *FOM-b*, represented in diff notation.

```
- if ((t == 1) && (a + b > c))
+ if ((t > 1) && (a + b <= c))
```

(c) A strongly subsuming higher-order mutant, produced by combining *FOM-a* and *FOM-b*. Represented in diff notation.

```
assertEquals(INVALID,
    getType(1, 3, 1));
assertEquals(INVALID,
    getType(1, 1, 3));
```

(d) Test assertions that, when combined, reveal the first-order mutants, but not the strongly subsuming higher-order mutant.

```
assertEquals(ISOSCELES,
    getType(2, 2, 3));
```

(e) Test assertion that reveals the first-order mutants and the strongly subsuming higher-order mutant.

Figure 2.5: Example of how a strongly subsuming HOM can be created from two FOMs for the method in Figure 2.4. This also illustrates how tests can reveal FOMs, but not the strongly subsuming HOM that is created by combining them. Adapted from Jia & Harman's work [148].

Edwards et al. indicates that mutation scores are not significantly correlated with the fault detection capabilities of students' tests [153]. It is possible that the efficacy of using mutation testing and analysis to evaluate students' tests is influenced by other factors, perhaps such as the assignment itself. These studies also identify additional weaknesses in using mutants to assess students' tests. Aaltonen et al. note that coverage metrics may be easier to understand, and that more complex programs would have more mutants generated for them, with different programs having different distributions of mutants, creating a potential source of unfairness. Shams et al. note that students' tests may also fail to run against a model solution, and by extension, any mutants generated from it. They also note that mutation analysis may induce an additional manual cost in grading, due to equivalent mutants (defined in Section 2.9.3), which have the same behaviour as the model solution program. Such mutants would need to be removed prior to assessment, as they would falsely assess test suites as inadequate. Regardless, mutation analysis may still provide some benefit in assessing students' tests, as Aaltonen et al. show that mutation analysis can reveal tests which are intended to gain an unfair advantage in grading, such as tests which never fail on a given program.

2.9.7 Potential Applications in Automated Assessment

It may be possible to use mutants to evaluate the adequacy of grading test suites. Evaluating a grading suite's adequacy is important, as it provides some reassurance that the grading process is fair. An inadequate test suite may fail to identify students' faults, resulting in some students receiving grades that are too high. These students would also not receive feedback to help them to correct their mistakes in the future, resulting in less effective formative assessment. By contrast, students' solution programs with faults that are detected instead will receive lower grades; this results in unfairness and inconsistency in summative assessments. As a new programming task would not have any students' solutions, a grading test suite's adequacy cannot be

evaluated using real faults. Even where students' solutions are available, only real faults that have already been identified by either the test suite or costly manual analysis can be used to evaluate adequacy. Existing faults that have not been identified cannot be used to evaluate adequacy. Mutants therefore offer a viable alternative to real students' faults, since all non-equivalent mutants can be used as a proxy for known faults.

As mutants can be used by developers to guide improvements to their test suites [154], they could similarly be used by tutors to assist in improving their automated grading test suites. By gaining knowledge of which mutants are undetected, tutors can evaluate if a student may make a similar mistake, and if so, add new tests to detect them, providing more appropriate test-based feedback and more accurate grades.

For mutation testing and analysis to be effective, mutants must be capable of simulating students' mistakes, as they do real faults. This can be evaluated via empirical studies, similar to those discussed in Section 2.9.2.

2.10 Test Generation

Regression testing presents a unique opportunity; the availability of an existing (presumably correct) program allows for it to be used to automatically derive new test cases to identify divergences in behaviour in future revisions of the program [155]. The process of deriving such tests from an original program is test generation, and offers a means to save the time cost of writing regression tests. Furthermore, human testers may focus on possible faults that they expect could be introduced into a program; generative testing processes can instead explore the program or its specification as a whole, possibly closing the gaps in completeness of manually-defined tests [156]. Automated assessment can effectively be modelled as regression testing, with a tutor's reference solution acting as an original program, and students' solutions as a series of modifications to it. Therefore, test generation also offers a potential means of easily creating additional tests for automated assessment, and may improve

the detection of students' faults that a tutor does not expect.

One approach is random testing; randomly generating test inputs, and executing the source program (e.g. reference solution) with them to determine their according expected output [157]. Randoop [158, 159] builds upon this, by constructing a sequence of a program's methods to call, using the return values of methods observed by previous generations as parameters of later methods in new tests.

An alternative approach is to use search-based techniques [160]. Such techniques guide the generation of new tests by a fitness heuristic, such as individual test goals, or a test adequacy metric. New tests which improve such heuristics should also improve a test suite's ability to detect real faults. Some search-based testing techniques are inspired by biological phenomena. For example, artificial immune system (AIS) approaches are inspired by the immune system of the human body; tests are reproduced and modified to achieve coverage targets in a manner analogous to how immune cells are produced by the body to identify pathogens [161]. Another biologically inspired approach is to use genetic algorithms, as employed by EvoSuite, an evolutionary search-based test generation tool [162, 163]. EvoSuite combines and alters test cases between candidate test suites, guided by a fitness function to maximise the coverage of the next generation iteration's candidate test suites. EvoSuite also applies mutation analysis to prune the assertions of its generated test cases, by removing assertions that do not reveal any mutants. Also, undetected mutants reveal where better assertions are required, providing goals for the tool to generate new assertions. Such search based approaches are typically more effective than random testing, as shown by Almasi et al., who found that tests generated by EvoSuite detected 56.4% of real industrial software faults, while Randoop's tests only detected 38.0% [164].

Test generation still presents some challenges. First, generated tests may not perfectly reveal faults; in their experiment on real faults in an industrial software system, Almasi et al. found that tests generated by either tool could not detect more challenging faults, especially those that require the creation of a specific object, or particular boundary values, to identify [164]. This

means that, if applied to automated assessment, generated tests could miss students' mistakes, preventing students from receiving appropriate feedback and grades. Second, if used for automated assessment, generated tests may not necessarily reflect and isolate individual learning outcomes; they may cover several unrelated learning outcomes if they test several different methods. It may be possible to avoid this, by using mutation analysis to guide generated tests towards individual types of faults, or faults in particular locations, but this is out of scope for my research. Finally, generated tests can be hard to read and understand [164]. This makes generated tests inappropriate to provide directly to students, since students would likely be unable to understand what deficiencies of their code the tests truly reveal. However, they can still be used by tutors to augment their test suites, in order to reveal more students' faults. This could be particularly beneficial for grading to distinguish between the correctness of different students' solution programs.

2.11 Fault Localisation

Fault localisation aims to simplify debugging by outputting a set of suspicious program elements (e.g. lines) that are likely to contain a fault [165]. Spectrum-based fault localisation (SBFL) is a common approach to fault localisation that uses the recorded behaviour from a program's executions (e.g. test runs), called *program spectra*, and errors (e.g. failing tests) to construct a model. Program spectra consist of a series of a program's components, which are observed to have been hit (i.e. covered) or not for each individual execution of the program (i.e. test run), irrespective of an error being encountered during an execution. Consequently, program spectra can be constructed in a similar manner to code coverage, using program components that are identified similarly to coverage goals:

- *Block hit spectra* consider individual blocks of code (such as statements or lines). Each block is flagged if it has been run during an execution of the program [166].

- *Branch spectra* track the conditional branches executed during a set of runs of a given program. This can either be represented as a boolean flag indicating if the branch has been executed or not (*branch-hit spectrum*), or as a count of the how many times the branch has been executed (*branch-count spectrum*) [167].

- *Path spectra* represent the paths across a program's control flow graph that have been taken during its execution, such as a series of conditional branches within a method [168].

These spectra are stored in an $N \times M$ activity matrix, A, for N runs (i.e. tests) and M components (e.g. lines) [169]. Each element of this matrix, a_{ij}, denotes that a component, j, was executed during a run, i. Next, the results of each run of a target program (i.e. test) are stored in an error vector, e, such that an element denotes whether a run passes ($e_i = 0$), or fails ($e_i = 1$). The activity matrix and error vector effectively form a model that can be used to associate the results of runs with program components. This association is performed by a spectrum-based fault localisation algorithm, which computes the similarity between the error vector and the activity column of each component. One particularly prominent means of determining similarity is the Ochiai coefficient [170, 171]. These similarities are then used to rank components in order of suspicion; component activity columns with a high similarity to the error vector are more likely to contain a fault. Essentially, if a particular line is covered by every test that fails, it is more likely to contain a fault. Gouveia et al. show that if this data is visualised in an appropriate manner, developers may be able to find (and correct) software faults with much less effort [172]. Similarly, this could perhaps be used to provide students with feedback on where they have made an error.

2.11.1 Diagnosability

Perez et al. have proposed DDU (Density, Diversity, Uniqueness), a metric to estimate a test suite's diagnosability; its ability to accurately predict the

location of faults with fault localisation [173]. DDU combines three metrics, density, diversity, and uniqueness together:

- *Density,* ρ, which was previously used as a standalone diagnosability metric by Gonzalez-Sanchez et al. [174], evaluates the proportion of components that are hit across every run.

$$\rho = \frac{\sum_i^j A_{ij}}{N \times M},$$

where C = set of system components,
$M = |C|$,
T = set of test cases,
$N = |T|$,
A = an $M \times N$ activity matrix;
A_{ij} denotes whether component c_j was executed by test t_i

Where $\rho = 1$, every component is covered in every run, and where $\rho = 0$, none of the components are ever covered. Both of these absolute cases would be detrimental to fault localisation, since an error vector would be equally similar to every component's activity column; there is no information gain. Gonzalez-Sanchez et al. have shown that the optimal density is $\rho = 0.5$. In order to combine density with additional metrics, Perez et al. normalise ρ to ρ', where the maximum, $\rho' = 1$, indicates the ideal density, and the minimum, $\rho' = 0$, indicates that every observation in the activity matrix has the same value:

$$\rho' = 1 - |1 - 2\rho|$$

However, density can only accurately evaluate diagnosability if the observations in the activity matrix are distinct. Consider a series of tests, which each only hit the same components, and miss the same quantity of other components. In this case, the activity matrix may have an optimal density, $\rho' = 1$, but a similarity metric would not be able to fit an error vector to a single component column, especially

if the error vector does not have the same test failures as any of the columns. A metric that separates similar tests is also required.

- *Diversity* is evaluated by the Gini-Simpson index, $\mathcal{G}$ [173, 175]. In the context of diagnosability, diversity determines the likelihood that two randomly selected tests differ in their behaviour.

$$\mathcal{G} = 1 - \frac{\sum n \times (n-1)}{N \times (N-1)},$$

where n = the number of tests with the same activity

The ideal case ($\mathcal{G} = 1$) is for every test to have a unique activity signature ($n = 1$); each test executes a different set of components, though these sets can have some overlap. This improves information gain for fault localisation, since if tests behave differently, a similarity metric should be able to better distinguish between different components. In a case where $\mathcal{G} = 0$, every test behaves the same with respect to the components, so every component would be completely similar to each other.

However, combining ideal density and diversity does not guarantee that a test suite offers maximum information gain for fault localisation. In such a case it may still be possible for two or more components to pass and fail for the same tests, since the tests may have different behaviour for other components. Where components share the same activity, they are deemed identical by similarity metrics. If an error vector was found to be similar to these components, a fault localisation algorithm would be unable to determine which of the components is most likely to represent the fault. A diagnosability metric should also consider these ambiguous components.

- *Uniqueness*, $\mathcal{U}$, measures how many ambiguous groups of components

are present in an activity matrix.

$$\mathcal{U} = \frac{|G|}{M},$$

where G = a set of subsets of C, such that members of each subset have the same activity for all tests

In this definition, $|G|$ effectively represents the number of groups of components that are covered by the same tests. Where no ambiguous components exist ($\mathcal{U} = 1$), an activity matrix with ideal density and diversity should differentiate between every component, and a similarity metric should be able to match an error vector to its closest components, providing an effective fault localisation estimate.

Baudry et al. used uniqueness to measure diagnosability [176]. However, Perez et al. note that uniqueness alone may not be an effective diagnosability metric, since it does not guarantee that various components can be combined to localise multiple faults simultaneously.

Perez et al. combine these metrics to form DDU, by multiplying their results together:

$$DDU = \rho' \times \mathcal{G} \times \mathcal{U}$$

This approach combines the benefits of each submetric, while avoiding the issues that arise from using any of the metrics alone. Perez et al. found that optimising generated test suites for DDU resulted in improved localisation of real faults compared to optimising for branch coverage.

The submetrics of DDU may have some implications for test suites used in automated grading. Diverse test suites evaluate different components of the program; they can reveal students' faults in different locations. High uniqueness indicates that each component of a program is only covered by a particular set of tests, and a fault in such a location may have a subset of tests that distinctly cover it. High values of these metrics for a reference solution may benefit grading, since they would indicate that each test is

primarily focused on one aspect of the task's specification. This means that a student's fault in one aspect of the specification would be less likely to affect their grades for other aspects of the specification. Combined with the other metrics, density gauges how evenly a test suite covers a program. It is possible that unevenly covered test suites could result in unfairness. For example, if a student makes a mistake that is heavily covered, they will likely receive a much lower grade than a student who makes a mistake that is less covered, even if the mistakes themselves are similar.

2.11.2 Mutation in Fault Localisation

Mutants can be used to aid fault localisation, by taking the importance of program components into account; modifying an important component should cause more tests to fail [165]. Mutation based fault localisation (MBFL) techniques assign suspicion to mutants, since tests that kill the mutants should be able to reveal real faults. Two MBFL techniques, MUSE [177] and Metallaxis-FL [178] each generate a set of mutants for each component of a program. In summary, MFBL involves computing a suspiciousness score for each mutant, based upon how many tests that it fails which pass for the original program, and how many tests that it passes which fail for the original program [177]. This is based on the premise that:

- a test which fails due to a faulty component is more likely to pass on a mutated version of the component, and
- a test which passes on a correct component is more likely to fail on a mutated version of the component.

A component's overall suspiciousness score is then computed by combining the scores of its constituent mutants, and localisation is performed on these components using an error vector, in the same manner as in SBFL.

```
int abs(int x) {
  if(x > 0) { return x; }
  return -x;
}
```

(a) Example method; returns the absolute values of an integer.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
}
@Test void testNegative() {
  assertEquals(2, abs(-2));
}
```

(b) Test suite β; $\rho' = 1.0$, $\mathcal{G} = 1.0$, $\mathcal{U} = 1.0$.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
}
@Test void testZero() {
  assertEquals(0, abs(0));
}
@Test void testNegative() {
  assertEquals(2, abs(-2));
}
```

(c) Test suite ϵ; $\rho' = 1.0$, $\mathcal{G} = 0.67$, $\mathcal{U} = 1.0$.

```
@Test void testAllOne() {
  assertEquals(1, abs(1));
  assertEquals(1, abs(-1));
}
@Test void testAllTwo() {
  assertEquals(2, abs(2));
  assertEquals(2, abs(-2));
}
```

(d) Test suite ζ; $\rho' = 0.0$, $\mathcal{G} = 0.0$, $\mathcal{U} = 0.5$.

Figure 2.6: Fault diagnosability metrics of various test suites for the method originally defined in Figure 2.2. These diagnosability metrics are density (ρ'), diversity ($\mathcal{G}$), and uniqueness ($\mathcal{U}$).

2.11.3 Potential Applications in Automated Assessment

Fault localisation may offer some benefits for the automated assessment of students' programs, particularly in generating automated feedback. Since fault localisation has been shown to assist developers in finding faults [172], the output of a fault localisation tool could be used to help students understand where they have made a fault. This is important, since introductory programming students sometimes find conventional compiler messages hard to understand [179], and may not fully grasp software testing strategies.

Furthermore, fault diagnosability metrics could be used by tutors to understand how well their tests isolate individual faults. This may be useful to

create test suites where each test only covers an individual learning outcome or a set of similar faults. This could improve the fairness of automated grading, since it would otherwise be possible for a student to make a minor mistake for one learning outcome, which also fails tests that are intended to evaluate other learning outcomes. This would also ensure that students can easily interpret what test failures mean with respect to their mistakes. For example, consider the test suites in Figure 2.6, assuming each of the diagnosability metrics use line coverage as components, and that all of a line must be executed to be covered, for simplicity. First, suite β (Figure 2.6b) yields a maximum value (1.0) for each diagnosability metric, since it each of its tests evaluate a different line; it is balanced. The tests in suite ϵ (Figure 2.6c) are able to fully distinguish between each line, so its density and uniqueness metrics yield the ideal values. However, its diversity is only 0.67, since the second line is covered by two of the three tests; the tests evaluate the method's lines unevenly. Finally, suite ζ (Figure 2.6d) yields minimum possible values for each metric for this method, since both tests cover the method in exactly the same way; they cannot separate between the lines. If a student was to make a mistake in either line, a tutor would not know which line the mistake would effect; the tutor would not gain insight into the nature of the student's mistake. By contrast, with a suite like β, they would be able to understand exactly what mistake a student made from the failing test alone; perhaps revealing which learning outcome the student has not fully grasped. Naturally, this is a simple example; a real assignment would be more complex, with more possible groups of statements to distinguish between.

2.12 Conclusion

In this literature review, I have affirmed the importance of tutors having accurate knowledge of the mistakes that students can make in introductory programming assignments, in order for them to develop effective automated assessments that use unit tests and static analysis tools. However, there is also evidence that tutors do not always hold this knowledge in a manner that

fully reflects reality [23]. I have identified that any techniques which assist tutors in identifying and understanding potential students' mistakes would be of a practical benefit, and would result in automated assessments that are fairer, more consistent, and able to provide richer feedback to students.

In particular, mutation testing and analysis offer a promising means to directly simulate students' faults, which tutors would be able to use directly when developing their assessment test suites. Therefore, mutation testing and analysis will be the primary focus of my research.

I have also identified that other test adequacy metrics could be beneficial. For example, tutors could focus on improving code coverage with their tests first, then focus on identifying mutants once they achieve sufficient coverage [27]. Accordingly, I will also investigate code coverage in my research, such as by using it as a baseline adequacy metric to compare mutation testing against, with respect to a suite's ability to detect students' faults.

Finally, it is possible that the fault diagnosability of a test suite may impact the grades that it generates for students' solutions. I will also investigate how fault diagnosability metrics impact generated grades, alongside other test suite metrics, such as mutation score and code coverage.

Chapter 3

Datasets

In order to investigate my research questions, I gathered two datasets of anonymised students' programs across a series of programming assignments. I sourced these solution programs from The University of Sheffield's first year introductory Java programming module, with the assistance of Dr. Siobhán North and Dr. Mari-Cruz Villa-Uriol. I am not able to provide these datasets as a contribution of my work, since these assessments may be redesigned and reused in the future; publicly accessible versions of these solutions may be plagiarised by future students.

3.1 Semester One Dataset

I first collected a dataset of students' solutions for assignments in the module's first semester, which is summarised in Table 3.1. I will refer to this dataset as SemOne. Since the students are typically inexperienced in programming at this point in the module, the solutions' mistakes are indicative of very novice programmers.

These tasks are focused on guiding students' learning for basic programming concepts, and therefore have a low complexity. `Assignment1` requires students

Table 3.1: Subject classes of the SemOne dataset.

Class	Students' Solutions	Compilable Solutions
`Assignment1`	63	60
`Assignment2`	63	62

to read numeric input from the command line interface and a text file. This numeric input represents denominations of a fictional currency, which the students' solutions must perform calculations on, before writing the output to the user interface. `Assignment2` tasks students with writing a program that decodes input from a file, and uses this input to render a 2D bitmap in a graphical window.

The assignments in this dataset are fairly simple, and their specifications typically require students to write a program that is executed via a single entry point, Java's `main` method. Students are able to use their own methods and variables, but their names and access are not specified. This presents a challenge for automated testing; the solutions only have one testable method, and only the program's standard output can be used to form assertions. Consequently, only blackbox tests can be written for this single unit; it is incredibly difficult to isolate sub-procedures of students' programs and check their correctness. This makes it almost impossible to identify exactly what caused a student's program to fail a test from the results of a test alone. This introduces a severe limitation on using these solutions to evaluate testing techniques. Therefore, I will not use this dataset to evaluate the testing techniques that I explore in my research; I will only use this dataset to manually analyse the mistakes that are present in its students' solutions.

3.2 End of Year Dataset

I also collected solutions submitted by separate cohorts of students for three end of year programming assignments. This dataset is referred to as EndOfYear throughout my thesis. The assignments in this dataset are more

Table 3.2: Subject classes of the EndOfYear dataset.

Assignment	Subject Class	Students' Solutions			Tests		Reference Solution LoC
		Total	Compilable	≥ 1 Fails Test	Manual	EvoSuite	
Chess	`Board`	59	45	43	18	14	26
	`Queen`	59	40	34	9	2	41
Wine	`Cellar`	38	36	35	16	15	272
Fitness	`DataLoader`	40	38	38	7	1	71
	`Questions`	40	38	37	20	30	263

complex than those in SemOne. These assignments included specifications that required students to write methods and classes with particular names, with a given package structure, in order to ensure that individual components of the students' solutions could be tested in isolation, to allow for effective automated assessment. Students were typically provided with some of the program already implemented, to give them a common starting point. This sometimes included interface or abstract classes, which students were to implement the methods of in their own classes. These qualities offer the ability to easily test individual aspects of students' programs, making this dataset a much better candidate to evaluate the techniques that I explore in my research.

I selected individual subject classes from these assignments in order to evaluate solutions with a variety of attributes, such as different numbers of methods and lines, and the use of different programming concepts. Table 3.2 details these selected classes, with the number of solution programs and tests for each. Alongside the students' solutions for these each of these assignments, the dataset also includes a correct reference solution, the applications of which I discuss later in this chapter.

These assignments aim to evaluate students' abilities to write much more complex programs than those in SemOne. *Chess* tasks students with implementing an interactive game of chess. In my study, I have sampled two classes from this assignment. `Board` is effectively a data structure that represents the grid on which pieces are positioned, which is used to set and query the

locations of the chess pieces. `Queen` is one of the more complex piece objects, and requires students to iterate over possible moves, while correctly terminating the search for more moves in a given direction if the edge of the board or another piece is reached. *Wine* tasks students with writing a program to analyse a database of wine samples. The `Cellar` class handles the logic of this analysis, specifically in loading the data, parsing a text file of queries to apply to the data, and to identify the wine samples with the greatest and smallest measurements of particular properties. *Fitness* is focused on analysing data from a range of fitness trackers. Students' implementations of `DataLoader` should accurately parse the provided data files, and `Questions` should include a series of methods which each perform a particular query on this parsed data.

Assignments in EndOfYear each include multiple class files, including some that I am not using in my research. If I included these classes when I execute tests on students' solution classes, it is likely that issues in the other classes would influence the behaviour of the subject class under test. For example, poor implementation in `Board` could introduce test failures for `Queen`. Similarly, each of the subject classes should follow a specification in isolation; a student's solution may produce the correct output, but the individual classes may violate the specification. Following the previous example, a student may implement updating piece positions in `Queen`, even if the specification requires this to be handled by `Board`; this would represent a specification violation. To address these potential issues, I isolated the students' subject classes under test, and used the corresponding reference solution's implementation of their dependency classes. Where possible, I also included any new classes that a student created beyond the specification, since the class under test may also depend on such new classes.

Before performing any specific analysis of the students' solutions, one clear pattern is that the *Chess* assignment includes more students' solutions which do not compile. This is due to a shift in assessment approach; the *Chess* assignment was not designed with a focus on automated assessment, so its specification imposed comparatively few restrictions on how students could

implement the classes that I use in this study. In turn, more of these students' solutions violate particular specifications for the classes. For example, they may name methods incorrectly, use external libraries, or modify the provided auxiliary classes that should only be imported and referenced. Consequently, some of these solutions fail to compile, especially when they are isolated and recombined with the original, unmodified auxiliary classes.

3.3 Tests

For both the *Wine* and *Fitness* assignments, I used the unit tests which were originally used to grade the assignments, which I helped to design. However, I did not have access to the original grading tests for the *Chess* assignment. Furthermore, the tests for the other assignments did not achieve full coverage. In order to resolve these issues, I wrote additional tests for each subject class, with the aim of achieving full coverage.

I also generated additional unit tests for each subject class using EvoSuite, an automated test generation tool [162]. This increased the variety of tests that can be sampled to produce unique test suites for my empirical studies. The number of generated tests for each subject class is shown in Table 3.2, alongside the number of manually defined tests, which includes both the original grading tests and my additional tests.

In order to execute these tests on the students' solutions, I used *Gradeer*, an automated grading tool which I developed alongside this research, which I outline in Chapter 4. In summary, it provides generated grades and individual test results for each solution as a CSV file, which I perform my data analyses on. In addition, some of my experiments require the use of coverage metrics. For this, I execute the test suites on only the reference solution for each subject class, in order to simulate a situation where a tutor is using such a reference solution to develop their grading test suite. I use *JaCoCo* to gather line-based coverage data for the execution of each individual test [180].

Table 3.3: Mutants generated by Major.

Subject Class	Mutants		
	Total	Non-Equivalent	Unkilled Non-Equivalent
`Board`	57	53	0
`Queen`	94	92	2
`Cellar`	167	137	29
`DataLoader`	44	41	2
`Questions`	199	188	16

Table 3.4: Mutants generated by Pit.

Subject Class	Mutants		
	Total	Non-Equivalent	Unkilled Non-Equivalent
`Board`	204	176	0
`Queen`	386	350	28
`Cellar`	657	550	123
`DataLoader`	195	165	13
`Questions`	804	718	41

In order to ensure that the tests are valid, I executed each test on the appropriate reference solution. If any tests failed on this reference solution, they would be making assertions that violate the task's specification. This did not occur for any of the tests.

3.4 Mutants

In order to evaluate mutants in my empirical studies, I require generated mutants for each subject class in the EndOfYear dataset. Therefore, I use two existing Java mutation tools—*Major* (1.3.4) [143] and *Pit* (1.5.2) [147]—to generate mutants from the reference solution of each subject class. Table 3.3 and Table 3.4 outline the mutants that I generated using each tool.

3.4.1 Major

Major implements a set of simple mutation operators, such as operator replacement, value replacement, and statement deletion. Major also offers an option to output the source code of each mutant as well as the compiled mutants, which I enable, since it simplifies the process of inspecting mutants,

which is required when I check for potentially equivalent mutants. Major uses a modified Java compiler to generate mutants, which I execute using the following command:

```
path/to/major/bin/javac
-d majorOutput
-s majorOutput
-XMutator:ALL
modelSolution/package/SubjectClass.java
-cp sourceDependencies
-J-Dmajor.export.mutants=true
-J-Dmajor.export.directory=majorOutput
```

One notable caveat of Major is that it only supports Java 1.7 programs; programs that use newer features of the language are not supported. This does occur for some of my reference solution programs. For example, the reference implementation of `Cellar` uses lambda expressions to filter the contents of a list. In these cases, I replaced the unsupported code with semantically equivalent, version compatible code.

3.4.2 Pit

Pit offers a wide range of operators [144]; not only simple operators, like those implemented by Major, but also more complex operators. An example of one of these complex operators is *argument propagation*, which replaces a method call with one of its arguments that have the matching type. I include such operators in my empirical studies, in order to determine if they offer an advantage in improving grading test suites over only using simple mutation opertors. Some of these mutation operators are not enabled by default, however; the use of all operators must be specified when generating the mutants:

```
java -cp pitest-command-line-1.5.2.jar:pitest-1.5.2.jar:
  pitest-entry-1.5.2.jar:../jars/junit-4.12.jar:
  modelSolution:sourceDependencies:runtimeDependencies:testSuites
org.pitest.mutationtest.commandline.MutationCoverageReport
--reportDir pitOut
--targetClasses package/SubjectClass
--sourceDirs modelSolution
--features +EXPORT
--mutators ALL
```

3.4.3 Equivalent Mutants

The equivalent mutant problem is a well documented issue in mutation testing literature [26], and my research is by no means immune to the problem that it poses. Equivalent mutants are mutants that are semantically equivalent to the original program; it is impossible to write a test that kills the equivalent mutant while also passing on the original program. For my subject classes, it is also possible for mutants to be generated which are not strictly equivalent, but do not violate the specification of a programming assignment; they would not represent a fault. I also treat such specification-adhering mutants as equivalent for the purpose of my research. I found that a subset of mutants

generated by each tool for every subject class do not fail any test. In some cases, these unkilled mutants reveal potential inadequacies of the test suite, but others are equivalent.

If I do not isolate these equivalent mutants, they can impact the results of my experiments; test suites' mutation scores will be reduced by the presence of such undetectable, non-faulty, equivalent mutants. This effect could also be more prevalent for some subject classes or mutation tools, for which a greater proportion of equivalent mutants are generated, posing an additional potential threat to validity. I manually analysed each mutant that was not killed by any tests, to determine if it is indeed equivalent. To simplify this process, I used the `diff` utility, which provides the difference between two input files; the change the mutant has made. This is a fairly easy, if time-consuming, process for Major's mutants; I am able to trace their source code in order to determine whether their changes cause divergences from the reference solution's behaviour. However, Pit does not have the functionality to store mutants' source code; it only outputs the compiled Java bytecode of each mutant. Consequently, I used *jd-cli* [181], a command-line Java bytecode decompiler, to allow me to examine the changes that Pit's mutants made to the original reference program. If a mutant's change revealed by `diff` did not impact the functionality of the program in a meaningful way, I marked it as equivalent. I stored each equivalent mutant in a separate directory to the other, non-equivalent mutants. I then executed the tests on the non-equivalent mutants to yield the test results for only the non-equivalent mutants, which I use throughout my empirical studies.

The original reference solution of `Cellar` includes an overly complex implementation of its parser method. This introduces problems when I generate mutants; some mutation tools generate thousands of mutants for this original implementation, far more than those of the other subject classes. This would similarly significantly increase the time cost of manually analysing the potentially equivalent mutants for this class. Accordingly, I reimplemented this reference solution, with a simplified but functionally identical parser method to reduce the number of potential equivalent mutants without impacting the

validity of my results.

3.5 Ethics Considerations

Since I am using students' solutions to programming assignments, my research is subject to several conditions mandated by The University of Sheffield's ethics approval process:

- Only collect solutions submitted by students who provide informed consent to participate in my studies;
- Fully anonymise the collected data;
- Ensure that the collected data is not published;
- Store the collected data in a safe manner using The University of Sheffield's provided infrastructure;
- Delete all of the collected data after my research is fully completed.

Following the approval of my ethics applications [1], I sent consent forms and information sheets which detail my research and the scope of data collection to students enrolled in the Introduction to Java Programming module. I repeated this for three separate cohorts of students. After receiving students' completed consent forms, I downloaded the consenting students' submitted solutions for the relevant programming assignments from the module's submission system. I also executed a script to anonymise the solutions immediately after I downloaded them. This script renames files, and remove code comments that appear before a Java class declaration, since students were instructed to add comments that include their names at the top of their source files.

[1] IDs of relevant ethics applications: 17775, 23255, 25096 & 30749

3.6 Limitations

There are some limitations for these two datasets. One limitation common to both datasets is that, due to The University of Sheffield's ethics requirements, I was only able to use solutions submitted by students who completed consent forms. This may introduce self-selection bias, since students who completed the consent form may be more engaged with the course, and could make different mistakes to students who are less engaged with the course.

Since I collected both of these datasets from one module in a single institution, there are some potential limitations to their generalisability. I recommend that other researchers perform replication studies using data collected from students enrolled in other courses, institutions, and different types of educational organisations, such as MOOCs. EndOfYear does include solutions from three different cohorts of students; it offers some generalisability with respect to the students themselves.

Another possible limitation is that some students' solutions could contain some faults that are not detected by any tests. This should not severely impact my experiments, since I perform my data analyses on test suites which I sample from the wider set of tests for each subject class; these sampled test suites will typically detect fewer faults than the full test sets; I can still investigate the relationships between adequacy metrics (e.g. mutation score) and the detection of students' faults.

Chapter 4

Gradeer: An Open-Source Modular Hybrid Grader

This chapter is based upon my published paper:

> Benjamin Clegg, Maria-Cruz Villa-Uriol, Phil McMinn, and Gordon Fraser – *"Gradeer: An Open-Source Modular Hybrid Grader"* – International Conference on Software Engineering: Software Engineering Education and Training Joint Track with CSEET (ICSE-JSEET), 2021 [4]

This paper details my automated grading tool, *Gradeer*, which I first developed as an environment to generate grades for students' solutions using unit tests, upon which I performed my empirical studies. I later adapted the tool to assess students' solutions for the University of Sheffield's introductory Java programming module, following the guidance and feature requests of the module's leader, Dr. Mari-Cruz Villa-Uriol. These additional features include test weighting and manual grading components, which I did not use in my research directly, but which Dr. Villa-Uriol and the module's teaching assistants used extensively to assess students' solutions to various programming assignments.

4.1 Introduction

There are several different approaches to automated assessment, including the use of unit tests [18–21], static analysis tools [90, 102], and program correction [17]. These each have their own advantages; unit tests provide a simple means to evaluate the correctness of a student's program, while static analysis tools enable the provision of feedback on students' style mistakes. A tutor could combine these techniques in a bespoke automated grading script to effectively assess different learning outcomes, but this would be time consuming; a completely new script would be required for each programming task. Modular assessment systems [19, 81, 182, 183] simplify this process; different assessment approaches are implemented within the system, and only require configuration to combine them for new tasks. Similarly, such systems provide a means for new assessment approaches to be implemented, such as by plugins [81, 183], allowing for new and unique assessment functionality to be used. However, most of these grading systems are web-based; they excel at automated assessment, but prevent an educator from interacting directly with a student's program while it is executed. This poses a limitation, since some aspects of assessment simply cannot be performed automatically, such as the user experience of interacting with the GUI of a student's program.

To address this, I implemented *Gradeer*, a hybrid modular grading system that can run on a tutor's personal computer, allowing for the strengths of both automated and manual assessment to be employed. This chapter details the design of Gradeer, and its application to assess an end of year introductory Java programming assignment, where the tool's hybrid approach allowed for the use of a large number of consistent automated assessment criteria, and aided in the provision of detailed manual feedback to students. Gradeer is available on GitHub under the GPLv3 license, which allows users to write their own extensions and integrations for the tool [184].

4.2 The Gradeer Grading Tool

Gradeer is an assessment tool which provides tutors with the benefits of both automated and manual assessment in a single package. The tool achieves this using a modular design, allowing a user to choose how to assess a programming task using simple configuration files, or even define their own assessment modules for specific purposes. To allow for manual assessment, Gradeer is designed to be used by tutors on personal computers, where the user can interact with the program via a CLI. Gradeer is implemented in Java, and currently facilitates the assessment of Java programs. This section describes my design of Gradeer, alongside some of its benefits.

4.2.1 Checks

I designed Gradeer with a focus on modular grading components, called *checks*, each of which represents a single grading criterion. Different types of checks are currently implemented, defining how a criterion's *base score* (a decimal value between zero and one) can be determined for a given process and student's solution. Various checks of different types can be used together in a single run of Gradeer, constructing a markscheme to assess several learning outcomes. For example, users can configure Gradeer to use multiple checks to run various test suites, perform static analysis, and manually assess specific aspects of a solution. Users configure their checks in JSON files. Users can also implement new checks to add the functionality of unique and domain-specific grading tools.

One currently implemented type of check is the `TestSuiteCheck`, which executes a given JUnit test class on a student's solution via Apache Ant [185], then calculates a base score as the proportion of tests that pass. Tutors can assess individual learning outcomes by grouping tests that evaluate the same outcome into one class.

I also implemented check types for two static analysis tools, Checkstyle and

PMD [90, 102], in order to automatically assess the code quality of students' solutions. Such checks search the output of their respective tool for a user defined rule violation. The number of violations in each source file of a solution is recorded and used to compute a base score. Users can also define a minimum and maximum number of violations, which yield base scores of one and zero, respectively.

To support manual assessment, I implemented a `ManualCheck` type, which displays a user-defined prompt and score limit to the user when executed. This check then parses numeric input from the user and normalises it to a base score in the range of zero and one. For example, the following response would produce a base score of 0.6:

```
How informative are the variable names?
(0 = very poor, 10 = excellent)
# 6
```

Each check has an associated weight; a score multiplication factor to allow a test to have a greater or smaller impact on each solution's overall grade, as discussed in Section 4.2.2. This weight can be defined by the user. In addition, each check has associated feedback to provide to a student for their solution. For most checks, this feedback is determined by mapping a base score to one of several feedback values that have been pre-defined by the user. For example, the above manual check may provide students with feedback for the base scores, bs:

- $0.9 \leq bs \leq 1.0$: "Your variable names are informative."
- $0.5 \leq bs < 0.9$: "Some of your variable names could be more informative."
- $0.0 \leq bs < 0.5$: "Most of your variable names could be more informative."

Manual checks can also read text input from the user, allowing for additional

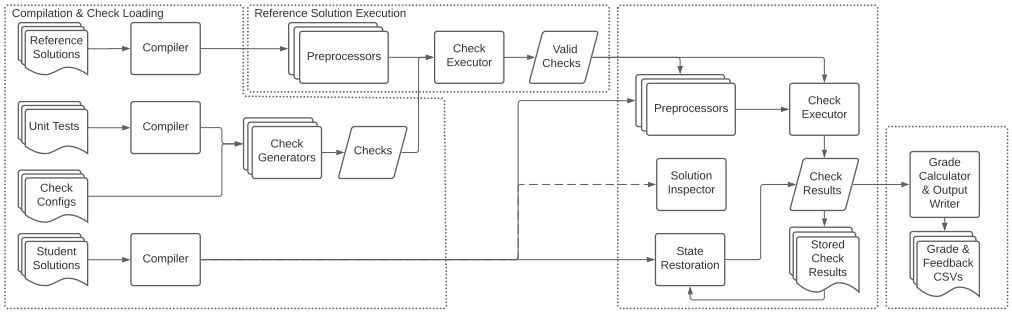


Figure 4.1: Overview of Gradeer's flow of execution. The dotted areas indicate different phases of the execution. Waved boxes are files, parallelograms are internal data, and regular boxes are processes.

feedback to be provided on an individual basis. For example, a tutor may enter "`a` is not an informative variable name, `leftMotor` would be better."

4.2.2 Execution

Figure 4.1 shows an overview of Gradeer's execution process.

Compilation & Check Loading

First, Gradeer compiles every student's solution and every reference solution (Section 4.2.2). At this stage, any solutions which do not compile are flagged. These solutions are reported to the tutor for review, and are excluded from further execution. Next, Gradeer loads every check defined in the JSON files. The tool also compiles the test classes that are provided by the user. If enabled, Gradeer automatically generates a test suite check for each test class which does not have a matching check already defined by the user.

Reference Solution Execution

The user can supply a set of one or more reference solutions, entirely correct solutions to the programming task being assessed. Users can choose to use multiple reference solutions to define different correct implementations of the programming task. In order to identify and remove invalid checks, Gradeer executes every check on each provided reference solution. Checks which attain a base score of less than one on any of the reference solutions are considered to be invalid, and are removed; they falsely claim that a reference solution is partly or completely incorrect. This prevents invalid checks from being used in the assessment of students' solutions, preventing them from unfairly losing or gaining grades, or being given inaccurate feedback. For example, uncompilable test classes will not pass on any solutions, so their checks are removed. The names of such invalid checks are stored in a file for the tutor to review and correct.

Solution Grading (for each Student's Solution)

Pre-checks In order for some checks to function properly, a series of pre-checks are executed on each solution. For example, checks for Checkstyle and PMD require pre-checks which execute their corresponding static analysis tool on the solution under test and store its output in memory.

Solution Inspection To support effective manual grading, Gradeer includes a *solution inspector* which can perform two processes, as configured by the user. The first executes a student's solution in a separate thread before running any manual checks. This allows the user to be able to interact with the solution, and to observe its user interface, which may be relevant to the rubric of manual checks. The solution execution thread is closed following the completion of every manual check on a given solution. The second opens each of the solution's source files in an external user defined text editor, such as Atom. This allows for the user to perform manual code inspection,

for example to determine the quality of variable names or comments. The solution inspector removes the need for the user to manually run a student's solution to interact with it, or open its source files to inspect it, saving time.

Checks The final step of a solution's grading process is to run every check on it. In order to improve execution time, Gradeer runs automated checks in parallel by default. Manual checks are only executed in the main thread, however – they require user input, and therefore could otherwise result in the occurrence of race conditions. In order to prevent some JUnit checks from taking too long to execute, Gradeer has a user configurable global test timeout, where any tests that take longer than this time are treated as failing. This is particularly important, since some students' solutions may contain bugs that prevent them from halting, such as incorrect loop conditions.

Output

After executing every check on every solution, Gradeer stores the appropriate grades and feedback for each solution in various CSV files. The final grade of each solution is stored in one CSV file. This grade is calculated by:

$$\begin{aligned} \textit{Grade}\ (\text{s}) &= \frac{\sum_{c \in C} w(c) \cdot \textit{Base Score}(c, s)}{\sum_{c \in C} w(c)}, \\ \text{where } s &= \text{Student's solution}, \\ C &= \text{Set of enabled checks}, \\ w(c) &= \text{Weight of check } c \end{aligned}$$

Similarly, the combined feedback of each solution across all checks is also stored in a CSV file. Gradeer also stores a CSV file with the individual base scores and feedback of every check for each solution. This file also includes the weight of each check. This allows for final changes to be made in spreadsheet software if absolutely necessary. For example, the user can post-process the students' grades by adjusting the checks' weights, and recalculating the

final grades in the same manner as Gradeer. Users can also gather valuable information on students' performance for the grading criteria, facilitating the provision of group feedback to the entire student cohort.

4.2.3 State Restoration

Following the completion of checks on a solution, Gradeer stores the results and feedback for every check in a JSON file. When Gradeer is executed with such files present, it uses them to restore these check results for every applicable solution, and skips the corresponding checks when processing these solutions again. This has numerous advantages:

- A tutor can effectively pause the grading process and come back to it at a later time. This is particularly advantageous when using manual checks, as programming tasks with many students' solutions can take hours to manually assess. State restoration allows this arduous process to be split into more manageable marking sessions.

- Assessment tasks can be allocated to multiple users, such as TAs. Tutors can adjust users' Gradeer configurations to use different solutions or checks. By allocating different manual checks to different users, grading can be completed more quickly without reducing consistency. By merging the users' state restoration files and re-running Gradeer, the final grades and feedback can be generated.

- If Gradeer halts unexpectedly, perhaps due to a wider system error, the user's grading progress is not lost.

- Tutors can either directly modify the result files to adjust the results of individual checks, or delete them outright to re-assess a solution. Running Gradeer again will update the final output files (as described in Section 4.2.2). Tutors can also choose to add new checks after initial executions of the tool to capture additional assessment requirements.

4.3 Case Study

In this section, I discuss the deployment of Gradeer to assess an end of year introductory Java programming assignment. The module's leader, Dr. Villa-Uriol, and the module's teaching assistants used Gradeer to assess 171 students' solutions for this programming task, which I helped to design.

4.3.1 The Assignment

The assignment was a redesigned version of the *Wine* assignment that is included in my dataset, but was undertaken by a different group of students. The task required students to parse a series of structured input files into a provided data structure, then implement a set of methods that query this data. The assignment also required students to plot graphs for this data in a GUI using Java's Swing library. A primary goal of the assignment was to provide students with experience in working on a multi-faceted project with co-dependent systems, which are more akin to real software than the simpler introductory programs used earlier in the course. As an end of year assessment, the assignment had a fairly wide span of learning outcomes. Such learning outcomes included the use of polymorphism, dynamic binding, bespoke data structures, the choice and use of various Java Collections, text manipulation, GUI programming, algorithm design, and the use of good quality code and programming style.

We first determined the overall assignment specification, then focused on creating a reference solution that captured this specification. We then created a set of grading unit tests, ensuring that they were valid and that the reference solution passed on each of them. Following this, we duplicated the reference solution in order to create a skeleton project, from which we removed the classes and methods that students were to implement.

4.3.2 Release

We distributed the skeleton project to students. We also provided the students with a set of input data files that were to be read by their implemented parsers. These data files were a subset of those that we later used when grading the assignment. Around a week after we released the assignment, we also provided students with a set of public tests. We designed these tests to ensure that students' code included the basic functionality of the assignment. This provided students with a degree of feedback as they worked on the assignment, and dissuaded students from submitting solutions which are not compatible with our grading environment, such as including incorrect class names.

4.3.3 Check Configuration

We configured Gradeer to use 45 checks:

- 26 test suite checks (each check executed one unit test),
- six PMD checks,
- six Checkstyle checks, and
- seven manual checks (for GUI functionality and subjective aspects of code review, such as variable names).

By using these checks together, we were able to use Gradeer to assess all of our learning outcomes. The manual checks were important in this regard, since the design of the GUI and some aspects of code quality cannot be fully graded automatically.

Table 4.1: Average time to perform each assessment task on each applicable solution.

Assessment Task	Average Time Per Solution
Compilable Solutions	
Compilation	∼1.6 seconds
38 Automated Checks	∼28.2 seconds
7 Manual Checks	∼2 minutes
Problematic Solutions	
Problem Identification	∼11.3 minutes
Solution Repair	∼11.4 minutes
Individual Feedback	∼10 minutes

4.3.4 Assessment

While Gradeer supports the use of all types of checks in a single execution, we split the checks across two separate execution configurations; one for automated checks and one for manual checks. This was necessary since we anticipated that some solutions would be problematic, containing issues that would prevent compilation or execution. Running manual checks on some of these solutions would have been a waste of effort if the solutions could not be executed properly. By splitting the checks we were able to first compile the students' solutions and run the automated checks to identify any problematic solutions, and to assess the working solutions. We identified 48 problematic solutions. We repaired these solutions where possible so that they could still be graded with Gradeer, but added a penalty for doing so when post-processing the grades. We repeated the automated grading for these repaired solutions. However, 11 of the solutions could not be repaired due to severe issues. We wrote individual feedback for each of these solutions to explain the nature of these problems. Finally, we re-executed Gradeer with only the manual checks on every working and repaired solution. Table 4.1 shows the average amount of time that various aspects of running the assessment with Gradeer took for each applicable solution, alongside the time taken to manage problematic solutions.

Once we completed grading the assignments, we performed some post-processing on the results. In particular, we added some more specific feedback and adjusted the weights of some of the checks. Providing the additional feedback revealed the possible benefit of being able to add specific feedback when running Gradeer, leading us to later implement the ability to add user entered feedback for manual checks. We also provided more detailed and general feedback to the entire student cohort using the distribution of solutions' base scores for individual checks. In addition, we used this check performance data to adjust the checks' weights. For example, we found that the scores of some checks would vary considerably between solutions, such as a PMD check for cyclomatic complexity, for which approximately half of the solutions achieved < 0.5. In such cases, we increased the check's weight, as it better differentiated students' solutions. However, we attempted to maintain similar total weights between the broader groups of learning outcomes, such as overall correctness and code quality, to assess students in a well-rounded manner.

4.3.5 Benefits of Gradeer

We found that Gradeer's hybrid grading approach provided several benefits when assessing this programming assignment:

Fast Manual Assessment

Gradeer provides a particular benefit in allowing for quick manual assessment. This is mostly due to Gradeer's solution inspector, which automatically executes students' solutions, and displays their source files in a text editor. Without this feature, a tutor must manually open the correct directory, enter a command to run the solution, and open the source files, before beginning the manual assessment. By removing the need to follow these steps for every solution, Gradeer removes a significant bottleneck in manual grading.

Automated Grading

By using automated grading wherever possible, we were able to reduce the number of manual checks. For example, we used some static analysis checks to evaluate the style of students' solution programs, such as ensuring that they used camel case formatting in variable names. By using these checks, the tutor did not have to look for these issues when performing the manual code inspection. Similarly, the use of unit tests to assess correctness of some elements of the program removed the need for the tutor to identify faults in these elements manually. The additional benefit of automated grading is that the checks are applied consistently across solutions. Any two students' solutions which have the same faults are assessed the exact same way.

High Quality Feedback

We found that Gradeer was capable of providing useful feedback to students. While automated checks only provide simple feedback, the large number of these checks gave students a very wide range of feedback; they could gain a good understanding of where they succeeded and where they can improve. This is supported by Falkner et al.'s findings that students' performance improves as more pieces of automated feedback are provided [186]. This feedback is further augmented by Gradeer's support for manual checks, the scores of which we used to determine which of several pieces of feedback to give to a student. The ability to provide manual feedback at runtime in the current version of Gradeer supports this even further.

Reusable

In the past, module leads typically used unique autograding scripts for each assessment. Developing these scripts is a time consuming process, and may involve repeated effort of implementing similar functionality across multiple assessments. Conversely, Gradeer can be reused in different assessments;

beyond the inclusion of new grading tests, Gradeer only requires modifications to simple configuration files to support a new assessment.

4.3.6 Challenges

When assessing the assignment, we found that uncompilable solutions introduced the greatest time cost. Around 48 of the 171 solutions initially could not be compiled or executed, due to missing files, syntax errors, or modifying files that should be unmodified. It is possible that such problems could be mitigated by preventing students from uploading broken solutions, such as by integrating Gradeer with the solution upload system, and reporting to students if an issue is detected.

Running the automated checks did take a considerable amount of time, at ∼28.2 seconds per solution using an AMD Ryzen 1700 CPU. The main source of this time cost is setting up the test execution environment. We plan to investigate a possible workaround for this issue in the future. In addition, the version of Gradeer that we used for this assessment did not support multithreading. After implementing multithreading, we observed an execution time of ∼10.9 seconds per solution on the same hardware.

We found that some static analysis rules can present a unique challenge in being used in an automated grader. In particular, Checkstyle's indentation rules can only be used with one tutor defined indentation width, while indentation widths may vary between solutions. This is an issue since several different indentation widths are commonly used in software development, any of which may be acceptable provided that they are used consistently. It may be possible to use multiple similar checks and only use the highest base score as a workaround.

4.4 Use in Experiments

I use Gradeer to execute tests on students' solutions in my experiments, by defining each unit test for a subject class as a check. I do not assign weights to any of these checks; they all contribute to generated grades equally. I also use it to execute tests on my generated mutants, by defining each mutant as an individual solution. I use the generated grades and individual check result CSV output in my data analysis for each experiment.

4.5 Conclusion

In this chapter, I have presented Gradeer, my modular grading tool that supports both the automated and manual assessment of students' programs. I have also outlined our experiences in using the tool to assess an end of year assignment for an introductory programming course, where it effectively supported the provision of quality feedback to students, and the automatic generation of grades. I have also described how I use Gradeer to facilitate the execution of my experiments for this thesis. Gradeer is available at `https://github.com/ben-clegg/gradeer` [184].

I plan to continue to maintain Gradeer, with a focus on modularising its other components, such as processes that are executed before checks, and Java-specific features. I also intend to add web integration to the tool, to enable it to be automatically executed when a student submits their solution program to a learning management system, informing them if their solution cannot be executed so that they can fix the issue.

Chapter 5

Investigating the Influence of Test Suite Properties on Automated Grading

This chapter is adapted from two of my published papers:

> Benjamin Clegg, Phil McMinn, and Gordon Fraser – *"The Influence of Test Suite Properties on Automated Grading of Programming Exercises"* – Conference on Software Engineering Education and Training (CSEET), 2020 [2]
>
> Benjamin Clegg, Phil McMinn, and Gordon Fraser – *"Diagnosability, Adequacy & Size: How Test Suites Impact Autograding"* – Conference on Software Engineering Education and Training (CSEET): Collaborative Special Track at the Hawaii International Conference on System Sciences (HICSS), 2022 [5]

The second paper [5] is a revision of the first [2], improving the analysis methodology, and using real students' solutions in place of artificial faults.

5.1 Introduction

In test-based automated grading, a student's grade can be derived from the proportion of tests in the grading test suite that pass for their solution program. Ideally, this should depend solely on the correctness of the program, but it is possible that the suite itself could be an influential factor, that could produce a source of potential inconsistency, inaccuracy and unfairness in grades. For example, test suites can vary in quality; some suites may fail to detect some faults [187]. This could result in some students receiving grades that are too high. Alternatively, a suite could detect detect some mistakes significantly more than others, overly punishing students that make such mistakes.

Figure 5.1 illustrates how different test suites can impact grades. In this example, the method in Figure 5.1a should return the absolute value of an integer, but a fault in one branch causes it to return the same value as its input parameter. If I consider grades to be calculated as the percentage of tests that pass, the suite in Figure 5.1b yields a grade of 100%; it only includes one test that never exercises the fault. If I extend this suite to execute more code, increasing coverage, it generates a more reasonable grade of 50% (Figure 5.1c). However, even with full coverage, extreme grades can still be generated; in Figure 5.1d, both of the tests are very similar, and make assertions that exercise all of the method, including the faulty line, so they both fail, generating a grade of 0%. Considering this, other metrics beyond adequacy may prove to be useful in guiding the fairness and consistency of grading. For example, diagnosability metrics (e.g. uniqueness) can provide some insight into how a test suite exercises a program. In this case, the suite in Figure 5.1d achieves minimal uniqueness, since every line is covered the same way by each test. In comparison, the suite in Figure 5.1c achieves the maximum possible uniqueness and generates the most reasonable grades; the return statements are covered by different tests. In reality, it is possible that similar issues could be present in tutors' grading test suites, perhaps due to insufficient TPACK in how to construct effective and balanced test suites.

I seek to understand how such differences in test suites affect students' grades. I originally investigated this by measuring grades generated for mutants [2]. However, such individual mutants do not necessarily perfectly reflect students' programs. For example, students' programs may contain multiple faults across several locations. I have revisited this study, using real students' solution programs in place of artificial mutants, to better reflect the real influence of test suites on grades. In this study, I consider two key research questions:

RQ1: Do grades vary with different test suites? I conducted a standard deviation analysis on the grades generated by sampled test suites for students' programs. I found that the mean standard deviation of grades for each solution is $\sim$10.1%; different suites generate a wide variety of grades.

```
int abs(int x) {
  if(x > 0) { return x; }
  - return -x;
  + return x;
}
```

(a) A fault present in a method that should return the absolute value of an integer.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
} // Passes
```

(b) Resulting grade: 100%.

```
@Test void testPositive() {
  assertEquals(2, abs(2));
} // Passes
@Test void testNegative() {
  assertEquals(2, abs(-2));
} // Fails
```

(c) Resulting grade: 50%

```
@Test void testAllOne() {
  assertEquals(1, abs(1));
  assertEquals(1, abs(-1));
} // Fails
@Test void testAllTwo() {
  assertEquals(2, abs(2));
  assertEquals(2, abs(-2));
} // Fails
```

(d) Resulting grade: 0%

Figure 5.1: Example test suites and a faulty method, illustrating an impact on generated grades. The fault is represented in diff notation; the red line starting with '-' is the correct statement, which is instead replaced by the green line beginning with a '+' below it.

Table 5.1: Dataset summary. I only include mutants that are detected by at least one test, and merge any mutants with equivalent test traces.

Task	Subject Class	Students' Solutions	Unique Mutants	Tests		LoC
				Man.	*Evo*	
Chess	`Board`	45	55	18	14	26
	`Queen`	40	46	9	2	41
Wine	`Cellar`	36	40	16	15	272
Fitness	`DataLoader`	38	19	7	1	71
	`Questions`	38	65	20	30	263

RQ2: Which properties of test suites impact grades? To further investigate exactly how test suites produce the effect I observed in RQ1, I performed a relative importance analysis of various test suite properties and the changes in generated grades. I observe that several factors of test suites influence the generated grades for students' solutions, including a suite's uniqueness, and its ability to detect other students' faults and artificial mutants.

5.2 Research Methodology

5.2.1 Experiment Procedure

In this empirical study, I use the compilable students' programs from my EndOfYear dataset, and their associated unit tests. I summarise this dataset in Table 5.1. This study requires a variety of test suites to investigate how suites influence grades; I generate such test suites by sampling from the wider set of unit tests for each subject class. I execute each test on every solution for each subject class, as well as the subject class's reference solution. I store the results of these tests, and use them to generate grades for each sampled test suite, as the proportion of a suite's tests that pass for a solution [18]:

$$G^s_\tau = \frac{|\mathbb{P}^s_\tau|}{|\tau|},$$

where s = the student's solution under test,

τ = a given test suite, such that $\tau \subset \mathbb{T}$,

$\mathbb{T}$ = set of all unit tests for the subject class,

G^s_τ = grade generated by τ for s, and

$\mathbb{P}^s_\tau$ = tests in τ that pass for s.

RQ1 As I aim to investigate how much different test suites generate different grades, I calculate the standard deviation of grades generated by my sampled test suites for each solution. I use this standard deviation instead of the absolute range of grades since some suites may only include tests that pass or fail, and would therefore have a typical grade range of $[0\%, 100\%]$. I also remove any test suites that only generate such extreme grades for every solution.

RQ2 In order to identify how different properties of test suites cause this grade variation, I perform a relative importance analysis using linear models that I construct from normalised test suite properties and changes in grades for each suite execution. I estimate a change in grades by computing the *grade delta* (ΔG^s_τ); the difference between the execution's generated grade and the median generated grade for every execution of the same solution:

$$\Delta G^s_\tau = |G^s_\tau - \tilde{G}^s_{\mathrm{T}}|,$$

where T = set of all test suites for the subject class, and

$\tilde{G}^s_{\mathrm{T}}$ = median grade generated for s by every suite in T.

5.2.2 Test Suite Properties

In order to address RQ2, I observe various properties of the sampled test suites, in order to evaluate the impact they have on the generated grades for each student's solution.

Coverage First, I consider code coverage, since it would be easy for tutors to measure and interpret when developing a test suite. I use line coverage (C_τ), since it is simple to compute and understand, especially in comparison to some other coverage metrics, such as path coverage.

$$C_\tau = \frac{|\mathbb{C}^m|}{|\mathbb{L}^m|},$$

where m = subject class's reference solution,
$\mathbb{C}^m$ = reference solution's lines covered by τ, and
$\mathbb{L}^m$ = all lines in the reference solution.

Mutation Score I also consider a suite's mutation score (M_τ) as an adequacy metric:

$$M_\tau = \frac{|\mathbb{F}_\tau^{\mathbb{M}}|}{|\mathbb{M}|},$$

where $\mathbb{F}_\tau^{\mathbb{X}}$ = set of programs in $\mathbb{X}$ detected by τ, and
$\mathbb{M}$ = set of mutants for the subject class,
with each mutant containing a single fault.

In order to evaluate the mutation score of a suite, I used the mutants that I generated using Pit [146], since these mutants are generated by a wide range of operators. I remove any mutants that are not detected by any tests, effectively normalising the mutation score to a range of $[0, 1]$. I also merge any mutants with the same behaviour for every test into a single mutant, such that every remaining mutant passes for a unique set of tests. This removes

a potential source of bias, since some types and locations of mutants would otherwise be considerably more prevalent than others. The number of unique mutants after merging for each subject class is shown in Table 5.1.

Detection Rate of Other Students' Faulty Solutions I use the detection rate of other students' faulty solutions ($D_\tau^{\mathbb{S}\setminus\{s\}}$) as an additional adequacy estimate, following the principle that a suite which detects other students' faults should also detect faults in a new student's solution:

$$D_\tau^{\mathbb{S}\setminus\{s\}} = \frac{|\mathbb{F}_\tau^{\mathbb{S}\setminus\{s\}}|}{|\mathbb{S}\setminus\{s\}|},$$

where $D_\tau^{\mathbb{S}\setminus\{s\}}$ = the proportion of other solutions detected by τ,
$\mathbb{S}$ = set of all solutions that fail at least one test in the complete test set for the subject class, and
$\mathbb{S}\setminus\{s\}$ = $\mathbb{S}$, excluding s.

I exclude the solution for which the metric is calculated (s), in order to prevent the solution itself from influencing the metric. If s is directly included in the calculation of this metric, the metric would not be independent from the grade generated for s – it would be a property of the solution and the test suite, not the test suite alone.

This metric abbreviated as "Other" and "Other Solutions" within some figures in this chapter.

Diagnosability I also consider the three diagnosability metrics that I describe in Chapter 2.11, since they offer insight into *how* a test suite covers a program. I hypothesise that since a suite with greater diagnosability should offer more accurate fault localization, it should also offer more consistent grades, as its tests are more capable of isolating individual faults.

Density (ρ_τ) measures the lines that are covered across every test in a suite:

$$\rho_\tau = \frac{\sum_{l|l\in\mathbb{L}^m}^{t|t\in\tau} A_{tl}}{|\tau| \cdot |\mathbb{L}^m|},$$
$$\text{where } A = \text{an activity matrix}(|\tau| \times |\mathbb{L}^m|);$$
$$A_{tl} \text{ denotes whether line } l \text{ was executed by test } t.$$

When $\rho_\tau = 0$, no lines are ever covered; when $\rho_\tau = 1$, every test covers every line. Gonzalez-Sanchez et al. [174] show that the optimal density to isolate faults is $\rho_\tau = 0.5$. I use normalised density, $\rho'_\tau = 1 - |1 - 2\rho_\tau|$, in this study; $\rho'_\tau = 1$ indicates the ideal density [173].

Diversity evaluates the probability that two randomly selected tests differ in their coverage behaviour, measured by the Gini-Simpson index, $\mathcal{G}_\tau$ [173, 175].

$$\mathcal{G}_\tau = 1 - \frac{\sum_{\mathbb{a}\in\mathbb{A}} |\mathbb{a}| \cdot (|\mathbb{a}| - 1)}{|\tau| \cdot (|\tau| - 1)},$$
$$\text{where } \mathbb{a} = \text{set of all tests, } t \subseteq \tau, \text{ that cover the same lines in } m;$$
$$(\forall l \in \mathbb{L}^m, \forall t, t' \in \mathbb{a} : A_{tl} = A_{t'l}), \text{ and}$$
$$\mathbb{A} = \text{set of all possible } \mathbb{a} \text{ for } \tau \text{ and } \mathbb{L}^m.$$

It is possible for some lines to share the same coverage for every test in a suite; these lines form an *ambiguity group* ($\mathbb{g}$). Having few, large ambiguity groups poses a potential issue for grading; if the lines within an ambiguity group implement different parts of a specification, tests may not be able to distinguish which aspect a fault is associated with. *Uniqueness* ($\mathcal{U}_\tau$) reveals how many ambiguity groups are present in the program.

$$\mathcal{U}_\tau = \frac{|\mathbb{G}|}{|\mathbb{L}^m|},$$
$$\text{where } \mathbb{g} = \text{set of lines, } l \subseteq \mathbb{L}^m, \text{ that are covered in the same way, } \forall t \in \tau;$$
$$(\forall l, l' \in \mathbb{g}, \forall t \in \tau : A_{tl} = A_{tl'}),$$
$$\mathbb{G} = \text{set of all possible } \mathbb{g} \text{ for } \tau \text{ and } \mathbb{L}^m.$$

Ideal uniqueness ($\mathcal{U}_\tau = 1$) indicates that every line is not covered in the same way by every test.

Size Finally, I also consider the size of a test suite, $|\tau|$, for two main reasons. First, the size of a suite may directly influence grades. For example, if a large test suite has one failing test suite for a solution, it will generate a higher grade than one with few tests and a single failure. Second, a suite's size may influence other properties, such as coverage and mutation score, as shown by Namin and Andrews [108]. By including $|\tau|$ as a property in my relative importance analysis, I am able to isolate its co-correlated impact on these other properties.

With the exception of the detection rate of other students' faulty solutions, I measure all of the properties using the reference solution, to simulate a tutor developing a new grading test suite. For metrics that require coverage information, I use JaCoCo [180] to record the coverage for every test execution, and store which lines are covered and uncovered by each test for the reference solution.

5.2.3 Grading Test Suites

Chen et al. describe a growth-based test suite sampling technique, in which a test suite is extended by randomly selecting an additional test that increases a given criterion [27]. This effectively simulates a test suite being iteratively improved; by storing a unique copy of the test suite whenever a test is added, it is possible to generate a large number of test suites with varying levels of adequacy. In order to investigate the possible changes in grades for my first research question, I used Chen et al.'s approach as inspiration for my test suite generator. The test suite generator performs the following procedure for a single generation run:

1. Randomly select a test from the whole set to use as a starting test suite.

2. Identify unused tests from the set that would improve an adequacy criterion, then add one of these to the test suite.

   (a) If the test suite has not achieved 100% line coverage, use line coverage as an adequacy criterion. A reference solution implements all of its task's learning outcomes; increasing coverage simulates adding tests that cover additional outcomes.

   (b) Otherwise, use mutation score as an adequacy criterion; some of the subject classes can be fully covered with only a small number of tests, so mutation score can provide a backup measure of adequacy.

   (c) If the test suite has already achieved 100% mutation score, reset the test suite's mutation score to 0%, and repeat step 2b. This is similar to the "stacking" approach described by Chen et al. [27,188], though for this experiment the purpose of this step is to maximise the number of generated test suites.

3. Store a copy of the test suite in its current form.

4. Repeat from step 2, until only one unselected test remains. I halt the generation here to avoid bias from a large number of identical suites that contain every available test.

I do not store the initial suites with only one test each, since they will introduce a sampling bias by only generating grades of 0% or 100%. In order to account for the random element of suite generation, I repeat this generation process 100 times, and use all of the resulting test suites to generate grades for each of the solutions.

In order to investigate the impact of test suite properties on grades for RQ2, I could not use the same approach, since constructing test suites with the goal to optimise coverage and/or mutation score would influence their impacts on the linear models. I instead opt to construct test suites by randomly sampling tests from the pool. For each of 100 generation runs, my random sampler constructed $|\mathbb{T}|$ test suites, the number of available unit tests for a subject class. My generator split these $|\mathbb{T}|$ test suites equally between several test

suite sizes; 20%, 40%, 60%, and 80%. My generator generates each individual test suite by constructing a pool of the available test suites, and randomly selecting a test from it, removing the test from the pool in the process. Once the target number of tests is reached, the generator compares the constructed suite against other suites constructed during the same generation run, and adds it to this wider set if no equivalent suites are present. This is repeated for the run until the target number of tests, or an iteration limit of $4|\mathbb{T}|$, is reached.

5.2.4 Relative Importance

To evaluate the impact of each test suite property on generated grades for RQ2, I perform a relative importance analysis [189] on the observed test suite properties, with respect to the solutions' grade deltas. Relative importance analysis allows for the impact of a linear model's explanatory variables on the response variable to be compared to one another directly. Specifically, I use the relative importance measure first proposed by Lindeman et al. [190, 191], as it allows the relative importance of explanatory variables (i.e. test suite properties) to be compared, even if they have some degree of correlation to one another, as is often the case for the test suite properties that I use in this study. For example, test suites that have a higher coverage tend to have a higher mutation score [108]. This offers a benefit over simply comparing the magnitudes of a linear model's normalised (β) coefficients, which I used in my original study [2]; β coefficients may not accurately capture the contributions for correlated explanatory variables.

Lindeman's relative importance metric relies on observations of a linear model's coefficient of determination (R^2), which represents the proportion of variation in the response variable that the model can predict from its explanatory variables; how effectively the model fits its input data. Measuring R^2 as explanatory variables are added to the model yields the model's sequential sum of squares ($seqR^2$). For example, a linear model derived from an explanatory variable, a, will have a given R^2, equal to $seqR^2_a$. Adding

more explanatory variables in a particular order, such as b and c, will each yield a new R^2, forming $seqR^2_{ab}$ and $seqR^2_{abc}$. Given that b and c both improve the model's accuracy, $seqR^2_{abc} > seqR^2_{ab} > seqR^2_a$. Since the $seqR^2$ of every explanatory variable is equal to the model's sum of squares, dividing the $seqR^2$ of all explanatory variables by R^2 yields their sequential R^2 contributions. The premise of relative importance is to decompose these into individual R^2 contributions. Following my example, the sequential contribution of c would be proportional to $seqR^2_{abc} - seqR^2_{ab}$, and for b, $seqR^2_{ab} - seqR^2_a$. However, ordering the explanatory variables poses a challenge; different orders of explanatory variables can yield drastically different contributions, since adding an explanatory variable in a different order would change its impact on R^2. Using my example, this would occur if $(seqR^2_{abc} - seqR^2_{ab}) \neq (seqR^2_{ac} - seqR^2_a)$; decomposition would yield two different contributions for c. This effect is especially prevalent if explanatory variables are strongly correlated; adding an explanatory variable that is correlated to one that is already present will increase R^2 by less than if the correlated variable was not already present. Lindeman's metric offers a solution to this problem; it produces an unweighted average $seqR^2$ from each possible ordering, yielding an average R^2 contribution for a given explanatory variable – its relative importance.

This relative importance measure also provides estimates of the relative importance in terms of the predictors' impacts on the variance of the response variable. This effectively reveals the proportional impact of the properties on the change in grades, even if the linear model does not perfectly predict the change in grades. This also allows me to compare the impact of each property across different subject classes.

I perform this analysis using Grömping's relative importance R package, *relaimpo*. This package also implements bootstrapping – repeated sampling of the linear model's residuals – which I use to derive a confidence interval for this analysis, with 2000 runs per subject class and a confidence interval of 95%. This effectively yields the estimated upper and lower bounds of each property's relative importance; should two properties' bounds overlap, it is possible that either property has a slightly higher or lower importance.

However, if the bounds of two properties are entirely disjunct, it is safe to assume that the property with the higher bounds has a greater relative importance.

I also calculate the Spearman's correlations between the test suite properties and grade delta. I do not use these correlations to determine the impact of the properties, but instead use them to further explain the impacts of the test suite properties, such as if higher measurements of a property correspond to increasing or decreasing the divergence in generated grades. Furthermore, I construct linear models with the test suite properties as predictors, and the grade delta as the response variable, in order to reveal the benefits of using a relative importance analysis instead of comparing the β coefficients of predictor variables.

5.2.5 Threats to Validity

One potential threat to validity is that sampled test suites may not necessarily reflect the construction of real grading test suites. I mitigated this by explicitly choosing a guided sampling technique for RQ1, as a test suite that a tutor would write to cover more learning outcomes should, in principle, increase in coverage and mutation score as more tests are added. However, for RQ2 I cannot use this approach, as I am investigating the impact of several properties, including those which guide the suites for RQ1. Consequently, a random sampling approach is the only viable option for this dataset.

For RQ2, I use fixed bootstrapping to yield estimate bounds according to a 95% confidence interval, shown by the range bars in Figure 5.3. Confidence intervals derived from fixed bootstrapping have been shown to not truly reflect their target confidence level [189], posing a potential threat to validity. I note that some of the properties may be more similar in how they impact students' grades than they appear in the data; properties with slightly less high importance estimates are possibly more important than those ranked above them. However, this should not heavily affect the general trends of relative importance; high or low estimates still provide a reliable indication

Table 5.2: Summary of grades generated by every sampled test suite for students' solutions, across all 30 runs. $\tilde{G}$ represents the median grade across all test suites and students' solutions. σ_G is computed by calculating the standard deviation of grades generated for each student's solution, then calculating the median of these values. All values are rounded to 1 d.p.

Subject Class	**Median Grade, $\tilde{G}$**	**Std. Dev, σ_G**
`Board`	83.3%	8.7%
`Queen`	100.0%	9.9%
`Cellar`	78.9%	13.6%
`DataLoader`	20.0%	14.2%
`Questions`	87.8%	3.8%
Mean	74.0%	10.1%

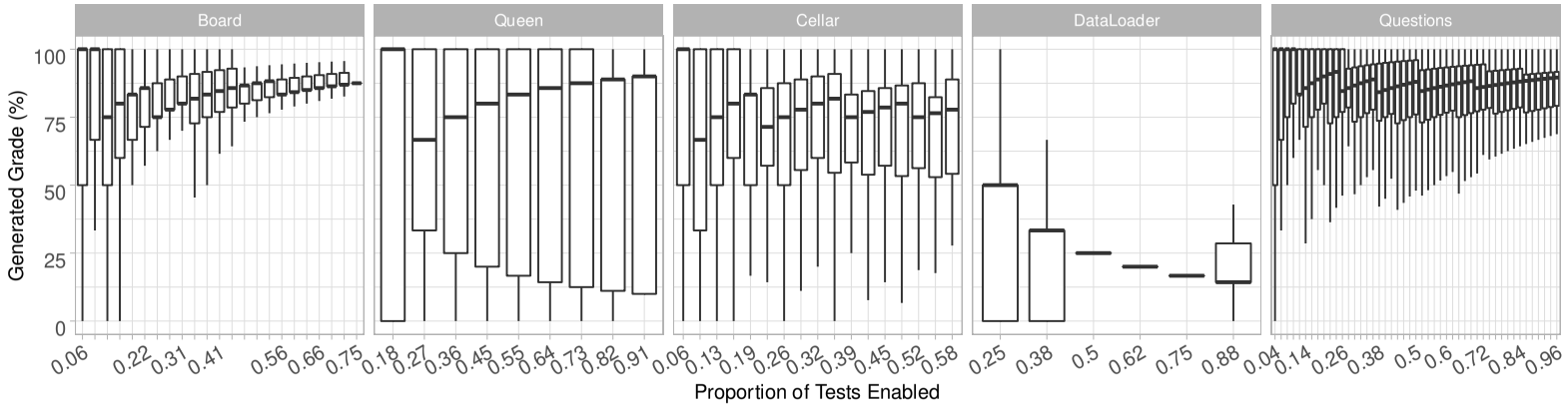
of how the observed properties influence generated grades.

5.3 Results

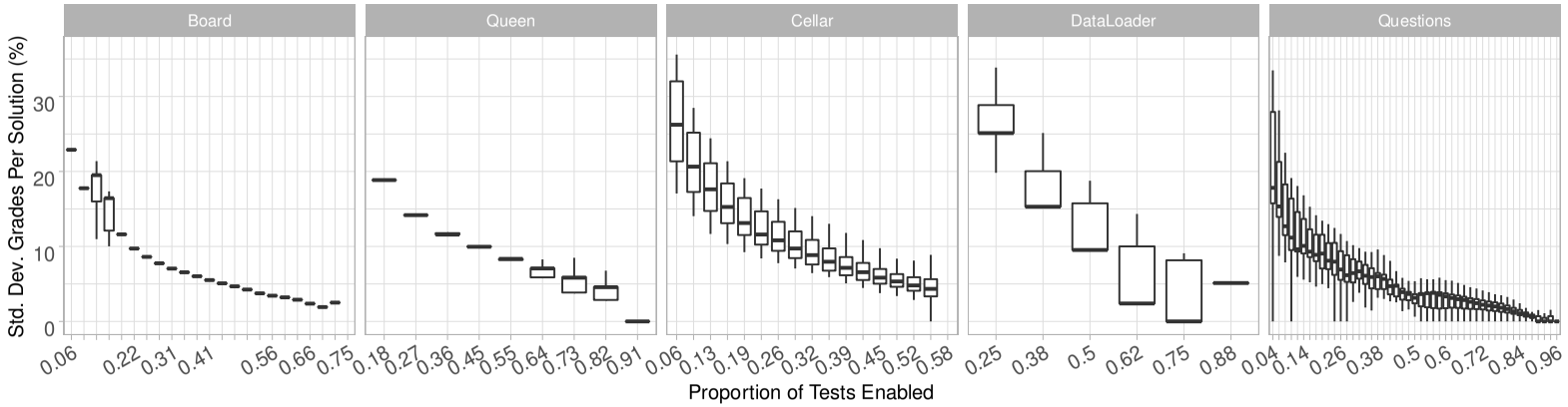
5.3.1 RQ1: To what extent do different test suites generate varying grades?

Table 5.2 shows the median grades and their standard deviations across all solutions and test suites, and their means across all subject classes. The mean standard deviation of grades per solution across all subject classes is 10.1%; different test suites yield grades that vary drastically for the same solution program.

Figure 5.2 shows the individual generated grades and grade standard deviations for each solution. I find that the subject classes have some variation in their behaviour, such as in the range of standard deviations at each test suite size, or the median grades. This indicates that a programming task itself may affect how suites evaluate students' solutions, perhaps because the specification of how a class should be implemented may influence the mistakes that students make. I observe a range of standard deviations for some of the subject classes, suggesting that the grades of some solutions



(a) Generated grades, per student's solution, per generated test suite.



(b) Standard deviation of generated grades for each student's solution, for all generated test suites.

Figure 5.2: Generated grade statistics of students' solutions for each subject class, across all 100 repetitions of suite generation via test sampling. For ease of presentation, I removed the outliers.

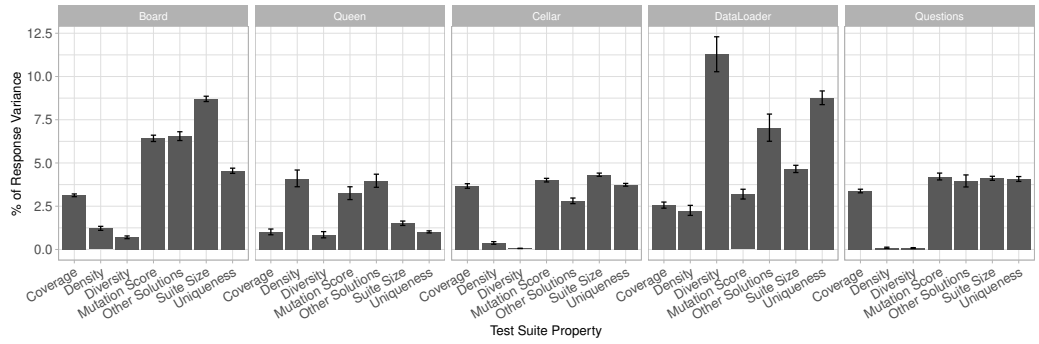


Figure 5.3: Relative importance of each test suite property, with respect to grade delta. The range bars denote the upper and lower bounds of a bootstrapped 95% confidence interval.

are influenced more by the test suite itself. For example, in `Cellar`, grades generated for some solutions by suites with 26% of the tests enabled have standard deviations of ∼16%, while others have standard deviations of ∼8%. This solution dependent variation in grades generated by different suites is a source of potential unfairness; some solutions' grades are affected by suites more than others. In comparison, this effect is less prevalent for `Board`, where most solutions have similar standard deviations in grades; the influence of the test suites on their grades is similar between different solutions. I note that the specification of `Cellar` is more complex than that of `Board`. Consquently, some students' solutions may contain more faults for particular aspects of the program's specification than others, and thus would be more susceptible to differences in test suites than other solution implementations. However, even for `Board`, different suites still generate varying grades for a given solution; suites themselves have an influence on grades. I consider how suites influence such behaviours in more detail in RQ2 and Section 5.4.

RQ1 Results: Grades generated by different suites vary considerably, with a standard deviation of ∼10.1% per solution. This standard deviation can also vary between different solutions; the grades of some solutions are affected by the test suite more than others.

5.3.2 RQ2: Which properties of test suites impact grades?

In order to determine the impact of each test suite property on the generated grades, I performed a relative importance analysis on multivariate linear models from all of the observations for each of the subjects, as described in Section 5.2.4. Table 5.3 and Table 5.4 show the relative importance estimates for each test property, with the detection rate of other students' faulty solutions included and excluded as a property, respectively. These tables also include the β coefficients of each property, and the mean correlation of each property to the grade delta across every subject class. The bounds of the predictors' relative importance bootstrapped confidence intervals are shown in Figure 5.3. Figure 5.4 further reveals the correlations of each property for each subject class.

Table 5.3: Relative importance analysis of linear models that predict grade delta from the observed test suite properties, across all students' solutions for all 100 random suite generation runs. This table includes relative importance estimates (*Est.*), linear model normalised coefficients (β), and mean Spearman's correlations (r_s). Significance levels of β and r_s are reported as: * = $p < 0.05$; ** = $p < 0.01$; *** = $p < 0.001$. $p < 0.01$ for each linear model.

Subject Class	$R^2_{\textbf{adj}}$		**Suite Size** $\lvert\tau\rvert$	**Coverage** C_τ	**Mut. Score** M_τ	**Other** $D_\tau^{\mathbb{S}\setminus\{s\}}$	**Density** $\rho_\tau{}'$	**Diversity** $\mathcal{G}_\tau$	**Uniqueness** $\mathcal{U}_\tau$
`Board`	31.24	*Est.*	8.7%	3.13%	6.42%	6.54%	1.21%	0.7%	4.55%
		β	***-4.82	***6.11	***-5	***-4.69	***11.25	***10.05	***-8.58
`Queen`	15.62	*Est.*	1.51%	1%	3.24%	3.95%	4.09%	0.84%	1.01%
		β	***-8.26	*5.3	***-15.62	***18.58	***13.38	***4.21	*-5.82
`Cellar`	18.96	*Est.*	4.32%	3.67%	4.01%	2.81%	0.37%	0.06%	3.74%
		β	***-7.24	***-5.99	-0.17	***-6.83	***16.47	-0.67	***7.66
`DataLoader`	39.68	*Est.*	4.65%	2.55%	3.18%	7.03%	2.25%	11.27%	8.77%
		β	***11.43	***12.57	***-16.78	***-10.4	***-2.28	***-41.15	***-128.78
`Questions`	19.88	*Est.*	4.11%	3.38%	4.21%	3.96%	0.09%	0.07%	4.07%
		β	***-2.87	***3.13	***-6.15	***-5.79	***8.14	***9.02	*-3.83
Mean	25.08	*Est.*	4.66%	2.74%	4.21%	4.86%	1.6%	2.59%	4.43%
		β	***-2.35	***4.22	-8.74	***-1.83	***9.39	-3.71	*-27.87
		r_s	***-0.3	***-0.3	***-0.3	***-0.25	*0	***-0.02	-0.27

Table 5.4: Relative importance analysis of linear models that predict grade delta from the observed test suite properties, as in Table 5.3, but with the detection rate of other students' faulty solutions **excluded**. $p < 0.01$ for each linear model.

Subject Class	$R^2_{\mathbf{adj}}$		**Suite Size** $\lvert\tau\rvert$	**Coverage** C_τ	**Mut. Score** M_τ	**Density** $\rho_\tau{}'$	**Diversity** $\mathcal{G}_\tau$	**Uniqueness** $\mathcal{U}_\tau$
`Board`	27.55	*Est.*	10.24%	3.68%	7.49%	0.41%	0.54%	5.19%
		β	***-6.57	***1.47	***-6.15	***4.4	***6.38	***-7.9
`Queen`	11.56	*Est.*	1.19%	0.99%	3.51%	3.37%	0.93%	1.59%
		β	***-8.6	***6.86	***-27.85	***6.91	***4.24	***62.43
`Cellar`	18.31	*Est.*	4.84%	4.15	4.73	0.24	0.05	4.31
		β	***-6.23	***-4.34	***-5.23	***11.27	*-2.15	*6.06
`DataLoader`	39.52	*Est.*	5.28	2.92	3.37	2.23	15.67	10.06
		β	***8.22	***10.87	***-16.05	-0.89	***-47.77	***-123.42
`Questions`	18.34	*Est.*	4.71	3.91	4.81	0.13	0.04	4.74
		β	***-2.32	***1.16	***-6.09	***14.97	0.73	***-9.55
Mean	23.06	*Est.*	5.25	3.13	4.78	1.28	3.44	5.18
		β	***-3.1	***3.2	***-12.28	*7.33	-7.71	***-14.47
		r_s	***-0.3	***-0.3	***-0.3	*0	***-0.02	-0.27

My results reveal how a relative importance analysis is more reliable than comparing the β coefficients of linear models in some cases. The β coefficients of correlated predictors may not accurately reflect their contributions to the predicted variable, or one of the correlated predictors may not make a statistically significant contribution to a linear model. This can be observed for mutation score and the detection rate of other students' faulty solutions in `Cellar`; these properties are correlated to one another ($r_s = 0.8$), and the β coefficient for mutation score is not statistically significant ($p = 0.64$). However, if the detection rate of other students' faulty solutions is not included as a predictor, mutation score's β coefficient becomes statistically significant ($p \leq 0.01$), as shown in Table 5.4. Relative importance does not suffer from this problem, by virtue of summarising the contribution of a single predictor across linear models constructed by adding predictors in every possible order. Accordingly, I observe that the impact of mutation score is greater than the detection rate of other students' faulty solutions for `Cellar`, and their impacts are similar overall.

In this experiment, I use grade delta as an estimate of grading inconsistency; by assuming that the median grade of a solution is a fair grade, the distance of individual grades to this median represent their inconsistencies. Grade deltas do have a limitation, however; if a solution's median grade is 0% or 100%, grade delta becomes a one-sided metric, equivalent to the proportion of tests that pass or fail for the solution. In effect, in such cases the metric does not represent the goal of my analysis. This issue occurs for `Queen`, where my randomly sampled test suites generate median grades of 100% for most solutions, similarly to the test suites that I use in RQ1. This is reflected in comparatively low relative importance estimates (and accordingly, R^2_{adj}) for the subject. Similarly, this effect also affects the correlations of the properties to grade delta. For other subject classes, where correlations are present, they are typically negative. This indicates that suites with higher measurements of the respective properties produce lower grade deltas; they generate more consistent grades. However, for `Queen` these correlations are typically positive; instead this only reveals that increasing the values of the properties increases the proportion of tests that fail for most solutions and test suites. This

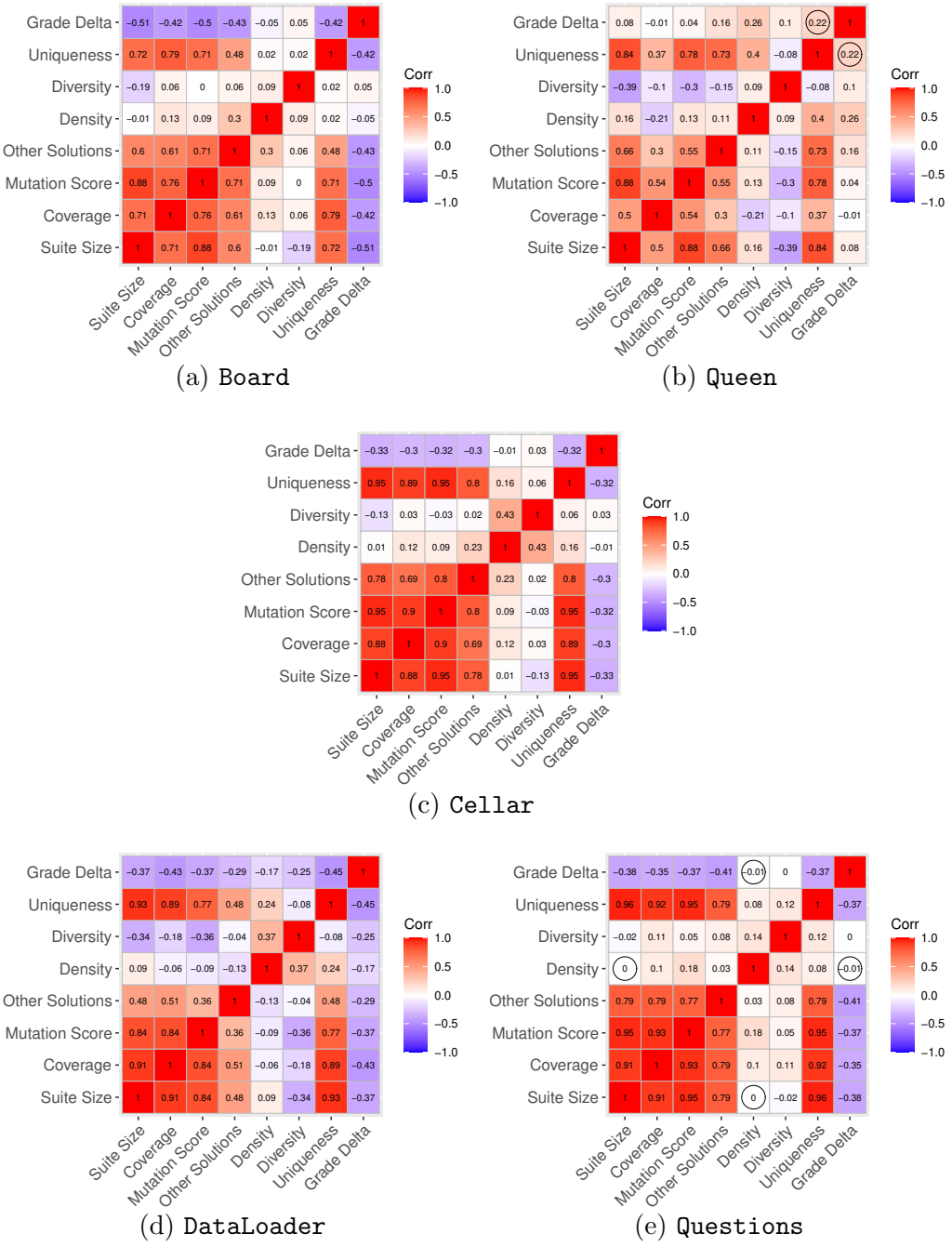


Figure 5.4: Correlations of all test suite properties and grade deltas. Circled values indicate low statistical significance ($p > 0.05$).

subject also affects the mean observations; the mean p-value of uniqueness's correlation to grade delta ($\bar{p} = 0.17$) is heavily inflated by its correlation for `Queen` ($p = 0.87$). Excluding `Queen` from my results shows that the other correlations for uniqueness are significant ($\bar{p} \approx 0.00$). In effect, for `Queen`, grade delta does not truly measure grading consistency, since it is skewed by such an extreme median grade. This subject class is an outlier in my dataset.

The adjusted R^2 of each linear model represents the grade delta's variance that is captured by the model, or in effect the combined impact of every property on the grade delta. This is equivalent to the sum of the relative importance estimates for each property. This is shown as a percentage by R^2_{adj} in Table 5.3; the mean across all five models is 25.08%; together, the properties account for 25.08% of the change in grades.

When evaluating the relative importance estimates, I find that, on average, the detection rate of other students' faulty solutions is the most influential property with respect to a change in a solution's grades, accounting for 4.86% of the variance in grade delta. This is followed by the suite's size, uniqueness, and mutation score; with impacts of 4.66%, 4.43%, and 4.21% respectively. Code coverage has a lesser impact on generated grades (2.74%), followed closely by diversity (2.59%). Finally, density has the least impact on the change in grades; 1.6% on average. The properties' contribution estimates vary between the subject classes. For example, diversity has a very high contribution for `DataLoader`, but a very low contribution for the other classes. These differences are reflected in the contributions of the complete linear models; the R^2_{adj} for `DataLoader` is the highest of all of the subject classes. It is possible that this divergence in behaviour could be due to aspects of the subject class itself having an impact on grading consistency.

As Figure 5.3 shows, there is some overlap between the bootstrapped confidence bounds for some of the test suite properties. In these cases, the true order of relative importance for the properties may be slightly different, with one of the overlapping properties possibly outperforming the other. For example this effect can be observed for mutation score and the detection rate of other students' faulty solutions for `Board`, `Queen`, and `Questions`. In these

cases, these two properties should be considered as having a similar impact on grading consistency, since despite the overall estimate for one property being higher, the true order of their importance could be the opposite.

RQ2 Results: Most properties have a statistically significant impact on grades, especially suite size, the detection rate of other students' faulty solutions, mutation score, and uniqueness. Suites with a higher measurements of these properties tend to generate more consistent grades.

5.4 Discussion

My results indicate that different test suites generate different grades, and that the properties of these suites influence the generated grades. Taking these test suite properties into consideration may help tutors to construct grading test suites that are capable of detecting students' faults, while avoiding exercising students' programs in an unbalanced manner. In this section I consider *how* each of the test suite properties can impact generated grades, and which test design strategies could be employed by tutors to control this impact.

5.4.1 Suite Size

A test suite's size has a relatively high impact on generated grades, accounting for ∼4.66% of the variance in grade delta. Since the number of tests in a test suite is correlated to other properties of the test suite, such as its coverage or mutation score [108], I include it as a property for my relative importance analysis, as a means of controlling for its impact. I note that achieving other metrics (e.g. high coverage and mutation score) will likely require a tutor to create a series of high quality tests, the quantity of which will depend on the programming task that they assess.

It is important to include the number of tests in a suite in the relative importance analysis of test suite properties and adequacy criteria in order to control for its effect.

5.4.2 Coverage

Coverage has a moderate impact on grading consistency compared to the other properties, with a relative importance estimate of ∼2.74%. While coverage does impact grading consistency, some other properties of a test suite have a greater impact. Coverage has a negative correlation with grade delta, indicating that test suites with higher coverage produce grades that are closer to the median for the subject class; grading is more consistent. This is likely due to uncovered faults being impossible for a test suite to detect; covering more lines improves a suite's ability to detect faults, and therefore generate grades that are not 100%, and closer to the median grade of the solution across every sampled suite in this study.

Maximising coverage is important to create consistent test suites, but other properties of a test suite may be more important, such as its ability to detect more subtle faults.

5.4.3 Mutation Score

Mutation score has a fairly high impact on grades, with a relative importance estimate of ∼4.21%. In addition, mutation score is negatively correlated with grade delta; improving a suite's ability to detect mutants also results in more consistent grading. Like coverage, this is likely due to the property's ability to predict the adequacy of a test suite; detecting more mutants will improve a suite's ability to detect students' faults, and therefore produce more consistent grades. This impact is greater for mutation score than coverage for each of the subject classes. I will investigate if detecting mutants does indeed improve a suite's ability to detect students' faults in more detail in Chapter 9.

The mutation score of a test suite has a relatively high impact on grading consistency, with a greater impact than coverage. Tutors may find mutation testing to be beneficial in creating quality grading test suites.

5.4.4 Detection Rate of Other Students' Faulty Solutions

The detection rate of other students' faulty solutions has the greatest impact on grading consistency of any test suite property on average, accounting for ∼4.86% of the variance in grade delta. This metric reflects the true adequacy of a test suite; its ability to detect students' faults. Since this metric is negatively correlated to grade delta, I can conclude that a test suite which detects more students' faults will produce more consistent grades.

However, this metric may be hard for tutors to use to improve their test suites. The metric would allow tutors to understand how many solutions have faults that are detected, but without manually identifying individual faults that are present in students' solutions, but it does not provide information for unknown faults that are present in students' solutions; these can only be identified and understood by using manual analysis. By contrast, artificial mutants serve as known faults; it would be easier for a tutor to write tests that target undetected, but known mutants than unobserved students' faults.

The detection rate of other students' faulty solutions has the greatest impact on a test suite's grading consistency; a test suite capable of detecting more faults generates more consistent grades. However, this metric is challenging to use in practice; a tutor would require a set of known existing faulty solutions to employ the metric.

5.4.5 Density

Normalised density has the lowest impact on grading consistency of any test suite property, with a relative importance estimate of ∼1.6, reflecting its lack of correlation to grade delta. Consequently, I can conclude that the average proportion of lines that each test in a suite covers has little bearing on the suite's ability to generate consistent grades. Instead, other qualities related to coverage—such as diversity and uniqueness—may be more important.

A test suite's density typically has little impact on its grading consistency.

5.4.6 Diversity

A test suite's diversity can have an impact on its grading consistency, representing ∼2.59 of the variance in grade delta, though this can be attributed exclusively to `DataLoader`, where it has the single greatest estimate of any property for any subject class, 11.27. From this, I can conclude that a test suite's diversity typically has almost no impact on grading consistency, except for in very specific circumstances. It is possible that this effect is related to how many tests in a sampled suite behave differently, and how well this sampled suite represents a typically sampled test suite (i.e. generate grades that are close to the median for each solution). For example, if the typical sampled test suite only contains tests that each have unique coverage behaviour, then test suites which have multiple tests with the same behaviour will generate grades that differ considerably from the median grade. This may be especially relevant to `DataLoader`, since it includes several tests that cover the same code, by virtue of the specification only defining the use of a single public method which can be called in a test.

Diversity typically has little impact on grading consistency, but could hold a considerable effect if a test suite contains a high proportion of similar tests.

5.4.7 Uniqueness

Uniqueness has a considerable impact on grading consistency, with a relative importance estimate of ∼4.43%. It is also negatively correlated to grade delta, indicating that test suites with more unique tests generate more consistent grades. Uniqueness likely leads to higher grading consistency since low uniqueness indicates that some aspects of a programming task are evaluated by every test, solutions with faults in such aspects may be overly punished by being more likely to be detected than solutions with different faults. High uniqueness also indicates that every aspect of a program is evaluated at least once; no fault would be completely uncovered by any test, and would therefore be more likely to be detected.

Attaining high uniqueness may pose a challenge for tutors however, since it is possible for a reference solution class to only have a single entry point; this entry point must be evaluated by every test. This may require some redesign of the task's specification and reference solution to avoid this problem, such as requiring that additional public methods are used. It may be beneficial for tutors to run an analysis to identify lines or methods that are executed or missed by every test, such as by adapting and using Perez's diagnosability tool [173].

A test suite's uniqueness has a considerable impact on grading consistency, though some redesign of a programming task may be required to allow a high degree of uniqueness to be achieved.

5.5 Conclusion

In this empirical study, I have shown that different test suites can generate significantly different grades for the same student's solution, with a standard deviation of ∼10.1%.

I have also identified that various observable properties of test suites can

have a significant impact on generated grades, especially mutation score and uniqueness. Developing a test suite to maximise these properties will improve the consistency of the grades that it produces. The detection rate of other students' faulty solution programs has the greatest impact on grading consistency, but is difficult to apply in practice, since such existing faulty solutions may not be available, or known to a tutor.

While both a suite's mutation score and uniqueness have a significant impact on grading consistency, in this thesis I will primarily investigate the application of mutation testing to improve grading test suites, since it should be comparatively easy for a tutor to apply mutation testing in the development of their test suites. I will further investigate how generated mutants relate to faults in students' programs. In order to do this, I first need to understand what programming mistakes students make.

Chapter 6

What Programming Mistakes Do Students Make?

This chapter is adapted from part of my published work:

> Benjamin Clegg, Siobhán North, Phil McMinn, and Gordon Fraser – *"Simulating Student Mistakes to Evaluate the Fairness of Automated Grading"* – International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET), 2019 [1]

In my original paper, I only examined mistakes present in my SemOne dataset. I have extended this analysis to also include my EndOfYear dataset in this chapter.

6.1 Motivation

In order to investigate the suitability of using mutation analysis to improve test-based automated grading, I must first understand the mistakes that students make, since lacking this knowledge will prevent me from identifying which existing mutation operators and tools may replicate students' faults,

and where it may be possible to define new mutation operators for this purpose.

While existing work has also investigated students' mistakes, these studies often have their own limitations. For example, the mistake categories investigated by Brown et al. [23] are derived from previous interviews with tutors [93], and thus are entirely founded on their beliefs. Since tutors may have gaps in their TPACK [1], these beliefs may not be entirely accurate; which Brown et al.'s work reveals. Furthermore, with a few exceptions, most of the mistakes revealed by their study would be easily detected, since they would prevent a student's program from compiling. Knowledge of such severe mistakes would not necessarily aid the development of grading test suites, since they would prevent tests from even being executed on a student's program. McCall and Kölling's work on the same dataset (Blackbox) also primarily identifies students' faults that prevent their solution programs from compiling, though they identify many more mistake categories. Keuning et al. also studied this dataset, instead focusing on code quality issues present in students' solutions. However, this study itself is limited by the fact that it only includes mistakes that are detected by PMD [2] this does not include general style faults that may be detected by other static analysis tools or manual analysis. This reveals a wider issue with mistake studies on large datasets; it is only feasible to use automated analysis techniques due to the incredibly large number of students' solution programs. This is still an effective means of detecting such mistakes, or performing a statistically meaningful analysis of their frequencies, but it may limit one's ability to understand why a student's program is faulty. Similarly, examining test failures alone may not truly reveal why or how a student has made a mistake; it is possible for several different types of mistakes to cause the same tests to fail.

In this chapter, I present an alternative analysis of students' mistakes from my two datasets. These datasets are much smaller than those used in the aforementioned prior work. While this limits my ability to draw wide, statistically significant conclusions from my results, this property does allow

[1] Technological Pedagogical Content Knowledge, as defined in Chapter 2.3.3

[2] A static analysis tool that detects code quality issues [102], as discussed in Chapter 2.6.2

me to perform a manual analysis. This manual analysis provides me with the opportunity to identify new mistake categories, especially those which can cause grading tests to fail.

6.2 Methodology

In order to perform this analysis, I first attempted to compile each solution class, in order to identify any clear compilation errors that are presented by the compiler's output messages. I then executed each of the solution classes. For classes in the EndOfYear dataset, I used the appropriate test suites, and recorded their results. However, the SemOne dataset does not include grading test suites. Instead, I used the same input data for every solution class for each of these assignments, and recorded the command-line output of each execution. I considered the test results and program output when performing a manual analysis on each solution class, since this helped me to identify where functionality mistakes were likely present. In my manual analysis, I not only considered such functionality mistakes, but also violations of style conventions, or general code quality issues.

Whenever I encountered any mistake, I classified it to an appropriate *mistake category*. If such an issue did not correspond to an existing mistake category, I described a new mistake category that accurately summarised it. I repeated the analysis for each of the solutions, in order to check that each of the newly added mistake categories were correctly identified across every solution. Furthermore, I also found that some mistake categories could be split into multiple categories, with each resulting category containing mistakes that exhibit very similar behaviour. For example, I originally defined a mistake category as "logic / control flow error", but found that its mistakes are better described by splitting it across several subcategories, including "lack of break/continue statements", "incorrect order of operations", and "else statement matches incorrect if statement." Some of my identified mistake categories have few instances in my datasets. I still included these as their own mistake categories, since they reveal unique mistakes that students can

make, and which could appear in other datasets.

6.3 Identified Mistakes

In total, I identified 56 mistake categories; 30 that impact the functionality or correctness of a student's program, eight that prevent compilation, and 18 style and code quality flaws. I define these mistake categories in Tables 6.1 & 6.2, Table 6.3, and Table 6.4, respectively.

I also list the frequencies of these mistake categories in Tables 6.5 & 6.6, Table 6.7, and Table 6.8. There are qualities of this analysis that are revealed in these results. First, I found that some mistake categories were not applicable to every subject class. For example, `Board`, `Queen`, and `Questions` are not susceptible to students' solutions using an *Incorrect Filename* to read input, since their specification does not require files to be read. Similarly, no solution for `Assignment1` and `Assignment2` *Modifies Another Class* which should not be modified, since the specifications of these programming tasks only require the creation and use of a single class each.

For other mistake categories, there are many instances of the mistake for some subject classes, but very few for others. This is due to some subject classes simply being more prone to the fault than others, due to requirements from the task's specification. One particularly prevalent example of this is for *Does Not Correctly Remove Collection / Array Contents;* 43 solutions for `Board` exhibit this mistake, whereas it never appears in the other classes. While solutions for other subject classes do use collections or arrays, only `Board` requires for contents to be removed from them in its specification.

In addition, some mistake categories appear rarely, with only a few instances across either of my datasets. One example of this is for one student's implementation of `Queen`, which has a *Mispositioned Break / Continue* statement; a nested for loop has its break statement in the incorrect branch.

Finally, I also observed that some mistakes contribute to others. For example,

code that *Exceeds Range* when reading a data structure may do so due to *Incorrect Calculation* in defining the range to read.

6.4 Limitations

My manual analysis of these mistakes is subject to some limitations. As with any manual analysis approaches, it is possible that I have failed to identify some mistakes; there may be additional mistake categories present in my datasets, or the frequencies of my mistake observations may be inaccurate. I attempted to mitigate this by making repeated passes over solutions that I previously examined after identifying new mistake categories.

Additionally, these mistake observations are unique to my datasets; other students, either in other cohorts for the course, or those enrolled in different courses, may make different mistakes, or make mistakes at a different frequency. This is an inevitable aspect of manual mistake analysis, so I must consider other courses, students' solutions, and programming tasks to be out of scope for the purpose of this analysis.

Table 6.1: Identified mistake categories that impact the functionality of a student's program.

Mistake Category	**Description**
Task-specific Implementation Flaw	The student's implementation does not match the specification.
String Misspellings	String literals include misspellings.
Incorrect Filename	A literal used as a file path is incorrect.
Incorrect Literal	A literal value is incorrect.
Missing Output	The student's program does not include sufficient output (e.g. missing System.out.println()).
Specification Not Fully Implemented	Parts of the specification are not implemented, e.g. method bodies are empty.
Overeager Input Validation	The program imposes validation that rejects input which should be accepted.
Insufficient Exception Handling	Does not catch some exceptions.
Insufficient Validation	The program does not validate input sufficiently; bad inputs are accepted.
Exceeds Range (Loops, Arrays, Strings)	The program attempts to read data outside the range of a data structure.
Flawed Conditional Logic	A conditional statement is incorrect. Can be due to incorrect boundary values, using the wrong comparator, etc.
Empty If Block	The block of an if statement is empty.
Incorrect Order of Operations	Some of the program's statements are in the incorrect order.
Lack of Branch	The program executes some statements unconditionally, when they should be executed under a particular condition.
Mispositioned Break / Continue	A break or continue statement is in the wrong location, causing a loop to end at the incorrect time.
Missing Break / Continue	A break or continue statement is missing, preventing a loop from terminating at the right time.
Does Not Reset Variable	The program does not reset a reused variable to its original value when it should.
Statements in Incorrect Branch or Outside Correct Branch	Some statements are in the incorrect branch, or are not in a branch when they should be.
Initialisation Error	A path exists in which the program attempts to read or modify an uninitialised (null) object, or an object is needlessly reinitialised.
Incorrect Method Signature	The program implements a method in the specification with the incorrect parameters.
Incorrect Calculation	A calculation in the program is incorrect with respect to the specification.
Type Fault	Not casting properly (e.g. integer division).
Incorrect Packaging (Non-severe)	Some classes are in the incorrect packages, but not severe enough to prevent compilation.

Table 6.2: Identified mistake categories that impact the functionality of a student's program (continued from Table 6.1).

Mistake Category	Description
Incorrect Return	Returning incorrect variable, or returning null instead of empty list.
Incorrect Use of Static	Methods or variables are static when they should not be.
Missing Public Access Modifier	A method or member has no access modifier when it should be public.
Does Not Correctly Remove Collection / Array Contents	The program fails to remove elements from a Collection (e.g. list.remove()) or set values in an array to null.
Modify Parameter Variable, Not Local	A method directly modifies one of its parameters, instead of a local copy of the parameter. This can especially cause issues if the specification requires the method to be called using a constant as a parameter.
Modification of Reference, Not Copy	The program modifies a reference to an object rather than a copy, causing some objects to hold the incorrect data (e.g. lists that should be copies of another list).
Bad Standard Library Call	Uses a standard library function improperly.

Table 6.3: Identified mistake categories that prevent a student's program from compiling.

Mistake Category	Description
Incorrect Classname	Incorrect internal class name, or incorrect file name of a class.
Incorrect Packaging (Severe)	Some classes are in the incorrect packages, preventing compilation when moved to an autograding environment.
Uncompilable Dependency Classes	Some of a class's dependency classes cannot be compiled.
Uses External Libraries	The program uses external libraries, violating the specification.
Modifies Another Class	The student has modified a class that the specification requires to not be modified. Their other classes may rely on these incorrect modifications.
Using Class Name as Identifier Name	The program includes an identifier (e.g. variable) that has the same name as a class.
Undefined Variable	The program attempts to read a variable which is never defined.
Missing Syntax	The program lacks some important syntax elements.

Table 6.4: Identified mistake categories that affect the style / code quality of a student's program.

Mistake Category	**Description**
Literal Value Repetition	Literal values are repeated instead of using a constant.
Statement Repetition	A block of statements is repeated instead of using a method.
Incorrect Identifier Style	Identifiers are named incorrectly (e.g. in UpperCamelCase instead of camelCase).
Poor Identifier Naming	Identifiers have names that are uninformative (e.g. a).
Poor Indentation	Identation violates code style conventions, or indentation is at different depths for no valid reason.
Constants Defined as Variables	Members are defined as variables instead of constants.
Overly Long Lines	Lines of code are over 100 characters wide.
Few Useful Comments	The program requires more, informative code comments.
Unnecessary Variables	Variables are defined that are never used.
Inefficient Code	The student's program is inefficient (e.g. unnecessary loop iterations, unneeded operations).
Lacks Annotation Tags	The program lacks annotation tags (e.g. @Override for methods that are defined in a super-class)
Inconsistent / Poor Whitespace	The student uses whitespace (e.g. spaces and empty lines) inconsistently, too little, or too much, at a detriment to readability.
Unnecessary While Loops	The student's program uses while loops where for loops or recursion are more appropriate.
Unnecessary Method Declarations	The program includes a method that does not serve any purpose; it does not improve readability, meaningfully perform a specific action, nor reduce repetition.
If-Else Instead of Switch-Case	The program uses a long if-else structure, instead of using a switch-case.
Poor Type Choice	e.g. String instead of boolean, or Double instead of double. (Excludes not using an enum.)
Lack of Enums / Improper Use of Enums	Does not use enums, or queries enums in an improper way (e.g. comparing its name to a String).
Complex / Inefficient Control Flow	Control flow is needlessly complex, increasing risk of mistakes (e.g. using counters and booleans with conditionals instead of a break/continue or methods).

Table 6.5: Frequencies of mistakes that impact the functionality / correctness of solution programs.

Mistake Category	**Frequency**						
	SemOne		EndOfYear				
	Assignment1	Assignment2	Board	Queen	Cellar	DataLoader	Questions
Task-specific Implementation Flaw	0	0	2	1	5	3	9
String Misspellings	6	0	0	0	0	0	0
Incorrect Filename	1	2	NA	NA	0	0	NA
Incorrect Literal	4	7	0	0	4	0	7
Missing Output	6	3	0	0	1	0	0
Specification Not Fully Implemented	2	3	7	7	4	0	5
Overeager Input Validation	3	0	1	0	0	0	1
Insufficient Exception Handling	0	0	0	0	2	38	5
Insufficient Validation	0	3	0	2	18	37	3
Exceeds Range (Loops, Arrays, Strings)	3	4	0	8	3	11	3
Flawed Conditional Logic	0	0	1	5	4	1	1
Empty If Block	0	0	0	0	0	0	1
Incorrect Order of Operations	0	0	2	0	1	0	1
Lack of Branch	1	0	0	1	0	0	0
Mispositioned Break / Continue	0	0	0	1	0	0	0
Missing Break / Continue	0	0	0	5	0	0	1
Does Not Reset Variable	0	0	0	0	0	0	1
Statements in Incorrect Branch or Outside Correct Branch	0	0	1	1	1	0	1
Initialisation Error	0	0	1	0	1	0	1
Incorrect Method Signature	NA	NA	2	0	0	0	0
Incorrect Calculation	21	13	0	6	3	0	13
Type Fault	0	0	0	0	0	0	1
Incorrect Packaging (Non-severe)	0	0	2	2	0	1	0

Table 6.6: Frequencies of mistakes that impact the functionality / correctness of solution programs (continued from Table 6.5).

Mistake Category	**Frequency**						
	SemOne		EndOfYear				
	Assignment1	Assignment2	Board	Queen	Cellar	DataLoader	Questions
Incorrect Return	0	0	1	24	1	0	1
Incorrect Use of Static	NA	NA	5	0	0	1	0
Missing Public Access Modifier	NA	NA	0	0	3	0	0
Does Not Correctly Remove Collection / Array Contents	0	0	43	0	0	0	0
Modify Parameter Variable, Not Local	0	0	0	0	1	0	0
Modification of Reference, Not Copy	0	0	0	0	3	0	0
Bad Standard Library Call	0	0	0	0	1	0	2

Table 6.7: Frequencies of mistakes that prevent the compilation of solution programs.

Mistake Category	**Frequency**						
	SemOne		EndOfYear				
	Assignment1	Assignment2	Board	Queen	Cellar	DataLoader	Questions
Incorrect Classname	3	4	0	0	0	0	0
Incorrect Packaging (Severe)	0	0	2	2	3	2	1
Uncompilable Dependency Classes	NA	NA	7	11	0	0	0
Uses External Libraries	0	0	4	6	0	0	0
Modifies Another Class	NA	NA	2	3	0	1	2
Using Class Name as Identifier Name	0	0	1	0	0	0	0
Undefined Variable	0	0	1	1	0	0	0
Missing Syntax	2	1	0	0	0	0	0

Table 6.8: Frequencies of mistakes that impact the style / code quality of solution programs.

Mistake Category	**Frequency**						
	SemOne		EndOfYear				
	Assignment1	Assignment2	Board	Queen	Cellar	DataLoader	Questions
Literal Value Repetition	55	45	37	23	25	22	25
Statement Repetition	44	34	3	48	34	22	34
Incorrect Identifier Style	15	20	5	6	9	8	8
Poor Identifier Naming	5	13	17	10	12	12	7
Poor Indentation	20	31	10	18	17	10	8
Constants Defined as Variables	9	31	1	0	1	4	0
Overly Long Lines	22	17	6	21	37	37	39
Few Useful Comments	3	2	19	8	7	12	18
Unnecessary Variables	1	0	5	42	18	3	6
Inefficient Code	13	18	21	3	5	3	5
Lacks Annotation Tags	NA	NA	0	51	21	2	1
Inconsistent / Poor Whitespace	6	9	19	13	21	15	11
Unnecessary While Loops	12	1	0	20	20	17	10
Unnecessary Method Declarations	0	0	2	29	1	3	1
If-Else Instead of Switch-Case	NA	39	1	2	10	6	1
Poor Type Choice	0	2	1	0	6	0	1
Lack of Enums / Improper Use of Enums	NA	57	0	0	3	0	0
Complex / Inefficient Control Flow	2	1	0	6	0	0	0

Chapter 7

Deriving Mutation Operators From Students' Mistakes

As with the previous chapter, this chapter is adapted from my published work:

> Benjamin Clegg, Siobhán North, Phil McMinn, and Gordon Fraser – *"Simulating Student Mistakes to Evaluate the Fairness of Automated Grading"* – International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET), 2019 [1]

I have extended my previous work, by investigating more existing mutation operators to match my new mistake categories, and define additional new mutants to simulate functionality mistakes.

7.1 Motivation & Methodology

Since I aim to apply mutation testing to simulate students' mistakes, I must first consider how mutation operators introduce artificial faults, and how this corresponds to the mistakes that students make. In this chapter, I investigate

which existing mutation operators have the potential to simulate students' mistakes that impact functionality. In order to achieve this, I evaluate each of my identified mistake categories in turn, and identify which operators in two existing mutation tools—Pit and Major—adequately capture the faults which each given mistake category captures.

I also identify which students' mistakes cannot be fully associated with existing operators. For this group of unmatched mistakes, I define new mutation operators, with the aim of simulating these mistakes where possible. I approach this operator construction by considering how the students' mistakes manifest, using my observations to define a rule that emulates their introduction into an otherwise correct program.

Table 7.1 provides a summary of the existing matching operators that I identify, alongside the new operators that I define for each mistake category.

7.2 Out of Scope Mistakes

First, I note that some mistake categories have unique properties which hinder the definition of associated mutation operators. I consider these mistake categories to be out of scope with regards to mutation testing and analysis, primarily due to infeasibility:

- *Task Specific Implementation Flaw:* Such mistakes are specific to unique aspects of a given programming assignment's specification. These mistakes would each need their own unique mutation operator, which targets a specific misconception about an aspect of a task's specification or how it should be implemented.

- *Overeager Input Validation:* This would require a mutation operator to capture the valid domain of an arbitrary input according to a task's specification, and add a check that this input is within a small subset of this domain.

Table 7.1: Overview of mutation operators for each of my mutation classes, including whether existing mutation operators should simulate the mistakes. New operators are displayed in italics.

Mistake Category	**Existing Operators?**	**Operator(s)**
Task-specific Implementation Flaw	No	Out of Scope
String Misspellings & Incorrect Filename	Yes	(Major) Literal Value Replacement; *(New) String Misspelling*
Incorrect Literal	Yes	(Major) Literal Value Replacement; (Pit) Constant Replacement
Missing Output	Yes	(Pit & Major) Statement Deletion
Specification Not Fully Implemented	Partial	(Pit & Major) Statement Deletion; (Pit) Null, Default & Empty Return; *(New) Multiple Statement Deletion*
Overeager Input Validation	No	Out of Scope
Insufficient Exception Handling	Partial	(Pit & Major) Statement Deletion; *(New) Targeted Statement Deletion - Exception Throwing; (New) Try Extraction*
Insufficient Validation	Yes	(Pit) Remove Conditionals
Exceeds Range (Loops, Arrays, Strings)	Partial	(Pit) Conditionals Boundary; (Pit) Remove Conditionals; *(New) Iterator Advancement*
Flawed Conditional Logic	Yes	(Pit) Conditionals Boundary; (Pit) Negate Conditionals
Empty If Block	Yes	(Pit) Remove Conditionals
Incorrect Order of Operations	No	*(New) Statement Transpose*
Lack of Branch	Yes	(Pit) Remove Conditionals
Mispositioned Break / Continue	Partial	(Pit) Remove Conditionals; *(New) Statement Transpose; (New) Partial Targeted Branch Extraction; (New) Partial Targeted Branch Nesting*
Missing Break / Continue	Yes	(Pit & Major) Statement Deletion; *(New) Targeted Statement Deletion - Break / Continue*
Does Not Reset Variable	Yes	(Pit & Major) Statement Deletion
Statements in Incorrect Branch or Outside Correct Branch	No	*(New) Branch Extraction; (New) Branch Nesting; (New) Partial If-Else Block Switch*
Initialisation Error	Partial	(Pit) Experimental Member Variable Mutator; *(New) Remove Variable Initial Value; (New) Re-Initialise Variable*
Incorrect Method Signature	No	*(New) New Parameter Creation*
Incorrect Calculation	Yes	(Pit) Math; (Major) Arithmetic Operator Replacement
Type Fault	No	*(New) Remove Casts*
Incorrect Packaging (Non-severe)	No	Out of Scope
Incorrect Return	Yes	(Pit) Null, Default & Primitive Returns
Incorrect Use of Static	No	*(New) Add Static Modifier*
Missing Public Access Modifier	No	*(New) Remove Public Access Modifier*
Does Not Correctly Remove Collection / Array Contents	Partial	(Pit & Major) Statement Deletion; *(New) Targeted Statement Deletion - Collection / Array Removal Calls*
Modify Parameter Variable, Not Local	No	*(New) Parameter Reassignment* Removal
Modification of Reference, Not Copy	Partial	(Pit) Experimental Argument Propagation; *(New) Replace Object Copy with Reference*
Bad Standard Library Call	No	Out of Scope

- *Incorrect Packaging (Non-severe):* The most common manifestation of this fault that I observed is students' classes not importing classes that are required by the specification, but which excluding does not prevent the program from compiling. In such cases, the student's program does not use any of the specification-mandated classes. A mutation operator that replicates such faults must delete an import statement, then either replace it with an import of another class, or remove calls to the deleted import. Producing compilable code from either of these options is unfeasible; for the first, the operator would have to synthesise an entire class to import that implements the same methods as the originally imported class. For the second, calls to a class defined by the specification are likely important, and removing them would likely generate uncompilable code.

- *Bad Standard Library Call:* While it may be trivial to implement a mutation operator to simulate some incorrect standard library calls, implementing a mutation operator that targets every possible improper call of Java's standard library is unfeasible; Java includes many classes, each of which contains multiple methods which may be used in several uniquely incorrect way. Targeting only a subset of these classes and methods introduces another problem; determining which classes and methods to target without introducing bias.

7.3 Existing Operators

I observe that some existing mutation operators would produce mutants that either fully or partially simulate students' programming mistakes [143, 144], as detailed in Table 7.1:

- *(Major) Literal Value Replacement & (Pit) Constant Value Replacement:* Replace literal values with a predefined default.

- *(Pit & Major) Statement Deletion:* Delete a single statement from the

program.

- *(Pit) Null, Default, Empty & Primitive Return Modification Operators:* Modify the return value of a method, by replacing it with a predefined value, according to the specified operator.
- *(Pit) Remove Conditionals:* Remove a conditional statement from a control flow statement. For example, replace `if (a == b)` with `if (true)` [144].
- *(Pit) Conditionals Boundary:* Replace a comparator with an alternate boundary, e.g. `>` to `>=`.
- *(Pit) Negate Conditionals:* Modify a comparator to its strict inverse, e.g. `==` to `!=`, or `<` to `>=`.
- *(Pit) Experimental Member Variable Mutator:* Replaces a variable assignment. For primitives, the assignment is replaced with a default "empty" value, such as `0.0`, and for objects it is replaced with `null`.
- *(Pit) Math & (Major) Arithmetic Operator Replacement:* Replace operators used in mathematical calculations, effectively simulating an incorrect calculation.
- *(Pit) Experimental Argument Propagation:* Replace a method call with one of its parameters that shares the same type as the original method's return value. For example, `sum(a, b)` would be replaced with `a`.

7.4 New Operators

I also find that some mistake categories are not emulated by existing mutation operators at all. Other mistake categories are only partially emulated by existing operators, with such partially matching operators being unable to simulate some manifestations of these mistakes. For both of these cases, I have defined new mutation operators to simulate such mistake categories, as summarised in Table 7.1:

String Misspelling Apply a small change to a string literal, to simulate simple spelling errors. This includes swapping two neighbouring characters, deleting characters, or replacing characters with another selected at random. I also note that an implementation of this operator should have a generation budget; this would entail randomly selecting a given number of possible mutated strings for each string literal in a reference program.

```
String s = "hello";
```

(a) Original

```
String s = "helol";
```

(b) Mutated

Figure 7.1: Example application of String Misspelling

Multiple Statement Deletion Similar to the existing *Statement Deletion* operator, except that it deletes more than one statement. This operator targets a given statement in the program; for example, the operator may be called on each statement sequentially, with each target statement generating a set of mutants. For a given target statement, the operator will generate a mutant by deleting the statement and one or more statements within the same block that succeed it. This is repeated such that every possible mutant that deletes a different number of statements is generated for the target statement.

```
int a = 2;
int b = 4;
int c = a + b;
```

(a) Original

```
int a = 2;
```

(b) Mutated

Figure 7.2: Example application of Multiple Statement Deletion, targeting the statement `int b = 4;`

Targeted Statement Deletion This operator is effectively equivalent to statement deletion, but specifically targets particular statements in order to provide a tutor with more clear information on what the unkilled mutant represents. In particular, I propose an operator to target three specific cases:

exception throwing, break / continue statements, and collection / array element removal calls.

```
if (badThing) {
  throw new
      RuntimeException();
}
```

(a) Original

```
if (badThing) {
}
```

(b) Mutated

Figure 7.3: Example application of Targeted Statement Deletion (Exception Throwing)

```
for (int i : values) {
  a = i;
  if (i > b) {
    break;
  }
}
```

(a) Original

```
for (int i : values) {
  a = i;
  if (i > b) {
  }
}
```

(b) Mutated

Figure 7.4: Example application of Targeted Statement Deletion (Break Statement)

```
String[] array =
{"a", "b", "c"};
array[1] = null;
```

(a) Original

```
String[] array =
{"a", "b", "c"};
```

(b) Mutated

Figure 7.5: Example application of Targeted Statement Deletion (Array Element Removal)

Try Extraction Extract the contents of a try block, to simulate exceptions not being handled correctly. This operator can produce uncompilable mutants

if a called method can throw a "checked" exception that must be handled. It may be possible to mitigate this by adding a try/catch block higher in a program's call graph, but this would greatly increase the complexity of the mutant's implementation, and may not be necessary; uncompilable mutants are naturally excluded from mutation testing and analysis, since they are never executed.

```
try {
  possiblyUnsafeMethod();
} catch (Exception e) {
  ...
}
```

(a) Original

```
possiblyUnsafeMethod();
try {
} catch (Exception e) {
  ...
}
```

(b) Mutated

Figure 7.6: Example application of Try Extraction

Iterator Advancement Duplicate a `next()` call of an object. This may force an iterator overflow when the last item of the iterator is encountered, or for the incorrect value within an iterator to be assigned to a variable.

```
while (iterator.hasNext()) {
  a += iterator.next();
}
```

(a) Original

```
while (iterator.hasNext()) {
  a += iterator.next();
  a += iterator.next();
}
```

(b) Mutated

Figure 7.7: Example application of Iterator Advancement

Statement Transpose Swap the order of two neighbouring statements.

```
a = 4;
a++;
int c = a + b;
```

(a) Original

```
a++;
a = 4;
int c = a + b;
```

(b) Mutated

Figure 7.8: Example application of Statement Transpose

Branch Extraction For a given `if` statement, move the statements that belong to its branch to immediately before or after the `if` statement. This operator can also be applied to an `else` statement instead. I also define a targeted variant of this for only the `break` and `continue` statement inside an `if` or `else` block, to specifically modify control flow.

```
if (c) {
  a--;
  b++;
}
int c = a + b;
```

(a) Original

```
if (c) {
}
a--;
b++;
int c = a + b;
```

(b) Mutated

Figure 7.9: Example application of Branch Extraction

Branch Nesting The reverse of *Branch Extraction*; statements are moved from a block of code to a branch within it. For this operator, the statements before or after a selected branching statement are moved. As above, I define a targeted variant for `break` and `continue` control flow.

```
a--;
if (c) {
  b++;
}
int c = a + b;
```

(a) Original

```
if (c) {
  a--;
  b++;
}
int c = a + b;
```

(b) Mutated

Figure 7.10: Example application of Branch Nesting

Partial If-Else Block Switch Selects a series of statements within each block of an `if-else` statement, and moves them to the other block. Every possible number of selected statements to move for each branch is selected. In addition, the operator supports moving statements from only one block to the other. The entire block is not swapped however, since that would be equivalent to inverting the conditional, which is already supported by existing operators.

```
if (c) {
  a++;
  b = 3;
} else {
  a = 4;
  b--;
}
```

(a) Original

```
if (c) {
  a++;
  b--;
} else {
  a = 4;
  b = 3;
}
```

(b) Mutated

Figure 7.11: Example application of Partial If-Else Block Switch

Remove Variable Initial Value As the name suggests, the initial value of a variable declaration is removed, simulating the access of an uninitialised variable.

```
double n = 11.2;
```

(a) Original

```
double n;
```

(b) Mutated

Figure 7.12: Example application of Remove Variable Initial Value

Re-Initialise Variable Reassign a variable to its initial value / object immediately before the variable is referenced.

```
int a = 4;
a++;
int c = a + b;
```

(a) Original

```
int a = 4;
a++;
a = 4;
int c = a + b;
```

(b) Mutated

Figure 7.13: Example application of Re-Initialise Variable

New Parameter Creation Add an extra parameter to a method declaration, and add a default value for this parameter to every call to the method.

```
public int subtract(int a,
    int b) { ... }
...
int v = subtract(5, 4);
```

(a) Original

```
public int subtract(int a,
    int b, int c) { ... }
...
int v = subtract(5, 4, 1);
```

(b) Mutated

Figure 7.14: Example application of New Parameter Creation

Remove Casts Remove casts from an expression.

```
int a = 3;
int b = 5;
double c = (double) a /
    (double) b;
```

(a) Original

```
int a = 3;
int b = 5;
double c = a / b;
```

(b) Mutated

Figure 7.15: Example application of Remove Casts

Add Static Modifier Add the `static` modifier to a member variable or method.

```
public int count = 0;
```

(a) Original

```
public static int count = 0;
```

(b) Mutated

Figure 7.16: Example application of Add Static Modifier

Remove Public Access Modifier Remove the `public` access modifier from a member variable or method.

```
public int count = 0;
```

(a) Original

```
int count = 0;
```

(b) Mutated

Figure 7.17: Example application of Remove Public Access Modifier

Parameter Reassignment Removal Remove a variable in a method that is initialised with the value of a parameter, and replace references to this variable with the original parameter.

```
public int aMethod(int
    param) {
  int a = param;
  a++;
  return param;
}
```

(a) Original

```
public int aMethod(int
    param) {
  param++;
  return param;
}
```

(b) Mutated

Figure 7.18: Example application of Parameter Reassignment Removal

Replace Object Copy with Reference Replace a new object created by copying an existing object (e.g. a method call such as `object.clone()`, or creating a new object such as `new ArrayList<>(object)`) with a reference to the original object itself. For most (non-primitive) objects, this will cause values of the original object to be modified, potentially modifying the state elsewhere in the program.

```
List<String> list = new
    ArrayList<>(originalList);
list.add("value");
```

(a) Original

```
List<String> list =
    originalList;
list.add("value");
```

(b) Mutated

Figure 7.19: Example application of Replace Object Copy with Reference

7.5 Operators for Compilation & Style Mistakes

It is also possible to define operators that aim to specifically simulate mistakes that impact compilation or code style. I present examples of such operators in the first paper that I published in this research project [1]. However, such students' mistakes can be identified using static analysis tools, or by evaluating compiler error messages [76, 192]. Therefore, I consider these operators out of scope for this thesis, since such mistakes can already be detected via these generalised approaches; using such operators would not aid tutors in detecting more programming mistakes made by students. By contrast, since grading test suites are unique to their respective programming tasks, mutants that simulate functionality faults can provide tutors with valuable information to improve fault detection in grading.

7.6 Conclusion

In this chapter, I have defined new mutants to replicate most of the mistakes that I found introductory programming students to make in Chapter 6, as shown in Table 7.1. However, simply defining mutation operators does not allow them to be generated from real programs, such as reference programming assignment solutions. In the next chapter, I detail my implementation of these mutation operators in a prototype mutation tool.

Chapter 8

MutaGen: Implementing Mutation Operators

8.1 Introduction

In order to evaluate my new mutation operators via empirical study, I must first create tangible mutants via these operators for the reference solutions of my subject classes. While manually seeded mutants can be used for such studies, they can behave differently to generated mutants [115]. Instead, I opt for an automated approach, since I can yield a large quantity of mutants with little effort. I have implemented each of my mutation operators as a prototype mutation tool, *MutaGen* (Mutant Generator) [193]. In this chapter, I discuss my approach to implementing these operators, and the challenges involved in creating an effective mutation tool.

8.2 General Program Operation

Upon execution, my tool reads a subject class's reference solution, and stores it in memory. My tool then executes each enabled mutation operator on the

solution's code. Each of these operators generates every applicable mutant from the original source code, not the class's compiled bytecode. Finally, my tool saves a Java source file for each mutant to their own labelled subdirectory. I can later compile and execute these mutants independently, such as by using an automated grading tool. This overall process is fairly simple and intuitive; the challenge lies in implementing my operators, which often require knowledge of the class' structure and the context of a mutation target.

8.3 Abstract Syntax Tree Manipulation

One means of processing the structure of a Java class is by evaluating and manipulating its *abstract syntax tree* (AST), using the *JavaParser* library [194]. JavaParser converts Java source code to an AST, in which every syntax component of the code is represented as a node in a tree, with different types of components (e.g. arithmetic expressions, assignment statements, and blocks of code) each having their own associated node type. JavaParser implements the functionality to evaluate and modify this AST, which I use in MutaGen to implement complex, context-aware mutation operators.

8.4 Operator Implementation Example

In this example, I will demonstrate how I used AST manipulation to implement my *Remove Casts* operator. This operator should remove every cast from a particular expression. For example, `(double) a / (double) b` within a variable assignment expression would be mutated to `a / b`. Figure 8.1 presents my implementation of this operator. Figures 8.2 & 8.3 show the abstract syntax tree representations of the expression, before and after the operator is applied.

JavaParser uses the visitor design pattern to process an AST. The visitor explores each node of the AST in turn, and runs its `visit` method with the

```
@Override
protected void visitorSetup() {
  // Set up the AST visitor
  visitor = new VoidVisitorAdapter() {
    @Override
    public void visit(ExpressionStmt n, Object arg) {
      super.visit(n, arg);
      // Generate a mutant from the visited ExpressionStmt
      generateMutant(n);
    }
  };
}

private void generateMutant(ExpressionStmt exprStmt) {
  // Copy the original statement to prevent direct modification
  ExpressionStmt modifiedStmt = exprStmt.clone();
  // Identify cast expressions in the ExpressionStmt's subtree
  List<CastExpr> castExprs = findCastExprs(modifiedStmt);

  if (castExprs.isEmpty())
    return;

  // Replace every cast expression with its subexpression
  for (CastExpr c : castExprs) {
    c.replace(c.getExpression());
  }

  // Create a mutant object to store the changes that are made
  addMutant(new ASTMutant(
    this.getOriginal().getCompilationUnit(),
    exprStmt,
    modifiedStmt,
    this.getType())
  );
}

private List<CastExpr> findCastExprs(ExpressionStmt exprStmt) {
  return exprStmt.findAll(CastExpr.class);
}
```

Figure 8.1: Implementation of my *Remove Casts* operator.

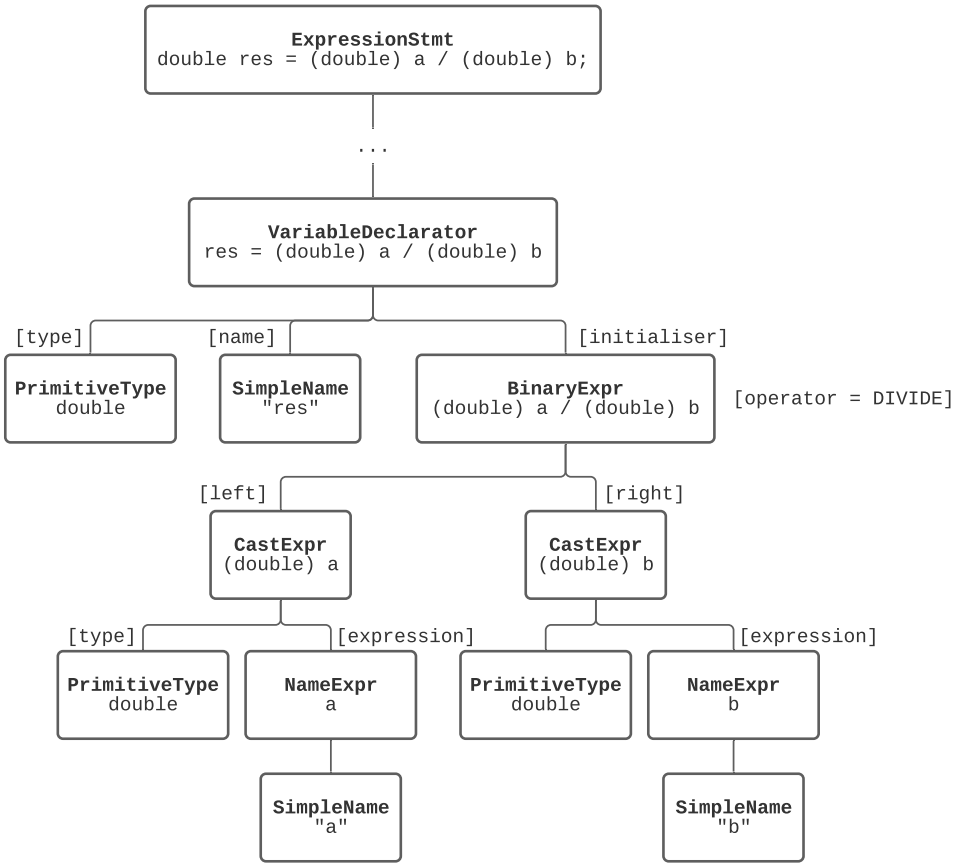


Figure 8.2: Abstract syntax tree representation of the variable declaration of `res`, with `(double) a / (double) b` as its initialiser, before the *Remove Casts* operator is applied.

current node as a parameter. This method is polymorphic; the particular `visit` method that has a parameter matching the node's type is executed. In this operator, I target `ExpressionStmt` nodes, with the visitor calling my `generateMutant` method on the matching node. The visitor design pattern calls this method once in isolation for each `ExpressionStmt` in the reference class. I target these `ExpressionStmt` nodes instead of `CastExpr` nodes directly, since my operator should remove every cast expression that is present in a single expression to generate a single mutant, rather than only one cast expression.

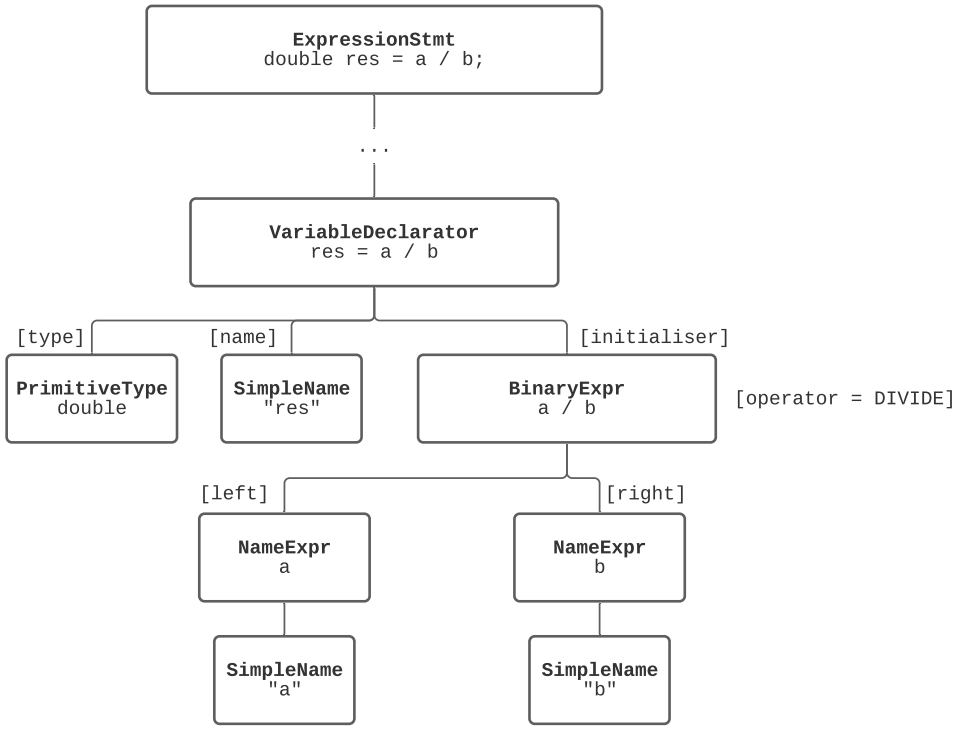


Figure 8.3: Abstract syntax tree representation of the modification that the *Remove Casts* operator makes to Figure 8.2.

The first step of my mutation procedure is to copy the matched node. If the node is not copied, any modifications made to the node (or its children) will not only be applied to the mutant under generation, but to the original AST; such changes would also be applied to future mutants.

The next step of my mutation rule is to identify every `CastExpr` within the `ExpressionStmt`, in order to determine which nodes must be removed from the expression. Should no casts be present, the method simply returns, since no mutant can be generated. Otherwise, the next step is to remove each of these identified casts. In order to do this, I replace the `CastExpr` with its associated expression. This seems counter-intuitive at first glance, but in reality the AST is structured such that a `CastExpr` (e.g. `(double) a`) has two components; the type (`double`), and the expression to cast (`a`). Replacing the `CastExpr` with its associated expression is equivalent to removing the cast

(`(double)`) itself; removing the entire node would also delete the originally casted expression instead.

Finally, as with each of my operators, I create a `Mutant` object that details the changes that have been made. When storing the mutant as a source file, MutaGen uses JavaParser to reconstruct the class's source code, supplying it with these modified nodes in place of the original nodes.

8.5 Limitations

MutaGen does have some limitations in its application. First, like every mutation tool, it is not immune to the equivalent mutant problem; it is possible for the tool to generate mutants that are semantically equivalent to the original program. When using MutaGen's mutants in my experiments, I manually inspect each mutant that passes every test, and remove any that are indeed equivalent.

Since my mutation operators tend to introduce significant changes to the program, some mutants generated by MutaGen can be uncompilable. Fortunately, this issue is easy to resolve; I use an automated compilation and test execution procedure via Gradeer, which allows me to automatically remove any mutants that cannot compile, so these mutants are excluded from my experiments.

Finally, since I conduct my studies on individual subject classes, I only designed MutaGen to support the mutation of an individual class in isolation. It is possible to extend MutaGen to support the mutation of multiple classes from a single program, but this would incur a significant time cost, and would not benefit my experiments.

Table 8.1: Mutants generated by MutaGen.

Subject Class	Mutants		
	Total	Non-Equivalent	Unkilled Non-Equivalent
`Board`	23	21	1
`Queen`	79	70	0
`Cellar`	421	342	39
`DataLoader`	108	94	2
`Questions`	330	295	0

Table 8.2: Comparison of the proportion of mutants generated by each tool that are equivalent, to 3 s.f.

Subject Class	Proportion of Equivalent Mutants (%)		
	Major	*MutaGen*	*Pit*
`Board`	7.02	8.70	13.7
`Queen`	2.13	11.4	9.33
`Cellar`	18.0	18.8	16.3
`DataLoader`	6.81	13.0	15.4
`Questions`	5.53	10.6	10.7
Mean	7.90	12.5	13.1

8.6 Generated Mutants

As with Major and Pit, I generated mutants from the reference solutions of my EndOfYear dataset using MutaGen, which I summarise in Table 8.1. I identified and removed equivalent mutants in the same manner as I describe in Chapter 3.4.

Equivalent mutants introduce a time cost of manual analysis to mutation testing; tools which generate fewer equivalent mutants would be easier to apply to evaluate the adequacy of a test suite. Therefore, I compare the proportion of equivalent mutants that are generated by each tool, as shown in Table 8.2. I find that MutaGen and Pit (with all operators enabled) produce a similarly high proportion of equivalent mutants, at 12.5% and 13.1%, respectively. By contrast, Major, which only implements simple mutation operators, generates considerably fewer equivalent mutants proportionally (7.90%). This suggests that simpler mutation operators are less prone to generating equivalent mutants than more complex operators. This is counter-intuitive, but is likely primarily due to particular operators. For example, my *Statement Transpose* operator can produce valuable faulty mutants in some cases, but it is possible that the order of some statements has no bearing on

a program's correctness.

8.7 Conclusion

I have successfully implemented my new mutation operators as a prototype mutation tool, MutaGen. These operators, which aim to replicate students' programming mistakes, manipulate the abstract syntax tree of a program to generate mutants. MutaGen does generate a greater proportion of equivalent mutants than Major, which only implements simple mutation operators, but these represent the minority of generated mutants; the vast majority of MutaGen's mutants can be applied to evaluate the adequacy of a grading test suite.

The source code of MutaGen is available at `https://github.com/ben-clegg/mutagen` [193].

Chapter 9

Evaluating the Suitability of Mutation Operators to Simulate Students' Mistakes

This chapter is based upon my published work:

> Benjamin Clegg, Phil McMinn, and Gordon Fraser – *"An Empirical Study to Determine if Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding"* – ACM Technical Symposium on Computer Science Education (SIGCSE), 2021 [3]

This chapter expands upon my published work, featuring updated data and additional experiments.

9.1 Introduction

While existing work has shown that mutants are a valid substitute for real faults in software projects when assessing a test suite's quality [26, 28], there

has been little investigation into whether this holds for evaluating a tutor's test suite's ability to detect faults in students' programs. Such supporting evidence is necessary to support the use of mutation testing and analysis to improve tutors' grading test suites. Accordingly, in this chapter I conduct an empirical study to investigate the effectiveness of artificial mutants in simulating students' faulty programs, using mutants generated by both existing tools and my prototype mutation tool, MutaGen. I aim to address six research questions:

RQ1: Are mutants coupled to students' faults? I first investigate how the coupling effect—a core principle of mutation testing—relates to artificial mutants and students' faulty programs. The coupling effect dictates that if a test suite is capable of detecting simple faults, it should also detect complex faults [113]. In the context of test-based automated grading, artificial mutants are such simple faults, and students' faults represent complex faults. However, due to practical limitations, I cannot isolate individual faults within students' programs. Accordingly, I investigate whether mutants are coupled to students' faulty programs. By showing that the coupling effect holds for mutants and students' programs, I can conclude that mutation testing can be applied to inform the development of grading test suites.

RQ2: Does MutaGen improve fault simulation? MutaGen implements my new mutation operators that specifically aim to simulate students' faults. Mutants generated by MutaGen should couple to students' faulty solutions which are not coupled to mutants generated by existing tools. I manually investigate how MutaGen's mutants can simulate students' faults, and analyse the mutants of each tool using a new coupling approach.

RQ3: Do mutants sufficiently capture the subtlety of students' faults? The coupling effect reveals that mutants can cause the same tests to fail as students' faults, but does not necessarily reveal that mutants are sufficiently subtle to inform the development of a test suite that can isolate

individual students' faults. Accordingly, I also consider probabilistic coupling, Chen et al.'s metric for how sensitive test goals are with respect to real faults [27]. This allows me to determine how closely the most representative mutant aligns with a student's faulty solution.

RQ4: Is mutation score analogous to real fault detection? I perform a simple correlation analysis of mutation score and the detection rate of students' faulty solutions for sampled test suites. A strong correlation between mutation score and the fault detection rate would suggest that increasing a suite's mutation score by adding more tests will improve its ability to detect students' faulty solutions. However, there are some limitations to this approach, such as the influence of other test suite properties which are correlated with both observations, such as the number of tests [27]. Consequently, I use the suite growth approach proposed by Chen et al. [27] to generate test suites, guided by adding tests that kill additional mutants. I also generate another set of test suites with random sampling to use as a baseline; the test suite is grown by adding a previously unselected test at random. I then compare how many students' faulty solutions are detected by these two groups of test suites; if mutation analysis is effective, a test suite guided by killing mutants will detect more students' faulty solutions than a test suite of the same size that was generated by random sampling.

RQ5: How do mutation score and code coverage compare in their effectiveness as adequacy metrics? I include coverage as an alternate test goal to guide a suite growth analysis to directly compare the adequacy performance of a suite's code coverage and mutation score. I also consider how using both code coverage and mutants to improve a suite influences its adequacy.

RQ6: Which mutation tool produces mutants that best represent students' faults? I further extend my suite growth analysis to compare the mutants generated by each mutation tool.

9.2 Experiment Procedure

In this chapter, I will outline the methodology that I employ in this empirical study. I use my EndOfYear dataset throughout this study.

9.2.1 Coupling

If a set of simple faults are detected by a test that also detects a complex fault, the complex fault is coupled to the simple faults. Existing work on coupling uses individual known faults to evaluate the coupling effect [28]. However, it is possible (and likely) that faulty students' solutions contain multiple complex faults; students' solutions cannot be considered as individual faults. It is therefore not sufficient for only one test that fails on a student's faulty solution to also fail on a mutant, as this would only show that the solution contains a coupled fault; it may also contain other uncoupled faults. While it may be possible to isolate individual faults in each student's program, and construct a series of pseudo-solutions that each only contain one fault, this would be intractable in practice. Instead, for **RQ1**, I perform my coupling analysis on each individual failing test of each faulty solution. For each faulty solution, I identify the failing tests. For each failing test, I determine if the solution is coupled with any mutants; i.e. the test also detects at least one mutant. If the solution is coupled with any mutants for this test, I define the test as a *coupling* test. Otherwise, I define it as an *uncoupling* test.

I use these observations to calculate the *coupling ratio* for each faulty solution, the proportion of coupling tests to failing tests for the solution. This allows me to determine to what extent a solution's faults are coupled to the available mutants. If a solution has a coupling ratio of > 0, it has at least one coupling test; it is partially coupled to the mutants, as it has at least one fault that is coupled to at least one mutant. If all of a solution's tests either pass or are coupling (coupling ratio $= 1$), then it is absolutely coupled to a set of mutants; every detected fault in the solution is coupled to at least one mutant. In my results (Table 9.1), I show the mean coupling ratio for each subject

class.

I find that for some subject classes, some generated mutants fail on every test. For example, some mutants cause an exception to be thrown when calling a class's constructor; executed tests immediately fail. I removed these trivially detected mutants prior to running my coupling analysis for **RQ1**, since they would introduce bias by being coupled to every fault in each student's solution. This procedure does pose a challenge for some students' faulty solutions, however; some solutions have a severe fault that causes every test to fail, including tests which only fail due to such severe faults. For such solutions, only trivial mutants can couple to them, though some of these may be coupled for entirely unrelated reasons. However, some of my operators which I have implemented in MutaGen directly simulate some of these severe faults. Therefore, for **RQ2**, I manually analyse faulty solutions that are otherwise uncoupled for this reason, and compare them to the trivial mutants, in order to determine if a mutant behaves similarly to the real fault. If a trivial mutant exists which introduces the same fault that is present in such a student's faulty solution, they are coupled.

Not all of the operators that I have implemented in MutaGen introduce such severe faults. In order to further evaluate the coupling performance of these operators compared to those of existing mutation tools for **RQ2**, I introduce a new variant of coupling analysis; *bidirectional coupling.* A set of mutants is bidirectionally coupled to a student's faulty solution, provided that both *a)* every test that fails for the student's solution fails for at least one of the mutants (as with traditional coupling); and *b)* no mutant in the set causes a test to fail that passes for the student's solution. Bidirectional coupling essentially evaluates that a set of mutants do not capture additional faults that are not present in a student's solution. The goal of this analysis is to determine whether a set of several mutants is capable of fully simulating the detected faults of a solution with respect to test failures. Accordingly, I do not examine the coupling rate of faulty solutions using individual test failures for this analysis; instead I use every test execution for a solution, treating the solution as a single entity that can contain one or more faults. In this

analysis, I check how many faulty solutions have at least one bidirectionally coupled set of mutants, using mutants generated by each of the mutation tools that I am evaluating. I include trivial mutants in this analysis, since they will only be bidirectionally coupled with faulty solutions that also fail every test.

In order to evaluate how killing individual mutants influences the detection of students' faulty solutions (**RQ3**), and how this can compare to covering individual lines, I use probabilistic coupling, as defined by Chen et al. [27] I group test goals into sets: each mutation tool's generated mutants, and covered lines of the model solution. I compute the detection probability for each test goal and each student's faulty solution, by selecting the tests that achieve a test goal (i.e detect a mutant or cover a line) and calculating the proportion of these that fail on the student's solution. Where no tests achieve a test goal, I use a probability of zero. I select the maximum probability of a test goal in each set for every student's faulty solution, since this allows me to directly compare the best case scenario for each group of test goals.

9.2.2 Correlation

By finding evidence of a relationship between the detection rates of mutants and students' faults, I can assert that mutation analysis is an effective measure of test adequacy. Accordingly, I investigate the correlation between these detection rates to partially address **RQ4**. I only use students' solutions for each subject class which fail at least one test in my dataset, in order to ensure that the maximum possible detection rate is consistent across subject classes. Otherwise, if solutions without any detectable faults were included in this analysis, there would be some bias due to some subject classes having more students' solutions that never fail tests than others. Since my dataset only contains one test suite for each subject class, I generate test suites which are subsets of the main test suite for each class. I control for test suite size, targeting a consistent number of test suites for each possible size from two tests to 70% of the total number of tests. For each suite size, I generate a

set of test suites by randomly sampling from the complete set of tests for the class. I discard duplicate test suites within a single run of this generation. My generation strategy aims to generate 80 randomly sampled suites for each class, split evenly between each possible test suite size. I repeat this process 100 times, generating 100 sets of ∼80 suites for every class. For `Queen` and `DataLoader`, some suite sizes cannot reach the target number of tests. Consequently, these subjects have fewer sampled test suites overall.

By evaluating the results of tests in each generated suite on students' faulty solutions and mutants, I am able to determine how many mutants and students' faulty solutions are detected by each suite. This allows me to calculate the mutation scores and real fault detection rate for each suite. Since the solutions may contain multiple faults, this real fault detection rate is truly the proportion of faulty students' solutions that have been detected. This presents a limitation to this empirical study; it would naturally be more beneficial to isolate individual faults within students' programs. However, such an approach would be impractical, since it would require manual extraction of each individual fault within each student's solution, followed by manually constructing a series of solutions that each contain one of these extracted faults; an intractable task.

I also evaluate the correlation of line coverage and real fault detection, in order to partly address **RQ5**. I only consider the model solution's coverage to simulate a tutor developing a test suite before any students' solutions have been collected. Similarly, to gain some insight for **RQ6**, I split my generated mutants into multiple sets to use in this correlation analysis; a set for each mutation tool, as well as sets which combine mutants generated by multiple mutation tools. This allows me to directly compare the correlations of the mutation scores for each of these tools, as well as the correlation for line coverage.

I order to determine which correlation approach to use, I first needed to determine if my data is normally distributed. I used the Shapiro-Wilk test [195], identifying that my observations for each variable do not fit a normal distribution; I require a non-parametric measure of correlation. Accordingly,

I compute the Spearman's correlation between the adequacy metric (i.e. the mutation score of a set of mutants, or the line coverage ratio) and the real fault detection rate.

Unlike my analysis of the coupling effect, I do not remove mutants that failed on every test prior to evaluating the correlation of mutation score to real fault detection. This approach better represents how a tutor would use a mutation tool to evaluate their grading test suites; determining how many mutants are detected by a test suite, and identifying those which are not.

9.2.3 Growth-Based Suite Generator

To address **RQ4**, **RQ5**, and **RQ6**, I also compare the fault detection capabilities of test suites that are guided by line coverage, or the mutants generated by each mutation tool. In order to create such test suites, I adapted the growth-based test suite generator that I describe in Chapter 5.2.3, inspired by the work of Chen et al. [27] Given a target adequacy metric (e.g. line coverage, mutation score for mutants generated by a specific mutation tool, or both), the generator performs the following procedure for a single generation run:

1. Randomly select a test from the whole set to use as a starting test suite.
2. Identify unused tests from the set that increase the target adequacy metric (e.g. have a higher mutation score, or cover more lines), then randomly select one of them and add it to the test suite.
    (a) Alternatively, if configured accordingly, switch from one adequacy metric (e.g. coverage) to another (e.g. mutation score) once the first has reached 100%, in order to simulate a tutor using both adequacy metrics to develop a test suite.
3. Store a copy of the test suite in its current form. This allows me to compare the fault detection performance of different test suites that have the same number of tests.

4. Repeat from step 2 until either:

   (a) The adequacy metric has reached 100% (or the second adequacy metric has reached 100%, if two are used); or

   (b) Only one unselected test remains. This prevents the generator from always generating the exact same test suite for every run.

Since this suite growth approach involves random selection, I run the sampler 100 times for each subject class, to yield a wide range of test suites for my analysis.

This generation approach allows me to compare the adequacy metrics with respect to how they guide the improvement of test suites, in two ways. First, I can identify which adequacy metric guides the generation of the smallest test suite which can still detect the maximum number of faulty students' solutions. Second, I can compare the fault detection performance of test suites generated according to different adequacy metrics at every possible test suite size. These analysis approaches allow me to understand which adequacy metric can guide the creation of an effective test suite with the least effort from a tutor (i.e. writing the fewest tests).

9.3 Threats to Validity

In this chapter, one of my goals is to evaluate how effectively mutants generated by MutaGen can simulate programming faults made by students. Due to the limited nature of my dataset, I must compare MutaGen's mutants to students' faulty solutions from part of the same dataset which I originally used to identify students' mistakes. This presents a potential threat to validity, since I used my analysis of students' mistakes to define the mutation operators that I implemented in MutaGen. Using the same faulty solutions in the definition and evaluation of these mutants limits the generalisability of the evaluation's results.

Table 9.1: Coupling Results

Class	*% Solutions Coupled*	*Coupling Ratio*			
		Major	*MutaGen*	*Pit*	*All Mutants*
`Board`	100.0	1	1	1	1
`Queen`	100.0	1	1	1	1
`Cellar`	91.4	0.99	0.949	0.99	0.997
`DataLoader`	100.0	1	1	1	1
`Questions`	100.0	1	0.472	1	1
Mean	98.3	0.998	0.884	0.998	0.999

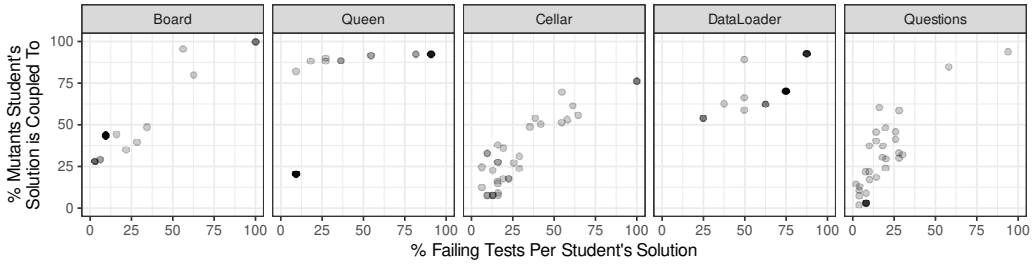
One approach to reduce this threat to validity would be to use cross-validation; using different solutions' mistakes to define and evaluate the mutation operators. However, such an approach would not be appropriate in this context; it is unlikely that a mutant would not couple to a set of faulty solutions which explicitly exclude solutions with mistakes that the mutant's operator aims to replicate.

9.4 Results & Analysis

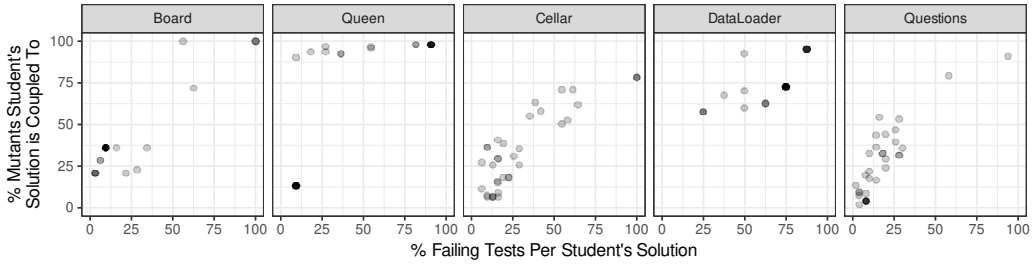
9.4.1 RQ1: Are mutants coupled to students' faulty programs?

Table 9.1 presents the results of my coupling analysis. I find that for four of my subject classes, every solution is coupled to a set of mutants. Of `Cellar`'s 35 solutions that fail at least one test, three are uncoupled. These three uncoupled solutions each fail every test; one or more of these tests do not fail for any non-trivial mutant. These three solutions are coupled to trivial mutants, but since such mutants may introduce entirely unrelated faults, I will further investigate their coupling in **RQ2**.

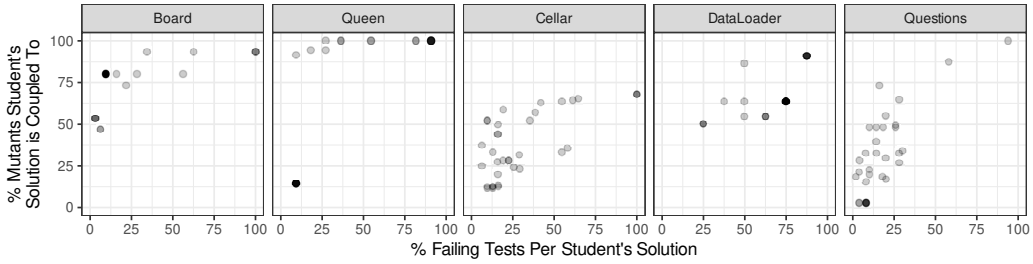
I also consider the relationship between a solution's test failures and how many mutants it is coupled to. Figure 9.1 presents my observations. First, I note that there appears to be a linear relationship between test failures and coupling for most of the subject classes, typically irrespective of the mutation



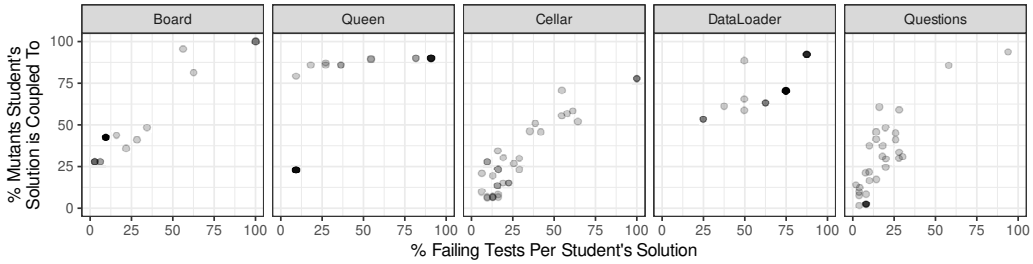
(a) All mutants



(b) Major's mutants



(c) MutaGen's mutants



(d) Pit's mutants

Figure 9.1: Observations of coupled mutants and failing tests for each student's solution. Each datapoint has the same opacity; darker points represent more datapoints.

tool. This occurs since some mutants will only be detected by particular tests, and therefore will be more likely to couple to solutions which fail more tests. `Queen` presents an exception to this; solutions which fail the minimum number of tests are coupled to few mutants, but solutions which fail any more tests are coupled to the overwhelming majority of mutants. For this class, it is likely that most mutants will be detected by particular tests, so any solutions which fail such tests will also be coupled to these mutants.

RQ1: Non-trivial mutants are coupled to most of the students' faulty solutions. There is typically a linear relationship between the number of tests a solution fails and the number of mutants it is coupled to.

9.4.2 RQ2: Does MutaGen improve fault simulation?

The most apparent observation I can make is that, as shown in Table 9.1, MutaGen's non-trivial mutants are coupled to fewer solutions, with a lower coupling rate for both `Cellar` (0.949), and `Questions` (0.472). This is likely due to MutaGen's operators not targeting some fault classes, such as simple calculation mistakes, which are likely more prevalent in solutions for these subject classes. I did not implement such operators in MutaGen, since they are already adequately represented in these existing mutation tools. This result indicates that specifically simulating particular types of mistakes that students make does not always provide a clear benefit over using operators that introduce more generalisable and subtle faults.

Upon manually examining the three solutions that are not coupled to any non-trivial mutants, MutaGen's true benefit is revealed. All three of these solutions fail every test due to not including a public access modifier in the class' constructor; every test attempts to call this constructor, but cannot for these three solutions. While this fault is not simulated by mutants generated by Pit or Major, MutaGen implements an operator that specifically introduces this fault; it generates a mutant that perfectly simulates the otherwise uncoupled fault. I excluded this mutant from my previous analysis

Table 9.2: Bidirectional Coupling Results

Class	Faulty Solutions	Solutions Bidirectionally Coupled to Mutants				
		All	*Major & Pit*	*Major*	*MutaGen*	*Pit*
`Board`	43	40	39	38	8	39
`Queen`	34	16	13	11	10	13
`Cellar`	35	12	12	9	6	12
`DataLoader`	38	36	16	1	9	16
`Questions`	34	8	8	8	4	8

since it fails every test; it and every other trivial mutant would couple to every solution if they were included.

I also manually inspected the coupling results for mutants generated by both Major and Pit. I found that two additional solutions for `Cellar` are not coupled to these mutants. These solutions both fail a test by modifying the reference of a `List` object, instead of a copy of the object. However, these solutions are coupled to some mutants generated by MutaGen, specifically those that are generated by my *Replace Object Copy with Reference* operator.

Table 9.2 shows the results of my bidirectional coupling analysis. This criterion is much stricter than traditional coupling, since it requires a combined set of mutants to fail *exactly* the same tests as a solution. While MutaGen's mutants alone have a fairly low bidirectional coupling rate compared to the other tools, adding these mutants to those generated by the other tools (*All* vs. *Major & Pit*) improves the bidirectional coupling rate for three of the subject classes. This effect is especially prevalent for `DataLoader`; adding MutaGen's mutants more than doubles the number of bidirectionally coupled solutions. Using MutaGen in conjunction with other, more generalised mutation tools can generate groups of mutants that, when combined, can more accurately simulate students' faults.

RQ2: While MutaGen's mutants are not coupled with students' solutions as much as mutants generated by the other mutation tools, MutaGen implements unique operators that directly simulate otherwise uncoupled faults. MutaGen offers a clear benefit when combined with other mutation tools.

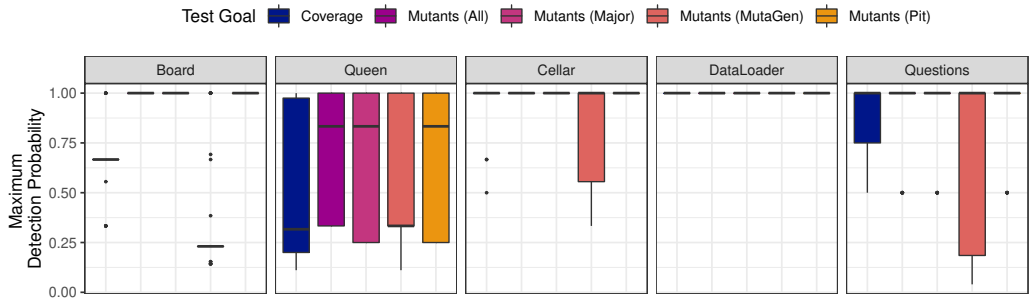


Figure 9.2: Maximum solution detection probabilities of every student's solution for each set of test goals.

9.4.3 RQ3: Do mutants sufficiently capture the subtlety of students' faults?

Figure 9.2 presents the results of my probabilistic coupling analysis. I find that mutants generated by Major and Pit, and—by extension—the whole set of mutants have the highest overall maximum detection probabilities, with an average maximum detection probability of 100% for four of the subject classes. The only class where they do not achieve an average maximum of 100% is `Queen`, where they still have the highest average maximum detection probabilities. This high probabilistic coupling reveals that, for most students' solutions, there exists a mutant for which every test that detects the mutant also fails on the student's solution. Such mutants would not introduce faults outside the scope of the students' solutions.

This effect does not occur in most of the subjects for MutaGen; its mutants rarely yield a 100% maximum detection probability. MutaGen's operators generate more complex faults, which will often cause more test failures than unrelated and simpler students' faults. A similar effect occurs for coverage, though to a lesser extent, and for a different reason; many tests can cover a line, but not all of these tests will detect subtle faults.

Another clear observation I make from this analysis is that many solutions for `Queen` have relatively low maximum detection probabilities for each set of test goals. This indicates that many solutions for this subject are very subtle;

Table 9.3: Mean Spearman's correlations (r_s) of the detection rate of faulty students' solutions and both adequacy metrics; mutation score and code coverage. The correlations for mutation score are shown for each mutation tool, and a combination of mutants generated by every tool. p = p-value.

Class	**Mutation Score**								**Coverage**	
	All		*Major*		*MutaGen*		*Pit*			
	r_s	p	r_s	p	r_s	p	r_s	p	r_s	p
`Board`	0.62	<0.01	0.55	<0.01	0.46	<0.01	0.64	<0.01	0.64	<0.01
`Queen`	0.46	<0.01	0.46	<0.01	0.37	<0.05	0.44	<0.01	0.25	0.10
`Cellar`	0.75	<0.01	0.76	<0.01	0.72	<0.01	0.74	<0.01	0.71	<0.01
`DataLoader`	0.29	<0.05	0.20	0.13	0.25	0.06	0.30	<0.05	0.49	<0.01
`Questions`	0.80	<0.01	0.80	<0.01	0.77	<0.01	0.80	<0.01	0.84	<0.01
Mean	0.58	<0.05	0.55	<0.05	0.51	<0.05	0.58	<0.01	0.58	<0.05

even more subtle than simple mutants in some cases. It is possible that this is caused by the solution classes only functioning correctly with their original student-written dependency classes, which I replaced for this experiment.

RQ3: Major and Pit's mutants achieve the highest probabilistic coupling for students' solutions; they are often more subtle than students' faults, and do not cause extra tests to fail. While I found that MutaGen can improve bidirectional coupling in **RQ2**, in isolation its mutants are often more complex than students' faults, causing additional test failures. I also find that coverage is weaker than mutation in terms of probabilistic coupling; students' faults are often subtle, and are not always detected merely by being executed by a test.

9.4.4 RQ4: Is mutation score analogous to real fault detection?

Table 9.3 shows the mean correlations between mutation score and the detection rate of students' faulty solutions. Figure 9.3 shows this correlation for each student's solution. I find that there is a positive, statistically significant correlation between mutation score and the detection rate of students' faulty solutions; 0.58 for all mutants. This supports my hypothesis

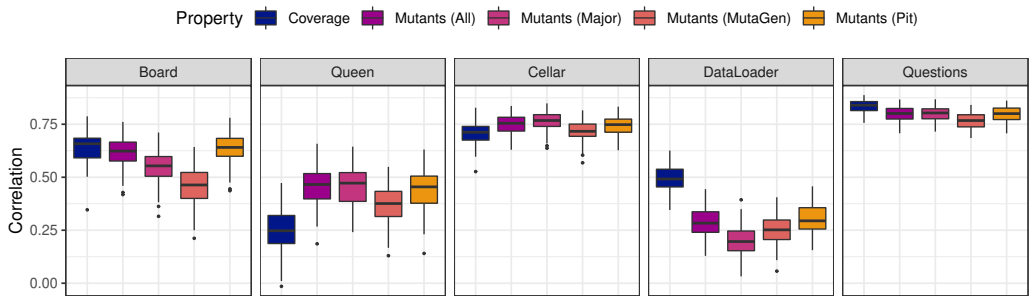


Figure 9.3: Correlations to detection rate of faulty students' solutions for each test suite, across 100 repetitions.

that improving the mutation score of a test suite improves the detection of students' faults.

The strength of this correlation varies between the subject classes. The strongest correlation is for `Questions` (0.80), while the weakest occurs for `DataLoader` (0.29). It is likely that this is due to the nature of these classes. `Questions` is a series of independent methods that perform calculations on a set of data, and return the result; mutants are particularly effective in simulating such calculation faults. In comparison, `DataLoader` primarily requires the implementation of a file parser; students tend to make more complex logical errors, fail to throw exceptions properly, or fundamentally misunderstand how the parser should operate. While I implemented several operators to address some of these issues in MutaGen, particularly exception management and branch ordering, the vast majority of mutants do not directly replicate such faults.

RQ4: There is a statistically significant positive correlation between mutation score and the detection rate of students' faulty solutions, of 0.58 on average. The magnitude of this correlation varies between the subject classes.

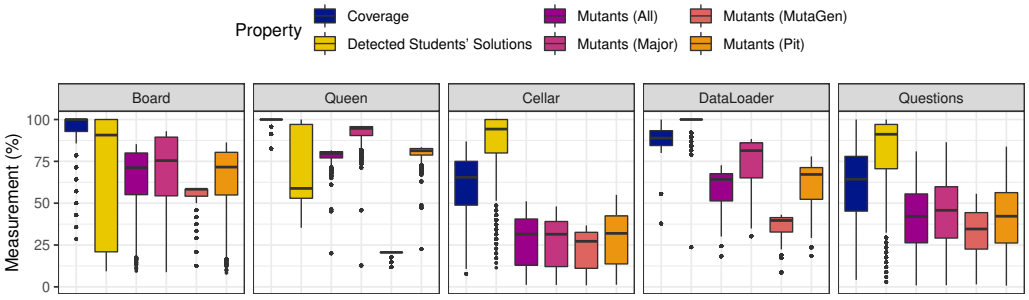


Figure 9.4: Observed property measurements for each test suite, across 100 repetitions.

9.4.5 RQ5: How do mutation score and code coverage compare in their effectiveness as adequacy metrics?

As shown in Table 9.3, the correlations for mutation score (with all mutants) and coverage are equal. This reveals that mutation score and coverage serve as similarly effective adequacy metrics for this dataset. This is reflected for `Board`, `Cellar`, and `Questions`; each of these subjects yield approximately the same correlations for mutation score and coverage. However, coverage does not have a statistically significant correlation for `Queen` ($p = 0.10$). The reason for this is revealed by Figure 9.4, which summarises the individual observations of each of the test suites. In `Queen`, there is a range of the proportion of students' solutions that the suites detect, but there is almost no change in coverage; the coverage of each suite is very high, 100% for most of the test suites. Since there is so little variance for coverage, no statistically correlation can be made. `DataLoader` also presents a divergence; mutation score has a considerably lower correlation than coverage (0.29 vs. 0.49). This is likely due to students' faults being fairly easy to detect, providing some affinity for coverage, while not aligning with mutants in their manifestation. For example, I found that many students' solutions did not correctly throw the expected exceptions when invalid data is parsed. While this is directly simulated by some mutants, the majority of mutants are simply unrelated.

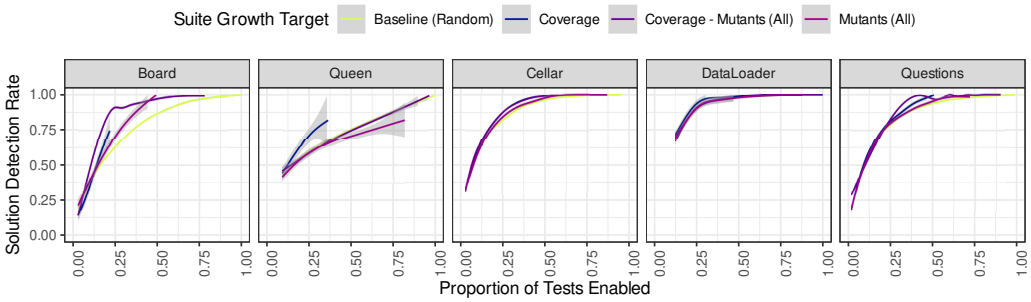


Figure 9.5: Performance of suites generated by growth strategies that target mutants and covered lines. The intervals marked in grey denote the margin of error.

Considering the limitations of only analysing the correlation of adequacy metrics [27], I also employ an analysis of suite growth, the results of which are shown in Figure 9.5. These results show that for three of the subject classes, `Cellar`, `DataLoader`, and `Questions`, the suites grown according to coverage and mutant detection perform similarly, with each strategy converging to maximum fault detection with approximately the same number of tests. Coverage focused suites do, however, appear to reach this peak fault detection slightly earlier than suites that are grown according to mutation score. `Board` and `Queen` exhibit different behaviour; coverage focused suites detect more faults with few tests, but at their maximum size they detect fewer solutions than the maximum for suites grown according to mutant detection. It is likely that these subject classes are easy to cover; since the suite growth ends once 100% coverage is reached, coverage based suites are smaller. Similarly, small coverage optimised suites may detect so many faults due to many students' solutions containing easily detected faults, which are revealed if they are merely executed. Other solutions may contain much harder to detect faults; some suites with 100% coverage cannot detect them, but suites with a high mutation score can.

I also combine both uncovered lines and unkilled mutants as a merged suite growth strategy; suites are grown to achieve 100% coverage first, then are grown to detect mutants once this is achieved. Suites that are grown according to this merged strategy perform the best for most of the subject classes; they

converge on 100% detection with fewer tests than the other strategies. This approach also detects more faulty solutions for `Queen`; each of the growth strategies are exhausted before reaching maximum suite adequacy when used independently, but not when they are employed together. Both of these observations suggest that it is beneficial to first use coverage to build a grading test suite, and to then use mutation testing to further improve the test suite.

RQ5: Mutation score and coverage are similarly correlated to the detection rate of students' faulty solutions. Independently, growing test suites to kill additional mutants or to detect more lines typically results in similar fault detection performance, though mutant focused growth does yield larger test suites that detect more faults in some cases. However, combining both coverage and mutants to grow test suites yields the most effective test suites.

9.4.6 RQ6: Which mutation tool produces mutants that best represent students' faults?

First, I consider the correlations of each tool's mutation score to the detection rate of faulty solutions, as presented in Table 9.3 and Figure 9.3. Major's mutants fail to achieve a statistically significant correlation for `DataLoader`, but achieve a similar correlation to the other mutation tools for the other subject classes. Major only generates mutants that replace values and operators, or delete statements; it is likely that such faults are not reflected in the students' faulty solutions for `DataLoader`. MutaGen's mutants have the lowest correlation overall. This is to be expected; I explicitly did not implement operators that are already implemented by existing tools. Pit yields the highest overall correlation. This is likely due in part to it generating the most mutants from the greatest number of operators (29); it is more likely that Pit's generated mutants will cause similar test failures to the students' solutions. Combining the mutants of every tool produces a similar correlation to that of using Pit's mutants alone.

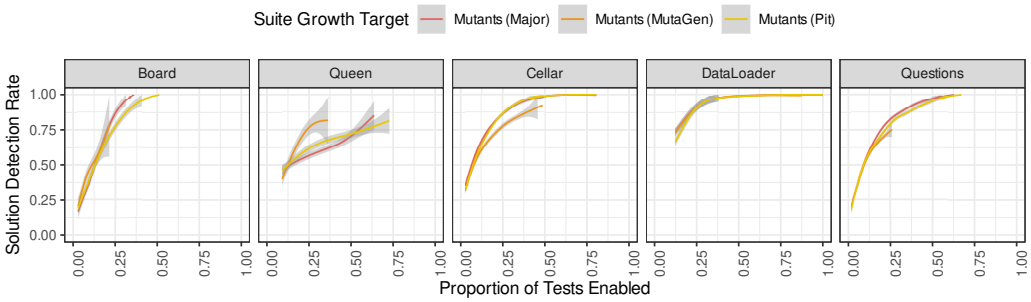


Figure 9.6: Performance of suites generated by growth strategies that target mutants generated by each mutation tool. The intervals marked in grey denote the margin of error.

As Chen et al. note, only comparing the correlations of adequacy criteria has considerable limitations [27]. Consequently, I also analyse the performance of suites grown according to mutants generated by each tool. Figure 9.6 shows the results of this analysis. I find that, despite its mutants' lower correlation to the detection rate of students' faulty solutions, Major slightly outperforms Pit; suites that are grown according to Major's mutants reach maximum fault detection with fewer tests. This suggests that it is sufficient to only use simpler mutation operators, such as those implemented by Major, rather than a wide range of operators that introduce more esoteric faults, such as those implemented by Pit's extended set of operators. This itself supports the presence of the coupling effect for students' solution programs; suites that detect simple mutants also detect more complex faults. I also find that for three of the subject classes, MutaGen clearly exhibits the weakest performance; suites that are grown until all of its mutants are detected fail to identify every faulty student's solution. This does not necessarily mean that MutaGen provides no benefit; for example, in `Queen`, suites that target MutaGen's mutants detect many students' solutions with few tests. Therefore, it is possible that combining MutaGen with other tools, as I originally intended, may offer a practical benefit to improving a grading test suite's adequacy.

In order to evaluate how using mutants from multiple tools benefits suite development, I repeat this suite growth analysis, comparing the use of different combinations of mutants as growth targets, as shown in Figure 9.7. Specifically,

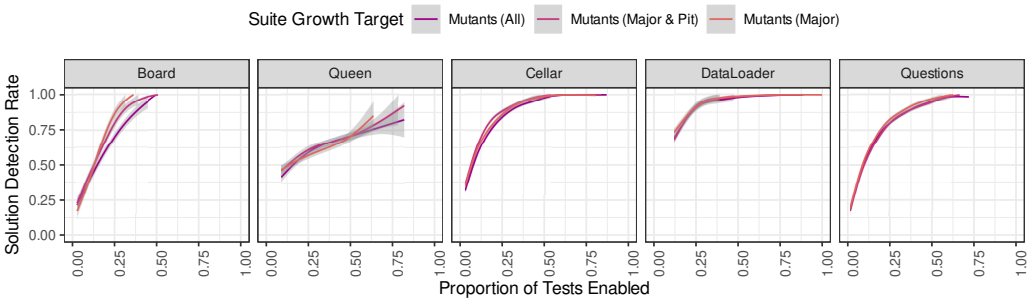


Figure 9.7: Performance of suites generated by growth strategies that target different combinations of mutants. I also include mutants generated by only Major, the best single tool, as a baseline to compare the combined sets against. The intervals marked in grey denote the margin of error.

I compare the use of every mutant to the use of only mutants generated by Major and Pit, in order to determine if adding MutaGen's mutants benefits fault detection. I find that including MutaGen's mutants offers no clear benefit for any of the subject classes, and is detrimental for `Board`, `Queen`, and `Cellar`; suites grown according to only Major and Pit's mutants either detect a higher maximum number of students' solutions, or reach maximum fault adequacy with fewer tests. While MutaGen can produce some mutants that directly replicate students' faults, some of the faults that it replicates are fairly uncommon within the dataset, and may not even be made by students for some subject classes. It is likely that the inclusion of these operators that replicate uncommon faults limits detection performance; including irrelevant mutants reduces the likelihood that a test which detects a relevant mutant is selected by the sampler.

I also compare combining mutants generated with Major and Pit against only using Major's mutants, which I previously found to be the most effective mutants from a single tool. This allows me to investigate if merging mutant sets improves fault detection. I find that adding Pit's mutants to Major's mutants only offers a considerable benefit for `Queen`, where it results in an improved maximum fault detection rate. For this subject class, it is likely that Pit's mutants accurately replicate some of the faults made by students, as shown by my bidirectional coupling analysis (Table 9.2), where Pit's mutants

are bidirectionally coupled to more solutions than those of any other tool. Including these mutants would make it more likely that my suite growth approach selects tests which detect such faults. However, for `Board`, including Pit's mutants increases the number of tests required to reach maximum fault adequacy; Pit's more complex operators may simply not reflect the mistakes that students make for this class. Consequently, I can conclude that the subject itself plays a role in the effectiveness of mutant adequacy, since a program's requirements may influence the mistakes that students make, and therefore, which operators best simulate these mistakes. Despite this, simpler mutants, such as those that I generated using Major, can sufficiently simulate students' faults in most cases.

This provides some supporting evidence for the *competent programmer hypothesis*, since simple mutants can effectively simulate most students' faults. Students may not always strictly be "competent programmers," but their mistakes are often simple enough that a set of mutants can simulate them. These simple mutants are applicable to evaluate the adequacy of grading test suites, and to guide their development.

RQ6: Despite their underwhelming correlation to fault detection, Major's mutants are typically the most capable of guiding a test suite to reveal more students' faulty solutions; simple mutants can sufficiently simulate students' mistakes. Combining these mutants with those of other tools improves fault detection in some cases, especially `Queen`, but not in general; it is likely not worth including more complex mutation operators unless they specifically target faults that students are especially likely to make for a programming task. Although some of its operators directly simulate some faults, MutaGen offers little benefit in improving the fault detection rate of a test suite.

9.5 Conclusion

In this chapter, I have conducted an empirical study to investigate the suitability of using mutants to simulate students' faults, and in turn their

applicability to improving grading test suites. Using the results of this study, I provide evidence that:

- The coupling effect holds for the vast majority of students' solution programs.
- My mutation tool, MutaGen, produces effective simulations of students' faults that other mutation tools cannot.
- Mutants can capture the subtleties of students' faults where line coverage does not.
- There is a positive correlation between the detection rates of mutants and students' faulty solutions.
- In isolation, using mutation testing and code coverage to guide test selection produces similarly effective test suites, but using coverage followed by mutation produces more effective test suites.
- Simple mutation operators, such as those implemented by Major, are generally sufficient to inform the development of a grading test suite; using more complex operators in addition to these yields diminishing returns.

Chapter 10

Conclusions & Future Work

10.1 Summary of Contributions

In this thesis, I have made eight key scientific contributions:

Contribution 4.1 *An open-source modular automated assessment tool that also enhances manual assessment.* I developed a modular automated assessment system, *Gradeer*, originally as a platform to gather execution data for my empirical studies. I later deployed Gradeer in an introductory Java programming module, where the module's lead and teaching assistants used the system to assess students' solution programs. This required the introduction of new features, including a feature to automatically execute a student's program and show its source code, to reduce the repetitive tasks involved with manual assessment.

Contribution 5.1 *Empirical evidence that different test suites can yield significantly varying grades for students' solution programs.* I conducted an empirical study using the grades generated by sampled test suites for students' solution programs. I found that the sampled test suites generated grades that

varied greatly for each solution, with a mean standard deviation of grades for each solution of ∼12.2%.

Contribution 5.2 *A statistical comparison of how various observable properties of test suites influence the grades that they generate.* I performed a relative importance analysis for several observable properties of sampled test suites and the change in grades that they produce for students' solution programs. I find that these properties have a significant impact on generated grades. In order of descending importance, these properties are:

- Detection rate of other students' faulty solutions,
- Size (the number of tests in a suite),
- Uniqueness (which increases as the lines of a program are covered by different tests to one another),
- Mutation score (the proportion of mutants that a suite kills),
- Coverage (the proportion of lines in the reference solution that a suite executes),
- Diversity (the likelihood that two tests cover different lines), and
- Density (the average number of lines covered by each test).

Contribution 6.1 *A qualitative analysis of students' programming mistakes.* I used a coding process to categorise students' programming mistakes that I observe in both of my datasets. I identified 30 mistake categories that impact the functionality of students' programs (i.e. could cause test failures), eight that prevent them from compiling, and 18 that violate style and quality conventions.

Contribution 7.1 *Definitions of new mutation operators to simulate students' programming mistakes.* I used the functionality mistake categories that I identified in Chapter 6 to derive new mutation operators, which would generate mutants to simulate these mistakes. Some of these mistake categories are already represented by operators that are implemented by existing mutation tools. Consequently, I only defined operators for the partially or completely unrepresented mistakes; I defined 17 new mutation operators.

Contribution 8.1 *A prototype mutation tool that implements my new mutation operators.* I implemented the mutation operators that I defined in Chapter 7 as a mutation tool, *MutaGen*. The tool manipulates the abstract syntax tree of a reference solution program's source code to generate mutants for each of my newly defined mutation operators.

Contribution 9.1 *Empirical evidence that the coupling effect holds for mutants and students' solution programs.* I performed an empirical study on the subject classes of my EndOfYear dataset and the mutants generated from their reference solutions, to investigate if the coupling effect holds for mutants and students' faulty solutions. I found that most students' faulty solutions are coupled to non-trivial mutants (i.e. mutants that do not fail every test). The remaining uncoupled faulty solutions fail every test, and are directly simulated by mutants generated by one of MutaGen's operators (*Replace Object Copy with Reference*). I also performed a probabilistic coupling analysis, which reveals how well any test goal (i.e. a covered line or a killed mutant) represents each detected fault in the students' solution programs. I found that there exists a mutant that achieves a maximum probabilistic coupling for the majority of students' faulty solutions, and that mutants outperform covered lines in this regard.

Contribution 9.2 *An empirical comparison of the effectiveness of code coverage and mutants generated by different tools in evaluating the adequacy of a grading test suite.* I used two empirical analysis approaches to evaluate

the effectiveness of mutants in simulating students' faults; a simple correlation analysis, and an evaluation of fault detection for suites grown according to an adequacy criteria. I found that there is a positive correlation between the detection of mutants and students' faulty solutions, and that coverage yields a similarly positive correlation to the detection of students' faulty solutions. This was reflected by my suite growth analysis, where I found that suites that I grew according to mutants and coverage perform similarly in their detection of students' faults. I also found that combining coverage and mutation—by first achieving maximum coverage, then maximising mutation score—yields test suites which are able to detect the most students' solutions that contain faults. Finally, I find that mutants generated by Major (which only includes simple mutation operators) provide a more accurate adequacy metric than the mutants generated by other mutation tools (which implement more complex mutation operators), and that including mutants from other tools fails to make a considerable improvement in this regard.

10.2 Suggestions for Tutors

From my findings, I am able to provide several suggestions for tutors aiming to develop or improve a grading test suite for a programming task. These suggestions can be applied using a reference solution which implements the specification of the programming assignment:

- First employ code coverage metrics, since they can quickly reveal where some aspects of the task's specification are not tested.
- Use mutation testing to guide the development of the test suite. Simple mutation operators (i.e. the default operators used by most mutation tools) are sufficient for this.
- Write tests that exercise the specification and reference solution in different ways. This is effectively measured by diagnosability metrics, such as uniqueness and diversity, though these may be difficult to

interpret.

10.3 Future Work

Throughout my research, I have identified several possible avenues for future work in improving test based automated assessment.

10.3.1 Further Exploration of Mutation Testing

There is a clear scope to further investigate how mutation testing and analysis can be applied to evaluating the adequacy of a grading test suite.

Evaluating Individual Mutation Operators By conducting an empirical study of how well mutants generated by individual operators simulate students' faults, the most effective mutation operators can be identified. This would allow for only the most effective set of operators to be used to generate mutants, which in turn would limit the number of generated mutants, reducing the costs associated with mutation testing and analysis.

Understanding Tutors' Use of Mutation Testing & Analysis It would also be beneficial to understand how tutors would apply mutation testing to the development of their grading test suites. For example, mutation testing does have some challenges; equivalent mutants must be manually identified before mutation analysis offers a benefit, and running mutants can take a considerable amount of time. It is possible for tutors to be overcome by these challenges; tutors may not consider mutation testing and analysis to be worth using due to them. Human studies and case studies would offer some insight into how tutors address such challenges, and how useful they find mutants to be. Performing a human study to investigate this would be a significant challenge; it would require the participation of tutors, who

often face time constraints that may be make them hesitant to participate in a human study. In addition, understanding an unfamiliar programming assignment's specification and writing tests to satisfy is incredibly time consuming, introducing practical limitations to running such a human study. It is likely more practical for a tutor who is interested in using mutation testing to perform a case study with one of their own programming assignments. Such a case study could investigate the challenges that they encounter, and how applying mutation analysis affects their test suites, such as in detecting students' faults, or in influencing the grades that it generates.

Higher Order Mutants Since some students' solutions contain multiple distinct faults, it is possible that combining multiple single mutants to produce higher order mutants (HOMs) may be beneficial. This may offer some advantage for mutation testing, since HOMs are less likely to be equivalent, and in some cases can be more subtle than first order mutants [148]; using HOMs may save a tutor's time. The performance of HOMs in simulating students' faulty solutions should be investigated.

10.3.2 Evaluating & Improving Fairness

Fairness Metric Defining a metric, or series of metrics, to evaluate the fairness of a grading test suite would be of a clear benefit to tutors; they can gain assurance that their test suites evaluate students' programs with as limited a degree of bias as possible. Such a metric should reveal both deficiencies in the test suite, and where possible faults (i.e. mutants) are detected unevenly. While coverage and mutation score fulfil the role of the former, diagnosability metrics, such as uniqueness and diversity may benefit the latter. Diagnosability metrics evaluate how achieved test goals relate to one another, such as which lines are covered the same way by multiple tests. I only used line coverage for the test goals in my analysis of diagnosability in Chapter 5; mutants can be used instead. For some programs, more mutants can be generated than there are lines to cover; using mutants to derive

diagnosability metrics would likely yield different results, which should be evaluated for how they reflect fairness.

Weight Generation It is possible to assign weights to individual tests to modify the impact that they have on grade generation. For example, Gradeer implements such functionality. A test suite that achieves 100% coverage and mutation score may still produce unfair grades if some types of faults, or some learning outcomes, are evaluated far more than others. Therefore, weighting tests can reduce unfairness; reducing the weights of tests that evaluate similar aspects of a task's specification can make their impact more similar to that of a single test that evaluates another aspect of the specification alone. Defining an accurate fairness metric would allow for the effectiveness of such weights to be predicted. Accordingly, such a fairness metric could be used to generate such weights; a search algorithm can be applied to determine which weights yield the highest fairness estimate.

Challenges Developing and applying such a fairness metric poses a particularly significant challenge; how should its effectiveness be evaluated? Existing grades cannot be used as a baseline; there is no guarantee that they are truly fair. One possible approach is to have several tutors manually grade a series of solution programs, and to use the mean grade for each solution as a benchmark "fair" grade. This is impractical, however, there is often no reason for multiple people to assess the same solutions, aside from doing so for a small sample of solutions to moderate grades.

10.3.3 Feedback Generation

Although my work has been primarily focused on the grading aspect of automated assessment, the generation of feedback is also incredibly important.

Mutants to Inform Feedback Messages Mutants may be able to assist tutors in defining feedback messages. For example, if a tutor wrote a test that only detects mutants generated by one operator, the change that this operator would perhaps describe the students' mistakes that the test detects. A tutor could add feedback associated with this test according to the operator. However, this is fairly unlikely to be practical; a tutor would be familiar with their programming task's specification already, and would know what aspects of the specification their tests evaluate.

Fault Localisation I have performed an initial investigation on the grading impact of diagnosability metrics, which were originally designed to estimate the effectiveness of applying fault localisation using a test suite. While I did not directly evaluate the application of fault localisation, it offers a promising approach for directly providing students with feedback. Fault localisation tools provide estimated locations of faults within a program that fails tests [169]. This output could serve as effective feedback for helping students to find, understand, and resolve their faults.

Bibliography

[1] Benjamin Clegg, Siobhán North, Phil McMinn, and Gordon Fraser. Simulating student mistakes to evaluate the fairness of automated grading. In *Proc. - 2019 IEEE/ACM 41st Int. Conf. Softw. Eng. Softw. Eng. Educ. Training, ICSE-SEET 2019*, pages 121–125. Institute of Electrical and Electronics Engineers Inc., may 2019.

[2] Benjamin S. Clegg, Phil McMinn, and Gordon Fraser. The Influence of Test Suite Properties on Automated Grading of Programming Exercises, nov 2020.

[3] Benjamin Simon Clegg, Phil McMinn, and Gordon Fraser. An Empirical Study to Determine if Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding. In *Proc. 52nd ACM Tech. Symp. Comput. Sci. Educ.*, pages 1055–1061, New York, NY, USA, mar 2021. ACM.

[4] Benjamin Clegg, Maria-Cruz Villa-Uriol, Phil McMinn, and Gordon Fraser. Gradeer: An Open-Source Modular Hybrid Grader. In *43rd Int. Conf. Softw. Eng. Softw. Eng. Educ. Train.*, pages 60–65. Institute of Electrical and Electronics Engineers (IEEE), may 2021.

[5] Benjamin Clegg, Gordon Fraser, and Phil McMinn. Diagnosability, Adequacy & Size: How Test Suites Impact Autograding. In *Proc. 55th Hawaii Int. Conf. Syst. Sci.* Hawaii International Conference on System Sciences, jan 2022.

[6] BCS Press Office. Record numbers choosing Computer Science degrees - new data reveals — BCS. [Online; accessed 2021-09-06] https://www.bcs.org/more/about-us/press-office/press-releases/record-numbers-choosing-computer-science-degrees-new-data-reveals/.

[7] Tingting Tong and Haizheng Li. Demand for MOOC - An Application of Big Data. *China Econ. Rev.*, 51(May):194–207, 2017.

[8] Sébastien Combéfis and Alexis Paques. Pythia reloaded: an intelligent unit testing-based code grader for education. In *Proc. 1st Int. Work. Code Hunt Work. Educ. Softw. Eng. - CHESE 2015*, pages 5–8, New York, New York, USA, 2015. ACM Press.

[9] Shuchi Grover. Toward A Framework for Formative Assessment of Conceptual Learning in K-12 Computer Science Classrooms. In *SIGCSE 2021*, volume 21. Virtual Event, 2021.

[10] Paul Ramsden. *Learning to Teach in Higher Education*. Routledge-Falmer, London and New York, 2002.

[11] Shuhaida Shuhidan, Margaret Hamilton, Daryl D ' Souza, and Daryl D'Souza. A taxonomic study of novice programming summative assessment. *Conf. Res. Pract. Inf. Technol. Ser.*, 95:147–156, 2009.

[12] Milena Vujosevic-Janicic, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. Software verification and graph similarity for automated evaluation of students' assignments. *Inf. Softw. Technol.*, 55(6):1004–1016, 2013.

[13] Stephen Edwards, Jaime Spacco, and David Hovemeyer. Can Industrial-Strength Static Analysis Be Used to Help Students Who Are Struggling to Complete Programming Activities? In *Proc. 52nd Hawaii Int. Conf. Syst. Sci.* Hawaii International Conference on System Sciences, 2019.

[14] Igor Solecki, João Porto, Nathalia da Cruz Alves, Christiane Gresse von Wangenheim, Jean Hauck, and Adriano Ferreti Borgatto. Automated Assessment of the Visual Design of Android Apps Developed with App

Inventor. In *Proc. 51st ACM Tech. Symp. Comput. Sci. Educ. (SIGCSE '20)*, pages 51–57, New York, NY, USA, feb 2020.

[15] Sébastien Combéfis and Arnaud Schils. Automatic programming error class identification with code plagiarism-based clustering. In *Proc. 2nd Int. Code Hunt Work. Educ. Softw. Eng. - CHESE 2016*, pages 1–6, New York, New York, USA, 2016. ACM Press.

[16] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, pages 465–480, 2018.

[17] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for MOOCs. In *SPLASH Companion 2016 - Companion Proc. 2016 ACM SIGPLAN Int. Conf. Syst. Program. Lang. Appl. Softw. Humanit.*, pages 39–40, New York, New York, USA, oct 2016. Association for Computing Machinery, Inc.

[18] David Insa and Josep Silva. Automatic assessment of Java code. *Comput. Lang. Syst. Struct.*, 53:59–72, 2018.

[19] Steffen Zschaler, Sam White, Kyle Hodgetts, and Martin Chapman. Nexus: a micro-service architecture for automated feedback and grading systems. [Online; accessed 2020-10-08] http://www.steffen-zschaler.de/publications/nexus_architecture.pdf, 2017.

[20] Draylson M. Souza, Katia R. Felizardo, and Ellen F. Barbosa. A systematic literature review of assessment tools for programming assignments. In *Proc. - 2016 IEEE 29th Conf. Softw. Eng. Educ. Training, CSEEandT 2016*, pages 147–156. IEEE, apr 2016.

[21] Maha Aziz, Heng Chi, Anant Tibrewal, Max Grossman, and Vivek Sarkar. Auto-grading for parallel programs. In *Proc. EduHPC 2015 Work. Educ. High-Performance Comput. - Held conjunction with SC 2015 Int. Conf. High Perform. Comput. Networking, Storage Anal.*, pages 1–8, New York, New York, USA, nov 2015. Association for Computing Machinery, Inc.

[22] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, Thu D. Nguyen, and Stephen Bar-Tos. Providing Meaningful Feedback for Autograding of Programming Assignments. In *SIGCSE 2018 - Proc. 49th ACM Tech. Symp. Comput. Sci. Educ.*, volume 2018-Janua, pages 278–283, New York, New York, USA, feb 2018. ACM Press.

[23] Neil C.C. C Brown and Amjad Altadmri. Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs. *ACM Trans. Comput. Educ.*, 17(2):1–21, may 2017.

[24] Hong Zhu, Patrick A.V. Hall, and John H.R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[25] Yucheng Zhang and Ali Mesbah. Assertions Are Strongly Correlated with Test Suite Effectiveness. In *Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015*, New York, New York, USA, 2015. ACM Press.

[26] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, sep 2011.

[27] Yiqun T Chen, Anita Tadakamalla, Michael D Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, René Just, Yiqun T Chen, Rahul Gopinath, Anita Tadakamalla, Michael D Ernst, Gordon Fraser, Paul Ammann, and René Just. Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. *35th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE '20)*, pages 284–296, 2020.

[28] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, 16-21-Nove:654–665, nov 2014.

[29] Lynne P. Baldwin and Robert D. Macredie. Beginners and programming: Insights from second language learning and teaching. *Educ. Inf. Technol.*, 4(2):167–179, 1999.

[30] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *ACM SIGCSE Bull.*, 37(3):14–18, sep 2005.

[31] Raymond Lister, William Fone, Robert McCartney, Otto Seppälä, Elizabeth S. Adams, John Hamer, Jan Erik Moström, Beth Simon, Sue Fitzgerald, Morten Lindholm, Kate Sanders, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *SIGCSE Bull. (Association Comput. Mach. Spec. Interes. Gr. Comput. Sci. Educ.*, volume 36, pages 119–150. ACM PUB27 New York, NY, USA, dec 2004.

[32] Geetha Kanaparan, Rowena Cullen, and David Mason. Effect of Self-efficacy and Emotional Engagement on Introductory Programming Students. *Australas. J. Inf. Syst.*, 23:1–21, jul 2017.

[33] Jessica Q. Dawson, Alice Campbell, Meghan Allen, and Anasazi Valair. Designing an introductory programming course to improve non-majors' experiences. In *SIGCSE 2018 - Proc. 49th ACM Tech. Symp. Comput. Sci. Educ.*, volume 2018-Janua, pages 26–31. Association for Computing Machinery, Inc, feb 2018.

[34] Soonhye Park and J. Steve Oliver. Revisiting the conceptualisation of pedagogical content knowledge (PCK): PCK as a conceptual tool to understand teachers as professionals. *Res. Sci. Educ.*, 38(3):261–284, jun 2008.

[35] L. Ma, J. Ferguson, M. Roper, and M. Wood. Investigating and improving the models of programming concepts held by novice programmers. *Comput. Sci. Educ.*, 21(1):57–80, mar 2011.

[36] David Starr-Glass. Moving From Passive to Active Blended Learning: An Adopter's Experience. *Cases Act. Blended Learn. High. Educ.*, pages 23–42, 2021.

[37] Mary Forehand. Bloom's Taxonomy From Emerging Perspectives on Learning, Teaching and Technology Jump to: navigation, search. *Emerg. Perspect. Learn. Teach. Technol.*, 2005.

[38] Lorin W Anderson. Rethinking Bloom's Taxonomy: Implications for Testing and Assessment. Technical report, University of South Carolina, 1999.

[39] Errol Thompson, Andrew Luxton-Reilly, Jacqueline L Whalley, Minjie Hu, and Phil Robbins. Bloom's Taxonomy for CS Assessment. In *Proc. ofthe tenth Conf. Australas. Comput. Educ.*, pages 155–161, 2008.

[40] Colin G. Johnson and Ursula Fuller. Is Bloom's taxonomy appropriate for computer science? In *Balt. Sea '06 Proc. 6th Balt. Sea Conf. Comput. Educ. Res. Koli Call. 2006*, volume 276, pages 120–123, 2006.

[41] Ursula Fuller, Colin G. Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L. Lewis, Donna McGee Thompson, Charles Riedesel, and Errol Thompson. Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bull.*, 39(4):152–170, dec 2007.

[42] Russel E. Bruhn and Philip J. Burton. An approach to teaching Java using computers. *ACM SIGCSE Bull.*, 35(4):94–99, dec 2003.

[43] Marko Hassinen and Hannu Mäyrä. Learning Programming by Programming: a Case Study. In *Proc. 6th Balt. Sea Conf. Comput. Educ. Res. Koli Call. 2006 - Balt. Sea '06*, New York, New York, USA, 2006. ACM Press.

[44] Martinha Piteira and Carlos Costa. Learning Computer Programming: Study of difficulties in learning programming. In *Proc. 2013 Int. Conf. Inf. Syst. Des. Commun. - ISDOC '13*, New York, New York, USA, 2013. ACM Press.

[45] John. B. Biggs and Kevin. F. Collis. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome).* Academic Press, New York, 1982.

[46] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. Not seeing the forest for the trees. In *Proc. 11th*

Annu. SIGCSE Conf. Innov. Technol. Comput. Sci. Educ. - ITICSE '06, volume 38, page 118, New York, New York, USA, 2006. ACM Press.

[47] Judy Sheard, Beth Simon, Angela Carbone, Errol Thompson, Raymond Lister, and Jacqueline L. Whalley. Going SOLO to assess novice programmers. In *Proc. Conf. Integr. Technol. into Comput. Sci. Educ. ITiCSE*, pages 209–213, 2008.

[48] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. A study of code design skills in novice programmers using the SOLO taxonomy. In *ICER 2016 - Proc. 2016 ACM Conf. Int. Comput. Educ. Res.*, pages 251–259, New York, NY, USA, aug 2016. Association for Computing Machinery, Inc.

[49] Tony Clear, Jacqueline L Whalley, Raymond Lister, Angela Carbone, Angela Carbone@infotech, Minjie Hu, Judy Sheard, Judy Sheard@infotech, Beth Simon, and Errol Thompson. Reliably Classifying Novice Programmer Exam Responses using the SOLO Taxonomy. In *21st Annu. Conf. Natl. Advis. Comm. Comput. Qualif.*, Auckland, New Zealand, jul 2008.

[50] Carmen Fernandez. Knowledge Base for Teaching and Pedagogical Content Knowledge (PCK): Some Useful Models and Implications for Teachers' Training. *Probl. Educ. 21st Century*, 60, 2014.

[51] Lee S. Shulman. Those Who Understand: Knowledge Growth in Teaching. *Educ. Res.*, 15(2):4–14, jul 1986.

[52] Malte Buchholz, Mara Saeli, and Carsten Schulte. PCK and Reflection in Computer Science Teacher Education. In *Proc. 8th Work. Prim. Second. Comput. Educ. - WiPSE '13*, New York, New York, USA, 2013. ACM Press.

[53] Mara Saeli. *Teaching Programming for Secondary School: a Pedagogical Content Knowledge Based Approach*. PhD thesis, 2012.

[54] Charles R Graham. Theoretical considerations for understanding technological pedagogical content knowledge (TPACK). 2011.

[55] Spyridon Doukakis, Alexandra Psaltidou, Athena Stavraki, Nikos Adamopoulos, Panagiotis Tsiotakis, and Stathis Stergou. Measuring the technological pedagogical content knowledge (TPACK) of in-service teachers of computer science who teach algorithms and programming in upper secondary education. *Readings Technol.*, may 2021.

[56] Handan Atun and Ertuğrul Usta. The effects of programming education planned with TPACK framework on learning outcomes. *Particip. Educ. Res.*, 6(2):26–36, dec 2019.

[57] Joachim Breitner, Martin Hecker, and Gregor Snelting. Der Grader Praktomat. *Autom. Bewertung der Program.*, 2017.

[58] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. Evaluating a new exam question: Parsons problems. In *ICER'08 - Proc. ACM Work. Int. Comput. Educ. Res.*, pages 113–124, New York, New York, USA, 2008. ACM Press.

[59] Fabienne M. Van der Kleij, Remco C.W. Feskens, and Theo J.H.M. Eggen. Effects of Feedback in a Computer-Based Learning Environment on Students' Learning Outcomes: A Meta-Analysis. *Rev. Educ. Res.*, 85(4):475–511, dec 2015.

[60] David Nicol and Debra MacFarlane-Dick. Formative assessment and selfregulated learning: A model and seven principles of good feedback practice. *Stud. High. Educ.*, 31(2):199–218, apr 2006.

[61] B S Bloom. Some theoretical issues relating to educational evaluation. *Educ. Eval. new roles, new means 63rd Yearb. Natl. Soc. Study Educ.*, (69):26–50, 1969.

[62] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proc. 10th Koli Call. Int. Conf. Comput. Educ. Res. - Koli Call. '10*, pages 86–93, New York, New York, USA, 2010. ACM Press.

[63] Neil C.C. Brown and Amjad Altadmri. Investigating novice programming mistakes: Educator beliefs vs student data. *ICER 2014 - Proc. 10th Annu. Int. Conf. Int. Comput. Educ. Res.*, pages 43–50, 2014.

[64] Stephan Krusche and Andreas Seitz. ArTEMiS - An Automatic Assessment Management System for Interactive Learning. In *Proc. 49th ACM Tech. Symp. Comput. Sci. Educ. - SIGCSE '18*, volume 2018-Janua, pages 284–289, New York, New York, USA, feb 2018. ACM Press.

[65] Ibrahim Albluwi. A Closer Look at the Differences between Graders in Introductory Computer Science Exams. *IEEE Trans. Educ.*, 61(3):253–260, aug 2018.

[66] Vreda Pieterse and Janet Liebenberg. Automatic vs manual assessment of programming tasks. In *Proc. 17th Koli Call. Conf. Comput. Educ. Res. - Koli Call. '17*, pages 193–194, New York, New York, USA, nov 2017. Association for Computing Machinery.

[67] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, 48(6):15–26, jun 2013.

[68] Brent Martin and Antonija Mitrovic. Tailoring Feedback by Correcting Student Answers. *ITS*, pages 383–392, 2000.

[69] Hamza Manzoor, Amit Naik, Clifford A. Shaffer, Chris North, and Stephen H. Edwards. Auto-grading jupyter notebooks. In *Proc. 51st ACM Tech. Symp. Comput. Sci. Educ. (SIGCSE '20)*, pages 1139–1144, New York, NY, USA, feb 2020. Association for Computing Machinery.

[70] Victor J. Marin, Tobin Pereira, Srinivas Sridharan, and Carlos R. Rivero. Automated personalized feedback in introductory Java programming MOOCs. *Proc. - Int. Conf. Data Eng.*, pages 1259–1270, 2017.

[71] Raymond Pettit, John Homer, Roger Gee, Adam Starbuck, and Susan Mengel. An empirical study of iterative improvement in programming assignments. In *SIGCSE 2015 - Proc. 46th ACM Tech. Symp. Comput.*

Sci. Educ., pages 410–415. Association for Computing Machinery, Inc, feb 2015.

[72] Umair Z Ahmed and Jooyong Yi. Verifix: Verified Repair of Programming Assignments; Verifix: Verified Repair of Programming Assignments. Technical report, arXiv preprint arXiv:2106.16199, 2021.

[73] Brent Martin and Antonija Mitrovic. Automatic problem generation in constraint-based tutors. In *Intell. Tutoring Syst. 6th Int. Conf. ITS 2002, Biarritz, Fr. San Sebastian, Spain, June 2-7, 2002. Proc.*, pages 33–45, 2002.

[74] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for MOOCs (Extended). *arXiv*, 2016.

[75] Randy Goebel, Ajay Chander, Katharina Holzinger, Freddy Lecue, Zeynep Akata, Simone Stumpf, Peter Kieseberg, and Andreas Holzinger. Explainable AI: The New 42? *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 11015 LNCS:295–303, aug 2018.

[76] Kirsti M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Comput. Sci. Educ.*, 15(2):83–102, jun 2005.

[77] Christopher Douce, David Livingstone, and James Orwell. Automatic Test-Based Assessment of Programming: A Review. *ACM J. Educ. Resour. Comput.*, 5(3):4, sep 2005.

[78] Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, 1960.

[79] Julio C Caiza and Jose Maria del Alamo Ramiro. Programming Assignments Automatic Grading: Review of Tools and Implementations. *7th Int. Technol. Educ. Dev. Conf.*, pages 5691–5700, 2013.

[80] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. Who tests the testers?: Avoiding the perils of automated testing. In *ICER 2018 - Proc. 2018 ACM Conf. Int. Comput. Educ. Res.*, pages 51–59. Association for Computing Machinery, Inc, aug 2018.

[81] Stephen H. Edwards and Manuel A. Pérez-Quiñones. Web-CAT: Automatically grading programming assignments. In *Proc. Conf. Integr. Technol. into Comput. Sci. Educ. ITiCSE*, page 328, New York, New York, USA, 2008. ACM Press.

[82] Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, and Athanasios Tsintsifas. The CourseMarker CBA System: Improvements over Ceilidh. *Educ. Inf. Technol.*, 8:287–304, 2003.

[83] Anne Münzner, Nadja Bruckmoser, and Alexander Meschtscherjakov. Can I Code? User Experience of an Assessment Platform for Programming Assignments. 91:18:1–18:0, 2021.

[84] Yoonsik Cheon and Gary T Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. (November 2001), 2002.

[85] Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. Testing of mobile applications in the wild: A large-scale empirical study on android apps. In *IEEE Int. Conf. Progr. Compr.*, pages 296–307, New York, NY, USA, jul 2020. IEEE Computer Society.

[86] Peter M. Chen. An automated feedback system for computer organization projects. *IEEE Trans. Educ.*, 47(2):232–240, may 2004.

[87] Stephen H. Edwards. Teaching Software Testing: Automatic Grading Meets Test-first Coding. In *Proc. Conf. Object-Oriented Program. Syst. Lang. Appl. OOPSLA*, pages 318–319, New York, New York, USA, 2003. ACM Press.

[88] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises

- Extended Version. *Annu. Conf. Innov. Technol. Comput. Sci. Educ. ITiCSE*, 11-13-July(March):41–46, 2016.

[89] Susilo Veri Yulianto and Inggriani Liem. Automatic grader for programming assignment using source code analyzer. *Proc. 2014 Int. Conf. Data Softw. Eng. ICODSE 2014*, pages 0–3, 2014.

[90] Checkstyle. Checkstyle. [Online; accessed 2020-10-16] https://checkstyle.sourceforge.io/.

[91] Debora Weber-wulff, Katrin Köhler, and Christopher Möller. Collusion Test 2012 Collusion Detection System Test Report 2012. pages 1–15, 2012.

[92] John B. (John Burville) Biggs. *Teaching for quality learning at university : what the student does*. 2011.

[93] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull. (Association Comput. Mach. Spec. Interes. Gr. Comput. Sci. Educ.*, pages 153–156, 2003.

[94] Ella Albrecht and Jens Grabowski. Sometimes it's just sloppiness studying students' programming errors and misconceptions. In *Proc. 51st ACM Tech. Symp. Comput. Sci. Educ. (SIGCSE '20)*, pages 340–345, New York, NY, USA, feb 2020. Association for Computing Machinery.

[95] Davin McCall and Michael Kölling. Meaningful categorisation of novice programmer errors. In *Proc. - Front. Educ. Conf. FIE*, volume 2014. Institute of Electrical and Electronics Engineers Inc., feb 2014.

[96] Davin McCall and Michael Kölling. Meaningful Categorisation of Novice Programmer Errors - data, 2014.

[97] Google Java Style Guide. [Online; accessed 2021-09-23] https://google.github.io/styleguide/javaguide.html.

[98] Abe Leite and Saúl A Blanco. Effects of Human vs. Automatic Feedback on Students' Understanding of AI Concepts and Programming Style.

In *Proc. 51st ACM Tech. Symp. Comput. Sci. Educ.*, volume 20, pages 44–50, New York, NY, USA, feb 2020. Association for Computing Machinery.

[99] Paul W. Oman and Curtis R. Cook. A Programming Style Taxonomy, jul 1991.

[100] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B Rowe, and Nasser Giacaman. Understanding Semantic Style by Analysing Student Code. In *ACE '18 Proc. 20th Australas. Comput. Educ. Conf.*, 2018.

[101] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Code Quality Issues in Student Programs. In *Proc. 2017 ACM Conf. Innov. Technol. Comput. Sci. Educ. - ITiCSE '17*, pages 110–115, New York, New York, USA, 2017. ACM Press.

[102] PMD. PMD. [Online; accessed 2020-10-16] https://pmd.github.io/.

[103] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, pages 245–256, 2004.

[104] IEEE Power Engineering Society Software Engineering Standards Subcommittee. IEEE Standard for Software Unit Testing. Technical report, IEEE-SA Standards Board, 1986.

[105] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. *Proc. - Conf. Softw. Maintenance, ICSM 1992*, pages 299–308, 1992.

[106] Gregg Rothermel, Roland H. Untcn, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[107] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code Coverage for Suite Evaluation by Developers. In *Proc. 36th Int. Conf. Softw. Eng.*, pages 72–82, New York, NY, USA, may 2014. ACM.

[108] Akbar Siami Namin and James H Andrews. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proc. eighteenth Int. Symp. Softw. Test. Anal. - ISSTA '09*, New York, New York, USA, 2009. ACM Press.

[109] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. PathAFL: Path-Coverage Assisted Fuzzing. In *Proc. 15th ACM Asia Conf. Comput. Commun. Secur. ASIA CCS 2020*, pages 598–609. Association for Computing Machinery, Inc, oct 2020.

[110] Hiralal Agrawal, Richard A. DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E.W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. Design of Mutant Operators for the C Programming Language. *Des. Mutant Oper. C Program. Lang.*, page 74, 1989.

[111] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2009*, pages 220–229, 2009.

[112] Allen T. Acree, Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Mutation Analysis. Technical report, Georgia Institute of Technology, sep 1979.

[113] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer (Long. Beach. Calif).*, 11(4):34–41, 1978.

[114] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, pages 189–200. IEEE Computer Society, dec 2014.

[115] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. - 27th Int. Conf. Softw. Eng. ICSE05*, pages 402–411, New York, New York, USA, 2005. ACM Press.

[116] M. Delamaro and J. Maldonado. Proteum - A Tool for the Assessment of Test Adequacy for C Programs User's guide. In *Proc. Conf. Performability Comput. Syst.*, 1996.

[117] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *2011 Int. Symp. Softw. Test. Anal. ISSTA 2011 - Proc.*, pages 342–352, New York, New York, USA, 2011. ACM Press.

[118] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. Memory mutation testing. *Inf. Softw. Technol.*, 81:97–111, 2016.

[119] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are Mutation Scores Correlated with Real Fault Detection? In *2018 ACM/IEEE 40th Int. Conf. Softw. Eng. Are*, 2018.

[120] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. Softw. Eng.*, 40(1):23–42, 2014.

[121] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 3103, pages 1338–1349. Springer Verlag, 2004.

[122] Douglas Baldwin and Frederick Sayward. Heuristics for Determining Equivalence of Program Mutations. Technical report, Research Report 276, Department of Computer Science, Yale University, 1979.

[123] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test. Verif. Reliab.*, 7(3):165–192, 1997.

[124] Rob Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Softw. Test. Verif. Reliab.*, 9(4):233–262, dec 1999.

[125] David Schuler and Andreas Zeller. Covering and Uncovering Equivalent Mutants. *Softw. Testing, Verif. Reliab.*, 23(5):353–374, aug 2013.

[126] Marinos Kintis, Mike Papadakis, and Nicos Malevris. Employing second-order mutation for isolating first-order equivalent mutants. *Softw. Testing, Verif. Reliab.*, 25(5-7):508–535, aug 2015.

[127] Mike Papadakis, Marcio Delamaro, and Yves Le Traon. Mitigating the effects of equivalent mutants with mutant classification strategies. *Sci. Comput. Program.*, 95(P3):298–319, dec 2014.

[128] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. In *Proc. - 2016 IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2016*, pages 142–151. Institute of Electrical and Electronics Engineers Inc., aug 2016.

[129] Bernhard J.M. Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2009*, pages 192–199, 2009.

[130] René Just, Michael D. Ernst, and Gordon Fraser. Using State Infection Conditions to Detect Equivalent Mutants and Speed up Mutation Analysis. mar 2013.

[131] Timothy A. Budd, Richard J. Lipton, Richard DeMillo, and Frederick Sayward. The design of a prototype mutation system for program testing. *Manag. Requir. Knowledge, Int. Work.*, pages 623–623, dec 1978.

[132] A. Jefferson Offutt and K. N. King. A Fortran 77 interpreter for mutation analysis. In *Pap. Symp. Interpret. Interpret. Tech. SIGPLAN 1987*, pages 177–188, New York, New York, USA, jul 1987. Association for Computing Machinery, Inc.

[133] K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Softw. Pract. Exp.*, 21(7):685–718, jul 1991.

[134] A Jefferson Offutt, Jeff Voas, and Jeff Payne. Mutation Operators for Ada. Technical report, George Mason University, 1996.

[135] Sunwoo Kim, John A Clark, and John A McDermid. The Rigorous Generation of Java Mutation Operators Using HAZOP. Technical report, University of York, 1999.

[136] Yu-Seung Ma, Yong-Rae Kwon, Jeff Offutt, Yu Seung Ma, Yong Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, volume 2002-Janua, pages 352–363. IEEE Computer Society, 2002.

[137] Yu Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: An automated class mutation system. *Softw. Testing, Verif. Reliab.*, 15(2):97–133, jun 2005.

[138] Diego Rodríguez-Baquero and Mario Linares-Vásquez. Mutode: Generic JavaScript and node.js mutation testing tool. In *ISSTA 2018 - Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, volume 18, pages 372–375, New York, NY, USA, jul 2018. Association for Computing Machinery, Inc.

[139] Shazly Ahmed, Islam A.T.F. Taj-Eddin, and Manal A. Ismail. MuHyb: A Proposed Mutation Testing Tool for Hybrid Mobile Applications. In *ACM Int. Conf. Proceeding Ser.*, pages 67–72, New York, NY, USA, nov 2020. Association for Computing Machinery.

[140] Mutpy. MutPy: a mutation testing tool for Python 3.x. [Online; accessed 2021-09-23] https://github.com/mutpy/mutpy.

[141] Stryker Team. Stryker Mutator. [Online; accesed 2021-09-23] https://stryker-mutator.io/#.

[142] René Just. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. *2014 Int. Symp. Softw. Test. Anal. ISSTA 2014 - Proc.*, pages 433–436, 2014.

[143] René Just. *The Major Mutation Framework (Documentation).* http://mutation-testing.org/doc/major.pdf, 2018.

[144] Henry Coles. PIT - Mutation operators. [Online; accessed 2021-09-06] http://pitest.org/quickstart/mutators/.

[145] Urs Metz. Naked receiver mutator by UrsMetz · Pull Request #218 · hcoles/pitest · GitHub. [Online; accessed 2021-09-22] https://github.com/hcoles/pitest/pull/218, 2015.

[146] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A practical mutation testing tool for Java (Demo). In *ISSTA 2016 - Proc. 25th Int. Symp. Softw. Test. Anal.*, pages 449–452, New York, NY, USA, jul 2016. Association for Computing Machinery, Inc.

[147] Henry Coles. PIT Mutation Testing. [Online; accessed 2021-02-11] https://pitest.org/.

[148] Yue Jia and Mark Harman. Higher Order Mutation Testing. *Inf. Softw. Technol.*, 51(10):1379–1393, 2009.

[149] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. Subtle higher order mutants. *Inf. Softw. Technol.*, 81:3–18, jan 2017.

[150] Quang Vu Nguyen and Duong Thu Hang Pham. Is Higher Order Mutant Harder to Kill Than First Order Mutant? An Experimental Study. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 10751 LNAI, pages 664–673. Springer Verlag, 2018.

[151] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. *Proc. ACM Int. Conf. Companion Object Oriented Program. Syst. Lang. Appl. Companion, SPLASH '10*, pages 153–160, 2010.

[152] Zalia Shams and Stephen H. Edwards. Toward practical mutation analysis for evaluating the quality of student-written software tests. In

ICER 2013 - Proc. 2013 ACM Conf. Int. Comput. Educ. Res., pages 53–58, New York, New York, USA, 2013. ACM Press.

[153] Stephen H. Edwards and Zalia Shams. Comparing test quality measures for assessing student-written tests. In *36th Int. Conf. Softw. Eng. ICSE Companion 2014 - Proc.*, pages 354–363, New York, New York, USA, 2014. Association for Computing Machinery.

[154] Ben H. Smith and Laurie Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empir. Softw. Eng.*, 14(3):341–369, jun 2009.

[155] Kunal Taneja and Tao Xie. DiffGen: Automated regression unit-test generation. *ASE 2008 - 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng. Proc.*, pages 407–410, 2008.

[156] Anthony J H Simons. JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Autom. Softw. Eng.*, 14(4):369–418, oct 2007.

[157] Joe W. Duran and Simeon C. Ntafos. An Evaluation of Random Testing. *IEEE Trans. Softw. Eng.*, SE-10(4):438–444, jul 1984.

[158] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. *Proc. Conf. Object-Oriented Program. Syst. Lang. Appl. OOPSLA*, pages 815–816, 2007.

[159] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. *Proc. - Int. Conf. Softw. Eng.*, pages 75–84, 2007.

[160] Phil McMinn. Search-based software test data generation: A survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, jun 2004.

[161] Konstantinos Liaskos and Marc Roper. Automatic Test-Data Generation: An Immunological Approach. In *Test. Acad. Ind. Conf. Pract. Res. Tech. - Mutat. (TAICPART-MUTATION 2007)*, pages 77–81. IEEE, sep 2007.

[162] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. *SIGSOFT/FSE 2011 - Proc. 19th ACM SIGSOFT Symp. Found. Softw. Eng.*, pages 416–419, 2011.

[163] Gordon Fraser and Andrea Arcuri. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):1–42, dec 2014.

[164] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proc. - 2017 IEEE/ACM 39th Int. Conf. Softw. Eng. Softw. Eng. Pract. Track, ICSE-SEIP 2017*, pages 263–272. Institute of Electrical and Electronics Engineers Inc., jun 2017.

[165] Spencer Pearson, Jose Campos, Rene Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and Improving Fault Localization. *Proc. - 2017 IEEE/ACM 39th Int. Conf. Softw. Eng. ICSE 2017*, pages 609–620, may 2017.

[166] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, nov 2009.

[167] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Softw. Testing, Verif. Reliab.*, 10(3):171–194, sep 2000.

[168] Thomas Reps, Thomas Ball, Manuvir Das, and James Lams. The use of program profiling for software maintenance with applications to the year 2000 problem. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 1301:432–449, 1997.

[169] Rui Abreu, Peter Zoeteweij, and Arjan J.C. Van Gemund. Spectrum-based multiple fault localization. *ASE2009 - 24th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pages 88–99, 2009.

[170] Rui Abreu, Peter Zoeteweij, and Arjan Van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim Int. Symp. Dependable Comput.*, pages 39–46. IEEE, 2006.

[171] Akira Ochiai. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bull. Japanese Soc. Sci. Fish.*, 22:526–530, 1957.

[172] Carlos Gouveia, José Campos, and Rui Abreu. Using HTML5 visualizations in software fault localization. *2013 1st IEEE Work. Conf. Softw. Vis. - Proc. Viss. 2013*, 2013.

[173] Alexandre Perez, Rui Abreu, and Arie Van Deursen. A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches. In *Proc. - 2017 IEEE/ACM 39th Int. Conf. Softw. Eng. ICSE 2017*, pages 654–664. Institute of Electrical and Electronics Engineers Inc., jul 2017.

[174] Alberto Gonzalez-Sanchez, Hans Gerhard Gross, and Arjan J.C. Van Gemund. Modeling the diagnostic efficiency of regression test suites. In *Proc. - 4th IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2011*, pages 634–643, 2011.

[175] Lou Jost. Entropy and diversity. *Oikos*, 113(2):363–375, may 2006.

[176] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proc. - Int. Conf. Softw. Eng.*, volume 2006, pages 82–91, New York, New York, USA, 2006. IEEE Computer Society.

[177] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the Mutants: Mutating faulty programs for fault localization. *Proc. - IEEE 7th Int. Conf. Softw. Testing, Verif. Validation, ICST 2014*, pages 153–162, 2014.

[178] Mike Papadakis and Yves Le Traon. Metallaxis-FL: Mutation-based fault localization. *Softw. Test. Verif. Reliab.*, 25(5-7):605–628, aug 2015.

[179] Paul Denny, James Prather, and Brett A Becker. Error Message Readability and Novice Debugging Performance. In *Proc. 2020 ACM Conf. Innov. Technol. Comput. Sci. Educ.*, volume 7, New York, NY, USA, 2020. ACM.

[180] Marc R. Hoffmann, Evgeny Mandrikov, and Mirko Friedenhagen. EclEmma - JaCoCo Java Code Coverage Library. [Online; accessed 2021-09-06] https://www.jacoco.org/jacoco/.

[181] Josef Cacek. intoolswetrust/jd-cli: Command line Java Decompiler. [Online; accessed 2021-09-06] https://github.com/intoolswetrust/jd-cli, 2021.

[182] Steffen Zschaler, Sam White, Kyle Hodgetts, and Martin Chapman. Modularity for Automated Assessment: A Design-Space Exploration. In *Softw. Eng. 18*, pages 57–61, 2018.

[183] Stephen H. Edwards. What is Web-CAT? - Web-CAT. [Online; accessed 2020-10-15] http://web-cat.org/projects/Web-CAT/WhatIsWebCat.html.

[184] Benjamin Simon Clegg. Gradeer Repository. [Online; accessed 2020-10-18] https://github.com/ben-clegg/gradeer.

[185] The Apache Software Foundation. Apache Ant. [Online; accessed 2020-10-16] https://ant.apache.org/.

[186] Nickolas Falkner, Rebecca Vivian, David Piper, and Katrina Falkner. Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units. *SIGCSE 2014 - Proc. 45th ACM Tech. Symp. Comput. Sci. Educ.*, pages 9–14, 2014.

[187] Kyle Dewey, Phill Conrad, Michelle Craig, and Elena Morozova. Evaluating Test Suite Effectiveness and Assessing Student Code via Constraint Logic Programming. *ITiCSE 2017*, 6:pages, 2017.

[188] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *Proc. - Int. Conf. Softw. Eng.*, pages 60–71. IEEE Computer Society, 2003.

[189] Ulrike Grömping. Relative Importance for Linear Regression in R: The Package relaimpo. *JSS J. Stat. Softw.*, 17(1):1–27, 2006.

[190] R. H. Lindeman, P. F. Merenda, and R. Z. Gold. Introduction to bivariate and multivariate analysis. 1980.

[191] Jeff W. Johnson and James M. Lebreton. History and Use of Relative Importance Indices in Organizational Research:. *https://doi.org/10.1177/1094428104266510*, 7(3):238–257, jun 2004.

[192] Sara Mernissi Arifi, Ismail Nait Abdellah, Azeddine Zahi, and Rachid Benabbou. Automatic program assessment using static and dynamic analysis. In *2015 Third World Conf. Complex Syst.*, pages 1–6. IEEE, nov 2015.

[193] Benjamin Simon Clegg. ben-clegg/mutagen: A Java mutation analysis tool for simulating the mistakes of beginner programmers. [Online; accessed 2021-09-06] https://github.com/ben-clegg/mutagen.

[194] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *JavaParser: Visited Analyse, transform and generate your Java code base.* Leanpub, 2017.

[195] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, dec 1965.