ABSTRACT

PRESLER-MARSHALL, KAI. Supporting Computer Science Education Through Automation and Surveys. (Under the direction of Sarah Heckman and Kathryn Stolee.)

Software engineering is a growing field, with ever-increasing demand for capable engineers who can design, implement, and test the software that is needed for the modern world. With this increasing demand for software engineers, there is a corresponding increase in the demand placed on computer science programs that graduate these engineers. However, the increase in undergraduate enrollment in computer science programs has generally outpaced the increase in instructors. Unfortunately, this can have negative educational impacts by reducing the support that instructors can offer each student. Automation has resulted in significant benefits, allowing developers to work more efficiently and deliver higher-quality software, but automation is not as prevalent within computer science education as it is within industry. To help promote better educational outcomes, particularly by improving the feedback that students receive on their work, I adopt software engineering automation techniques into computer science education and evaluate their efficacy.

With ever more students enrolled in computer science programs comes a more widespread use of team-based learning (TBL) and larger teams. While TBL has numerous educational benefits, it is not an educational panacea. Larger teams increases the risk of team challenges, including ineffective communication and non-participation, which has the potential to hamper educational outcomes. To address this, I propose and evaluate using survey techniques to gain insights into how teams work and the challenges that students face in this environment, and enable just-in-time support for struggling teams. This approach can provide instructors with feedback on team challenges, and also encourages self-reflection on the part of students.

Together, these approaches support my thesis: **Using software engineering automation and survey techniques in computer science education results in improved student learning outcomes, early prediction of struggling teams, and more effective instructional materials.**

My first two research contributions focus on applying software engineering automation to support individual students. My *test flakiness study* investigates the impact that configuration options have upon the stability of Selenium tests. This supports improved educational outcomes by giving students more consistent feedback and greater confidence in the code and tests that they write. My *automated program repair study* investigates the mistakes that students make when learning SQL, and introduces an automated pro-

gram repair tool for SQL queries. It demonstrates that automated repair can be applied to special-purpose languages such as SQL, and that students find automatically-repaired SQL queries to be understandable, suggesting that they may have promise as an instructional technique.

My final three research contributions use survey techniques and software engineering automation to focus on supporting software engineering student teams. My *collaboration reflection study* investigates the use of a team collaboration reflection survey (TCRS) for identifying software engineering student teams that are struggling to collaborate effectively. It shows that most (89%) teams which later receive poor grades can be flagged through the TCRS, typically by the halfway mark of the project, and students appreciated that the TCRS encourages self-reflection. In my *team challenges study*, to better understand team challenges that were uncovered through the TCRS, I interview students who had recently completed a team-based software engineering course about their teaming experiences. This provides novel insights into how teams work together, and the types of issues that students face and how they attempt to overcome them. It demonstrates that the issues student teams face are largely in-line with educational theory, and informs improvements to instructional materials to help students work together more effectively. Finally, in my *contributions analysis study* I develop an algorithm and a tool to summarise individual students' code contributions to team-based projects. I then conduct a study to evaluate whether this information can help TAs grade projects more consistently and provide students with better feedback. My algorithm performs abstract syntax tree-based program analysis to offer more meaningful summaries of individual contributions than state-of-the-practise approaches. This study demonstrates that automated contributions summaries help TAs grade more consistently and provide more actionable feedback to students. Additionally, by helping instructors evaluate students more consistently, this can help identify teams that are struggling to work together effectively.

Taken together, these studies demonstrate that software engineering automation and surveys can result in benefits in computer science education. In particular, I demonstrate that this can provide students with feedback that is more consistent, and thus more actionable. Additionally, I demonstrate that surveys can provide insights into the challenges that teams face working together, thereby helping instructors provide guidance on how to work more effectively.

Supporting Computer Science Education Through Automation and Surveys

by
Kai Presler-Marshall

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2022

APPROVED BY:

Laura Bottomley	Collin Lynch
Sarah Heckman Co-chair of Advisory Committee	Kathryn Stolee Co-chair of Advisory Committee

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisors, Dr. Katie Stolee and Dr. Sarah Heckman, to whom I owe unpayable debts. Katie, thank you for encouraging me to pursue a dream I didn't know that I had, and for encouraging me over and over to pursue a PhD. Sarah, thank you for remaining relentlessly positive. Thank you for the tireless support and encouragement that you have provided ever since I entered the Computer Science department. I thank you both for helping me find my passion in computer science education research. And thank you both for the tireless feedback throughout, helping me learn from my mistakes and giving me responsibilities for Software Engineering where I can learn and prove myself.

I would like to thank the other faculty members of NC State University who have contributed so much to where I am now. To Dr. Laurie Williams, thank you for asking me to be your TA in Software Engineering in Fall 2016. I am certain that if I did not have that experience, and the subsequent encouragement from Katie, I would never have considered a career as an educator and be in this position. To Dr. David Sturgill, I still fondly remember my experience taking C and Operating Systems with you. Your lectures and class projects were some of the best I've had, and I thank you for putting up with my shenanigans. To Dr. Barbara Adams, thank you for trusting me to teach a summer class before I had entered into the PhD program. I hope my class lived up to your expectations. Finally, I would like to thank Dr. Dan Harris for lending his statistical expertise for the study in Chapter 7.

I would also like to thank my friends in the Computer Science program who have been so supportive and provided such positive pressure upon me. To Eric Horton, I thank you for providing friendly competition and encouraging me to do my best all the time. Thank you for all of the help that you provided for the research in Chapter 3. To Sarah Elder, I thank you for your support as I was a new and clueless TA. To Gina Bai, Justin Middleton, George Matthew, and Peipei Wang, I thank you for your friendship and support over the past four years, and the feedback and help that you have provided. Gina, you have been a fantastic friend and colleague as we have worked on several papers together. Finally, to Samim Mirhosseini and Nischal Shrestha, thank you for your friendship. Nischal, you have been a great source of encouragement these final few months as everything is coming together.

I would like to thank my committee for their support and guidance. To Sarah and Katie, I cannot say too many positive things. To Dr. Laura Bottomley, thank you for bringing a broader perspective from outside CS. To Dr. Collin Lynch, thank you for holding me to the highest standards and helping me come up with a much better evaluation plan in Chapter 7.

Finally, I'd like to thank my family for their support throughout. To Mom and Dad, who have been supportive in both words and actions throughout. To my brother and sister, who entertain my ramblings about topics I am sure they don't care about. And to my grandparents, for their love and support.

CHAPTER

1

INTRODUCTION

The demand for professional software engineers has grown remarkably since the turn of the 21st century. According to estimates from the Bureau of Labor Statistics, in 2021 there were nearly two million software engineers working in the United States [Sejb], a nearly three-fold increase compared to 2001 [Seja]. Additionally, the field is expected to continue to grow rapidly, with US government estimates suggesting a growth of more than 20% between 2021 and 2030 [Sejb]. The increasing demand for software engineers has resulted in an increasing demand for computer science programs that can train them. Indeed, the number of computer science bachelors degrees awarded in the United States has more than tripled between 2010 and 2020, and enrollment from non-majors continues to grow as well [Zwe21]. Taken together, this information shows an explosive growth in the fields of software engineering and undergraduate computer science education.

However, faculty hiring has been unable to grow at the same rate as undergraduate enrollment. The number of computer science PhDs awarded each year in the United States has grown by only 20% since 2010 and has remained largely flat since 2013 [Zwe21]. Additionally, a majority of PhD graduates continue to enter industry rather than academia. Consequently, although faculty hiring continues to grow [Wil20], it has done so at a slower rate than the increase in enrollment [Zwe21], and a constrained supply of PhD graduates puts further pressure on departments' abilities to hire additional faculty going forward.

The consequence of this is that instructors' time is split between more and larger classes, leaving them unable to offer students the individual attention they otherwise could. The negative impacts of this manifest in multiple ways, impacting both individual students and student teams.

To help alleviate this problem of large and growing classes, enable better learning outcomes for students, and offer more support for instructors, I propose and evaluate two novel approaches for improving computer science education. I argue that these techniques can help both students and instructors, by helping students meet learning outcomes and providing instructors with support to help them more effectively teach large and growing courses. To do this, I use automation techniques from software engineering and surveys that encourage students to self-reflect, a key component of self-regulated learning [Pan17].

Automation is widespread within software engineering, with developers taking advantage of tools and techniques that can help them design, implement, test, debug, and maintain software more effectively than is possible working unassisted [des04; Vas15]. While the breadth of these tools is immense, they all have the same fundamental goal: allowing developers to work more efficiently and deliver higher quality software by letting them avoid tedious or error-prone manual portions of their jobs.

While not as prevalent as in industry, computer science education also draws on automation, with techniques and tools that enable automated grading [Edw08; Gul18; Yi17; Par17] or automated team formation [Lou14]. Computer science education also uses some of the technologies used by professional developers to assist instructors and students [Hec18a]. Beyond helping students work through problems more rapidly, automation also has a role to play in helping students more effectively learn material. Through the lens of cognitive load theory [Swe88], automation may help students focus their mental efforts toward better long-term information storage. In this dissertation, I consider applications in education for three types of software engineering automation: automated testing [Kim13b; Cha06; Mao16], automated program repair [Nod20; Mar19; Ngu13; Ke15], and program analysis [Tra20; Sin17; Fei16; Fal14].

Another technique used within education is surveys, which can help promote self-reflection, an essential component of self-regulated learning [Pan17]. This benefits students by getting them to pause and reflect on their learning, taking stock of what is working and what is not working. Surveys can also aide instructors, giving them regular, informal feedback on the progress and difficulties that students face which can then inform "monitoring and arbitration" to help resolve complex issues teams are facing [Fra13; Bur03].

While surveys have been used to encourage self-reflection and reveal team challenges, little has been done to evaluate whether these benefits can transfer to computer science

education. In particular, I use surveys to gain insights into the challenges that teams face. Prior work has shown that team-based learning offers substantial pedagogical benefits [Hry12; Aya15], but is not an educational panacea. Previous research has demonstrated that students may struggle to work effectively in teams [Taf16; Oak04; Dor12; Che15; Tuc06; Abb17; Dzv18], thus depriving them of a positive, team-based learning experience. By using surveys to gain insights into the challenges that students face working together, I help craft more successful teaming environments. In this work, I consider team challenges through the lens of Tuckman's model of team development [Tuc65; Lea] to understand where teams struggle to work together effectively.

In this dissertation, I propose and evaluate adapting additional automation techniques into computer science education, and demonstrate the benefits that they can bring to students. My work features automated testing (Chapter 3), automated program repair (Chapter 4) and automated program analysis (Chapter 7). I study the impact of Selenium configuration on test flakiness in the context of iTrust2 [Hec18b], and identify an optimal configuration for running tests that eliminates flakiness and provides students with more consistent feedback on the status of their projects. I also show that automated program repair can produce repairs students find understandable, which may pave the way for their use in intelligent tutoring systems [Cro18]. Finally, I use automated program analysis to help teaching assistants (TAs) grade projects more consistently and provide students with more actionable feedback. This can help students figure out what is expected of them in team-based environments, and help instructors identify students who are not making the expected contributions to their team.

Additionally, I draw upon survey techniques to offer novel insights on how software engineering teams work and the challenges that they face. In Chapter 5, I propose and evaluate a Team Collaboration Reflection Survey (TCRS) to proactively identify teams that are struggling to work together effectively. Based off of the results observed, in Chapter 6, I draw upon survey techniques to conduct more detailed interviews with students to better understand the challenges they faced with their teams and how the course teaching staff can best assist them. Finally, I use reflection surveys in Chapter 7 to understand how TAs use automated contributions summaries and how improvements can be made to them to improve their grading workflow.

Taken together, the information gained from these approaches leads to my thesis statement:

1.1 Thesis

Using software engineering automation and survey techniques in computer science education results in improved student learning outcomes, early prediction of struggling teams, and more effective instructional materials.

1.2 Research Questions

To support this thesis statement, the rest of this dissertation is structured around five studies. Each subsequent chapter explains how that study contributes to the thesis statement.

1.2.1 Study 1: Test Flakiness

NC State University's undergraduate software engineering course uses Selenium tests [Hun18] to provide end-to-end testing of web applications. However, I have observed that Selenium tests can be flaky [Fow11], with tests passing and failing with no changes to the underlying code or tests. Flaky tests give inconsistent feedback, and deny students confidence in the correctness of their code. This inconsistent feedback can frustrate student learning. Additionally, when students are graded on maintaining a passing build, it can unfairly impact their grades on the course project. In this study, I aim to identify a more stable configuration for running Selenium tests in the course project. To do so, in Chapter 3, I consider five research questions focused on how to achieve an optimal run configuration for automated Selenium tests.

- **RQ1**: *What is the impact of the WebDriver on Selenium stability?*
- **RQ2**: *Which Selenium wait methods are the most stable?*
- **RQ3**: *What is the impact of hardware (CPU and memory) on Selenium stability?*
- **RQ4**: *What effect does the host operating system have on Selenium stability?*
- **RQ5**: *What impact does restarting the browser between individual tests have on Selenium stability?*

To answer these research questions, I conduct an experimental study to determine the impact of various factors on Selenium test stability. I identify that Selenium tests run faster on more powerful hardware hardware. More importantly, they also run more reliably, with

tests passing more consistently and experiencing fewer failures. I also identify that Chrome outperforms other browsers in both test execution time and stability, and the manner in which the test tries to *wait* for the page to finish rendering also impacts stability. Finally, I identify that disabling browser restarts and using the same browser session to run all tests in a suite results in faster runtime and completely eliminates test flakiness. This work results in both a better understanding of how to configure Selenium for automated testing, and a more stable application to use as a teaching tool in NC State's Software Engineering course. While this study does not evaluate improvements to student learning, prior work has demonstrated that more consistent feedback improves learning outcomes [Vat21]. Answering these research questions supports the thesis statement by demonstrating that software engineering automation, in particular improvements to automated testing, can help provide more effective instructional materials.

1.2.2 Study 2: Automated Program Repair

Prior work has studied the types of mistakes that students make when first learning new programming languages, which helps educators identify common pitfalls and tailor their lessons accordingly [Bro14a; Alt15; Bro17]. Additionally, prior work has considered the use of automated program repair within computer science education [Gul18; Yi17]. However, neither of these techniques have been applied to SQL, the primary language used for relational databases. As SQL is widely used by both computer scientists and end-user programmers [Tai19; Mig20; Sos], I seek to better understand both the types of mistakes beginners make and whether automated program repair can address them. In Chapter 4, I consider three research questions related to the types of mistakes that students make while learning SQL, and the ability of automated program repair to fix them and be used as a teaching tool.

- **RQ6**: *What types of mistakes do beginners make when working with SQL?*
- **RQ7**: *How well can SQLRepair fix errors introduced by beginning SQL programmers?*
- **RQ8:** *Do students find SQLRepair-repaired queries to be more understandable than queries written by other students?*

To answer these research questions, I conduct a controlled study where I teach undergraduate computer science students basic database concepts and introduce them to the language SQL. Next, I task students with using SQL to solve problems, asking them to write queries that are capable of performing illustrated transformations from one table to

another. I analyse the queries they wrote to understand the types of mistakes that students make, and develop a tool, SQLRepair, to perform automated repair on SQL queries. I find that students struggle with both the syntax and semantics of SQL, and that automated repair is capable of fixing some of the queries that students write. Finally, I demonstrate that students find repaired queries from my tool as understandable as queries written by other students, suggesting that automated repair may be useful as an educational technique. This study supports the thesis statement by demonstrating that software engineering automation, in this case automated repair, has the potential for improved learning outcomes and more effective instructional materials.

1.2.3 Study 3: Collaboration Reflection

Practically without exception, professional software engineering is run in teams, bringing together developers with disparate skills to solve complicated problems [Lay00; Ric12; Ram20]. To prepare students for this, most computer science programs include team-based learning opportunities [Wil00; Sim02; Iac20]. However, despite the educational benefits of team-based learning, it introduces new challenges. Dysfunction in student teams can impede the learning experience they offer [Tuc06; Oak04; Iac20; Mar16; Dzv18]. In Chapter 5, I consider three research questions to focus on proactively identifying struggling software engineering student teams.

- **RQ9:** *Can weekly reflection surveys identify software engineering teams in need of instructor assistance?*
- **RQ10:** *Can weekly reflection surveys identify software engineering teams that need assistance sufficiently early?*
- **RQ11:** *Can weekly reflection surveys help support a better experience for software engineering teams?*

In this work, I develop a collaboration reflection survey that asks students to consider how effectively they think their team is working together. I then conduct a classroom study, where I integrate the TCRS into two projects in a junior-level software engineering course. To ensure students take the time to self-reflect, the TCRS is made a mandatory component of weekly project activities for both projects. The TCRS is delivered electronically via email, and lightweight automated parsing provides a list of teams for the course teaching staff to follow up with. I find that it is capable of identifying a large majority of teams (89%) that go on to receive poor grades. Additionally, I find that a majority of students find the TCRS

useful for self-reflection or helping keep them on track. This study also provides insights into common issues that teams face, which can then be proactively addressed. This study supports the thesis statement by demonstrating that survey techniques can result in early prediction of struggling teams and more effective instructional materials.

1.2.4 Study 4: Team Challenges

In Chapter 5, I demonstrate that I can proactively identify a large majority of teams that go on to do poorly. However, my work also suggests that the issues that teams face may be broader than grades alone can reveal. Additionally, while it provides evidence that teams *do* face issues, it does not reveal what issues they are facing. While prior work has studied team challenges from the outside [Tuc06; Oak04; Iac20; Dzv18], to the best of my knowledge, no prior work has studied these challenges from the perspective of the teams themselves. In this study, I aim to address this shortcoming and help educators better understand the types of challenges that teams face and how they sought to overcome them. To do this, in Chapter 6, I consider four research questions to better understand how software engineering student teams work, and what challenges they face along the way.

- **RQ12**: *What team-related difficulties do students face on software engineering teams?*
- **RQ13**: *Why are some student software engineering teams able to overcome the issues that they face, while others are unable to do so?*
- **RQ14**: *What support do students on software engineering teams want from the course teaching staff for overcoming collaborative difficulties?*
- **RQ15**: *What are the characteristics of successful software engineering student teams?*

To answer these research questions, I interview 18 students who had recently completed a team-based software engineering course, and asked them about the processes their team followed, what if any challenges they faced, and how they tried to overcome these challenges. From this, I follow a grounded theory approach, identifying common sources of dysfunction, steps that teams took to try and overcome the challenges they faced, and reasons why some of these steps proved insufficient. This study reveals novel insights into how teams work, and this study suggests improvements to project materials to help teams work more effectively. This study contributes to the thesis statement by using survey techniques to result in improved student learning outcomes and more effective instructional materials.

1.2.5 Study 5: Student Contributions Analysis

Prior work has demonstrated that team-based learning (TBL) offers substantial educational benefits [Hry12; Aya15], but is not without challenges. In order to discourage freeriding [Hal13] and encourage engagement throughout the project, instructors must be able to identify individual students' contributions. Version control systems such as Git [Tor] have made this process easier by associating each *commit* with its specific *author*, but there is still manual effort required to accurately identify the totality of what each student has contributed. Because this is a largely manual process, instructors or TAs may miss contributions that students have made, providing inconsistent or contradictory feedback [JAC88] that impedes learning [Vat21]. To help assist this process, in Chapter 7, I consider four research questions to identify whether tool assistance can help TAs grade projects more effectively.

- **RQ16**: *Can automated summaries of student contributions enable faster grading by TAs?*
- **RQ17**: *Can automated summaries of student contributions enable more consistent grading by TAs?*
- **RQ18**: *Can automated summaries of student contributions enable less frustrating grading from the perspective of a TA?*
- **RQ19**: *How do automated summaries of student contributions enable better feedback?*

To answer these research questions, I develop an algorithm for summarising code contributions individual developers make to team-based projects. I then develop a reference implementation, AutoVCS, which targets projects written in Java, and conduct a study to evaluate its efficacy. Through a controlled user study, I demonstrate that automated summaries can help TAs grade student contributions more consistently, which can assist with detecting teams with an unequal work distribution. Additionally, TAs that grade with automated summaries provide students with more actionable and more nuanced feedback, which can help students learn how to work more effectively in teams. This study contributes to the thesis statement by demonstrating that software engineering automation can result in improved learning outcomes and early prediction of struggling teams.

1.3 Contributions

The major contributions of this dissertation include:

- A demonstration that software engineering automation can help provide students with more consistent feedback (Chapters 3 and 7).
- A demonstration that automated program repair can be applied to SQL, and that repairs can be made of sufficient quality to have promise in intelligent tutoring systems (Chapter 4).
- A lightweight collaboration reflection survey for identifying struggling student software engineering teams (Chapter 5) and an algorithm that helps with identifying cases of nonparticipation (Chapter 7).
- A demonstration that surveys can encourage self-reflection and provide insights into the challenges that student teams face (Chapters 5 and 6).
- A discussion of some of the the challenges faced by students in computer science, including a classification of mistakes students make as they learn and first work with SQL (Chapter 4) and the challenges that students face collaborating on software engineering teams (Chapter 6).

1.4 A note on pronouns

The work in this dissertation would not have been possible without the tireless support of my advisors, and the work in Chapter 3 would not have been possible without the assistance of Eric Horton. In recognition of their contributions, the remainder of this work uses first-person plural pronouns.

CHAPTER

2

RELATED WORK

The related work is organised into two sections. Section 2.1 discusses related work in automation in software engineering. Section 2.2 discusses prior work on collaboration and team-based learning, with a focus on computer science; common team challenges; and the pedagogical foundations of team-based learning.

2.1 Software Engineering Automation

Automation is widespread within software engineering, with developers taking advantage of tools and techniques that can help them design, implement, test, debug, and maintain software more effectively than would be possible working unassisted [des04; Vas15]. Just as few developers today would choose to implement a complicated piece of software in assembly language, few would choose to do their job without some sort of automated support. Automation techniques in software engineering vary widely, but perhaps the most common is integrated development environments (IDEs) that offer intelligent code completion, automated refactoring, and code generation [Ngu19; Mic21; Ijc; Rob08]. Automated testing [Ber05b; Kim13b; Cha06; Mao16] is also widely used, often run in combination with a continuous integration environment [Seo14], which has helped practically eliminate major integration issues [Sul16; Duv07]. Software engineers also make use of program

analysis tools, which can help catch bugs without even running code [Nag05; Joh13] and automated repair, which can help fix bugs that make their way through [LG12; Lou20; Wei09; Mec16]. Despite the breadth of these tools, they all have the same fundamental goal: allowing developers to work more efficiently and deliver higher quality software by letting them avoid tedious or error-prone manual portions of their jobs. We consider three areas of software engineering automation in further depth: automated testing (Section 2.1.1), automated program repair (Section 2.1.2), and program analysis (Section 2.1.3).

2.1.1 Automated Testing

Automated testing is widely used in software engineering [Kim13b; Cha06; And10; Ber05b; Mao16; Ngu14], where it is an integral portion of continuous integration environments. Berner et al. [Ber05b] explain that automated testing is "intended to save as much money as possible" by reducing the need for manual testing, and can help developers deliver higher-quality software by testing at a greater scale than is possible with manual tests. Prior work in testing has explored automation at all levels of the testing hierarchy, from unit tests [Kim13b; Cha06; And10] through system tests, which typically test an application's GUI [Ngu14; Vil17]. Prior research in automated testing has also explored automated test generation, using tools not just to run tests in a repeatable fashion, but using them to generate testcases as well. For example, Marinov and Khurshid [Mar01] explore automatically generating inputs for Java tests. Buy et al. [Buy00] use program analysis to identify multiple methods that interact with the same fields in a class, and then generate testcases to exercise these methods in different combinations. Fully automated testcase generation complicates the process of testing against an oracle, and as such automated testcase generation often uses a simple oracle, focusing on ascertaining whether certain inputs will cause a program to crash [Mao16; Ngu14].

End-to-end system testing of web applications is typically performed using Selenium, a test framework for scripting actions against webpages, allowing a user to specify actions to perform and the output that is expected. Vila et al. [Vil17] discuss the need for automated system testing of web applications, and provide information on the use of Selenium as well as common problems associated with it. One issue they discuss is that Selenium tests have a tendency to run slowly. Kuutila et al. [Kuu16] consider this problem in more detail, benchmarking Selenium tests using a variety of programming languages and automated browsers to identify the performance tradeoffs, and report that the Chrome browser, controlled by Python bindings, typically performs the best. Leotta et al. [Leo14; Leo13] discusses issues with reliability and maintenance of Selenium tests, discussing approaches for improving

the robustness of element locators [Leo14] and strategies for locators that are easiest for developers to maintain over time [Leo13]. Other recent work has considered Microsoft Playwright [Vei21], another tool for scripting web browsers for automated tests, although it has yet to be as widely adopted as Selenium.

While automated tests can offer many benefits, they are also susceptible to nondeterminism, or *flakiness*. Fowler argues [Fow11] that flaky tests are "useless" because they no longer serve as meaningful regression tests, and are a "virulent infection" that can compromise the effectiveness of an entire test suite as developers lose confidence in it. They discuss that common causes of nondeterministic tests are asynchronous behaviour and tests that are not properly isolated, and thus cause interference for each other. Luo et al. [Luo14] conduct an empirical study of test flakiness, and conclude that is a relatively common issue, often caused by asynchronous behaviour. To help researchers address the impacts of flakiness, Cordy et al. [Cor19] present FlakiMe, a tool for introducing a controlled degree of flakiness into test suites, to measure the impact it has on techniques such as mutation testing and automated program repair. To help practitioners, Bell et al. [Bel18] present DeFlaker, a tool which labels tests as flaky by using Java AST analysis to determine when failing tests do not execute any recently-changed code. Meanwhile, Herzig and Nagappan [Her15] use association rule mining to associate test failures with code changes, and thereby identify tests failures that are likely due to nondeterminism. By contrast, in Chapter 3 we have already identified tests that are known to be flaky, and our goal is to address the flakiness observed through modifications to the Selenium configuration or underlying system configuration.

2.1.2 Automated Program Repair

Software engineering automation is also used in the field of automated program repair (APR), or creating programs that can automatically find and patch bugs in other programs. Past work in automated program repair has considered approaches for repairing C [Wei09; LG12; Lon16] and Java [Jia18; Xua16; Lou20; Kim13a] programs, with some work done in specialty languages such as Python [Dro20]. Automated program repair techniques can be classified into two main approaches [Gaz17]. Generate-and-validate approaches generate a large number of candidate repairs, and then evaluate their correctness by executing testcases [Wei09; LG12; Kim13a; Lon16; Qi14; Sah17]. By contrast, correct-by-construction (or semantics-driven) approaches [Mec16; Xua16; Ngu13; Ke15] encode the problem to solve, and then attempt to solve it, often using a SMT solver such as Z3 [DM08].

Regardless of the approach, APR techniques produce repairs, or patches, that are at

best *plausible*, in that they satisfy all provided testcases. Patches must then be evaluated by human developers to ensure they actually solve the problem intended, and are not merely correct by coincidence. As this manual process can be quite time intensive, several APR techniques focus explicitly on the quality of the generated patch. In one approach, Kim et al. [Kim13a] manually review over 60,000 developer-written patches for Eclipse, and classify the types of repairs performed to generate *patch templates*. Their tool then generates patches based on these templates, ensuring that all generated patches resemble what developers write. Long and Rinard [Lon16] take a similar approach, focusing on human-written patches to open source projects, and use a *maximum likelihood estimator* approach to produce a statistical model of their characteristics. Patches generated by their tool are then ranked by their resemblance to human-written patches, and tested in that order, to save test execution time and prioritise returning a patch that resembles what a developer would write. Both Kim et al. [Kim13a] and Long and Rinard [Lon16] report producing patches acceptable to developers at a higher rate than state-of-the-art approaches. Drosos et al. [Dro20] similarly focus on the quality of synthesised Python code from their tool, Wrex, which helps data scientists by synthesising code to perform an illustrated transformation. They demonstrate that data scientists are more productive with Wrex than without it, demonstrating that their tool-generated code is of sufficient quality to be useful.

While most APR techniques are used on small, open-source projects [Le 15], several projects have considered their use in industrial settings. Marginean et al. [Mar19] at Facebook Research have extended their automated testing tool Sapienz [Mao16] to perform automated repairs, creating SapFix. They use SapFix to automatically repair crashes in six applications that are each upwards of ten million lines of code, and reflect on their experiences. They report that many bugs still require developer intervention to resolve, but that SapFix can solve many crashes before users experience them. Meanwhile, Noda et al. [Nod20] consider the use of automated program repair at Fujitsu. They report that a state-of-the-art solution, ELIXIR [Sah17], works poorly in this context, producing few fixes and even fewer that developers consider acceptable. Based on the experience, they create ELIXIR+, which features several improvements that increase the rate at which developers accept patches it generates.

To the best of our knowledge, no prior work has investigated automated program repair for SQL queries, yet SQL queries are error prone [Tai19; Tai18]. The closest work to that in Chapter 4 we am aware of is SCYTHE [Wan17], which synthesises SQL queries given a *(source, destination)* table pair. However, SCYTHE is limited in that it does not support common operators such as projection, and supports only a single `(source, destination)` pair,

while SQLRepair supports arbitrarily many.

2.1.3 Program Analysis

Program analysis is the process of "analysing software to learn about its properties" [Ald19]. Consequently, program analysis is a broad field, and includes activities such as manual code inspections, automated testing, and static analysis tools. At a high level, program analysis can be broken down into *static analysis*, which is analysis done without running a program, or *dynamic analysis*, which involves running the program and observing its behaviour. While there is much interesting work on instrumenting programs to perform dynamic analysis [Net07; Rav15; Luk05], we focus here on static analysis techniques. Static analysis is typically performed on application source code, as the process is more difficult to perform on compiled binaries [Kin08; Kin12]. Static analysis is often used to check for bugs [Wög05; Aye08; Tra20; Sin17; Zam17] and enforce style guidelines [Bal13], and even check for security vulnerabilities [Che04; Liv05; Oye18; Li20]. Additionally, static analysis approaches may be integrated into a continuous integration pipeline [Zam17]. Because static analysis tools do not rely on executing a program to analyse it, they can often run much faster than dynamic analysis, and can discover problems in code that is difficult to execute under normal conditions [Che04]. Conversely, however, static analysis tools report bugs that a program *may contain* and consequently are prone to *false positives*, or reporting problems that a program does not actually contain. To address this, Grech et al. [Gre18] propose a technique that combines static and dynamic analysis techniques, and report a substantial improvement in precision.

While primitive static analysis tools can rely on `grep` to perform simple keyword matching, this approach is unable to distinguish between method calls, comments, and variable names, and consequently offer limited capabilities [Che04]. Instead, most static analysis tools integrate with a compiler, leveraging a stream of tokens or a full AST [Flu07; Fal14] to better analyse a program. AST-based analysis has many applications beyond searching a program for bugs, and has been used to explore API usage [LÏ1] and how code evolves over time [Mar13; Meq20]. Most similar to our work in Chapter 7, Feist et al. [Fei16] provide an AST-based analysis algorithm which traverses Git history and compares file revisions to understand changes. However, they focus their evaluation on open-source project evolution, while we focus on understanding contributions in education, and consequently the types of data extracted and how it is used is very different.

2.2 Collaboration and Team-Based Learning in Computer Science Education

Professional software engineering is a team-based activity, often drawing together a team of diverse and distributed members [Lay00; Ric12; Ram20]. To help prepare students for this reality, team-based learning is a key learning outcome assessed by ABET accreditors [Abe], and is consequently taught in many computer science programs [Wil00; Sim02; Iac20; Bat22]. Educational theorists have recognised that team-based learning (TBL) offers substantial pedagogical benefits [Hry12; Aya15], so it is often used as teaching technique regardless of the learning goals for the course [Bat22; Lin21; Har]. However, prior work has demonstrated that student teams are at great risk of dysfunction, and many students struggle to work together effectively [Abb17; Dzv18; Mar16]. In Chapters 5 and 6 we seek to gain further insights into the challenges that teams face.

Substantial prior work has studied teaming in a variety of educational contexts. Abbasi et al. [Abb17] studied the challenges that student teams face and their ability to resolve them. Other work has studied student attitudes towards teaming [Pfa03; Owe15; Bur03; Rud17], including focusing on how to improve students' motivation and engagement with their teams [Pau06; Che15]. Other work has focused explicitly on teaming pedagogy [Tuc06; Gil13], focusing on improving pedagogical practise [Raf13], the educational theory behind successful teams [Hry12; Aya15], and how to help students form the most effective teams [Oak04].

Because of the pedagogical benefits that it offers, TBL is featured in many undergraduate computer science programs, where it has been shown to offer numerous benefits. Early work in collaborative learning in computer science education dates to work on pair programming by Williams and Kessler [Wil00]. They demonstrated that students who worked on tasks in pairs applied a positive "pair pressure" and motivated each other to do their best. Consequently, students in this environment performed better than those who worked on their own, producing higher-quality software that was capable of passing more testcases. Students who engaged in pair programming regularly turned to each other for help, thus easing the burden on the course teaching staff. While these conclusions are now core to computer science pedagogy, they served as one of the first looks at team-based learning (TBL) in computer science education. Lin et al. [Lin21] focus on collaborative learning within an upper-level algorithms course. In their course, students are encouraged, but not compelled, to self-select study groups to work through practise problems before completing their own individual homework assignments. They report that students who

chose collaborative learning outperformed those who did not. While stating that "any collaboration improved individual performance", they concluded that groups of four or five students performed the most effectively. Meanwhile, Harris studies the use of teamwork in operating systems and assembly courses [Har] and reports that it "reinforced students' learning" across the curriculum. Gitinabard et al. [Git20] study how small student teams function, and build a classifier to label different types of collaborative behaviour. Their work provides insights into how students work together, and may serve as an early warning when teams are not working together effectively. Finally, Tafliovich et al. [Taf16] discuss how to evaluate and grade students on teams, and student preferences for how their grades are calculated depends on how effectively their team has been working together.

In many computer science curricula, software engineering courses used to teach students how to work effectively in teams. Consequently, most prior work on teaming in computer science focuses on upper-level project-based software engineering and senior capstone courses. Most prior work in the area explicitly talks about project-based learning as a way to teach the entire software engineering experience, describing these hands-on activities as ones that offer the best learning experiences [Ṕ20; Sim02; Kha20; Par18; Hun21]. Several studies discuss having student teams work on projects where there is a real client or customer that the teams work with [Hun21; Bat22; Abe09]. Hundhausen et al. [Hun21] discuss the difficulties of evaluating projects fairly when each team works on a different project. To address this, they propose a rubric based on the Goal-Question-Metrics framework that focuses on the team's process and product but is sufficiently general to work regardless of the underlying project. A literature review of teaming in software engineering by Groeneveld et al. [Gro19] recognised that many computer science students graduate with solid technical skills, but poor communication, conflict resolution, and other interpersonal skills (so-called "soft skills"). To address this, they report a growth of team-based learning, often in the form of extensive project-based courses, including final capstone courses. Other papers discuss using team-based learning to create a more engaging environment [Ṕ20], increasing students motivation, or broadening participation from historically under-represented groups [Bat22].

2.2.1 Team Challenges

Prior work has identified that although team-based learning offers pedagogical benefits, it also introduces new challenges. Possibly the best known issue is one of *freeriding* [Hal13], where students contribute little and count on their teammates completing the project in their absence. A common solution for addressing this is giving students individual grades

in addition to a team grade; to assist with this, past work has considered approaches for visualising contributions from individual members [Par18] or allowing students to provide anonymous feedback for each other [Tuc06; Rob17; Din14].

However, other issues may cause teams just as much trouble. Sims-Knight et al. [Sim02] discuss a view among educators that students will learn how to work on a team merely by being on one, without explicit instruction on how to effectively work with other students. More modern work has addressed this with activities to teach students how to function as a team, as we discuss in Section 2.2.2.

Despite activities to teach effective teaming skills, many teams struggle to work together effectively. Prior work has demonstrated that up to 40% of teams in project-based courses are characterised by "internal strife" and fail to work together effectively [Tuc06], often caused by a lack of communication or effective project management [Oak04; Iac20]. Iacob and Faily [Iac20] report that dysfunction is a risk in student software engineering teams, where low engagement or poor communication can hamper individual and team outcomes, but do not go into detail about the issues that contribute to poor engagement. Marques [Mar16] proposes having a "monitor" conduct weekly meetings with teams of software engineering students, observing them work and providing feedback on the overall team function and contributions of each member. They report mentored teams produced higher-quality software, and performed substantially better on their final project, but provide little elaboration on the details of the challenges that students faced in either case. As these approaches require mentors to help teams, Maguire et al. [Mag19] discuss how to train mentors and ensure they have the skills to help teams overcome various challenges. Meanwhile, Gitinabard et al. [Git20] focus on VCS data to identify how small teams collaborated, and cases where one member of a team failed to contribute. Most similar to our work in Chapter 6, Dzvonyar et al. [Dzv18] explore team forming and success in software engineering. They discuss considerations when forming teams for a project-based course, and survey teams at the end of a software engineering course, asking questions about team synergy and any challenges the team faced. They report that team synergy was generally high, but some teams struggled with low motivation and performed poorly. While they present a discussion of how the teaching staff formed teams, there is no discussion of how the teams themselves operated and how this may have impacted any challenges faced.

2.2.2 Teaming Theory

Team-based learning (TBL) is a learner-centred pedagogy, where students direct their own learning under the guidance of an instructor who serves as an "expert facilitator" [Hry12].

TBL is grounded in constructivist theory, which argues that students cannot merely absorb information passively, but must actively discover it. This theory says that learning is done through dialogue rather than a dissemination of facts. Prior work has demonstrated that this is typically a more effective pedagogy and results in better learning [Hry12; Aya15]. For these reasons, many project-based courses use TBL extensively, as TBL teaches students not just the skills of how to function in a team, but can teach other skills more effectively. However, despite the benefits it offers, researchers have recognised that TBL is not an educational panacea. Successful teamwork depends upon regular communication, particularly when members work asynchronously [Gil13].

To work successfully together on teams, students must be capable of conflict resolution, which requires both identifying challenges, and successful resolution of them [Pau06; Raf13]. Prior work has demonstrated that this can be a persistent issue that teams may struggle to overcome [Tuc06; Oak04; Iac20; Mar16]. In order to help students navigate these challenges and function more effectively in teams, many educators include team forming activities [Rap07; Pin06; Hog08], self-and-peer assessment [Din14; Taf16], or discussions of teaming theory (such as Tuckman's model of teaming [Tuc65; Tuc77], discussed in Raferty [Raf13] and Hansen [Han06]). In Chapter 6, we use Tuckman's model for characterising where teams faced challenges. Tuckman argued that teams progress through four stages: *forming*, as the members of the team meet each other but largely act independently, *storming*, as conflicts and disagreements arise between members, *norming*, where conflicts are resolved and the team starts to function as a cohesive whole, and finally *performing*, where members are motivated and engaged and the team works together effectively. Tuckman noted that some teams may skip the *storming* stage entirely, while others may face more intense "storms" and never emerge from this stage. Later, Tuckman and Jensen added a fifth stage, *adjourning*, where the group disbands upon the completion of their tasks [Tuc77].

CHAPTER

3

EXAMINING FACTORS IMPACTING SELENIUM RELIABILITY

This study[1] explores the impact that Selenium configuration options have on **automated test** stability. As a result, we found an optimal configuration, which contributes to **more effective instructional materials** by providing a far more stable project for instructors and their students in the software engineering course at NC State University.

> Satisfies part of thesis: Using **software engineering automation** and survey techniques in computer science education results in improved student learning outcomes, early prediction of struggling teams, and **more effective instructional materials**.

3.1 Study Rationale

This work studies Selenium, a popular framework for automating web browsers for testing [Hun18]. We study Selenium in the context of the undergraduate software engineering

[1] This study was published in substantial part as Presler-Marshall, K., Horton, E., Heckman, S. & Stolee, K. "Wait, Wait. No, Tell Me. Analyzing Selenium Configuration Effects on Test Flakiness". *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. 2019, pp. 7–13.

class project, iTrust2, at NC State University. In this work, we seek to improve the feedback quality provided by the continuous integration environment by reducing test flakiness and test execution time. iTrust2 is open source and designed to provide students with an industry-like experience by exposing them to a large system and continuous integration (CI) [Hec18b]. However, flaky tests are a source of frustration among students. While this may reflect actual industry experiences, for an educational context that introduces students to software engineering principles, ambiguous feedback from flaky tests can impede effective learning. Psychologist and educational theorist Lev Vygotsky [Stu02] argued that that the best way for students to learn material is through *scaffolding*. This approach argues that instruction must be guided by a teacher, and new material should be broken down and gradually integrated into what the student already knows. This contrasts to an approach of expecting a student to learn many new pieces all at once, where the extent of the new material may overwhelm the student's ability to learn *any* of it. While flaky tests may be a fact of life in industry [Fow11; Luo14; Bel18], their presence in an educational context is counter-productive to students' learning. Flaky tests effectively serve to provide students inconsistent, or even contradictory, feedback, as they make it unclear whether the underlying application actually works correctly or not. Prior work suggests that inconsistent feedback makes it more challenging for students to assess what they are doing correctly, and consequently impedes their ability to integrate feedback and learn from it [Vat21]. Vygotsky's theories suggest that rather than having students grapple with all of this at once, it is more helpful to students' education for them to learn software engineering skills in a consistent and predicable environment, and only *then* grapple with the additional challenge of understanding and overcoming flaky tests.

There are many potential sources of flakiness, but a typical situation involves a test case that attempts to verify the presence of a UI element before the browser has completely loaded, causing it to fail. To expose the source of flakiness in iTrust2, we start by analyzing the impact of the WebDriver (the automated browser controlled by Selenium) and waiting strategy (how the test attempts to wait for the page to be ready. We then test the best WebDrivers and wait strategies on three different hardware configurations and operating systems.

In summary, this study provides the following contributions[2]:

- A more stable version of iTrust2 that is suitable for use as a teaching tool in undergraduate software engineering classes and which performs well on our CI environment.
- A comparison of four methods of waiting for expected conditions on webpages when

[2]Study materials are available at `http://github.com/ncsu-csc326/iTrust2-SeleniumAnalysis`

performing automated testing with Selenium.

- A comparison of runtime and stability of iTrust2's Selenium tests on three different hardware and operating system configurations.
- A recommendation on the optimal configuration of Selenium for applications similar to iTrust2.

3.1.1 Introduction

NC State University (NCSU)'s undergraduate software engineering course uses iTrust2, a large Java EE medical records application, as one of its primary teaching tools. A successor to the original iTrust application that saw the course through ten years [Hec18b], iTrust2 was introduced to students in Fall 2017 as part of a larger course redesign. iTrust2 uses the Spring and AngularJS frameworks used in many enterprise applications. It consists of about 30,000 lines of Java and JavaScript code tested with nearly 500 Selenium tests.

Each semester, twenty five groups of students, working in teams of four or five, push their work to GitHub, where it is then automatically built and tested using Jenkins CI [Hec18a]. When students develop far in advance of a deadline, Jenkins is able to return feedback quickly (within 15 minutes); however, as the deadline approaches and load on the system increases, feedback becomes less timely. This exacerbates the issue of test flakiness by giving students less time to respond to any failures and ascertain their cause.

Consider the method in Figure 3.1, `fillHospitalFields`, extracted from iTrust2. It is a subroutine in a test case which verifies that submitting the form results in a new hospital record being registered with the system. The method starts with the implicit assumption that the browser is currently on the page for creating a new hospital object. It then asks Selenium to find and fill the input fields for name (lines 2-5), address (lines 6-8), state (lines 10-12), and zipcode (lines 14-16). Finally, it instructs Selenium to click the submit button and trigger a form submission (line 18).

In this example, Selenium may fail because it cannot find the element it looks for (e.g., an element with an `id` called `address`). Such a failure could be caused by several mistakes in the test or application: the wrong page was specified by the test case; the locator used to select the element was faulty; or there was a bug in the underlying application being tested. However, it is also possible that the test and application are both fine, and the befoe the browser had finished loading the page, Selenium checked, and failed. Selenium has no way of knowing if the UI will ever be ready, so this leaves it to developers to tell Selenium when to check.

```
1  public void fillHospitalFields () {
2      final WebElement name = driver.findElement( By.id( "name" ) );
3      name.clear();
4      name.sendKeys( "Tim Horton's" );
5
6      final WebElement address = driver.findElement( By.id( "address" ) );
7      address.clear();
8      address.sendKeys( "121 Canada Road" );
9
10     final WebElement state = driver.findElement( By.id( "state" ) );
11     final Select dropdown = new Select( state );
12     dropdown.selectByVisibleText( "CA" );
13
14     final WebElement zip = driver.findElement( By.id( "zip" ) );
15     zip.clear();
16     zip.sendKeys( "00912" );
17
18     driver.findElement( By.className( "btn" ) ).click();
19 }
```

Figure 3.1 A method from one of the iTrust2 Selenium tests that automates filling in fields on a web page.

The motivation of this work is to proactively identify and remove test flakiness and improve performance in iTrust2 by finding and implementing optimal Selenium and system configurations. All tests are known to be capable of passing, but under the conditions of student computers and our CI environment they do not all pass consistently.

3.2 Methodology

In this section, we provide background information on Selenium (Section 3.2.1), present our research questions (Section 3.2.2) and discuss our study design (Section 3.2.3).

3.2.1 Background

A WebDriver class implements the Selenium interface `WebDriver`. Each driver provides an interface for Selenium to control a single web browser. All WebDrivers in this study are named for the web browser they control. For example, ChromeDriver is the WebDriver implementation for the Google Chrome browser. Because WebDrivers have a 1:1 mapping with their browsers, we refer to the driver and the browser interchangeably.

3.2.1.1 Drivers

We use the drivers for Chrome, Firefox, PhantomJS, and HtmlUnit in this study. The HtmlUnit driver represents a headless browser designed specifically for automation; PhantomJS is a more capable browser designed for the same purpose; Chrome and Firefox are two popular and widely-used web browsers. The default configuration is used for HtmlUnit and PhantomJS, as both are already headless. Chrome and Firefox are run in headless mode (a requirement for our CI environment), with Chrome additionally specifying the options `window-size = 1200x600` and `blink-settings = imagesEnabled = false`, both of which were selected to improve runtime. We omitted drivers for browsers not supported by every major OS, such as Apple Safari and Microsoft Edge.

3.2.1.2 Wait Strategies

Several waiting strategies are considered. The first, *No Wait* is the default behavior of immediately failing when an element is not found. *No Wait* informs a baseline against which to compare other wait strategies. *Thread Wait* calls Java's `Thread::sleep`, which pauses thread execution for a fixed period of time.

Explicit Wait is a Selenium construct that tells the driver to wait for an explicit amount of time or until some condition has been satisfied (whichever occurs first). Explicit waits are supported by all drivers. We use explicit waits to verify the presence of elements. For example, the following will wait for an element with the `name` of `notes` to appear:

```
1 WebDriverWait wait = new WebDriverWait( driver, 2 );
2 wait.until( ExpectedConditions .visibilityOfElementLocated( By.name(
  ↪ "notes" ) ) );
```

Angular Wait is a Selenium construct that tells the driver to wait until the Angular web framework has completed all requests. It is only supported by the Chrome driver, and can be used as follows.

```
1 new NgWebDriver( (ChromeDriver) driver
  ↪ ).waitForAngularRequestsToFinish();
```

Unlike Explicit Waits, Angular Waits do not wait for a specific element to appear: rather they wait until *all* dynamic requests are finished. This has the potential to work better if the locator that would be passed to an Explicit Wait is overly general and thus would locate an element before the page is truly ready.

3.2.1.3 Current Configuration

The existing test suite for iTrust2 uses the HtmlUnit driver with a combination of *No Wait* and *Explicit Wait* strategies to verify the correctness of elements. HtmlUnit has been used for its reasonable runtime performance on our Jenkins CI systems, with 59 *Explicit Waits* scattered through individual tests where the *No Wait* approach proved insufficient.

3.2.2 Research Questions

We explore the following research questions, measuring test flakiness and runtime performance as the dependent variables, in the context of iTrust2:

- **RQ1**: *What is the impact of the WebDriver on Selenium stability?*
- **RQ2**: *Which Selenium wait methods are the most stable?*
- **RQ3**: *What is the impact of hardware (CPU and memory) on Selenium stability?*
- **RQ4**: *What effect does the host operating system have on Selenium stability?*
- **RQ5**: *What impact does restarting the browser between individual tests have on Selenium stability?*

Our research questions can be grouped into three broad categories: the impact of WebDriver configurations (RQ1 and RQ2), the impact of system configuration (RQ3 and RQ4), and further optimization (RQ5).

3.2.3 Study

We refer to a single run of an application's entire test suite as a *build*. A group of multiple builds is referred to as an *evaluation*. All evaluations in this work contain 30 builds. An *evaluation* is used to determine runtime performance, measured in seconds (i.e., average test execution time over all the builds in an evaluation). Because we know that all tests can pass, any test failure is seen as indicative of test flakiness. Thus, test flakiness is defined as the sum of all test failures over an execution.

We refer to the choice of driver and the waiting strategy employed as *WebDriver configuration*. We refer to choice of CPU, memory, and operating system as *system configuration*. When considering a combination of WebDriver and system configurations, we say *Selenium configuration*, or just *configuration* if it is not ambiguous.

Table 3.1 Selenium & System Configuration Options.

		Options				
	Factor	1	2	3	4	5
A	Wait	No Wait	Explicit	Thread.sleep	Angular	-
B	WebDriver	HtmlUnit	Chrome	Firefox	PhantomJS	-
C	Memory	2GB	4GB	8GB	16GB	32GB
D	Processor	AMD C60 (NB)	Intel E5-1620 (HP)	-	-	-
E	OS	Windows 10	Linux 4.13	-	-	-

Table 3.2 Mapping Selenium & System Configuration Options to Research Questions.

Factor	RQ1	RQ2	RQ3	RQ4	RQ5
Wait	*	*	2,4	2,4	2,4
WebDriver	*	*	1,2	1,2	1,2
Memory	2	2	*	3	3
Processor	1	1	1,2	2	1,2
OS	2	2	2	1,2	1,2

Our configuration options are summarized in Table 3.1. Rows A and B correspond to WebDriver configuration and will guide RQ1 and RQ2 (Section 3.3.1), rows C and D are hardware configuration and will guide RQ3, and row E guides RQ4 (Section 3.3.2). Table 3.2 maps configurations used (Table 3.1) to each research question we answered. For example, RQ1 has a * for the WebDriver factor, meaning all four options are considered. For Memory, RQ1 uses option 2, representing 4GB. Not all options are possible (e.g., HtmlUnit with Angular waits), but this provides a general outline for how each RQ was evaluated.

3.2.3.1 Setup

To address RQ1 and RQ2, we modified the current iTrust2 codebase and introduced a common superclass for all test classes. This superclass provides a WebDriver factory method and a method for performing waits. To obtain a list of flaky tests, we mined the Jenkins test logs from our undergraduate software engineering course in Spring 2018. Any Selenium test failure observed that appeared unconnected to the Git commit that triggered the build was considered a potentially flaky one and was included in our study. We manually analyzed 1,000 Jenkins test logs, and found 19 distinct locations in the source code where tests appeared to be flaky. Before every flaky location, we inserted a call to the waiting method in our superclass. We branched the codebase, implementing the WebDriver factory and waiting method for every valid combination of driver and waits (see Table 3.1, rows A & B,

yielding 13 valid configurations[3]).

To address RQ3 and RQ4, we procured two computers. First, an Acer netbook provisioned with Ubuntu 17.10, known hereafter as "NB-Linux". It was deliberately selected for its poor performance, as it represents the lower end of what students have been observed using. Second is an HP workstation system provisioned with Ubuntu 17.10 (known hereafter as "HP-Linux") and Windows 10 (known hereafter as "HP-Windows"). It was selected for giving performance similar to most student systems and to evaluate the impact of operating system when hardware is controlled for. Turbo Boost was disabled on both platforms, and Hyper-threading on the HP Z420 (the Acer's CPU does not support any form of SMT[4]), to result in a more consistent execution environment. Evaluations on HP-Linux were tested with 2, 4, 8, 16, and 32GB memory. Evaluations on HP-Windows were run at 8GB memory as a comparison against HP-Linux at 8GB.

To address RQ5 we reconfigured our WebDriver factory to not restart the browser between individual tests and retested on our three environments (NB-Linux, HP-Linux, and HP-Windows).

3.2.3.2 Execution

After each build, the execution time and list of failing test cases were recorded. At the end of an evaluation (30 builds), test flakiness was recorded as the sum of all failures seen in each build, and runtime was computed by averaging the runtime of each build within the evaluation.

We ran evaluations for each configuration on our procured systems, using performance results on NB-Linux to inform our selection of run configurations on the HP workstation.

3.3 Results

The results from our study can be grouped into three broad categories: the impact of WebDriver configurations (RQ1 and RQ2), the impact of system configuration (RQ3 and RQ4), and further optimization (RQ5).

[3] (Chrome + Firefox + HtmlUnit + PhantomJS) * (No Wait + Thread.sleep + Explicit) = 12 + Chrome * AngularWait = 13

[4] Simultaneous Multithreading, a hardware technique allowing the simultaneous execution of two or more logical threads per physical CPU core. Hyper-threading is Intel's implementation of the technology.

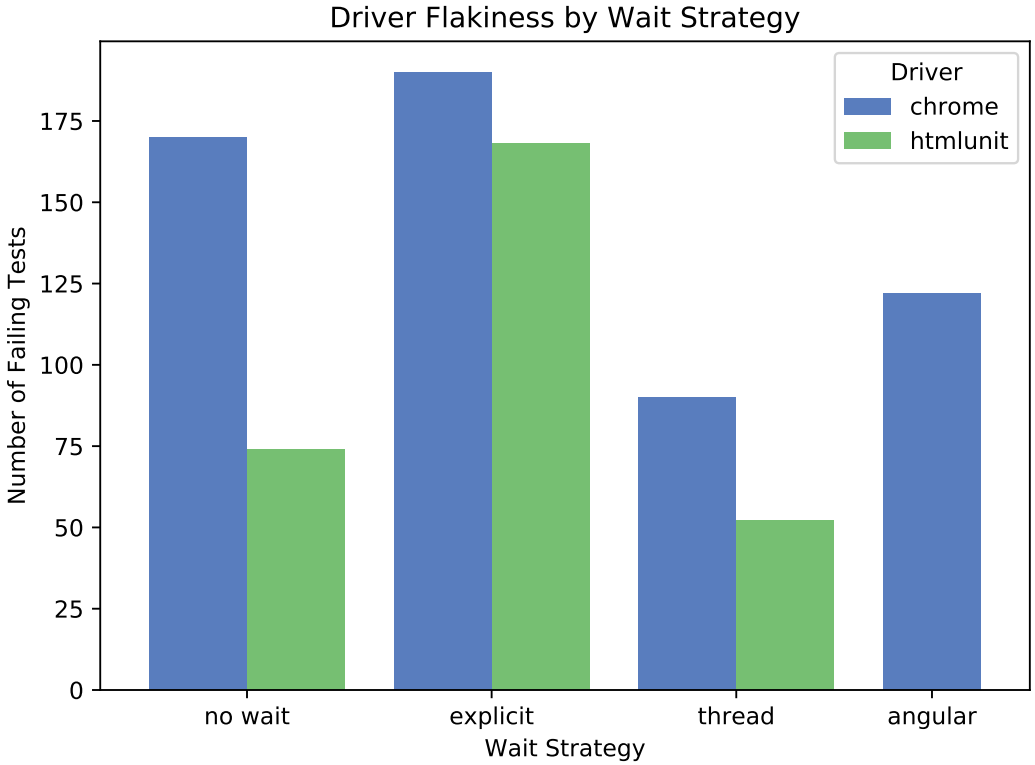


Figure 3.2 Total failing test cases for supported waiting strategies with Chrome and HtmlUnit. Each bar represents one evaluation. Evaluations were run on NB-Linux.

3.3.1 RQ1 & RQ2: WebDriver Configuration

RQ1 addresses the impact of WebDriver choice on test stability and runtime. This corresponds to varying rows A and B of Table 3.1. Even on our fast HP-Linux system, running the test suite with Firefox or PhantomJS took well over an hour a build, making them unsuitable for use in a CI environment.

HtmlUnit, the least featured and least resource-intensive WebDriver used, experienced fewer flaky tests overall than Chrome on NB-Linux. This was consistent across all wait strategies on this system; not once did Chrome deliver a more stable testing experience. These results are shown in Figure 3.2; the Y-axis reports the total number of tests that failed across each evaluation. Because each test was run thirty times, a test that failed in multiple builds will increase the count on each observed failure. Runtime differences between the WebDrivers were minor; an average build time of 31 minutes for HtmlUnit and 35 minutes for Chrome.

RQ1 Summary: *HtmlUnit yields fewer flaky tests than Chrome on NB-Linux, regardless of wait strategy. HtmlUnit is approximately 10% faster than Chrome in terms of test runtime.*

RQ2 addresses the impact of the specific waiting strategy on test flakiness. We evaluated waiting strategy for Chrome and HtmlUnit by running both browsers with all supported waits on NB-Linux. Figure 3.2 presents the total number of test flakes seen in each evaluation.

We see that Thread Waits perform the best for both HtmlUnit and Chrome, while Explicit Waits performed the worst. The performance of Thread Waits was unsurprising – by suspending test execution for a relatively long (5 second) period of time, we give the browser hopefully ample time to catch up to where the test expects it to be. Surprisingly, No Wait manages to perform better than Explicit Wait, particularly so for HtmlUnit, where it resulted in under half of the flakes seen with Explicit Waits. Explicit Waits used the exact same timeout (5 seconds) as Thread Waits, so we expected to see similar results for both. While waiting approach had a sizable impact upon stability, its impact upon runtime was minor on both browsers: no more than an 8%[5] difference was observed between the fastest (no waits) and slowest (explicit waits and thread waits) regardless of browser.

We also acknowledge that RQ1 and RQ2 were evaluated on poor hardware. As we show in the next section, better hardware leads to better performance and less flakiness for Chrome. At the same time, using a slower system for evaluation is valuable as it mimics some students' situations.

We note here that while Thread Waits provide the lowest flakiness score, the number of unique tests impacted by the failures is actually the highest. That is, the Thread Wait failures are more insidious, being both more likely to occur for any test and less predictable. On the other hand, the Explicit Waits and Angular Waits are more predictable; a test failing with an Explicit or Angular Wait is more likely to fail again within the evaluation. Since we want to suggest a wait strategy that has predictable behavior, we move forward with Angular and Explicit Waits when evaluating system configurations.

RQ2 Summary: *Thread waits give the lowest flakiness for both HtmlUnit and Chrome, with Explicit Waits giving the highest.*

[5]Calculated as $(T_{slow} - T_{fast})/T_{fast}$, where T_{fast} is the runtime of the faster build, and T_{slow} is the runtime of the slower build

3.3.2 RQ3 & RQ4: System Configuration

Anecdotally, we expected that slower hardware results in more unstable and slower builds. We consider three types of system configuration: the CPU, the amount of RAM, and the OS managing test processes. We prune our search space by focusing our testing on Angular Waits and Explicit Waits.

3.3.2.1 RQ3a: Increasing CPU

Evaluations of flaky tests in Chrome and HtmlUnit on NB-Linux and HP-Linux (at 4GB RAM) are presented in Figure 3.3. Note that there are no results for HtmlUnit and Angular Waits, as Chrome is the only browser to support them. For both browsers, moving to HP-Linux resulted in substantially fewer flaky tests across the evaluations, regardless of wait strategy. The impact was particularly pronounced for Chrome, with test flakiness falling by over 70%[6] in both configurations, but HtmlUnit still saw a very respectable improvement of 55%.

The average build time over all evaluations was reduced by 80-84% when moving to faster hardware (i.e., D1 to D2). With Chrome, the average build times were reduced from 35 minutes to 7 minutes; with HtmlUnit, they dropped from 31 minutes to 5 minutes. Thus, not only does Chrome give far more stable build results on fast hardware, it does so quickly as well. Fast build times are imperative for CI environments, and better hardware is an easy way to achieve this.

RQ3a Summary: *A faster CPU results in a substantially faster build on both HtmlUnit and Chrome.*

3.3.2.2 RQ3a: Increasing Memory

Figure 3.4 shows the impact of manipulating the amount of available memory for each of three configurations of driver and wait method. The Y-axis shows the sum total of test failures over all 30 builds in an evaluation. The X-axis shows the memory configuration. Each evaluation was run three times (totaling 90 builds of iTrust2 and its test suite), and each data point on the graph is an average over the three evaluations. No significant outliers were observed in any configuration. For example, with *HtmlUnit + explicit*, there were an average of 61 test failures per evaluation with 2GB of memory. This is an average of two per build.

[6]Calculated as $(F_{high} - F_{low})/F_{high}$, where F_{high} is test flakiness from the flakier build, and F_{low} is the flakiness from the less flaky build

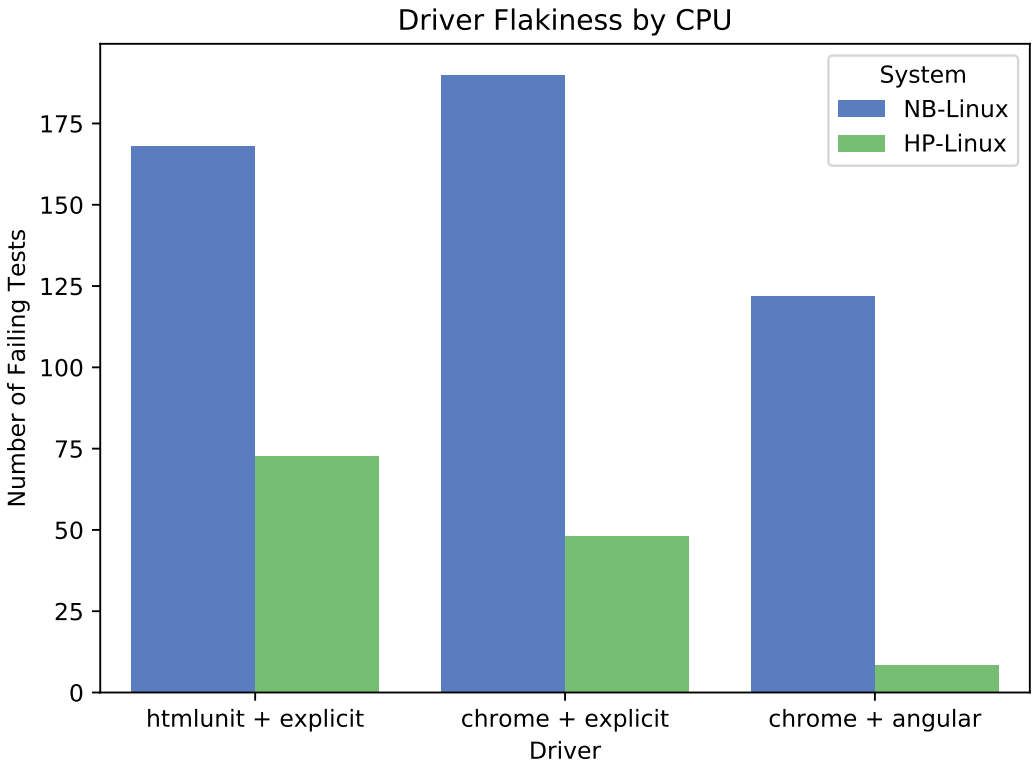


Figure 3.3 Number of failing tests for HtmlUnit with explicit waits and Chrome with explicit and Angular waits for NB-Linux and HP-Linux systems (fixed 4GB memory, increasing CPU performance).

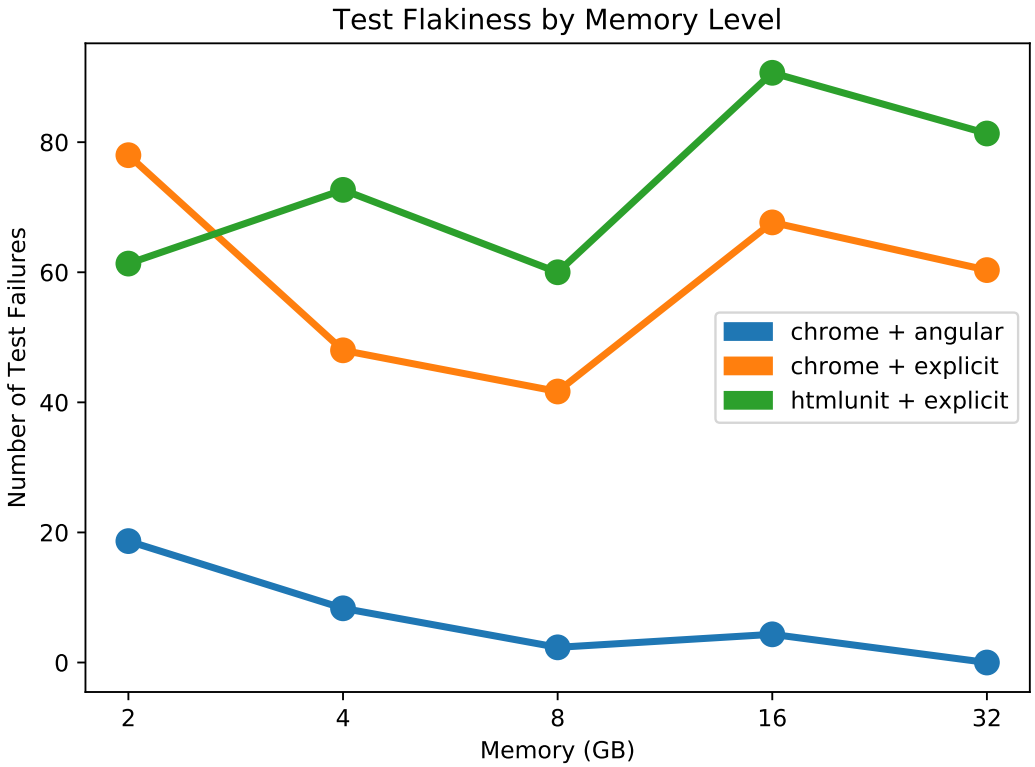


Figure 3.4 Total number of test case failures for all builds in evaluations of Chrome with explicit and Angular waits and HtmlUnit with explicit. All evaluations were run on the HP-Linux platform (D2/E2).

There is no clear trend on flakiness for *chrome + explicit* or for *htmlunit + explicit*, but the trend for *chrome + angular* is decreasing flakiness as memory is increased. If we had cut off the evaluation at 8GB, we would have concluded that *chrome + explicit* has fewer flakes with more memory, but the behavior at and above 16GB is not clear. Further exploration is needed.

HtmlUnit had more flakes, on average, with extra memory. It is the most lightweight browser tested and does not appear require significant resources. However, our evaluations do show the number of test failures varied wildly, suggesting that HtmlUnit may not give consistent results, even with sufficient hardware. This is supported by the data for RQ2, where the naive No Wait approach outperformed the Explicit Wait.

RQ3b Summary: *More memory results in tests that fail less regularly for Chrome + Angular, but not for the other configurations.*

3.3.2.3 RQ4: Host Operating System

RQ3 specifically considers hardware. However, all results are presented for systems running Linux. Next, we turn to RQ4 to generalize past Linux. While the primary goal was to improve the experience on the CI server used for automated feedback and grading projects[7], a secondary goal was to give students a more stable testing environment for their local development.

Section 3.3.2.1 indicates that Chrome on a system with sufficient CPU and memory was the most stable configuration for Linux. However, an attempt to replicate this on Windows was unsuccessful. Our evaluations found 1923 test failures (across a 30-build evaluation) on Windows (E1) vs 12 on Linux (E2) with other factors held constant (A4/B2/C3/D2). The Windows build time was much higher, averaging 44 minutes versus six for Linux. Other browsers and waits also performed poorly on Windows, giving either high runtime, flakiness, or both. For instance, HtmlUnit was fast, but flaky, and Firefox and PhantomJS were still too slow. We turn now to a solution that helped all platforms, but particularly Windows.

RQ4 Summary: *Windows gives worse performance than Linux with respect to flaky tests and runtime given comparable hardware.*

3.3.3 RQ5: Restarts

Attempting to generalize our results to Windows worked poorly, with all configurations resulting in high test failures, unacceptable runtime, or both. We consider now the impact of restarting the browser between each test versus a configuration that does not do so.

By default, each test starts by launching a fresh WebDriver and logging in as a user of the appropriate type. Here, we modified the test suite to share a single Chrome instance across all tests rather than launching the browser between every test. To ensure a consistent starting environment, we introduced a method that would log out of the iTrust2 web portal if a user was logged in and ensured it was run before every test.

Figure 3.5 shows average test execution times when running with and without restarts on each of our systems (NB-Linux, HP-Linux, HP-Windows). Our results show a decrease in build time for every system under test. Even HP-Linux, which already saw the best runtime, saw the tests run 49% faster. Similar to HP-Linux, on NB-Linux we saw a 46% reduction in test execution time. However, HP-Windows saw the biggest proportional improvement, from 44 to five minutes, an improvement of 89%. In addition to faster speeds, none of the platforms had any flaky tests in this configuration. Results here did not generalize to other

[7] Our CI environment runs CentOS Linux 7.5

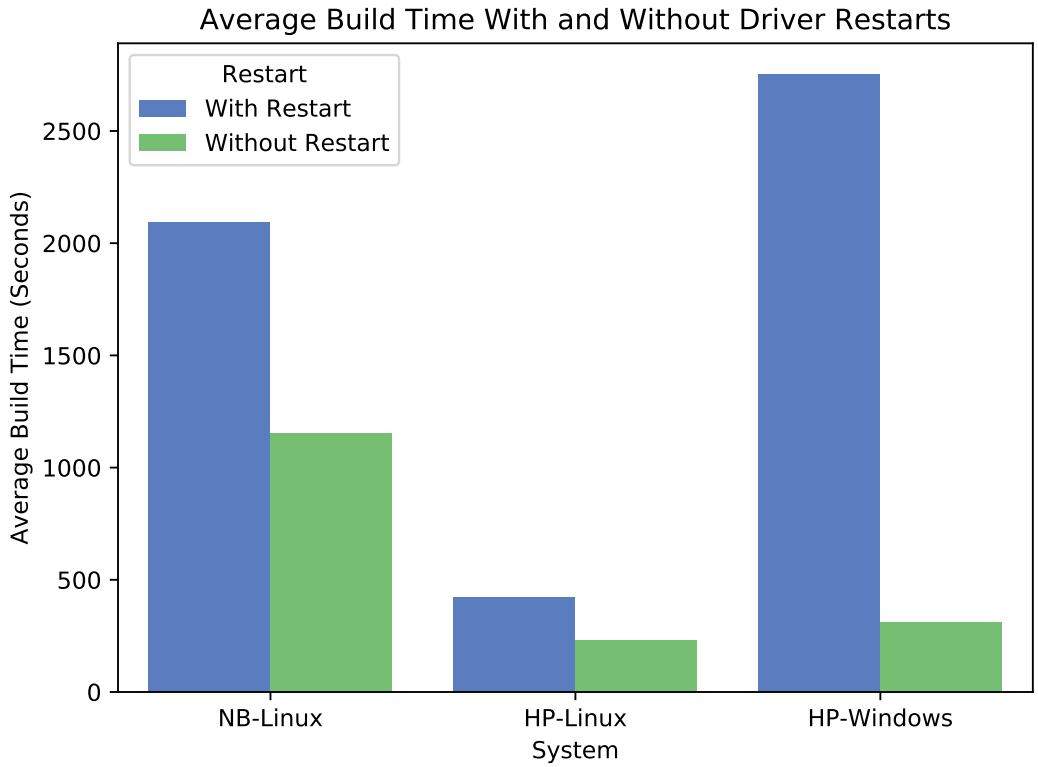


Figure 3.5 Average build time, in seconds, of Chrome with Angular waits on each of the three systems under test. Evaluations **Without Restart** used a single instance of a WebDriver across all tests in a build. Evaluations **With Restart** created a new WebDriver instance before every test.

WebDrivers, where test flakiness and runtime both remained high. For instance, HtmlUnit on Windows remained flaky (giving over a thousand test failures across a single evaluation), while Firefox still was slow (with build time over an hour) on all platform configurations.

RQ5: *By using Angular waits and not restarting the Chrome browser between tests, we get substantially faster performance and no test failures.*

3.4 Discussion and Future Work

Our work provides the first investigation of which we are aware into the effect of Selenium configuration on both reliability and runtime. We end our investigation with an optimal configuration for our use case of running a large Selenium test suite in a CI environment for an undergraduate software engineering class: Chrome + Angular waits + no browser restarts + Linux + fast processor. We have implemented this configuration in our current version of iTrust2 and reconfigured our CI environment accordingly. Knowledge of this

configuration has already proven a valuable resource for the class by providing us with a substantially more stable teaching application that performs much better on our CI environment. However, there is still much left for future work.

3.4.1 Explicit Waits

In Section 3.3.1, Explicit Waits give worse performance in terms of total test failures than any other approach. We presumed that test flakiness resulted from the browser having insufficient time to load a page before the test started interacting with it, so any waiting approach should perform better than none at all. We struggle to explain why performance here was so poor, particularly in relation to the Thread Wait approach, so further investigation is needed.

3.4.2 Hardware and Operating System

We see in Figure 3.4 that the number of test failures increases when going from 8GB to 16GB. Every other memory increase for Chrome, and all but one for HtmlUnit, resulted in a decrease in number of failing tests. While the three evaluations performed at each configuration suggests this is not due to random variation, we cannot be certain without further exploration.

3.4.3 WebDrivers

We limited our search to WebDriver implementations which run on all major operating systems. It remains open as to whether an untested driver performs better on any of the operating systems included in the study. If this is the case, future applications may benefit by detecting and using the best driver for the platform on which they are running.

3.5 Threats to Validity

The threats to validity of our experiments are as follows:

3.5.1 Conclusion Validity

Due to the time taken to run experiments, most of the conclusions on test flakiness in this paper were drawn from a single 30-build evaluation (the conclusions to RQ3b were drawn from an average across three evaluations). This may not be sufficient to observe all tests

that could be flaky. However, it does represent a number of builds similar to the number of teams in the software engineering course each semester. Still, extending the number of builds per evaluation may lead to different conclusions.

3.5.2 Internal Validity

In our experiments, we were careful to vary only a single factor at a time (WebDriver, wait strategy, CPU, memory, operating system, or browser restarts). We also controlled for other factors, such as software versions, memory and disk performance, and internet connection. All testing was performed in an automated and repeatable manner. We thus believe that the differences we observed come from the experimental factor that was varied.

3.5.3 Construct Validity

We assume that test flakiness occurs because the test attempts to interact with an element that has not yet appeared on the page; other factors, such as test order, may also impact correctness. However, we eliminated all observed test case flakiness solely by changing Selenium settings, and without changing the order in which the tests are run, which suggests that our tests do not face this issue.

3.5.4 External Validity

Our study has focused on a single artifact, iTrust2. While the Spring and AngularJS frameworks used in iTrust2 are widely used, we have not attempted to replicate our results past iTrust2. We have, however, performed some generalization, applying our optimal configuration to two expanded versions of iTrust2 that were developed in parallel to our work. Our solution, when applied, eliminated all test flakiness and improved runtime. Replicating our work on projects other than iTrust2 would see if the results generalize further.

3.6 Conclusions

Our work has implications for automated testing of web applications using Selenium. We evaluated four WebDrivers and four different approaches for waiting on elements to appear on a page. We demonstrated that there are differences between the studied WebDrivers: HtmlUnit driver performs best where system resources are heavily constrained and the browsers must be run in their default configuration, while Chrome works best on faster systems or where configuration can be optimized. We demonstrated that hardware has a

significant impact upon the runtime and reliability of a test suite and that a faster CI environment makes a meaningful difference. Our work has already contributed a substantially more stable teaching application to our undergraduate software engineering course, giving students confidence in the tests they write and the results they see.

CHAPTER

4

AUTOMATED REPAIR OF STUDENT-AUTHORED SQL QUERIES

This study[1] explores the types of mistakes that students make when first learning SQL, investigates the possibility of applying automated program repair to SQL queries written by students, and evaluates whether students find automated repairs to be understandable. It does so to evaluate whether **automation**, in particular automated repair, can be applied to student-authored programs, and demonstrates that in cases where peer instruction is not available, students may be able to **learn from** automated repairs. Additionally, an understanding of the types of mistakes that students make can help instructors produce **more effective instructional materials** that counter common misconceptions.

> Satisfies part of thesis: Using **software engineering automation** and survey techniques in computer science education results in **improved student learning outcomes**, early prediction of struggling teams, and **more effective instructional materials**.

[1] This study was published in substantial part as Presler-Marshall, K., Heckman, S. & Stolee, K. "SQLRepair: Identifying and Repairing Mistakes in Student-Authored SQL Queries". *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 2021, pp. 199–210.

4.1 Study Rationale

A common problem in education is that students may struggle to identify how to get started on or how to finish a problem they are working on. While some confusion is good for students, and is a necessary component of learning, students who continue to struggle and remain confused are more likely to end up demoralised than they are to learn from the experience [D'M14]. If instructor assistance is available, this is a problem that can often by overcome by a back-and-forth discussion. This approach is best exemplified by Socratic teaching techniques [Nel80; Del16], getting students to articulate what they have accomplished so far, where they got stuck, and what they have tried to do next but are struggling to do effectively. However, in large and growing classes, there may not be sufficient teaching staff resources available to offer this. Consequently, automation has a role to play in computer science education, where students can receive at least some support regardless of whether teaching staff resources are available.

The most common automated educational support offered are *hints*, suggesting approaches for students on approaches that they can try, usually in order, to get closer to get closer to a solution. Hints can be provided in two formats: a relatively static list of steps to try, or more dynamic approaches, exemplified by Intelligent Tutoring Systems (ITSs) [Bar10; Eag12; Pri18; And85; Cro18]. The breakthrough of ITSs is that they customise hints based on where a student currently stands: hints are generated based on how *past students* have gotten from the current challenge to a working solution [Bar10; Eag12]. These approaches typically take "big data" approaches, and use information on how problems have been solved in the past to offer feedback on how students might try to solve them now. Consequently, they face a "cold start problem" – with insufficient data, they are unable to offer students any suggestions on what to try. In this study, we explore the use of automated program repair as a way of providing students with feedback and guidance as they learn to solve SQL problems, which sidesteps the problem by generating a solution on the fly rather than relying on past data. We demonstrate that automated repair is capable of fixing the types of mistakes that students make when first learning SQL, and demonstrate that the repair quality is sufficient that it has the potential to be used as an educational technique.

This study makes the following contributions[2]:

- quantitative and qualitative classifications of the types of errors introduced by beginning SQL programmers,
- a tool capable of repairing 29.1% of the observed errors in SQL queries,

[2]Our tool, datasets, and study tasks are available at `http://github.com/kpresle/sqlrepair`

- a benchmark dataset of realistic SQL errors gathered from undergraduate computer science students, and
- a demonstration that tool-repaired SQL queries are equal in understandability to human-written queries.

4.2 Introduction

Understanding how beginners work with a new programming language and the types of mistakes that they make can help instructors better tailor their lesson plans to avoid previous pitfalls [Chr19; Ani19]. We consider SQL, a widely-used language for interacting with relational databases. SQL is taught in many undergraduate computer science programs [Tai19; Mig20], but may not be part of the core curriculum. It is regularly used by professional and amateur developers alike [Sos], including those with little formal computer science background [Har15a; Bau15].

While the types of mistakes that students make when working with languages such as C and Java are relatively well studied [Bro14a; Alt15; Bro17], we know less about mistakes made in special-purpose languages such as SQL. We seek to understand the types of mistakes that undergraduate students, who are relatively familiar with Java, make when working with SQL. Understanding these mistakes can help educators ensure that they have the resources necessary to support computer science students and end-user programmers alike, which may include automated support [Gul18].

In addition to an analysis of student mistakes, we propose a tool, SQLRepair, which can automatically fix some of the errors students introduce.[3] While there are tools for automated repair of programs in languages such as C and Java [Jia18; Lon16; Mec16; Wei09], to the best of our knowledge, no existing techniques attempt to repair errors in SQL queries. Our repair process first attempts non-synthesis repair based on a predefined ruleset. As needed, it uses a satisfiability modulo theory (SMT) solver [DM08] to further synthesize repairs.

We frame our work around the following research questions:

- **RQ6**: *What types of mistakes do beginners make when working with SQL?*
- **RQ7**: *How well can SQLRepair fix errors introduced by beginning SQL programmers?*

[3]We adopt terminology used in existing work on SQL education: students make a *mistake* while solving a problem, introducing one or more *error*s into the query. Note that this diverges from terminology frequently used in testing literature where the term would be *fault* instead of *error*. We choose *error* for consistency with existing work.

- **RQ8:** *Do students find SQLRepair-repaired queries to be more understandable than queries written by other students?*

To answer our research questions, we conducted an empirical evaluation to understand student mistakes (RQ6), evaluate SQLRepair's ability to repair the errors in the student-written queries (RQ7), and determine the repair quality (RQ8).

Students in two undergraduate computer science courses at NCSU were given a short introduction to SQL and then asked to write queries to solve problems associated with a sample database. For each problem, students were provided an example (`source`, `destination`) table pair that demonstrated the desired transformation (similar to programming by example (PBE) techniques) [Gul16] and were asked to write a SQL query that would complete the transformation. Incorrect queries were followed by additional examples (up to three) to demonstrate the intended behavior. Any SQL query that did not correctly solve the problem was analyzed for errors and considered a candidate for repair. Students were then asked to evaluate up to four human-written or tool-generated queries, judging each for understandability.

4.3 Methodology

To provide a dataset for analyzing mistakes (RQ6) and evaluating SQLRepair (RQ7, RQ8), we conducted a two-phase study with students from two undergraduate computer science courses.

In Phase 1, we conducted a study with students from the Summer 2019 offering of a 2nd-year Java programming course. This phase demonstrated the viability of our approach, gave us preliminary data for RQ6 and RQ7, and motivated additional enhancements to our tool. In Phase 2, we put repairs produced by SQLRepair directly in front of students to understand whether our tool-generated repairs are understandable (RQ8). Students were recruited from the Fall 2020 offerings of a 2nd-year Java programming course and a 3rd-year Software Engineering course. This study was approved by the NCSU IRB Office as protocol #19062.

4.3.1 Phase 1

We collected a dataset of SQL queries written by introductory programmers to understand the type of mistakes students make by analysing the errors they introduce, and ascertain SQLRepair's ability to repair the errors.

Table 4.1 Major concept in each problem and the total number of `(source, destination)` tables in the problem specifications.

Problem	Major Concept	Number of Table Pairs
1	Single-condition select	3
2	Select with projection	2
3	Inequality	3
4	Projection and inequality	2
5	Compound select	2
6	Compound select with AND	2
7	Distinct	2
8	Ordering	2
9	Joins	2
10	Grouping	2

4.3.1.1 Design

Eighteen students were given a lecture on SQL functionality and syntax, including compound select queries, various datatypes, `JOIN`, `COUNT`, `DISTINCT`, and `GROUP BY`. Students were informed that we were interested in studying how beginners work with SQL and the types of mistakes that they make. Next, they were given a ten-problems to solve; each problem had a `(source, destination)` table pair and students were asked to write a SQL query that would accomplish the transformation. Each problem had two or three pairs of `(source, destination)` tables that acted as test cases that must be passed simultaneously for the query to be considered correct. The major concept of each problem is shown in Table 4.1. For example, the major concept introduced in Problem 10 was grouping, and there were two sets of `(source, destination)` table pairs for evaluating the query. The problems and data used were based on the UMLS dataset, a health and biomedical vocabulary dataset made available free-of-charge by the NIH, which was chosen for offering a large amount of structured data [Bod04].

Students were shown one `(source, destination)` table pair at a time. Each student received a paper handout that contained the first pair for each problem. To avoid learning effects, the problems were given in a random order. Students submitted their queries into a web application. If the application detected that the first pair had been solved successfully, the query was then tested against subsequent pairs. If a query failed a subsequent pair, that pair was revealed to the student. Students spent approximately 40 minutes working on all problems and were reminded every ten minutes to move on to the next problem if they had been stuck for more than five minutes. Students were compensated with participation credit.

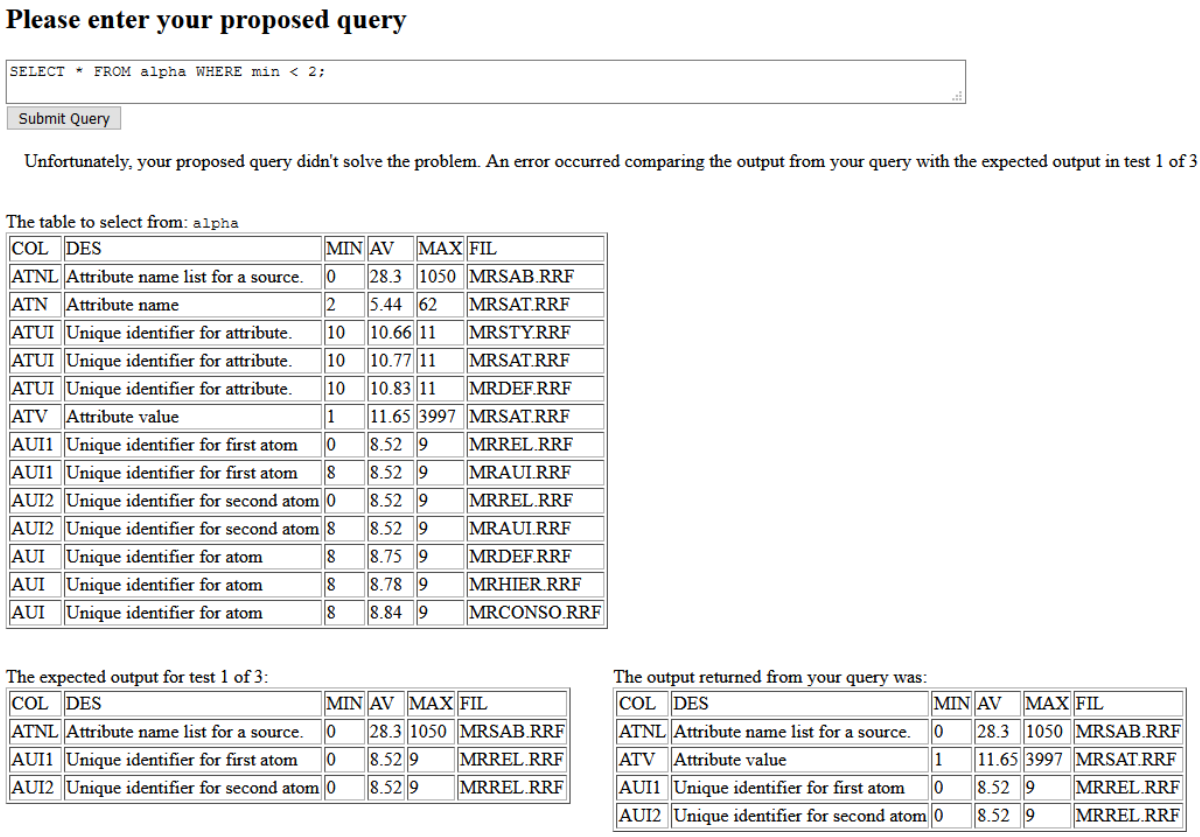


Figure 4.1 The application for students to submit SQL queries.

The web application is shown in Figure 4.1. In this example, a student submitted the query `SELECT * FROM alpha WHERE min < 2;`, which was incorrect, as communicated through the message, *"Unfortunately, your proposed query didn't solve the problem . . . "*; the actual output from executing the query is shown alongside the expected output (destination table). If the query produces the correct output for all table pairs, the student was congratulated and told to move on to the next problem. The application records the participant's unique ID, submission time, proposed query, and whether the problem was solved correctly or not. At the end of the study, students completed a brief demographics survey, which asked questions such as their prior programming experience, their experience with SQL, and whether they had any comments on the introduction to SQL or the problems themselves.

4.3.1.2 Participants

We recruited participants from a 2nd-year Java programming course (CS2). CS2 is the second computer science course taken by majors and minors at NCSU. By this point, students are

exposed to programming in Java. Eighteen students from the Summer 2019 offering of CS2 participated, but only 12 students submitted one or more SQL queries as part of the study. Of the 12 active participants, three identified as female. Ten students said they had three or fewer years of programming experience (min: 0, max: 8, average: 2.6) and none had more than a year of professional programming experience. One student reported prior database experience.

4.3.1.3 Dataset

We collected 362 SQL queries written by 12 different students. Of these, 35 were correct. Of the 327 incorrect queries, 124 had syntax error(s) and 203 had semantic error(s). Students submitted between 7 and 65 queries (average: 32.2, median: 28.5). Students made between one and 21 attempts per problem (average: 4.6, median: 3.5) and attempted between two and ten problems (average and median: 6.5).

4.3.2 Phase 2

In Phase 2, we build on Phase 1 and further evaluate SQLRepair by putting repaired queries directly in front of students to assess query quality.

4.3.2.1 Design

Phase 2 was similar to Phase 1 in that students were given the same introductory SQL lecture and the same set of problems to solve. However, some changes were made to the study format and content, as follows:

Due to the COVID-19 pandemic, Phase 2 was performed online via Zoom. After the introduction to SQL and the study, each participant was assigned to an individual breakout room to work in for the remainder of the session. To ensure that each participant was engaged and working, the first author rotated between each room at least once to answer any technical questions that arose. Students could also use Zoom's "Ask for help" functionality to request assistance.

While the study problems were identical to Phase 1, we made operational changes to suit the online format:

- Instead of a paper handout, each student received the randomly ordered problems as a PDF.
- Instead of students entering their participant ID manually, the web application automatically included each student's random ID in each problem submission.

- The post-study demographics survey was converted from a paper handout to a Google Form. Students were asked to include their participant ID in their submission.

Additionally, after composing queries for a problem, students evaluated the understandability of several solution queries for that problem (Section 4.3.2.2).

4.3.2.2 Evaluating SQLRepair

We wanted students to assess the understandability of tool-repaired queries by comparing them against human-written queries. As a majority of software engineering effort is spent on maintenance [Gla02], we consider understandability, as a proxy for ease of maintenance, to be paramount. We seek a minimally-invasive way of gathering information on students' program comprehension as they evaluate queries without the feeling of being watched [Wil01]. Thus, we opt for short surveys deployed after each question and separately at the end of the study.

First, we populated a database with data from Phase 1, giving us 29 unique correct queries and 19 unique repaired queries SQLRepair produced from incorrect queries. Next, we modified the web application to use SQLRepair to attempt to repair incorrect queries that students wrote during the study. We did this through brief post-problem surveys: after solving each problem, students were asked to rate the understandability of up to four different queries using a modified Likert scale, with 1 indicating the query was very difficult to understand and 7 that it was very easy to understand. As an alternate workflow, after making at least five attempts at a problem over at least five minutes, students were presented with an "I'm tired of this problem" button. Upon clicking it, they would be given the voting options shown, despite having never solved the problem correctly.

The four possible queries presented to students were:

- **MyCorrectQuery**: A correct query written by the student (available if they solved the problem correctly).
- **MyRepairedQuery**: A repair of an incorrect query written by the student (available if they got the problem wrong at least once, and SQLRepair was able to repair one of their queries.[4])
- **OtherCorrectQuery**: A correct query written by someone else (a participant from Phase 1 of the study; a query from this category was always available).

[4]Incorrect queries were considered starting with the most recent incorrect submission, and repairs were attempted until a query was successfully repairable, or, to ensure sufficient responsiveness of the web application, the repair process had failed ten times.

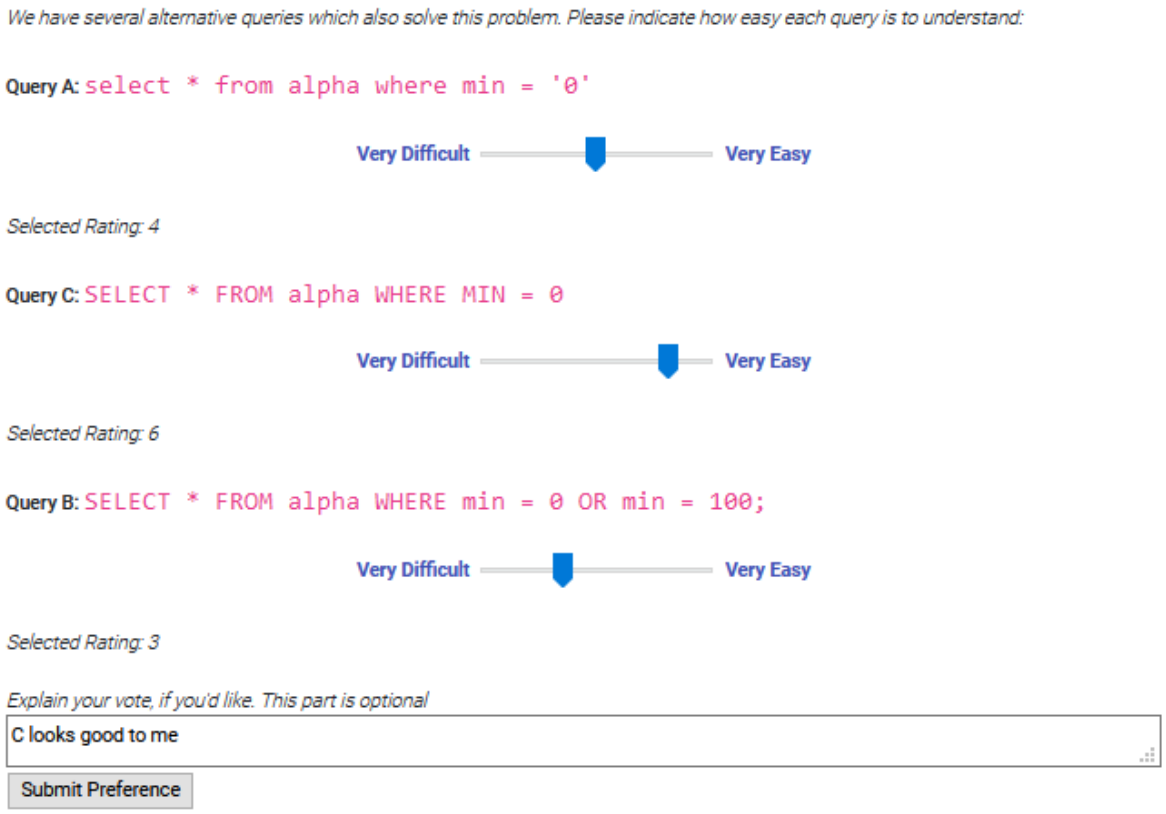


Figure 4.2 An example of how students voted on the understandability of queries.

- **OtherRepairedQuery**: A repair of an incorrect query written by someone else (a participant from Phase 1 of the study; a query from this category was always available).

The queries were labeled A through D, and presented in a random order. An example with three queries is shown in Figure 4.2. For queries written by others, query selection was pseudo-random: each query was associated with a count of how many times it had been shown to a student for voting, and each time a query was needed for voting, the application selected the query with the smallest vote count. Identical queries were consolidated (for instance, if the first and fourth queries were identical, the query would only appear once).

4.3.2.3 Participants

In Fall 2020, we distributed recruitment emails to students in two undergraduate courses: CS2 and a 3rd-year Software Engineering course (SE). SE is a fifth-semester course, and by this point, students have been exposed to Java, C, x86 assembly, and JavaScript. Additionally, prior to our study, the SE students received an in-class lecture on SQL, although not hands-on practice with it. Students in both classes were invited to sign up for one of four two-hour

Table 4.2 A breakdown of all of the queries submitted.

	Course		
	CS2	SE	**Total**
Correct	157	94	251
Syntax Error	680	137	817
Semantic Error	1,185	529	1,714
Total	2,022	760	2,782

virtual lab sessions held. In all, 104 students signed up to participate in a session; 71 students attended and participated for at least thirty minutes. The first of four sessions was used as a pilot for the improved SQLRepair tool and new format. Feedback was collected and data from this group was discarded. Participants from Phase 1 could not participate in Phase 2.

Seventy-three students from CS2 signed up; 46 ultimately participated. Thirty-one students from SE signed up; 24 ultimately participated. After discarding data from the pilot study, we retained data from 33 CS2 and 19 SE students. Students in CS2 reported up to seven years of prior programming experience (average and median: 2); students in SE reported up to eight years (average: 5, median: 4). Sixteen participants from CS2 and five from SE identified as female.

4.3.2.4 Dataset

We collected 2,420 SQL queries from 52 students. Of these, 216 were correct; of the 2,204 incorrect queries, 693 had syntax error(s) and 1511 had semantic error(s). Students submitted between 1 and 118 queries (average: 42.4, median: 37.5). Students attempted between 1 and 10 problems (average and median: 7) and made between 1 and 50 attempts per problem (average: 6.4, median: 4).

The 33 students from CS2 submitted 1,660 queries. Of these, 122 were correct; of the 1,538 incorrect queries, 556 had syntax error(s) and 982 had semantic error(s). Students submitted a median of 41 queries (max: 118) and attempted between 1 and 10 problems (average: 6.8, median: 7.5).

The 19 students from SE submitted 760 queries. Of these, 94 were correct; of the 666 incorrect queries, 137 had syntax error(s) and 529 had semantic error(s). Students submitted a median of 36 queries (max: 79). Students attempted between 1 and 10 problems (average: 7.4, median: 7).

4.3.3 Data Summary

A summary of all of the queries collected across both phases of our study, and their correctness or error category, is shown in Table 4.2. We performed a Mann-Whitney test between the two CS2 courses (Summer 2019 and Fall 2020 from Phase 1 and Phase 2, respectively) looking for significant differences on successes per problem. Our analysis revealed that the differences between them were not statistically significant ($p = .31$), so the data from both were combined for further analysis. The data from SE remained significantly different ($p = .0016$) and was kept separate.

4.3.4 Analysis

We use the errors that students introduce into SQL queries they write as a proxy for the mistakes made while solving the problem. To identify student mistakes for RQ6, we executed each student-written query against the source and destination tables using a MySQL 5.7 database. Any query where the database returned an error message was considered to have syntax error(s).[5] For the remaining queries, we compared the actual output table to the expected output for the problem. When they were different, the query was considered to have semantic error(s).

To identify syntax error categories, we manually grouped queries with similar errors together. For example, students submitted the queries:

```
SELECT CUI1, RUI FROM bravo where REL='RO', 'SY';
SELECT CUI1, RUI, FROM bravo WHERE CUI2 == C0364349;
```

Both queries have an extra comma, so were grouped together. We continued this process for all queries with a syntax error. If there were three or more queries in a category, we gave the category a name. Categories with fewer than three were grouped together into a miscellaneous category.

For semantic errors, we manually investigated the query and the output table it produced and grouped together queries with similar errors. For example, students submitted the queries:

```
SELECT LAT FROM juliett;
SELECT LAT, STT FROM juliett
```

[5]This understates the number of SQL syntax errors as MySQL 5.7 supports functionality not part of the official SQL specification, such as wrapping strings in double quotes or using operators such as `&&` instead of `AND`. We are aware of no databases which follow the SQL specification exactly.

item	price	quantity	country	seller
apples	7	500	US	Joe's Fruits
bananas	3	400	MX	Nancy's Produce
oranges	11	300	MA	Ahmed's Fruits
grapes	1	200	US	Raj's Vinyard

item	price	quantity	country
grapes	1	200	US

Figure 4.3 Example source (top) and destination (bottom) tables.

Both queries return only a subset of the columns expected (`LAT, STT, ISPREF`) so they were grouped together. A miscellaneous category was created by grouping together all categories with less than three queries.

A single query can contain multiple errors (for instance, a broken operator and unquoted string literal) so some queries were counted for multiple categories. However, when classifying errors, a single query could be counted towards either the syntax error category *or* semantic error category, but not both.

4.3.5 SQLRepair

SQLRepair follows the correct-by-construction approach to automated program repair [Gaz17]. The subset of supported SQL includes queries with compound `WHERE` clauses, integer and string datatypes, `ORDER BY`, and `DISTINCT`.

To explain how SQLRepair builds constraints from the `(source, destination)` tables and SQL query, consider the following example. A user of SQLRepair submits the source and destination tables shown in Figure 4.3 and the SQL query `SELECT * FROM fruitSellers WHERE country=US && quantity < 800`. SQLRepair proceeds in two steps: (1) non-synthesis repair, and (2) synthesis repair.

4.3.5.1 Non-Synthesis Repair

SQLRepair attempts three types of non-synthesis repair over the following types of errors: operator mismatches that result in parse errors, column mismatches that can cause an otherwise correct query to be incorrect, and string repair where a string literal shows up without proper quotes.

4.3.5.1.1 Operator Mismatch

SQLRepair replaces any C/Java-style operators in the provided query with their SQL equivalent. For example, C/Java use `==` for equality checks and `&&` for logical AND. SQL uses `=` and `AND`, respectively. SQLRepair thus replaces operators such as these. In the example, `&&` is replaced with `AND`, giving us the query, `SELECT * FROM fruitSellers WHERE country=US AND quantity < 800`.

4.3.5.1.2 Column Mismatch

SQLRepair attempts to repair any issues with the column list prior to the `WHERE` clause. When a column does not exist, a syntax error occurs. However, column mismatch does not always start with a syntax error. In the the running example, the source table has five columns while the destination table only has four; however, the SQL query has a `SELECT *` clause, SQLRepair detects and fixes this mismatch. Thus, the query is updated to `SELECT item, price, quantity, country FROM fruitSellers WHERE country=US AND quantity < 800`. In addition to correcting the column list following `SELECT`, SQLRepair can also rename columns to match the destination table using `AS`.

4.3.5.1.3 String Repair

SQLRepair attempts to repair any issues where a string literal is present in the query either unquoted or quoted incorrectly. SQL requires strings to be surrounded with single quotes. Thus, SQLRepair removes double quotes and surrounds what appear to be unquoted string literals with single quotes. The query is thus updated to `SELECT item, price, quantity, country FROM fruitSellers WHERE country='US' AND quantity < 800`.

Resolving operator mismatch, column mismatch, and fixing strings resolves syntax errors, but often synthesis is needed to fully correct the semantic errors.

4.3.5.2 Synthesis Repair

SQLRepair uses a SMT solver, Z3 [DM08] to synthesize parts of a query in need of repair [Gaz17]. The synthesized parts, or patches, are composed of individual constants, operators, and column names. The `(source, destination)` tables are used as test cases that must be simultaneously satisfied for a query to be successfully patched.

For each query, SQLRepair builds a system of constraints to represent the query logic. Given a set of example `(source, destination)` tables E and a SQL query q, SQLRepair

checks that: $\forall e \in E, q \wedge source_e \rightarrow destination_e$. If the equation evaluates to true, Z3 returns SAT and q is correct; otherwise q is incorrect and a candidate for repair.

If q is a repair candidate, SQLRepair inserts holes into q, for example by replacing a constant with `CONST_i`, forming q', and provides q' to the solver. If q' is repairable by SQLRepair, Z3 returns SAT and the solver has identified values for the holes in the satisfiable model. If q' is not repairable by SQLRepair, Z3 returns $UNSAT$. SQLRepair supports five types of synthesis repairs. After each repair stage, the process terminates if a successful repair can be made. Repairs are performed in the following order:

4.3.5.2.1 Constant Synthesis

For constants that are compared to columns, SQLRepair replaces each constant in the `WHERE` clause with `CONST_i`. If a query contains `CONST_1 OP_1 CONST_2`, SQLRepair does not replace either of the constants. Synthesis is supported for integers and strings, although synthesized strings must be exact matches without wildcards.

4.3.5.2.2 Operator Synthesis

SQLRepair replaces each operator in q's `WHERE` clause with `OP_j`. SQLRepair supports synthesising operators for both string and integer types. SQLRepair supports `=` and `!=` when dealing with strings, and `=`, `!=`, `>`, `>=`, `<`, and `<=` when dealing with integers.

4.3.5.2.3 Column Synthesis

SQLRepair inserts holes for the columns. For example, a query $q =$ `...quantity OP_1 CONST_1` is replaced with $q' =$ `...COL_1 OP_1 CONST_1`, where `COL_1` represents one of the columns in the source table. If SQLRepair fails to find a solution, column synthesis is repeated for each subclause in the original query, in order.

4.3.5.2.4 Clause Removal

SQLRepair will remove subclauses one at a time to attempt a solution. For a query with `n` subclauses, if a correct solution cannot be found for `n` subclauses, but can be found with `1...n-1` subclauses, SQLRepair will remove subsequent clauses that impede correctness. If this step fails, the removed clauses are added back to the query before proceeding with *Clause Synthesis*.

4.3.5.2.5 Clause Synthesis

Some queries require additional `WHERE` clauses or conditions. In this case, SQLRepair functions most similarly to Scythe [Wan17], and will synthesize new subclauses. Suppose in the column synthesis step, SQLRepair inserts holes such that $q' = \ldots$`WHERE COL_1 OP_1 CONST_1`, but is not able to find any columns, operators, and constant values that result in a solution. At this point, SQLRepair attempts to make a repair by synthesizing in a new subclause. More formally, SQLRepair will take a clause …`WHERE COL_1 OP_1 CONST_1` from the previous step, and add a new subclause, giving $q' = \ldots$`WHERE COL_1 OP_1 CONST_1 BOP_1 COL_2 OP_2 CONST_2`, where `BOP_1` is a binary operator (`AND` or `OR`) and `COL_2 OP_2 CONST_2` represents the abstracted form of a new subclause to be synthesized. If values can be found, they are inserted into the query, and the repair is complete. If no such values can be found, the query will be expanded again. This process repeats until either a solution is found, or the query reaches the maximum of five subclauses, at which point the process is aborted and the repair is marked as failed.[6]

In the example, after repairing *Operator Mismatch* and *Column Mismatch* and performing *String Repair* the query: $q =$`SELECT item, price, quantity, country FROM fruitSellers WHERE country='US' AND quantity < 800` is incorrect. Thus, SQLRepair creates: $q' =$`SELECT item, price, quantity, country FROM fruitSellers WHERE country = 'US' AND quantity OP_1 CONST_1`. When Z3 returns SAT, SQLRepair uses the satisfiable model to replace `OP_1` $\rightarrow !=$ and `CONST_1` $\rightarrow 500$, creating a correct query.

4.3.5.3 Analysis

To identify queries to repair for RQ7, we considered any query that had a syntax error or semantic error. We report on what SQLRepair can fix from Phase 1 and Phase 2. Unlike with error classification, as discussed in Section 4.3.4, a repaired query could be counted towards both the synthesis and non-synthesis categories, depending on precisely what repair operations were performed.

With RQ8, we seek to understand the quality of the repairs produced by SQLRepair. Students in Phase 2 were shown multiple queries simultaneously (see Figure 4.2) and asked to rate the understandability of each one on a seven-point Likert scale. Because students were shown multiple queries simultaneously, we are interested in the relative ratings given to each one. Thus, we perform a series of paired Mann–Whitney U analyses to understand how queries from one category compare to queries from another category.

[6] In our experiment, the maximum number of added clauses in a successfully patched query was three.

Table 4.3 Classifications of syntax errors introduced by students across both phases.

Error Type	CS2 Number (%)	SE Number (%)	Total Number (%)	Example
Broken operator	188 (27.6%)	29 (21.2%)	217 (26.5%)	`SELECT RUI FROM bravo WHERE CUI1 == 'C0000039'`
Column reference error	118 (17.3%)	16 (11.7%)	134 (16.4%)	`SELECT DISTINCT CUI FROM juliett, india WHERE juliett.CUI = india.CUI`
Quotes on strings	87 (12.8%)	40 (29.2%)	127 (15.5%)	`SELECT * FROM foxtrot WHERE TTY = PT`
Incomplete query	91 (13.4%)	6 (4.4%)	97 (11.9%)	`SELECT DISTINCT WHERE MRRANK_RANK < 384;`
Wrong order	83 (12.2%)	14 (10.2%)	97 (11.9%)	`select LAT, STT, ISPREF distinct from juliett`
Table reference error	68 (10.0%)	16 (11.7%)	84 (10.3%)	`SELECT STT, ISPREF FROM juliett WHERE india.CUI = juliett.CUI`
Extra commas	62 (9.1%)	13 (9.5%)	75 (9.2%)	`SELECT CUI1, RUI, FROM bravo WHERE CUI2 = 'C0364349'`
Missing commas	20 (2.9%)	9 (6.6%)	29 (3.5%)	`SELECT RSAB TFR CFR FROM delta WHERE TFR > 470`
Miscellaneous	38 (5.6%)	1 (0.7%)	39 (4.8%)	`SELECT CUI, STN, TUI from hotelORDER BY TUI DESC`

4.4 Results

In this section, we present quantitative and qualitative results showing the types of errors students introduce (RQ6), the types of repairs by SQLRepair (RQ7), and the repair quality (RQ8).

4.4.1 RQ6: SQL Mistakes

The students in SE were more successful at solving the problems than the students in CS2 (see Table 4.2). Among queries submitted by SE students, 12.4% (94 of 760) were correct, compared to 7.8% (157 of 2,022) from CS2 students. Additionally, perhaps due to exposure to more programming languages, the SE students introduced syntax errors at a lower rate (20.6% of all queries with errors, vs. 36.4% among CS2 students). We performed a test of two proportions and found that the difference in overall success rates between groups was statistically significant ($p < .001$). For this reason, results for students from each course are presented separately.

Table 4.3 and Table 4.4 show the syntax and semantic errors students introduced, respectively. Because individual queries can contain multiple errors, a query can be counted in more than one category. Each row in the table shows one of the categories, how many queries had errors of that type, a corresponding percentage, and a representative example from the category. For example, the first row of Table 4.3 is our syntax error category of a *Broken operator*; we saw 217 of these, representing 26.5% of the 817 queries with syntax errors. The query `SELECT RUI FROM bravo WHERE CUI1 == 'C0000039'` was placed into the *Broken operator* group because the query uses a `==` where it should have used a `=`.

We notice similarities between our categories in Table 4.3 and those reported by Taipalus and Perälä [Tai19]; for instance, we both observed *a column reference error*, *wrong ordering* of SQL keywords, and *miscellaneous syntax errors*. Likewise, there is overlap between our categories and those of Ahadi, et al. [Aha16]; the *column reference error* rank high in both lists, and their general *syntax error* category appears similar to our *broken operator* category. Unfortunately, because they do not offer examples of their categories it is impossible to map our categories to theirs precisely. The types of semantic errors that we saw are shown in Table 4.4. The most common issue was *Wrong subclauses in the* `WHERE` *clause*; this is the first row in the table and was observed in 1,234, or 72%, of queries. The prevalence here indicates that students had difficulty precisely describing the rows they wanted to include. A *Missing or extra operator* was the second most common issue, particularly among students in CS2. In contrast to our categories of semantic errors, which represent cases where the

analyzed query returns an incorrect result, Brass and Goldberg [Bra06] focus on queries that are correct but complicated or difficult to read. There is, however, overlap between our categories and those of Taipalus and Perälä [Tai19], such as a *missing join*. In addition, their category of *duplicate rows* is similar to ours of a *Missing [or extra] operator*.

The breakdown of successful queries and submitted queries on each problem is shown in Table 4.5. We note that certain problem types proved to be particularly challenging. For example, Problem 9, which necessitated use of a join, was widely attempted (with a total of 350 attempts from 42 different participants) but was solved correctly by only a single student. Problems involving compound `WHERE` clauses (Problems 5 & 6) proved difficult as well, with less than a third of students managing to solve each one correctly. The lower success rates on these problems compared to single-condition selects (Problems 1 & 2) likewise suggests that students struggle with understanding the interactions between multiple columns.

RQ6 Summary: *Students made eight main types of syntax mistakes, including misusing an operator and ambiguity with referenced columns, and seven main types of semantic mistakes, including using the wrong column(s) in a* `WHERE` *clause, using wrong constants, and missing operators such as* `GROUP BY` *or* `DISTINCT`*. Joins and compound clauses proved difficult for all students.*

4.4.2 RQ7: SQLRepair

Our evaluation dataset consists of 2,531 incorrect SQL queries. Of these, SQLRepair was able to find a repair for 737, giving an overall repair success rate of 29.1%. The different types of repairs made are shown in Table 4.6. The table is organized based on the repair types: the first three correspond to the three non-synthesis repairs supported, and the last five to the synthesis repairs. For example, the first row, *Column Mismatch*, is described in Section 4.3.5.1.2; this repair is made to 67 (13.7% of 488) queries from CS2 and 40 (16.1% of 249) from SE, totaling 107 (14.5% of 737) of all repaired queries. The representative example modifies the `SELECT` clause to return three columns instead of two.

4.4.2.1 Repaired Queries

The most common repair type observed was *Column Synthesis* (made to 393, or 53.3% of 737, queries), where SQLRepair synthesizes an expression using a new column, replacing an existing expression. The second most common repair type observed was *Clause Removal*, where SQLRepair identifies and removes a `WHERE` subclause that results in incorrect output.

The third most common synthesis repair type is *Clause Synthesis*, where a new subclause is generated for the `WHERE` clause. Together, these three repairs correspond to the very common *Wrong subclauses in WHERE clause* error observed across both classes (see Table 4.4), where the resolution is to add, fix, or remove an incorrect clause.

We also observe that while the majority of repairs performed (982 of 1,192 repairs, or 82.4%, from Table 4.6) involve a synthesis repair, non-synthesis repairs play an important part in success as well. In 107 cases, our tool fixes a *Column Mismatch error* by identifying a query that is returning the wrong set of columns and rewrites the `SELECT` clause accordingly. Although this is a non-synthesis repair, it fixes queries from the *Column reference error* category in Table 4.3 and the *Column mismatch* category in Table 4.4, thus covering both syntax and semantic errors. Fixing unquoted or misquoted string literals (*String Repair*) and incorrect C/Java style operators (*Operator Mismatch*) happen less often, but a fix from one of these categories is still made to 75 and 28 queries, respectively. Additionally, making non-synthesis repairs also opens up new possibilities for synthesis repairs: queries must be well-formed for synthesis repair to proceed, and non-synthesis repair fixes some cases where they are not.

More often than not, repaired queries requires a combination of repair operations. In fact, 433 (58.8%) of the successfully repaired queries contained multiple repair operations. For example, the query `select * from delta WHERE CFR < 1696` was repaired by fixing both the columns to return (a *Column Mismatch* repair and changing the 1696 to an 1865 (a *Constant Synthesis* repair).

4.4.2.2 Not Repaired Queries

The remaining 1794 queries that could not be repaired fall into two major categories:

4.4.2.2.1 Unsupported functionality

Some functionality necessary to solve the problems in Table 4.1 is not supported in SQLRepair, such as `GROUP BY` or joins. Students also used functionality that was neither necessary nor supported (such as `BETWEEN` and `LIMIT`), which rendered their queries unfixable.

4.4.2.2.2 Miscellaneous syntax errors

SQLRepair can fix some but not all syntax errors. Errors such as a misspelled SQL keyword (e.g., `...GROUPED BY...`) clauses placed in the wrong order (e.g., `SELECT * DISTINCT...`) are not fixed automatically by our tool.

4.4.2.3 Performance

We tested the performance of SQLRepair on an Intel i7-6700HQ running Linux Mint 18. Successful repairs are found in a median of 231 milliseconds (max: 1,602) and unsuccessful repairs in a median of 196 milliseconds (max: 1,912).

RQ7 Summary: *SQLRepair automatically fixes 29.1% of student queries with errors, covering both syntax and semantic errors.*

4.4.3 RQ8: Repair Quality

To understand the quality of the repairs produced by SQLRepair, once students in Phase 2 found a solution for a problem (or gave up), we presented them with several alternative solutions (see Section 4.3.2). Students rated each query on a scale of 1 (very difficult to understand) to 7 (very easy to understand) and optionally provided a free response rationale. We received a total of 281 voting responses (CS2: 183, SE: 98) and 81 rationales (CS2: 50, SE: 31).

Each query was from one of four categories (Section 4.3.2): MyCorrectQuery (MCQ), MyRepairedQuery (MRQ), OtherCorrectQuery (OCQ), OtherRepairedQuery (ORQ). On average, students found their own queries (MCQ) to be more understandable than their repaired queries (MRQ) (5.58 vs. 5.35, see Table 4.7), but the difference is not significant. Thus, a student's repaired query could be used as an alternate way to solve a problem without sacrificing understandability, a divergence from prior work in automated program repair suggesting that machine-repaired code is less understandable than human-written code [Fry12].

Looking more closely at the data, a pairwise analysis can determine the within-participant differences in understandability between each query category. Using the four query categories, we ran six paired Mann–Whitney U tests. As all the p-values are above 0.1, the data show that there is no statistical difference in understandability between human-written and machine-repaired SQL queries. For example, comparing the student's own query (MCQ) with their repaired queries (MRQ) yielded $p = 0.662$. While there is a 0.23 difference in averages, representing a quarter of a level on the 7-point Likert scale, the difference is not significant. Comparing a student's own correct query (MCQ) against a correct query written by others (OCQ) also reveals no difference in understandability. Therefore, we find evidence that repaired queries and queries written by others are all viable candidates for presenting students with alternate implementations of SQL queries.

Qualitatively, we observe that some students preferred their own solutions over all

others; we received written responses such as "*I literally wrote [this query]*" and "*I chose [this query] because it was exactly my solution*". However, this was not universally the case: one student remarked "*I can't believe how [bad] my answer is*". This suggests that automated repair can help students identify better solutions even after solving a problem correctly.

RQ8 Summary: *Queries repaired by SQLRepair are rated as equal in understandability compared to queries written by the students themselves, suggesting repaired queries could be useful for presenting students with alternate queries.*

4.5 Discussion

Here, we discuss the implications of our results, present opportunities for future work, and discuss threats to validity.

4.5.1 Implications

By analyzing the SQL queries written by students new to SQL, we see that certain topics are particularly challenging; most students struggled with joins, ordering, and compound clauses. When SQL is used, it is typically with more than one table, so teaching joins is a necessity [Lu93]. By contrast, students had less difficulty with operators such as `GROUP BY`. All concepts were introduced to students in a similar way, with background information provided through slides and live examples showing how they work in practice. These results suggest that some topics remained more difficult for students to understand and thus may require additional instruction.

To the best of our knowledge, SQLRepair is the first automated repair (APR) tool for SQL queries, and our results provide preliminary evidence that this can be useful in education. Patitsas, et al. report that presenting students with multiple solutions side-by-side can improve learning outcomes [Pat13]. In cases where peer instruction is unavailable, our results suggest repair tools may be able to provide alternative solutions for students to visualize. Providing hints or iterative refinement rather than just a new solution may further improve the process.

While the overall repair rate of SQLRepair is lower than many general-purpose repair tools, this is a first step and the availability of our dataset should allow future SQL repair tools to improve on our efforts reported here for educational and professional audiences.

4.5.2 Future Work

We have identified several promising directions for future work in program repair to support learners.

The single most challenging problem for students was one that involved joining two different tables together on a common column. This suggests that students struggle to see the big picture and how their data connects together. Tools such as MySQL Workbench allow reverse-engineering an entity-relation diagram from an existing database schema, and the produced diagrams can be used much like UML class diagrams to introduce new developers to an existing design. A database IDE that automatically shows the relationship between tables when two or more are included in a query could help users see and utilise the connections in their data.

We observed, and several students affirmed in their comments, that it is challenging to identify patterns within a table and thus pick out desired rows (i.e., forming queries from examples is challenging). Tooling that highlights similarities and differences between selected columns of two or more rows could help the user better identify relevant patterns.

More generally, our results suggest that program repair may be a useful educational tool for presenting alternate solutions to a problem. Notebooks such as Jupyter have become a popular way for performing exploratory data analysis, particularly among end-user programmers, because they allow intermingling code, written descriptions, and results [Ker18]. While most such notebooks focus on Python or R, SQL has a place within the data science world as well, and integrating synthesis or repair tools could help make the learning process easier for many students.

All of the repairs produced by SQLRepair follow the steps listed in Section 4.3.5. The order in which repairs are performed has the potential to impact the query that is ultimately produced. Future work could study whether performing repairs in a different order impacts the quality of the query produced by potentially producing more concise or understandable solutions.

4.5.3 Threats to Validity

In this section, we discuss different types of threats to validity.

4.5.3.1 Conclusion Validity

We performed statistical analysis using Mann-Whitney U tests, a non-parametric test for calculating a difference-of-means that is resilient to skewed data. To ensure consistency

across study sessions, we performed our SQL introduction using a common slide deck and set of examples. Despite the difference in the format of study sessions between Phase 1 and Phase 2, a statistical test showed no significant difference in how effectively each group of students solved problems.

4.5.3.2 Internal Validity

Students who participated in our study did so under two different incentive structures. In Phase 1, students participated in the study as part of an in-class activity. In Phase 2, students were offered extra credit for participating in an out-of-class activity. Consequently, there may be a selection bias for the students who participated in Phase 2.

4.5.3.3 Construct Validity

In this work, we use the understandability rating that a student gives a query as a proxy for the quality of the query that has been produced. However, this merely asks students to read the query and then offer a vote on it; we do not ask them to integrate the queries produced into a larger application or modify the query to solve a problem that is similar but not identical. Consequently, students who are using the queries in a different context may have different priorities for what makes a query understandable or not.

4.5.3.4 External Validity

The problems we had students complete were based off of the UMLS dataset; it is unknown whether the nature of the dataset contributed to the difficulty students faced when solving problems. The specific errors that students faced may not generalize to different problems. It is possible that the context of the data made problems more difficult than if students had been working with more familiar data. However, we expect the data to be equally unfamiliar to all students.

4.6 Conclusion

In this work, we have analyzed the mistakes that undergraduate students make when working with SQL for the first time by studying the errors they introduce. We found that the majority of queries contain one or more syntax or semantic error, and that semantic errors make up a majority of errors introduced. We found that junior-level students perform better than sophomore-level students, solving more problems correctly and introducing syntax errors at a lower rate. Among the more advanced SQL topics covered, students

particularly struggle with joins, thus suggesting a need for teaching students to see and utilise patterns in data. We have also demonstrated that SQLRepair can fix 29.1% of queries with errors. By demonstrating that APR techniques are applicable to SQL, we pave the way for additional automated repair of special-purpose programming languages. Finally, our results suggest that automated repair may support students as they learn SQL. Students rate our tool-produced repairs as good as queries written by themselves or other students, and thus automated repairs may make a compelling teaching tool when peer instruction and feedback is unavailable.

Table 4.4 Classifications of semantic errors made by students across both phases.

Error Type	CS2 Number (%)	SE Number (%)	Total Number (%)	Example
Wrong sub-clauses in WHERE	828 (69.9%)	406 (76.7%)	1,234 (72.0%)	`SELECT * FROM charlie`
Missing or extra operator (GROUP BY, DISTINCT, etc)	369 (31.1%)	122 (23.1%)	491 (28.6%)	`SELECT LAT, STT, ISPREF FROM juliett, india WHERE juliett.CUI = india.CUI GROUP BY LAT`
Wrong values in WHERE	241 (20.3%)	74 (14.0%)	315 (18.4%)	`SELECT DISTINCT SVER FROM golf WHERE SVER < 2000`
Wrong ordering	209 (17.6%)	68 (12.9%)	277 (16.2%)	`SELECT DISTINCT * FROM echo ORDER BY MRRANK_RANK DESC`
Column mismatch	70 (5.9%)	46 (8.7%)	116 (6.8%)	`SELECT * FROM juliett a, india b WHERE a.CUI = b.CUI`
Wrong operator in WHERE	83 (7%)	27 (5.1%)	110 (6.4%)	`select LAT, STT, ISPREF from india a, juliett b where a.CUI = b.CUI AND a.CVF = 256`
Missing join (implicit or explicit)	43 (3.6%)	21 (4.0%)	64 (3.7%)	`SELECT LAT, STT, ISPREF from juliett where TS='S';`
Miscellaneous	31 (2.6%)	3 (0.6%)	34 (2.0%)	`SELECT DISTINCT SVER FROM golf;`

Table 4.5 Successes per problem and per course. Each cell represents the ratio between the number of correct attempts and the total number of attempts. Success per participant represents the ratio between the number of students who attempted the problem and the number of students who solved it successfully. Success per attempt represents the sum of correct attempts to total attempts across CS2 and SE.

		Course			
Problem	**Major Concept**	CS2	SE	**Success per participant**	**Success per attempt**
1	Single-condition select	31/115 (27.0%)	14/25 (56.0%)	45/51 (88.2%)	45/140 (32.1%)
2	Select with projection	17/233 (7.3%)	9/163 (5.5%)	26/49 (53.1%)	26/396 (6.6%)
3	Inequality	17/140 (12.1%)	13/55 (23.6%)	30/41 (73.2%)	30/195 (15.4%)
4	Projection and inequality	23/145 (15.9%)	12/39 (30.8%)	35/41 (85.4%)	35/184 (19.2%)
5	Compound select	4/304 (1.3%)	7/101 (6.9%)	11/51 (21.6%)	11/405 (2.7%)
6	Compound select with AND	6/256 (2.3%)	8/113 (7.1%)	14/44 (31.8%)	14/369 (3.8%)
7	Distinct	23/153 (15.0%)	11/38 (28.9%)	34/52 (65.4%)	34/191 (17.8%)
8	Ordering	19/201 (9.5%)	12/80 (15.0%)	31/49 (63.3%)	31/281 (11.0%)
9	Joins	1/280 (0.4%)	0/70 (0.0%)	1/42 (2.4%)	1/350 (0.3%)
10	Grouping	16/195 (8.2%)	8/76 (10.5%)	24/45 (53.3%)	24/271 (8.9%)
Successful attempts	-	157/2022 (7.8%)	94/760 (12.4%)		

Table 4.6 Types of complete repairs from SQLRepair. Non-synthesis repairs are presented first, followed by synthesis repairs. Each section is sorted by the total number of repairs; percentages are computed over the total number of repaired queries. Because many successfully repaired queries contain two or more repairs, the totals in each column sum to more than 100%. The identifier associated with each repair type corresponds to the description in Chapter 4.3.5.

	Repair Type	**CS2 Number** (%)	**SE Number** (%)	**Total Number** (%)	**Representative Example**
Non-Synthesis	Column Mismatch (4.3.5.1.2)	67 (13.7%)	40 (16.1%)	107 (14.5%)	`SELECT CUI, TUI FROM...` → `SELECT CUI, TUI, STN FROM ...`
	String Repair (4.3.5.1.3)	33 (6.8%)	42 (16.9%)	75 (10.2%)	`...WHERE CUI2 = C0364349` → `...WHERE CUI2 = 'C0364349'`
	Operator Mismatch (4.3.5.1.1)	28 (5.7%)	0 (0%)	28 (3.8%)	`...WHERE min==0` → `...WHERE min = 0`
Synthesis	Column Synthesis (4.3.5.2.3)	252 (51.6%)	141 (56.6%)	393 (53.3%)	`...WHERE REL = 'RO';` → `...WHERE CUI2 = 'C0364349';`
	Clause Removal (4.3.5.2.4)	109 (22.3%)	98 (39.4%)	207 (28.1%)	`...WHERE CUI2 = 'C0364349' OR REL = 'RO'` → `...WHERE CUI2 = 'C0364349'`
	Clause Synthesis (4.3.5.2.5)	131 (26.8%)	65 (26.1%)	196 (26.6%)	`SELECT CUI1, RUI FROM bravo` → `SELECT CUI1, RUI FROM bravo WHERE CUI2 = 'C0364349';`
	Constant Synthesis (4.3.5.2.1)	114 (23.4%)	21 (8.4%)	135 (18.3%)	`...WHERE CFR < 1834` → `...WHERE CFR < 1865`
	Operator Synthesis (4.3.5.2.2)	39 (8.0%)	12 (4.8%)	51 (6.9%)	`...WHERE TFR < 1850...` → `...WHERE TFR <= 1965...`

Table 4.7 Average Likert-scale understandability scores per course and per query type, where 1 maps to Very Difficult and 7 maps to Very Easy; 4 is a neutral response (neither easy nor difficult). Query categories are defined in Section 4.3.2.

	Course		
	CS2	SE	**Overall**
MyCorrectQuery (MCQ)	5.62	5.54	5.58
MyRepairedQuery (MRQ)	5.32	5.41	5.35
OtherCorrectQuery (OCQ)	5.04	5.41	5.17
OtherRepairedQuery (ORQ)	5.03	5.38	5.15

CHAPTER

5

IDENTIFYING STRUGGLING TEAMS THROUGH WEEKLY REFLECTION SURVEYS

This study[1] explores whether struggling teams in a software engineering course can be **predicted early** in a project using a **weekly collaboration reflection survey**. Additionally, the collaboration reflection survey has revealed some common challenges that can offer **more effective instructional materials** to proactively ward off common problems.

> Satisfies part of thesis: Using software engineering automation and **survey techniques** in computer science education results in improved student learning outcomes, **early prediction of struggling teams**, and **more effective instructional materials**.

[1]This study was published in substantial portion as Presler-Marshall, K., Heckman, S. & Stolee, K. "Identifying Struggling Teams in Software Engineering Courses Through Weekly Surveys". *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. SIGCSE '22 [2022].

5.1 Study Rationale

Computer science education is increasingly focused on team-based learning, where students work collaboratively to achieve the learning goals for a course [Abe]. Such an approach provides an educational environment that more closely resembles professional software engineering workplaces [Ric12].

A team's success depends upon many factors, including good-faith participation, equitable contribution, and effective communication [Taf16]. When a team lacks these characteristics, the impact on the team can be negative: students are left frustrated by underperforming teammates or confused about the team's progress [Oak04]. Teams that are unable to communicate effectively will be at a disadvantage throughout the project [Dor12]. Furthermore, poor teaming experiences can harm a sense of engagement or belonging with the Computer Science community, particularly among historically under-represented groups [Che15]. Consequently, while grade adjustments can be made at the end of the project, a better approach is to identify and mitigate problems before the project ends.

Tuckman's model of teaming [Tuc65] is a prominent theory of teaming and group dynamics, and argues that teams move through four largely sequential stages. The fist stage, *forming*, is where the team originally starts to take place, and individual members figure out their place on it. In the *storming*, the team is typically characterised by conflict or friction as each member seeks to work in *their* preferred approach. As conflict gradually give way to working together effectively, the team moves into the *norming* stage, where members establish how to work together effectively. Finally, in the *performing* stage, the teams work in a smooth and cohesive unit, with each member playing their role to accomplish the necessary outcomes. Later, Tuckman and Jensen revisited the model and added a fifth stage, *adjourning* where the team disbands after having accomplished its goals [Tuc77]. Tuckman's model has remained influential in education and project management for capturing the stages that are practically universal to team experiences [Bon10]. However, practitioners have recognised that the *storming* phase is the most difficult one for teams to overcome, and the teams that are unable to successfully move through this phase continue to perform poorly for the remainder of their interaction [Lea].

In this work, we seek to understand whether weekly collaboration reflection surveys can effectively identify struggling teams in our software engineering course. Drawing upon teaming theory, we seek to identify teams that are unable to successfully navigate the *storming* phase and emerge as a coherent and successful group. In particular, we seek to understand whether we can observe several commonly-observed issues: (1) students whose work quantity or quality does not rise to the expected standards, (2) teams where members

are uncertain about what they should be accomplishing, and (3) teams where members are not attending meetings or following expected communications patterns [Tuc06; Abb17]. While peer evaluations can identify students who fail to contribute adequately to the success of their team, or who go above and beyond to ensure success [Bru10], these tend to focus on evaluating individual teammates rather than the dynamic of the entire group. By contrast, our survey focuses on whole-team collaboration, as a team is more than the sum of its members. Finally, taking time to reflect on what is working and not working is a key component of self-regulated learning and may promote better learning outcomes [Pan17].

Our results show the survey is effective at identifying struggling teams, and by the halfway mark of the project in most cases. A majority of students reported the TCRS helped keep them on track.

The contributions of this study are as follows:

- A weekly team collaboration reflection survey (TCRS) suitable for undergraduate software engineering courses, and an approach for flagging struggling teams based upon it (Figure 5.1),
- A demonstration that the TCRS is capable of identifying teams that later face difficulties, and can in most cases do so by the halfway mark of the project, and
- A blueprint for an intervention that may be helpful for engaging with teams that are struggling to collaborate effectively.

5.2 Methodology

In this section, we start by introducing our course context (Section 5.5) and research questions (Section 5.2.2). We also discuss study design (Section 5.2.3), including the TCRS, and the intervention that we attempted (Section 5.2.4).

5.2.1 Background

At NC State University, a research-intensive university in the south-eastern United States, undergraduate Computer Science students are required to take a Software Engineering course, typically during their third year. The course covers fundamentals in software engineering, such as how to design, implement, and test a medium-sized object-oriented system; how to write requirements; and how to appropriately break down a project into manageable components, all in the context of team-based projects that each span several weeks. The first project, an *onboarding project* (OBP), introduces the process expectations

and technology stack. The second, a larger *team project* (TP), asks students to complete a more comprehensive project with a larger team. The OBP is completed in teams of two or three students; the TP in teams of five or six. Projects are broken into iterations, each typically lasting one week, that cover different learning objectives: requirements and planning, design, testing, and implementation. Students are evaluated in five categories: technical deliverables (including both code and technical documents); technical processes; project management; team collaboration; and peer review. Each of these high-level grade categories includes both team and individually graded components. At the end of the project, the course teaching staff reviews peer evaluations and contributions to determine whether individual adjustments are needed (positive or negative). We seek to supplement the peer evaluations with a more informal metric for identifying struggling teams.

The software engineering course at NC State University typically has between 120 and 160 students a semester, led by one PhD professor and three to five teaching assistants (TAs). Consequently, the student:teaching staff ratio is typically between 30:1 to 25:1. This necessitates light-weight approaches for detecting struggling teams. To support the project-based learning, the course features weekly lab sessions; led by the TAs, the labs provide time for teams to review technical deliverables from the previous week and plan tasks for the next week. Lab sessions are typically conducted in-person, but due to the university's response to the COVID-19 situation were conducted online via Zoom from Spring 2020 through Spring 2021.

To this end, with the backing of the DELTA Center, a Teaching Technology group at NC State University that offers support for educational technologies and course redesigns, we introduced weekly reflection surveys (TCRS) into both class projects. The TCRS, discussed in further detail in Section 5.2.3.2, provide students an opportunity to reflect on how their project is going, and gives the teaching staff regular, *in-situ* feedback on how teams are working together. Surveys have been administered sporadically, with inconsistent followup, from Fall 2017 to Fall 2019. We have recently revised the survey, and, starting with Fall 2020, administered the survey every week to enable drawing more reliable conclusions.

5.2.2 Research Questions

We frame our work around the following research questions:

- **RQ9:** *Can weekly reflection surveys identify software engineering teams in need of instructor assistance?*
- **RQ10:** *Can weekly reflection surveys identify software engineering teams that need assistance sufficiently early?*

Table 5.1 A summary of the participants involved in our study across the Fall 2020 and Spring 2021 semesters. *OBP* and *TP* are the two projects in our course, as discussed in Section 5.2.1.

	Fall 2020	Spring 2021
Students	120	162
IRB Opt-Outs	2	5
Teams - OBP	42	57
Teams - TP	21	28
Struggling Teams - OBP	9	13
Struggling Teams - TP	8	8

- **RQ11:** *Can weekly reflection surveys help support a better experience for software engineering teams?*

These questions evaluate the utility of a *weekly team collaboration reflection survey (TCRS)* we developed, based on prior work in teaming and collaboration [Owe15; Pfa03; Rud17; Bur03]. We administered the survey on a weekly basis to students in the junior-level software engineering course at NC State University. We then analysed the results of the TCRS, comparing against grades and peer evaluations [Ada18] to understand its effectiveness.

5.2.3 Study Design

In our study, we deployed the TCRS to each student weekly throughout both course projects. This section describes the survey design, participants, analysis, and an intervention to help struggling teams. This study was approved by the NCSU IRB Office as protocol #12876.

5.2.3.1 Participants

We ran our study in Fall 2020 and Spring 2021, using the class described in Section 5.2.1. Of the 120 students in the course in Fall 2020, two students declined to let us analyse their data for research purposes. As data within a team cannot be separated for our purposes, their entire teams were excluded from further analysis. The Spring 2021 semester had 162 students enrolled; five students opted out of letting us analyse their data. A summary of the participants from each semester is shown in Table 5.1. The final two rows of the table describe the number of teams that we identified as "struggling" in each semester, as described in Section 5.2.3.3.

Weekly tasks questions, answered with checkboxes in response to *This week I have:*
Q1: ☐ Designed a usecase (or a portion of one) ☐ Fixed a bug in the system ☐ Implemented a usecase (or a portion of one) ☐ Written black-box tests
☐ Written automated tests ☐ Other: _____
Q2: ☐ Completed all my assigned tasks ☐ Completed some of my assigned tasks ☐ Asked a teammate for help completing my tasks ☐ Helped a teammate complete a portion of their tasks
Q3: ☐ Met live with my team ☐ Participated in checkins with my team
☐ Opened a pull request and asked my team for feedback on my code ☐ Asked my team for feedback on my non-code work ☐ Reviewed technical artifacts for my teammates

Planning questions, answered with a five-point Likert scale: ○ **Much less** ○ **Less** ○ **About as much as** ○ **More** ○ **Much more**
Q4: This week, I have gotten done __ than I think I should have
Q5: This week, my team overall has gotten done __ than I think we should have
Q6: Next week, I intend to get done __ than I did this week

Collaboration satisfaction questions, answered with a five-point Likert scale:
○ **Strongly disagree** ○ **Disagree** ○ **Neither agree nor disagree** ○ **Agree** ○ **Strongly agree**
Q7: This week, I knew what I needed to get done
Q8: Overall, I think that everyone has been contributing adequately to the success of the project
Q9: In our team we relied on each other to get the job done
Q10: Team members kept information to themselves that should be shared with others
Q11: I am satisfied with the performance of my team
Q12: We have completed the tasks this week in a way we all agreed upon

Miscellaneous questions:
Q13: My progress this week has been impeded by:
☐ Difficulties with technologies or course materials ☐ Demands of other classes ☐ Other personal responsibilities or distractions ☐ Teammates who didn't complete their responsibilities ☐ Communication difficulties with my teammates ☐ Difficulty scheduling tasks so that I wasn't waiting for my team to complete their work ☐ Other: _____ ☐ None

Q14: How do you feel about your team's collaboration process in this project?

Figure 5.1 Team Collaboration Reflection Survey.

5.2.3.2 Surveys

The survey was originally developed by the DELTA Center at NC State University using prior work in teaming and collaboration [Owe15; Pfa03; Rud17; Bur03]. The survey was revised prior to Fall 2020 to add additional questions on what students were working on and to focus on key questions from prior work on identifying struggling teams. The questions on the survey are shown in Figure 5.1. The survey was deployed through Qualtrics, and all questions other than **Q14** were mandatory. Weekly response rates were approximately 90%; for example, for the TP in Spring 2021 response rates ranged from 87% to 93%, averaging 91% across the project.

To support a repeatable way of flagging teams, we needed a way to quantify TCRS

responses. To do this, we broke down the survey into "positive questions" (ones where we would expect a successful team to answer either "*Agree*" or "*Strongly Agree*", such as **Q7**) and "negative questions" (where we would expect a successful team to answer either "*Disagree*" or "*Strongly Disagree*") and assigned numerical scores to the Likert scale responses. For positive questions, an answer of "Strongly Agree" was assigned a score of 4, "Agree" a score of 2, and so on down to -4 for "Strongly Disagree". For negative questions, this scale was reversed, with "Strongly Disagree" receiving a score of 4. The process was repeated for each question, and the scores for each were summed. Questions that did not fall into either the positive or negative categories, such as asking students what they had accomplished over the week, were excluded from the scoring process. Any survey where the overall score was 0 or less, indicating that the student felt that more things were going wrong than right, was flagged as indicating issues.

5.2.3.3 Observed Struggle Oracle

The surveys are intended to flag, or predict, teams that are struggling. To determine if the survey correctly identifies such teams, we need an oracle, which we formed from two metrics:

Low Project Grades: Most teams typically do well on the Team Project, with approximately 90% scoring an A or B. The Onboarding Project has more variability in project grades. In both cases, we use a cutoff of one and a half standard deviations below the mean project grade to identify struggling teams.

Peer Evaluations: Students evaluate themselves and their peers, rating each member between *1 (Infrequently)* to *6 (Above and Beyond)* on metrics such as their contributions and timeliness. The OBP has one peer evaluation, completed at the end of the project; the TP has two: one at the halfway mark and one at the end of the project. Each student received the average of the scores from their teammates (and, for the TP, averaged across both peer evaluations). Students who scored at least one and a half standard deviations below the class average were selected as struggling, and their team was selected for analysis.

These metrics have been used as measures of success in team-based learning environments [PÁ0; Sit04]. If either metric flags a team or team member, we consider that an indication of struggle; we refer to these as teams with *observed struggling* or *observed struggling behaviour*. The number of teams with observed struggling behaviour can be seen in the last two rows of Table 5.1.

To verify the oracle formed using these metrics, we cross-checked the team classifications with another data source, the end-of-project reflections (for the OBP, done through a

semi-open-ended Google Form; for the TP, done through a three-page written document). The reflections asked students to consider the entire project and how they and the team had worked to meet their goals. As an open-ended task, this gave students more opportunity to explain team dynamics, and let us verify the teams with observed struggling. To confirm our observations, we read through the end-of-project reflections submitted by each member of the eight teams that were observed to struggle on the TP in Fall 2020. For seven of the eight teams, at least three members (out of the five or six members of the team) mentioned issues such as the team falling behind on deliverables or communication difficulties. On the eighth team, one member received poor peer evaluations, but there were no issues reported in the final reflections. To contrast this against the rest of the class and establish a baseline, we randomly selected five other teams with no observed struggle and read through their reflections. Students on one of the teams reported in their reflections that they faced communication issues as the project progressed; no issues were reported by any other students. Consequently, we consider the metrics of grades and peer evaluations to be reasonably accurate, if imperfect, for identifying teams struggling during the project.

5.2.4 Intervention

To assist struggling teams, we developed a checklist intervention with sample questions to ask teams. Based on prior work [Iac20], the checklist focuses on getting students to articulate what specifically they are responsible for, how they have been meeting and collaborating with their teams, and helping them schedule tasks to allow for concurrent work. The TAs used this checklist during the lab sessions; additionally, they were encouraged to follow up with teams via email to help hold members accountable to their plans.

In Spring 2021, we conducted a structured experiment where struggling teams in half of the lab sections received the targeted follow-up intervention (experimental labs) and struggling teams in the other half of the lab sections did not (control labs). We then measured the impact of the intervention by comparing end-of-project grades and peer evaluations between the groups.

5.3 Results

Here, we present results on the efficacy of the weekly TCRS at identifying teams in need of assistance (RQ9), identifying them early into the project (RQ10), and impacts on student success (RQ11).

5.3.1 RQ9: Identifying Struggling Teams

First, we sought to understand whether weekly reflection surveys can successfully identify software engineering teams in need of instructor assistance.

We find that identifying and reporting struggling relies on the entire team for accurate results. For example, for the OBP in Fall 2020, nine students (on eight teams) were flagged for receiving a peer evaluation at least one and a half standard deviations below the class average. The nine students submitted a total of 47 TCRS responses over the course of the project. Twelve responses across five flagged students (and four distinct teams) identified that the team was facing challenges. Consequently, only four of the nine OBP teams we observed struggling[2] were flagged through the TCRS responses of their most unproductive members. Factoring in TCRS submissions from the team increased this to eight of nine teams, which suggests that reporting works best as a whole-team effort.

A breakdown of the teams that were identified through the TCRS and a comparison to teams with observed struggle is shown in Table 5.2. Each sub-table shows the results for one semester; for example, Table 5.2a shows the results from Fall 2020. Within each subtable, the left side shows results for the OBP, the right side, for the TP. For example, the left half of Table 5.2a shows that for the OBP, 8 teams were flagged by the TCRS and had observed struggle. One team had observed struggle that was not flagged by the TCRS, 10 teams were flagged by the TCRS but had no observed struggle, and 23 teams were neither flagged nor had observed struggle. The right half of Table 5.2a shows results for the TP. Results for Spring 2021 are presented similarly in Table 5.2b, where we see every single team with observed struggle on the TP was flagged by the TCRS. In total, across two projects and two semesters, the TCRS flagged 34 of 38 teams, or 89.5%, with observed struggle.

There is an inherent tradeoff between precision and recall: if the TCRS flags more teams, it will increase the recall (the number of struggling teams that the TCRS detects). However, this will come at the cost of lower precision (more teams flagged with no observed struggling). Given our use case, we prefer a survey that has high recall over one with high precision. As the cost of engaging with a team is low, rather than miss teams truly in need of instructor assistance, we prefer to engage with more teams that potentially don't need the help. That said, it is possible that the TCRS-flagged teams actually do need help, but their struggles did not translate to poor grades or poor peer evaluations. Consequently, while many teams are flagged that do not ultimately demonstrate struggling outcomes – 35 over the course of two projects and semesters, giving a precision of 49.3% – the tradeoff suits the circumstances. The recall, by contrast, is much better – 89.5%. In Chapter 6 we

[2]The final team with observed struggle was flagged solely through poor grades, not peer evaluations.

Table 5.2 Success of surveys (*TCRS*) at predicting team struggle relative to struggle observed (*observed struggle*, see Section 5.2.3.3). Percentages are relative to the number of teams in total for that project in each semester.

		OBP - F20 TCRS		TP - F20 TCRS	
		Flagged	Not Flagged	Flagged	Not Flagged
Observed Struggle	Yes	8 (19.0%)	1 (2.4%)	7 (33.3%)	1 (4.8%)
	No	10 (23.8%)	23 (54.8%)	3 (14.3%)	10 (47.6%)

(a) Results for Fall 2020

		OBP - S21 TCRS		TP - S21 TCRS	
		Flagged	Not Flagged	Flagged	Not Flagged
Observed Struggle	Yes	11 (19.3%)	2 (3.5%)	8 (28.6%)	0 (0%)
	No	14 (24.6%)	30 (52.6%)	8 (28.6%)	12 (42.9%)

(b) Results for Spring 2021

discuss the struggling teams that were not flagged and measures to support similar teams.

RQ9 Summary: *The TCRS manages to identify most teams, 89.5% across two projects and two semesters, that exhibit observed struggling behaviour at the end of the project.*

5.3.2 RQ10: Identifying Teams Early

If the TCRS only reveals issues during the last week of a six-week project, it is unlikely that it will be of any practical use to the teaching staff. We seek to determine if the TCRS can identify struggling teams *sufficiently early*, defined as the halfway mark of the project. To answer RQ10, we find the first occurrence of a TCRS response indicating a problem for each team with observed struggle.

We present results for when teams were identified through the TCRS in Table 5.3. Each column tracks a one-week iteration within each project, and the rows the semesters where the TCRS was used. The final two columns represent the number of teams with observed struggle that were not detected (**ND**) by the TCRS at any point during the project, and the percentage flagged by the halfway mark (**H?**). The final row summarises teams flagged in each half of the project. Table 5.3a presents results for the Onboarding Project; Table 5.3b presents results from the Team Project. For example, the first row in Table 5.3a shows that

Table 5.3 The first week that each team with *Observed Struggle* was flagged through the TCRS. Each column header represents a one-week iteration in the respective project. Teams in the *ND* column were not detected through the TCRS. The *H?* column shows the percentage of teams flagged by the TCRS by the halfway mark of the project.

	W0	W1	W2	W3	W4	W5	W6	ND	H?
F20	1	5	1	-	-	1	-	1	78%
S21	-	2	-	5	2	2	-	2	54%
Total	14/22 (63.6%)				8/22 (36.4%)				

(a) Onboarding Project

	W0	W1	W2	W3	W4	W5	ND	H?
F20	1	1	4	-	1	-	1	86%
S21	2	4	2	-	-	-	-	100%
Total	14/16 (87.5%)			2/16 (12.5%)				

(b) Team Project

in Fall 2020, one team was first flagged during Week 0, five teams during Week 1, and so on. Ultimately, seven of the nine teams, or 78%, were flagged by the TCRS by the halfway mark. The percentages are based on the teams with observed struggle, representing the oracle.

On the whole, the TCRS does a compelling job, identifying 28 of the 38 teams, or 73.7%, by the halfway marks of their respective projects. There is some difference between projects: 63.6% of teams on the OBP were identified by the halfway mark, compared to 87.5% of teams on the TP. This may be because the first several weeks of the OBP are spent on tasks the students find comparatively easy – requirements analysis, wireframing, and writing system tests – and consequently when implementation tasks start to pick up for Week 4, the workload increases and team dynamics can become strained.

If we move our goalposts one week later, the detection recall for the S21-OBP goes from 53.8% to 76.9%. For the scope of this project, the teaching staff would still have two weeks to help the teams improve. This suggests that the details of the project impact early detection. The TP involves more difficult tasks comparatively early on; of the eight teams with observed struggle in F20, seven were flagged by the TCRS, and six of those by the halfway mark. All teams from S21 were flagged by the halfway mark.

RQ10 Summary: *The TCRS identifies a majority of struggling teams – 53.8% to 100%, depending on project and semester – by the halfway mark of the project.*

5.3.3 RQ11: Survey Impact on Team Success

In this section, we evaluate whether the TCRS has a positive impact on software engineering teams. We consider two factors: 1) does engaging with flagged teams improve their grades, and 2) do students find the TCRS useful for self-reflection or staying on track. We found that there was no improvement in students' grades, but a majority of students (64.4%) found the TCRS helpful.

As discussed in Section 5.2.4, we conducted an intervention in Spring 2021 where students in half of the labs received followups from the lab TA and students in the other labs did not. To understand the impact, we conducted unpaired Mann-Whitney U tests between the grades received by students in the control group and students in the experimental group, and found there was no statistically significant improvement in either end-of-project grades or peer evaluation grades ($p > .1$ for both metrics and projects).

Students found the TCRS a useful tool for self-reflection or keeping their team on track. Starting with Fall 2020, we added a question to the end-of-project reflection for the TP asking students whether the TCRS "*helped keep you and your team on track*". To complement the intervention, we read the reflections submitted by each student in Spring 2021. In total, we received 118 responses that explicitly mentioned the TCRS. Of the 118 students, 76, or 64.4%, believed that it was a useful tool for self-reflection or keeping them or their team on track. For example, students mentioned that the TCRS "*help[ed] keep our team on track*" or "*it forced us to demonstrate what we've done*", suggesting that it may be useful for getting students to reflect on their teaming experience. Of the remaining students who did not find it helpful, some said the project was going well and "*we rarely had any communication issues to reflect on*". Others remarked that even though issues were brought up, a member of their team remained intransigent and the situation did not improve. However, most students appreciated the value of the TCRS. In Chapter 6 we discuss plans to encourage and support self-reflection.

RQ11 Summary: *Most (64.4%) students believe the TCRS helps keep their team on track or provides a positive chance for self-reflection.*

5.4 Discussion

In this section, we discuss the implications of our results, threats to validity, and propose future work.

5.4.1 TCRS Success

We observed four false negatives across our two projects and semesters: teams with observed struggle that were not flagged by the TCRS. Three of these teams were from the OBP, and two of these teams had only two students (as opposed to the typical teams of three); we posit this puts the students in a more difficult position as there is one fewer person to contribute to the team's tasks. In the future, we will make teams of three or four students. For the final team, on the TP, we read through the end-of-project reflections submitted by everyone on the team as well as their weekly TCRS responses. Issues were mentioned in the final reflection, as well as in some of the open-ended comments in the TCRS, but not the main Likert-scale questions. We have incorporated natural language processing [Nlt] into our flagging process to alert us to these issues.

5.4.2 Facilitating TA Engagement with Teams

As discussed in Section 5.2.4, we conducted a targeted intervention in Spring 2021 to have TAs engage with struggling teams. Several times during the semester, we also checked in with the TAs to see if they were using the checklist and how the discussions with teams in lab were going. Anecdotally, the TAs mostly reported that teams said things were "fine" and were hesitant to discuss issues. It is possible that the TAs need more training in crucial conversations [Pat12] so that they can more effectively discuss challenging dynamics with teams. Additionally, explicitly tracking when issues came up week after week could help the TAs take an increasingly hands-on approach for helping teams overcome their issues. Finally, if TAs are more forceful in reminding students that there are consequences for non-participation, it may encourage recalcitrant students to engage with team and the project.

5.4.3 Threats to Validity

In this section, we discuss different types of threats to validity.

5.4.3.1 Conclusion

In this work, we use project grades and peer evaluations as a proxy for *observed struggle* to identify teams that are in need of assistance from the teaching staff. While we can use both of these measures objectively, we have observed that they may not capture the true picture of what difficulties a team is facing. Indeed, when we read through end-of-project reflections, we found a team that received fine grades and no concerning peer

evaluations, but two members still reported that they were facing issues communicating and collaborating effectively. Work remains to be done in finding an oracle of team distress that is objective and accurate.

5.4.3.2 Construct

While we observe that many students were willing to reveal issues in their teams to the teaching staff, this was not universally the case. One student reported in their final reflection: "*I did not do this [mention a struggling teammate] however, because I did not want to create tension*". Prior work suggests that women and students from historically underrepresented minorities may be less assertive [Par15; Lea11], and consequently potentially less comfortable alerting the teaching staff of perceived issues. Further work to detect team issues that can complement self-reporting is necessary.

5.4.3.3 Internal

In Fall 2020, we merely deployed the TCRS each week to get a baseline observation for how capable it is at identifying struggling teams. Consequently, we did not look at the responses until the end of the semester, and no followups were performed based on them. Several students remarked in their end-of-project reflections that they wished there were consequences for issues identified, or that there had been more prompt followup. It is possible that students, frustrated that they were not getting any followup, started taking the TCRS less seriously as the semester progressed.

5.4.3.4 External

This study was conducted in the context of one course at one university over two semesters. While we received promising results, replication needs to be performed to validate the use of a survey for flagging teams and promoting self-reflection.

5.4.4 Future Work

We have identified several promising avenues for future work.

Students mentioned in their end-of-project reflections that they found the TCRS useful for weekly self-reflection and staying on top of tasks that needed to be completed (Section 5.3.3). We intend to probe this further, and conduct follow-up interviews with students at the end of a future semester to understand the TCRS' use as a self-reflection tool and how to further improve it.

As mentioned in Section 5.4.3, a fundamental limitation of the TCRS is that it requires students to be willing to share the issues they believe their team is facing with the teaching staff. While our results suggest many students are willing to do so, this places a burden on students that may be particularly unwelcome for women and students from underrepresented groups. Consequently, future work that focuses on identifying successful and unsuccessful patterns from version control systems can give early warning signs of team struggle in a way that does not require students to self-report the issues. We intend to tackle this next as a way to complement the TCRS for detecting and overcoming team struggle.

5.5 Conclusion

In this work, we have designed a weekly reflection survey for identifying struggling teams in undergraduate software engineering courses. By matching survey results against project grades, we have found that the survey can flag teams with observed struggle in most cases (with an overall success rate of 89.5% across two projects and two semesters), and typically can do so early enough in the project that the course teaching staff may be able to intervene and help the team perform better. We devised an intervention to try and foster discussion in struggling teams and identify a plan for overcoming their collaborative difficulties. Our intervention did not result in any grade improvements, yet most (64.4%) students nonetheless reported that the surveys helped keep them on track and provided a chance for weekly self-reflection. We are planning improvements to the survey and the course in light of these findings to offer better support for teams.

CHAPTER

6

PROMOTING SUCCESSFUL TEAMING OUTCOMES FOR SOFTWARE ENGINEERING STUDENTS

This chapter[1] explores the difficulties that undergraduate students face when collaborating on software engineering team projects. In this study, we interview students who had recently completed a team-based software engineering course to understand how they ran their teams, what challenges they faced, if any, and how they tried to overcome those challenges. In addition, we consider how students engaged with the collaboration reflection survey from Chapter 5, to understand how it encourages students to self-reflect on their project experiences and how this can contribute to successful teaming outcomes. The work in this chapter provides a better understanding of the types of challenges that students face, and motivates improvements to project materials to benefit future students. This study demonstrates the value of the collaboration reflection **survey**, and provides insights that have inspired improvements to course materials, resulting in **more effective**

[1]This study was published in substantial portion as Presler-Marshall, K., Heckman, S. & Stolee, K. "What Makes Team[s] Work? A Study of Team Characteristics in Software Engineering Projects". *Proceedings of the 2022 ACM Conference on International Computing Education Research*. ICER '22 [2022].

instructional materials that will contribute to **improved learning outcomes** for future cohorts of students.

> Satisfies part of thesis: Using software engineering automation and **survey techniques** in **computer science education** results in **improved student learning outcomes**, early prediction of struggling teams, and **more effective instructional materials**.

6.1 Study Rationale

Professional software engineering is, almost without exception, a team-based activity, drawing together diverse teams to solve large problems. To help prepare students for this reality, teaming is taught in many computer science programs and is also a skill assessed by ABET accreditors [Abe].

In this work, we focus on student teams, and we observe that in software engineering courses, not all teams manage to work together effectively. Some students may have a sufficiently dysfunctional team experience that they are not able to learn key skills of how to manage and run a multi-member team. Students regularly complain about *freeriders*, or team members who fail to contribute equitably to the project, resulting in more work and stress for everyone else [Hal13]. Peer evaluations may be able to discourage freeriding [Taf16; Hal13], but are not a general-purpose tool for addressing all teaming challenges. Indeed, while teams may be hampered by the explicit non-participation of one of their members, they may also be frustrated by a general sense of confusion and disorganisation that negatively impacts the entire team [Oak04]. However, the precise details of the challenges that software engineering teams face have not been the focus of much research, which limits educators' ability to help teams overcome them.

6.2 Introduction

In this work, we look beyond issues of non-participation, and seek to understand what makes teams work. We do so by identifying transient and persistent challenges faced by software engineering teams and attempts to overcome them. Additionally, we identify the characteristics of successful teams, which may serve as a model that educators can encourage students to adopt. We focus our efforts around the following research questions:

- **RQ12**: What sort of team-related difficulties do students face on software engineering teams?

- **RQ13**: Why are some teams able to overcome the issues that they face, while others are unable to do so?
- **RQ14**: What support do students want from the course teaching staff for overcoming collaborative difficulties?
- **RQ15**: What are the characteristics of successful teams?

We answered these research questions by conducting one-on-one interviews with students who have recently completed a team-based undergraduate software engineering course. These questions aimed to understand their experiences, successes, challenges, and how they tried to overcome these challenges.

Our results show that while some teams manage to work together successfully throughout the project, communication issues and poor time management caused challenges that other teams struggled to overcome. Additionally, we find that self-reflection, a critical component of self-regulated learning [Pan17], helps some teams overcome challenges, but is not capable of motivating recalcitrant teammates.

Our contributions are as follows:

- A discussion of the characteristics of software engineering student teams that worked well together, and teams that struggled to work effectively.
- A discussion of the steps students attempted to overcome challenges.
- A discussion of how teaching staff can help struggling teams.
- Data suggesting that teams may face more collaborative challenges than was previously understood, calling for researchers to better understand issues teams face.

To the best of our knowledge, this is the first paper to study the characteristics of software engineering student teams from the inside, discussing with students to understand their experiences and challenges. All prior work we are aware of looks at external factors of team success, such as team grades [Iac20; Mar16; PM22a], peer evaluations [Iac20; PM22a], whether projects could be deployed [Mar16], or similar metrics [Dzv18]. By contrast, our one-on-one interviews with students provide novel insights into what teams do, and, from their perspective, what challenges were faced. This gives us a far richer perspective on how teams function, and lets us demonstrate empirically that software engineering teams function according to educational theory and many educators' intuitions.

6.3 Related Work

Practically all professional software engineers work in teams, bringing together a diverse set of skills to enable engineering the software systems of the modern world [Lay00; Ric12; Ram20]. While software teams have long been geographically distributed [Lay00], the COVID-19 situation has accelerated this trend, with more developers opting for fully remote work [Mil21]. Prior work suggests that remote work accentuates the challenges developers face working together [Bao20; Neu05], making it crucial that developers enter the workforce with teaming experience. As a result, most software engineering classes include some form of team-based learning, from pair-programming efforts to longer-running, many-member teams [Kha20; Par18; Wil00; Sim02; Iac20; Hun21].

However, despite that teaming is a key learning outcome [Abe], some students have a dysfunctional team experience that imperils their ability to learn teaming skills. Prior work has demonstrated that up to 40% of teams in project-based courses are characterised by "internal strife" and fail to work together effectively [Tuc06], often caused by a lack of communication or effective project management [Oak04; Iac20]. Iacob and Faily [Iac20] report that dysfunction is a risk in student software engineering teams, where low engagement or poor communication can hamper individual and team outcomes. They identify that there may be team dysfunction, but do not study its causes. Marques [Mar16] proposes having a "monitor" conduct weekly meetings with teams of software engineering students, observing them work and providing feedback on the overall team function and contributions of each member. They report mentored teams produced higher-quality software, and performed substantially better on their final project, but provide little elaboration on the details of the challenges that students faced in either case. Prior work [PM22a] investigates the use of a *team collaboration reflection survey (TCRS)* for identifying struggling teams. They report that the TCRS can identify struggling teams, but provide little insight into the types of challenges teams faced. Maguire et al. [Mag19] discuss how to train the mentors required by several of these approaches.

Computer science education researchers have considered student teams primarily by focusing on externally-visible characteristics, such as grades, peer evaluations, or version control system (VCS) commit history. Iacob and Faily [Iac20] and Marques [Mar16] focus on end-of-project grades as a measure of team success. Meanwhile, Gitinabard et al. [Git20] focus on VCS data to identify how small teams collaborated. Most similar to our work, Dzvonyar et al. [Dzv18] explore team forming and success in software engineering. They discuss considerations when forming teams for a project-based course, and survey teams at the end of a software engineering course, asking questions about team synergy and any

challenges the team faced. They report that team synergy was generally high, but two teams struggled with low motivation and performed poorly. However, they do not discuss how teams themselves operated and how this may have impacted any challenges faced. Finally, Berglund [Ber05a] conducts an in-depth study of an upper-level networking course. They consider how teams distribute leadership responsibilities and whether they function as a cohesive whole. However, their work considers only two aspects of how teams function, and the course context is very different from software engineering courses, where teaming is a primary learning outcome.

Prior work has considered the characteristics of student teams more broadly in engineering education. Borrego et al. [Bor13] present a comprehensive literature review of teaming in engineering education, and consider the learning outcomes and "negative behaviours" commonly associated with them. They show that teaming is widespread in engineering education, particularly in introductory courses and senior-level capstone courses. They report that social loafing, or *freeriding*, is the primary form of dysfunction faced by teams. They counter that freeriding can be reduced by having projects that are sufficiently complex that each student has a unique role [Kar93; Kar95], and that academically unbalanced teams (those featuring both high and low performing students) are at the greatest risk of freeriding [Pie10]. Beyond the issue of freeriding, they also consider how to promote teaming environments that lead to positive educational outcomes. They report that interpersonal conflicts between members of the team leads to "*reduc[ed] productivity and satisfaction*" [DW12] but that disagreements over how to solve tasks can help students consider a broader range of possible solutions and thus improve outcomes [DD03]. This finding has been echoed by other work [Hon04]. More recently, Walsh et al. [Wal21] consider impacts on team dynamics in engineering education during the COVID-19 situation. They report that teams experienced many of the same challenges that we observed, including issues with time management and timeliness, communication difficulties, forming effective relationships, and burnout and a lack of motivation. Finally, Pazos and Magpili [Paz15] propose interviewing engineering students to understand how technology can support better teaming.

Team-based learning (TBL) is a learner-centred pedagogy, where students direct their own learning under the guidance of an instructor who serves as an "expert facilitator" [Hry12]. TBL is grounded in constructivist theory, which argues that students cannot merely absorb information passively, but must actively discover it. This theory says that learning is done through dialogue rather than a dissemination of facts. Prior work has demonstrated that this is typically a more effective pedagogy and results in better learning [Hry12; Aya15]. For these reasons, the software engineering course we study in this work uses TBL exten-

sively. Despite the benefits, researchers have recognised that TBL is not an educational panacea. Successful teamwork depends upon regular communication, particularly when members work asynchronously [Gil13]. Additionally, teams must be capable of conflict resolution, which requires both identifying and resolving challenges [Pau06; Raf13]. In order to help students navigate these challenges, many educators include team forming activities [Rap07; Pin06; Hog08], self-and-peer assessment [Din14; Taf16], or discussions of teaming theory (such as Tuckman's model of teaming [Tuc65], discussed in Raferty [Raf13] and Hansen [Han06]). In this work, we use Tuckman's model for characterising where teams faced challenges. Tuckman argued that teams progress through four stages: *forming*, as the members of the team meet each other but largely act independently, *storming*, as conflicts and disagreements arise between members, *norming*, where conflicts are resolved and the team starts to function cohesively, and finally *performing*, where members are engaged and the team works together effectively. Tuckman noted that some teams may skip the *storming* stage entirely, while others may face intense "storms" they never overcome. Later, Tuckman and Jensen added a fifth stage, *adjourning*, where the group disbands upon the completion of their tasks [Tuc77].

6.4 Background

At NC State University, a large, research-focused university in the United States, undergraduate Computer Science students are required to take a Software Engineering course, typically during their third year. The course covers fundamentals in software engineering, including how to design, implement, and test a medium-sized object-oriented system; how to write requirements; and how to appropriately break down a project into manageable components, all in the context of two multi-week team-based projects. The first project, an *onboarding project* (OBP), introduces the process expectations and technology stack. The second, a larger *team project* (TP), tasks students with a more comprehensive project with a larger team. The OBP is completed in teams of three or four students; the TP in teams of five or six. Prior to team formation, the teaching staff distributes a Google Form to students, inviting them to fill out who they *would* or *would not* like to work with. The teaching staff makes a best-effort to build teams that satisfy these preferences. Avoidance requests are always satisfied, and students will usually get at least one, if not more, of the teammates that they request to work with. Projects are broken into iterations, each typically lasting one week, that cover different learning objectives: requirements and planning, design, testing, and implementation. Over the course of the projects, students are evaluated in five categories: technical deliverables (including both code and technical documents);

technical processes; project management; team collaboration; and peer review. At the end of the project, the course teaching staff reviews peer evaluations and contributions to determine whether individual adjustments are needed (positive or negative). We focus here on the *team project*; a larger team provides more interesting dynamics, and a more recent project is easier for students to remember.

The *team project* features several aspects designed to promote positive teaming outcomes. The first iteration features a team forming activity based on prior work [Rap07; Pin06; Hog08]. The teaming activity is facilitated by the course teaching staff, and encourages students to reflect on collaborative experiences in prior classes to identify the characteristics of successful teams. The activity features questions to facilitate discussion on what each student wants to learn, how they want to run their teams (including how they want to meet and communicate out of lab, how they want to resolve conflicts, and team roles), and timeliness expectations. Students are encouraged to establish individual feature-based roles to focus on specific tasks, as well as an overall team lead role. They are also encouraged to split their team approximately in half into two subteams to work in parallel, and establish leadership on each subteam. Teams are required to establish a real-time communication approach (i.e. something to supplement email) and produce a written document reflecting their discussion and the rules they have established, but are otherwise free to establish rules as they see fit. All members of the team are expected to sign the rules.

The Software Engineering course typically has between 120 and 200 students a semester, led by one PhD professor and three to five teaching assistants (TAs). To support team-based learning, the course features lab sessions each Thursday led by the TAs. Labs have 20-25 students each and provide time for teams to review work from the previous week and plan tasks for the next week. To ensure teams are prepared for the Thursday labs, weekly tasks are due Wednesday nights. Lab sections are run synchronously and while they have typically been run by a single TA, starting in Fall 2021, they are run by pairs of TAs. Due to the COVID-19 situation, labs were run online from Spring 2020 to Spring 2021, but have returned to in-person in Fall 2021. However, in keeping with safety protocols, students with a COVID exposure were asked to join their team by Zoom instead of attending physically.

In prior work [PM22a], we introduced a *team collaboration reflection survey (TCRS)* to the class projects. Administered through Qualtrics, the TCRS is mandatory and asks students to reflect on their contributions and how their team collaborated. The TCRS is capable of identifying a large majority of teams that ultimately perform poorly (with the team as a whole receiving a poor project grade, or one or more members receiving a low peer evaluation grade). In this paper, the TCRS is used as a tool as we seek insights into the

challenges that teams face.

All authors of this paper are regular members of the teaching staff for the studied course. The first author is head TA for the course; the third author was the instructor of record in the Fall 2021 semester we studied. The second author is a regular instructor for the course, but was not a member of the teaching staff in Fall 2021.

6.5 Methodology

In this section, we discuss how we improved the TCRS, classified teams based on their project experience, recruited potential participants, and conducted and analysed interviews.

6.5.1 TCRS Improvements

The TCRS features an open-ended question asking students to reflect on their project experiences over the past week. As suggested in prior work [PM22a], for the Fall 2021 semester we introduced natural language processing using VADER [Nlt] as an additional way to identify struggling teams from this response. VADER is a sentiment analysis tool, and produces a 3-tuple of (`positive`, `negative`, and `neutral` scores) representing the sentiments detected in a piece of text. However, in our context, rather than individual sentiment scores we need to answer "*Is this TCRS response describing a problem the team is facing?*" Thus, we need a binary classifier that combines together the individual sentiment scores to determine if a comment is predominantly negative (that is, describing a problem, which the teaching staff would like to know about) or not (describing instead that the team is working well, or effectively saying nothing at all).

To construct and evaluate a classifier, we built a labeled dataset. We read through the 579 open-ended responses on the TCRS from Spring 2021, and manually labeled each one as expressing a predominantly positive sentiment, a predominantly negative sentiment, or no sentiment. This gave us a dataset of 437 positive comments, 93 negative comments, and 49 neutral comments. We then ran VADER on each comment, and built a binary classifier from the `positive`, `negative`, and `neutral` scores it produced. As in prior work, we prefer a high recall (a large majority of negative comments correctly labeled) over high precision, so we tuned our classifier until the recall exceeded 90%, which gave a precision of approximately 55%.[2] While the precision is relatively low, the classifier successfully narrows

[2]These figures represent training error, rather than test error; the skew of our dataset towards positive comments means there is an insufficient number of negative comments for a typical training/test split.

down approximately 110 comments submitted each week to no more than ten negative comments for the teaching staff to review, and a quick manual inspection lets us discard comments that were incorrectly labeled as negative.

6.5.2 Team Classification

In prior work [PM22a], we used a two-part grades-based oracle for identifying struggling teams: low project grades and peer evaluation grades. In both cases, a threshold of 1.5 standard deviations below the class average was used; a team was flagged through the oracle if the overall team grade or any member's peer evaluation grade was below the threshold.

We adopted this same model, with the improvements discussed in Section 6.5.1, and then compared teams flagged through the oracle against teams flagged through the TCRS. The TCRS is due each week as part of project tasks, and was analysed to identify struggling teams. Therefore, while the oracle represents team struggle at the end of the project, the TCRS represents a metered lens into struggle throughout. We cross-referenced teams flagged through the oracle to teams flagged through the TCRS, splitting the 24 teams in the course into four distinct groups:

- **Group 1**, *eight teams*: Teams that were not flagged through the grades-based oracle, and were flagged ≤ 1 time through the TCRS. These are teams that ultimately did well, and any issues faced appeared to be transient.
- **Group 2**, *seven teams*: Teams that were flagged through the grades-based oracle, and were flagged ≥ 2 times through the TCRS. These are teams that ultimately received poor grades, and issues were seen consistently through the TCRS.
- **Group 3**, *three teams*: Teams that were flagged through the grades-based oracle, and were flagged ≤ 1 time through the TCRS. These are teams that ultimately received poor grades, but issues showed up at most briefly through the TCRS.
- **Group 4**, *six teams*: Teams that were not flagged through the grades-based oracle, but were flagged ≥ 2 times through the TCRS. These are teams that appeared to struggle during the project itself, but the issues did not manifest themselves in low grades at the end.

6.5.3 Recruitment Process

From each group, we randomly selected three teams for analysis, with the exception of Group 4, where we made an administrative error and only selected two teams. We then

Table 6.1 The number of students contacted, and who participated in interviews, from each of the groups studied. Also shown is the number of teams represented in our interviews.

	Students Contacted	Students Interviewed	Teams Represented
Group 1	17	8	2
Group 2	15	3	2
Group 3	16	4	3
Group 4	10	3	2
Total	58	18	9

sent individual recruitment emails to each member of the selected teams, inviting students to discuss their project experiences. Recruitment emails were sent in January 2022, and interviews were conducted in late January, approximately two months after the conclusion of the project. We did not ask students to hide their participation in the study from their teammates.

This study received IRB approval. Participation was voluntary, and students were not compensated for participating. Willing students signed up for an interview timeslot from a provided calendar; we then sent them Zoom information and a consent form. Students were asked to sign and return the consent form before their interview slot. Every student who signed up participated in an interview. As shown in Table 6.1, 18 of the 58 students we invited participated, for a response rate of 31%. The 18 interviews represent nine of the eleven teams we contacted.

6.5.4 Interview Process

To ensure that all students were asked the same core set of questions, we prepared a semi-structured interview outline, shown in Figure 6.1. Students were free to direct the conversation, so questions were not always asked in the same order or with exactly the same wording, but we asked the same core questions in each interview. As discussed in Section 6.5.5, after the first three interviews, we added questions on teams' communication and leadership approaches (Q5 and Q6). All interviews were conducted by the first author.

6.5.5 Analysis

To analyse interviews, we followed a grounded theory approach [Cha14], transcribing and performing preliminary analysis concurrently with interviewing. To reduce bias, we replaced all student names and pronouns with gender-neutral pseudonyms. As suggested

by Saldaña [Sn09], we began with structural coding [Nam08], identifying a preliminary set of codes and categories from the interview script and common themes. During this process, we added Q5 and Q6 to the interview script, shown in Figure 6.1. To supplement our initial set of codes and categories, we followed an open coding approach [Kha09], letting insights from each interview guide our analysis, and revisiting prior interviews to see if and how each newly-discovered topic was discussed.

On five of nine teams represented in the interviews, we interviewed two or more members. Consequently, we compared interviews within a team for consistency and contradiction. As expected, we found students emphasised and discussed different aspects of their experiences, but we found no contradictions between different members of the same team. We verified claims where possible, checking Git logs to confirm comments on the timeliness and distribution of labour. In no cases did we find information substantially different than what students told us.

As we compared student responses, we found that the nine teams could be arranged into four distinct categories, depending on whether they faced collaborative challenges and how effectively they overcame them. We use Tuckman's model of teaming [Tuc65], as discussed in Section 6.3, to characterise team experiences:

- **Category I: Ineffective Collaboration**. Teams that faced a substantial collaborative issue they were unable to overcome. These teams never successfully made it past the *storming* stage of Tuckman's model. We name these three teams *Alpha, Bravo* and *Charlie*.

- **Category II: Partially Ineffective Collaboration**. Teams that faced a substantial collaborative issue which they were able to partially, but not fully, overcome. These teams struggled to move through the *storming* stage of Tuckman's model, and while they made more progress than the teams in *Category I*, they faced conflict until the end of the project. We name these two teams *Delta* and *Echo*.

- **Category III: Effective Collaboration with Issues**. Teams that faced a collaborative issue which they were able to fully overcome. These teams lingered in the *forming* stage of Tuckman's model. We name these two teams *Foxtrot* and *Golf*. Foxtrot lingered in the *forming* stage for approximately two weeks, and *Golf* for a bit over one week.

- **Category IV: Effective Collaboration**. These teams never faced acknowledged collaborative issues. They progressed through the forming, norming, and performing

stages without difficulties, with little sign that they faced a *storming* stage at all. We name these two teams *Hotel* and *India*.

To answer **RQ12**, we focused on the challenges faced by **Categories I-III**. To answer **RQ13**, we focused on **Categories I-III**, looking at how they tried to overcome these challenges and comparing the fully successful attempts in **Category III** with partially successful attempts in **Category II** and unsuccessful attempts in **Category I**. To answer **RQ14**, we focus on students from all categories, soliciting feedback on steps that were or could be taken by the teaching staff to help address similar issues. Finally, to answer **RQ15**, we focus on what students from all categories described as the successful attributes of their teams, paying particular attention to teams from **Category III** and **Category IV**.

6.6 Results

Here, we present the results on challenges teams face (RQ12), teams' ability to overcome them (RQ13), support for teams from the teaching staff (RQ14) and the characteristics of successful teams (RQ15).

6.6.1 RQ12: Challenges Faced

First, we sought to identify the challenges that impeded effective teamwork. We focus here primarily on answers to Q1-3 and Q5-15 from our interview script.

6.6.1.1 Communication Difficulties

We find that poor communication underlies most team issues. Two teams from Category I and all four teams from Categories II and III reported that, at least for part of the project, their teams did not communicate effectively, leaving them unclear about their current progress. Adrian, on Team Alpha, described the communication difficulties on their team succinctly: "*I would text things in the chat, and there would be radio silence*". For some teams, communicating effectively was even a challenge within lab. On Team Bravo, Finnegan said that "*the first few labs it was like almost like silent...and I don't really know what's going on, [or] what are they doing*". Meanwhile, Spencer, from Team Delta, reported that their teammates were "*a little ashamed that they hadn't started*" and would not respond until the very last moment, once they had actually started. This lack of communication meant that "*a lot of times we didn't know if [individual tasks] were done or were going to be done before lab*". As we discuss in Section 6.6.2, Team Bravo managed to partially overcome these issues, but communication deficiencies impeded many teams.

We also find that the communication platform students used and how they used it impacted their communication efforts. Of the nine teams, eight chose Discord, a popular channel-based, text-and voice chat program [Dis]. Only one team, Alpha, used anything else: SMS text messages, which Adrian described as an "*awful*" solution that "*discouraged*" necessary conversations. However, despite that all remaining teams used Discord, some used it more successfully than others. Page, from Team Foxtrot, reported that their team had a "*big group*" channel for the entire team. However, rather than using this, or something else that would be visible to the team, Page and the other member on their subteam communicated via direct messages (DMs). They described DMs as the "*obvious*" approach, but later reported that "*[I] no idea what was going on on the second subteam because I didn't talk to any of them*". In Section 6.6.4 we discuss how more successful teams used Discord to communicate.

Finally, we observe that language barriers can contribute to communication challenges. We interviewed two teams with an ESL[3] student; one of them cited this as a major challenge. In Section 6.6.2 we discuss how Team Bravo partially overcame this issue. Meanwhile, Parker, from Team Golf, reported that their team experienced a minor language barrier with one student, but that "*we adjusted to it and we did okay*" and it did not impede their work.

6.6.1.2 Time Management

The second most prevalent issue that teams faced is one of time management and accountability. Two of the three teams from Category I and both teams from Category II reported that they had issues getting work done on time, with a tendency to wait until the last moment. Spencer, on Team Delta, explained that "*we all kind of were pushing what we need to do back*". They reported that the team often would not *start* tasks until after the Wednesday deadline, completing tasks between the deadline and the start of their lab session the next day: "*we fudged that a little bit...not really doing [our work] until like the next day, Thursday...and that didn't work*". When technical issues arose, it put their team in an untenable position, since "*you know how it is, there's not really time to figure it out*". Emery, from Team Charlie, also reported issues with procrastination: "*nobody would start it early...everything was done at the last possible minute*". Ultimately, Team Delta and Team Alpha managed to get their projects done, although Adrian reported that it took "*an all-nighter that night it was due*". However, on Team Charlie, Emery reported that "*we didn't have time to fix [several broken pieces]*".

[3]English as a second language

Exactly *why* teams struggled to get work done on time varied from team to team. On Team Delta, Spencer reported that "*the team doesn't keep you accountable*" and at one point on the project, "*other than Kennedy, nobody else really cared*". Spencer admitted that their team was "*more concerned about the grade than actually learning*". More pressure from the TAs may have been necessary. Emery was more optimistic about their team's process, and said that if their approach of "*just make sure you do it before the deadline*" was replaced with "*days with [the] team to work on things*" they may have worked more effectively.

6.6.1.3 Task Planning

Our results also suggest that teams struggle with task planning and organisation, which tended to exacerbate other problems. Spencer, from Team Delta, reported that "*we didn't really have a good execution plan*". They wished that their team had been "*more specific about the [meeting] agenda*". Meanwhile, Adrian, on Team Alpha, reported that while their team managed to figure out *what* needed to happen, "*those [time] estimates that we had were just completely wrong*" and resulted in an "*unequal distribution*" of labour. Additionally, Adrian reported that "*we didn't really think about "* task dependencies, which meant that they were often blocked because "*some things were dependent on other things and you didn't necessarily know that*". Incorrect time estimates caused troubles for Hayden and Team Echo as well. In their case, during the final iteration Nuru was assigned a "*process [that] took way less time than we thought*". Nuru then went to work at their day job, and was unable to help their teammates, who were "*helping with the process that took a lot longer*". While Hayden said that "*overall I don't blame [Nuru]*" because "*[they] did all work that we assigned*", this created difficulties for everyone else.

6.6.1.4 Other Issues

We find some evidence that team leadership strategies also impact team success. Adrian, on Team Alpha, reported that their teammate Casey established themself as team lead. Adrian reported that Casey largely refused to delegate tasks, and would go and "*[change] my code for me...and edit the things I've done*". Adrian described Casey's leadership style as "*abrasive*". Other teams had varying degrees of success with their leadership approaches. Hayden, from Team Echo, acknowledged that the decentralised leadership approach of their team "*usually doesn't work out*" but said "*it worked out perfectly*". However, Team Echo only managed to pull together a working project through a crunch at the end, and we question if things worked as smoothly as Hayden described. Meanwhile, on Team Golf, Parker reported their initial experience was very chaotic. They described their first two

team meetings as "*torture*", saying that "*some people were throwing out ideas, some people were just kind of silent*". Parker reported that their experience after establishing a team leader was much smoother.

Three teams also faced issues with mental health challenges. On Team Delta, Spencer reported that one team member "*sorta just disappeared*" and stopped attending class or participating in the team chat; attempts from the team and the teaching staff to contact them were unsuccessful. Emery, on Team Charlie, faced a similar issue with their teammate Alex, who the teaching staff was also unable to contact. On Team Bravo, Landon said they were struggling from "*burnout*". They described their performance as "*hot and cold*" and said that sometimes they were engaged with the team, and sometimes they "*[were] not able to perform*". We asked Landon about being referred to the counseling center for mental health support; they "*probably would not have taken advantage of it as I should have*" and "*would have denied*" help. We suspect that these issues were exacerbated by the COVID-19 situation.

On Team Echo, Hayden mentioned an issue unique to their team: two members of the team had jobs outside of school. Hayden reported that in the first couple weeks of the project, they "*[were] actually a bit hesitant*" because the team members with jobs "*never responded...until very late at night*". As discussed in Section 6.6.2, Team Echo was able to partially overcome this issue, but combined with task planning issues (see Section 6.6.1.3) it still presented a challenge.

RQ12: *We find that teams struggle with communication, setting and keeping to deadlines, and task planning. Some teams also faced challenges with leadership and burnout.*

6.6.2 RQ13: Overcoming Challenges

In this section, we discuss the two teams that completely overcame their challenges, the two teams that partially overcame them, and the three teams that faced challenges they were unable to overcome. We focus here primarily on answers to Q1-3, Q12-13, and Q17 from our interview script.

6.6.2.1 Successful Attempts

We find that two teams completely overcame their challenges by addressing deficiencies in communication and leadership. On Team Foxtrot, Page reported that in the first half of the project, they had "*no idea*" what the other subteam was doing, because they "*didn't talk to them*". The turning point was when the team "*completely missed*" one weekly task. Page

explained the grade "*really hit us*" and the team realised "*we really need to start [talking]*" to stay on top of tasks and "*make sure [a bad grade] is not a trend*". They remarked that while this oversight "*could have turned into a blame game very easily*" their team "*handled it very gracefully*". Everyone realised that "*no one was told to do this*" and consequently, responsibility fell on the team. Ultimately, Team Foxtrot overcame this issue through pair programming, which facilitated communication between the subteams. Additionally, the team "*talked actually quite a bit with*" a TA to make sure they "*were 100% prepared*". Team Foxtrot faced no further issues and worked together effectively henceforth.

Team Golf managed to overcome their issues just as effectively. Parker described the first two team meetings as "*torture*" as the team meandered aimlessly. Parker explained that when they sat down to complete the TCRS, they "*put [their] thoughts about that week together*" and realised the team needed a plan. Parker credits self-reflection for identifying the problem, and thought without it they "*could have just been really disappointed and demotivated*". To address this, their team instituted a "*rotating team leader*" who could steer discussions. Parker said that as their team figured out the project, they used the team leader less, but that it was still "*nice having it there...as a safety button*" if needed.

6.6.2.2 Partially Successful Attempts

Team Delta originally faced severe issues on several fronts: poor communication, last-minute work, and next to no collaboration. Spencer reported that they tried, largely unsuccessfully, to organise the team, "*volunteering to be in the library*" and asking for progress updates. They described the inflection point for their team as the week that they got sick. Spencer said "*when I wasn't there they had to step up*" and this "*kick-started*" the other members of the team into participating. Additionally, they remarked that "*we realised how there's a lot of work left to do*" and not much time. While Team Delta never heard from the missing member, Spencer said that "*once we started working*" their team started making some progress, although they conceded the team remained more focused on "*[the] grade than actually learning*". Ultimately, their desire for a better grade encouraged better collaboration.

Team Echo partially overcame their issues by working around the schedules of the members with jobs. Hayden said that after a couple of meetings in the library when "*these two people were just MIA*" their team scheduled meetings for "*weekends and evenings*" to accommodate everyone. Ultimately, while Team Echo struggled with time estimates until the end, they made progress working together.

6.6.2.3 Unsuccessful Attempts

While four teams managed to overcome many of the challenges they faced, three more did not. Team Alpha was ultimately unable to overcome Casey's "*abrasive*" leadership. Adrian said while everyone privately agreed that Casey was behaving unreasonably, they were "*scared or hesitant*" to call out the problem, only to "*be deflected, and be gas lit*" by the member causing it. Adrian self-described as "*not a very confrontational person*" and preferred to suffer through the problem rather than speak up. Meanwhile, Team Charlie had one member of the team drop the course, leaving more work for everyone else. As the team never established effective leadership or held to deadlines, they struggled with many tasks until "*the hour before [they were] due*". As we discuss in Section 6.6.3, Team Charlie may have needed external accountability.

Finally, Team Bravo was in a unique position. Finnegan reported that three members of their team "*were very involved, and very willing to like [do] anything*". They said "*it was productive for us three*" and the team identified tasks, Finnegan delegated them, and they shared progress. However, Finnegan explained that tasks assigned to the remaining two members "*wouldn't get done...and would put us in [a] bad position*". Finnegan conjectured this was partially due to a language barrier, as Glenn "*had a hard time understanding us*". Ultimately, Glenn collaborated with Max, who came from a similar cultural background. Landon also struggled to participate effectively on Team Bravo. Finnegan reported that Landon missed every out-of-lab meeting. Landon acknowledged being "*aware of the problem*" but was "*not in a position*" to solve it. There was no clear way the team could have overcome this challenge.

RQ13: *When everyone on a team is making an effort to participate, reflecting on what is working and what is not working can be enough for teams to figure out what they need to do differently. Other teams may need the pressure of an impending deadline, or external motivation from the course teaching staff, to encourage everyone to contribute. Finally, abrasive leadership and mental health challenges posed insurmountable barriers for other teams.*

6.6.3 RQ14: Support from Teaching Staff

In this section, we discuss how the course teaching staff can help teams work more effectively, by focusing on answers to Q4, Q8-11, and Q13-16 from our interview script.

In every lab, TAs meet with each team to check on their progress and offer guidance. However, as students do most of their work independently outside of lab, it is often difficult

for TAs to identify team dynamics and whether everyone is doing their part and the team is working effectively together. Consequently, in prior work [PM22a] we assembled a checklist-based intervention to supplement the TCRS, providing questions for TAs to ask teams on communication, collaboration, and project management. Additionally, the intervention encouraged TAs to conduct mid-week email checkins with struggling teams to help hold members accountable. Prior work found no improvement in grades from this intervention. In this work, we aim to identify why and how to fix it.

As part of our interviews, we asked students how we, as members of the course teaching staff, could help them overcome collaboration issues. For teams that faced no issues, we asked how we could help them overcome hypothetical issues similar to those we observed.

6.6.3.1 TA Interventions

Prior work demonstrated that this checklist-based intervention for helping teams overcome challenges was ineffective. Comments from students on struggling teams overwhelming tell the same story: they want more help, and more active engagement, from members of the teaching staff. On Team Charlie, Emery requested "*more of a guiding hand*" than a "*passing comment*". They explained that "*[no]body really noticed or paid attention to*" comments from TAs. On Team Alpha, Adrian expressed a similar sentiment. They said that group projects "*typically expect people to be confrontational about problems*", which they were not comfortable doing, remarking that "*it's easier to do things yourself than try to explain to [TAs]*" what is happening. Adrian requested that TAs "*be a facilitator*" because if "*a person in authority*" brings up problems, the team will be less likely to "*not just say it's all OK*". When we asked Blair, from Team Hotel, how TAs could help with a hypothetical team challenge, they commented "*not a lot of people would want to directly confront someone*" and suggested that TAs take a more active role.

We received slightly more conflicting requests from students on other teams. On Team Delta, Spencer requested that TAs "*step in earlier*" and let teams know when their grade "*might be affected in the future*". By contrast, on Team Hotel, Carson would prefer to let teams "*try and resolve it amongst [ourselves] for a week*" before the teaching staff intervenes. Educational theory supports Carson's suggestion, arguing that it is important for teams to *try* to overcome their challenges before getting help. Also on Team Hotel, Blair made a similar comment, saying "*if it's recurring and it's a problem*" then TAs should get involved, but they would like the team to try first.

In Fall 2021, a student remarked in their end-of-project reflection "*When we had a major contribution issue, I reflected on that in the [TCRS] and TAs were able to intervene*". This student clearly appreciated knowing their reflections were used to foster discussions.

However, we recognise that some students may feel they are being "called out" for what was said. Thus, we asked participants in this study if they would prefer we mention the TCRS, so that they know we are acting on them, or would they prefer that issues be brought up more generally, such as in the context of Github contributions. Spencer, on Team Delta, acknowledged both sides. They appreciated knowing that "*y'all take them seriously...you're actually reading them*", but said that if issues were brought up through the survey, they would "*immediately try and figure out who is doing this, who said this*". Consequently, they would prefer for us to "*depersonalise*" the comments. They said that it would be "*good*" to make a note of the reflections, but make it "*not the main reason*" or focus of the conversation. Eilian, on Team Hotel, described the tradeoff similarly. They said that "*it's really awkward*" for the TAs to say that "*according to the feedback some people weren't doing their part*". They suggested that TAs start by asking students what they have contributed and how they have collaborated, and then follow up with more probing questions and discussion if the answers did not appear to match what was in the TCRS. Eilian suggested that we could "*still acknowledge [the TCRS]*" but preferred that it would not be the primary focus. While Emery, on Team Charlie, said they "*wouldn't mind*" if comments were brought up through the TCRS, the overall consensus is students prefer for comments to be brought up without it.

6.6.3.2 Team Formation

Some students suggested changes to the team forming activity discussed in Section 6.4. Parker, on Team Golf, acknowledged that the team forming activity was "*a good idea*" but said it was "*really hard*" to do effectively on the first day of the project when "*you don't really know the people and you don't really know the project*". They said it would be helpful to take time in lab on the second week of the project to review the team rules and goals and identify "*are [these] still working?*". To encourage everyone to read project materials ahead of time and make the project easier to discuss, Hayden, from Team Echo, said that "*A small quiz...would have helped*". This suggestion was echoed by Sawyer and Corey, of Team Golf.

Spencer, on Team Delta, suggested encouraging members to share "*our specific strengths and weaknesses*", acknowledging that while they discussed "*our technical skills*" they never discussed "*how we worked, or if we were bad at getting started on things early*". They hesitated to call out teammates for not getting things done because "*you never want the first impression to be like 'Hey guys, you're not doing your crap'*" and expressed optimism that further team forming would help. On Team Foxtrot, Page said their team was formed from two smaller groups from the prior project: "*three kids already knew each other, and then I had someone from my previous group which was really nice*". They said this made

the team forming exercises "*really painless*", but acknowledged they "*didn't talk to any*" of the new team members for the first half of the project. They suggested "*swapping it [up] could have been more efficient*" as a way to introduce everyone. One of their teammates, Jamie, similarly said activities to "*break the ice between subteams, but without forcing them to cooperate*" may foster teamwork.

RQ14: *We find that most students want a more active role from the course teaching staff, using their position of authority to bring up issues and then guide teams to a solution, holding members accountable as necessary. Students also suggested improvements to the team formation activity as a way for everyone to get to know each other more quickly.*

6.6.4 RQ15: Characteristics of Successful Teams

In this section, we discuss the characteristics of successful teams, by focusing on responses to Q1-6 and Q16-17 from our interview script.

In many ways, the characteristics of successful teams were largely the opposite of the teams that struggled the most. Carson, on Team Hotel, said that their group "*did really well communicating*" throughout. Blair, also on Team Hotel similarly said they "*communicated really well using Discord*". Also on Team Hotel, Emerson said that with multiple Discord channels "*we could sort of compartmentalise different discussions*". On Team Golf, Jesse echoed this, saying that "*we weren't all trying to talk in the same channel*" but "*we could still see what the other [sub]team was doing*". On Team India, Riley explained everyone discussed the tasks they were working on and "*brought up and talked about*" any disagreements. On Team Hotel, Eilian said that they would "*just straight up tell them [team members] like 'stop'*" if they were distracting meetings.

In addition to regularly communicating, we find that the most successful teams also worked on tasks together. On Team Hotel, Carson said that members on their team would "*hop in the voice chat real quick [when] we needed help on this or that*". Eilian, also on Team Hotel, explained that their teammates were "*really attentive*" and would "*come on Discord...until 8 or 9 [PM]*" if someone got stuck. On Team Foxtrot, Page said that a prior internship gave them experience with some of the technologies the project used. To get their teammate up to speed, they would "*almost strictly work in pair programming at first*", and remarked that "*I think that was actually really good...I....was able to teach [them]*". On Team Golf, Sawyer said that their teammates served as effective mentors, and would be "*like a guide to me*". They credited this relationship for helping them learn a new technology that they struggled with on the previous project.

We also find that successful teams held themselves accountable. Emerson, on Team

Hotel, said they would hold scrum-style meetings once a week, and "*post updates in the chat on certain days*". Because of this, "*no one put their work to the last minute*". They said that the "*implicit shame*" of showing up to meetings unprepared ensured that everyone did their work. On Team India, Riley explained they "*would have a meeting every Monday afternoon*" to discuss progress and come up with a plan for "*anything that needed to be done*".

We find the most successful teams invested heavily in task planning. Sam explained that Team India would "*split up each thing and just estimate it...we tried to split every task [so they're] pretty small*". After estimating times, they would create a "*wheel [of] fortune thing with all of our names*" and assign tasks at random, ensuring that each person had "*the same amount of hours*" of work to complete. Sam said that this process was "*so much fun*" and ensured that everyone was engaged with the process. Team Golf took a more conventional but equally involved approach to task planning. Parker explained their team "*took every task [for the week]...and put all of them on a [white]board*", at which point each member of the team "*took turns grabbing what we wanted to do*". They said this approach was "*great for learning*" because everyone got a chance to do "*a little bit of everything*", but conceded that "*with our stochastic approach, it was a lot harder to coordinate*" dependencies. Jesse echoed that this planning approach worked well; they said that after the team planned out everything, Team Golf would photograph the plan and share it in Discord, at which point "*everybody knew what needed to be done*". Team Hotel used technology to facilitate their task planning. Carson explained that "*we utilised Github Projects*[4] *and the [Github] Issues like religiously*" and said this worked effectively.

We find that successful teams considered team leadership, but carried it out in a largely decentralised manner. On Team Hotel, Emerson served as initial team lead, delegating tasks and ensuring that everyone knew what to do. Over time, however, as the team "*got familiar with what worked for us*" leadership became a shared effort. On Team India, Sam explained "*a tendency to take leadership roles*", but said their role was more "*get[ting] conversations going and ideas moving*" than directing people. Meanwhile, Riley, also on Team India, described their leadership approach as decentralised, explaining that "*we all served as like passive leadership*". On Team Golf, Parker described how their team instituted a leader role after an initial rudderless week where the meetings were disorganised to the point of being "*torture*". Jesse, also on Team Golf, explained that the team leader's role was "*just to keep the meetings on track*" and figure out "*what needed to be talked about...and make sure that's what was talked about*". They said this approach "*worked really well*" for their team.

[4]A Kanban-board style task tracking system

RQ15: *Successful teams communicate regularly, sharing their current progress with the team. Additionally, members of these teams regularly collaborated on their tasks. We also see that these teams plan out tasks carefully and then hold themselves accountable. These teams consider leadership roles, but largely function in a decentralised manner, trusting each person to ask for help as needed.*

6.7 Discussion

In this section, we discuss the significance of our results and how they compare to our prior work (Section 6.7.1), consider whether team formation impacts team success (Section 6.7.2), discuss possible improvements to team forming (Section 6.7.3), and finally discuss threats to validity (Section 6.7.4).

6.7.1 Significance of Results

We observe that none of the characteristics of successful and unsuccessful teams, as discussed in Section 6.6, come as a surprise. Prior work in teaming, as discussed in Section 6.3, has identified that student teams are at risk of dysfunction [Tuc06; Oak04; Iac20; PM22a; Dzv18], particularly due to challenges with communication and project management. However, to the best of our knowledge, this is the first work to study student teamwork in software engineering from the perspective of the teams themselves, rather than external factors. Thus, we offer the first concrete evidence that the student-identified challenges faced in software engineering teams are consistent with broader pedagogy.

Our results here suggest that teams may face more challenges than grades alone reveal. All members of Team Alpha ultimately received an A on the project, but the team was nonetheless flagged several times through the TCRS for having collaboration issues. In this case the team did not work together as well as the grades indicated. This suggests that collaborative challenges may be greater than previously understood. Given the four distinct stages Tuckman [Tuc65] argues teams progress through, the six weeks of the project may simply not be long enough for all teams to overcome their challenges and work effectively.

6.7.2 Teammate Requests as a Predictor for Team Success

As discussed in Section 6.4, teams are formed by the teaching staff with student input. We sought to understand whether there was any relationship between team formation and success. We did so by looking at how each team was formed, and classifying it as A) at

random, B) around one group of students who requested each other, or C) around two or more groups of students who requested each other. For example, Team Alpha falls into Group B, as four students mutually requested each other and they were matched with one additional student. By contrast, Team Hotel was formed from a group of three students who requested each other, a group of two students who requested each other, and a student with no requests, so Hotel falls into Group C. We find the formation of the team has little bearing on how effectively it worked. Carson, on Team Hotel, was the only member of the team who did not have another "buddy" that they had requested, but reported being "*grateful to have a really good group*". By contrast, on Team Alpha, the conflict that Adrian reported was between four people who mutually requested each other. Likewise, on Team Echo, the conflict was between two people who mutually requested each other. On Team Golf, which faced and overcame a collaborative challenge that was not due to any member in particular, three members mutually requested each other, but peer evaluations and comments in interviews showed that all six members were happy together. Overall, we find no clear relationship between how a team was formed and its success.

We note that although the teams studied in this work were enrolled in a synchronous, in-person class, this class followed a year and a half of online classes. Prior work has argued that students struggle to form effective relationships when working online [Wal21], which are necessary for establishing the trust that supports positive teaming outcomes [Bor13]. It is possible that students with a more normal educational trajectory would have established relationships that support more effective teaming.

6.7.3 Student-Suggested Improvements

Most of the project changes suggested by students offer a clear pedagogical improvement. The suggestion that we add a small quiz to encourage students to read project materials is used in flipped classrooms [Wil13], and compelling students to prepare for the team forming activity may ward off the disorganisation that Parker described as "*torture*". Page, on Team Foxtrot, suggested that we make the final question on the TCRS, *How do you feel about your team's collaboration process in this project?*, mandatory, which may encourage further self-reflection. Hayden, on Team Echo, suggested that we allow teams to submit a redacted chat log demonstrating peer review in place of the review in lab, which can be accomplished with a rubric to ensure equitable grading [Fel18]. Indeed, if this encourages students to engage in more peer review outside of lab, it would have clear benefits by promoting more timely and collaborative work. We have found both of these behaviours are associated with the most successful teams, and encouraging students to engage in them

is advantageous.

Currently, both project deliverables and the TCRS are due Wednesday night, to prepare for labs on Thursday. Emery, on Team Charlie, said that they would "*put in the survey what I would expect people to submit*", but their teammates would often not follow through, and thus their responses did not accurately represent team progress. We recognise that as a *reflection* survey, it may make more sense to have it due after other tasks, but unfortunately with one lab per week, this is not feasible. Emery suggested that we could "*allow for like, a second survey after lab*" if students had any followup comments. Combining this with a dashboard for viewing team challenges over time could help the teaching staff track persistent issues.

6.7.4 Threats to Validity

In this section, we discuss threats to validity, using categories suggested by Wohlin et al. [Woh12].

Construct: When we ask students what worked effectively for their teams, and what they struggled with, they may interpret success differently than we do, focusing on what actions led them to a higher grade rather than better learning. We asked clarifying questions to focus them on collaborative behaviour when their answers did not match the questions.

Responses are subject to hypothesis guessing, particularly as interviews were conducted by a member of the course teaching staff. However, participants were forthcoming about both the strengths and weaknesses of their teams, suggesting a willingness to discuss their experiences frankly. Additionally, as discussed in Section 6.5.5, we compared responses from teammates, and checked what we could on Github, and found no misrepresentations.

Internal: In this work, we describe the characteristics of struggling teams and teams that worked together effectively. Students reported that when they took the steps towards behaviour we see associated with successful teams, they did better. This suggests a causal relationship.

The responses are subject to recall bias, as there was two months between the conclusion of the project and the interviews. We cross-referenced answers from students on the same team, and checked what information we could against Github, and found no misrepresentations. Not all information, such as how students met outside of lab, could be externally verified, so we must rely on what students said. Aside from what we were told by students, we do not have information on their other obligations, such as how many classes they are taking or whether they have day jobs.

External: This study was conducted with 18 students from one course and one semester

at one university. We caution that these findings may not be broadly representative of team-based software engineering projects, and encourage replication with students from different courses.

Reliability: The interviews for this study were conducted by the first author, and the data analyses primarily by them. However, after minor changes following the first several interviews, our interview script remained unchanged, and all subsequent interviewees were asked the same questions. This step improves reliability as all interviewees were asked a consistent set of questions.

6.8 Conclusion & Future Work

In this work, we have studied how students on software engineering teams work together. We have revealed that students face issues communicating, establishing and keeping to deadlines, and estimating the difficulty of tasks. Some teams are able to overcome these issues, partially or completely, by reflecting on what is working and not working, or through external motivators such as grades. However, mental health challenges and intransigent teammates remain a challenge, suggesting that instructors need to do more to offer support for struggling teammates and encourage better behaviour. Additionally, we discuss the characteristics of successful teams, and report that these teams stay in regular communication, using Discord to facilitate asynchronous discussion and holding meetings to work on tasks together as a team. Members on these teams hold each other accountable and support each other. We consider suggestions that students offer on how to provide more effective feedback and guidance.

This work has identified behaviours associated with struggling teams and ones associated with successful teams. We encourage future work to identify whether an intervention can steer teams towards these latter behaviours. Additionally, as our study is limited in scope, we suggest a replication of this work with a course where students work in the same teams all semester to see both what challenges teams face and whether they are more successful at overcoming them when they are working together for sixteen weeks as opposed to six.

Collaboration:

Q1: Could you tell me about your collaboration experience on the TP?

Q2: Could you tell me one thing about your collaboration experience that you think worked out well on the TP and one thing that could have been improved upon?

Q3: Is there anything that you would definitely do again, and anything that you would definitely change?

Team Formation:

Q4: At the start of the project we took a day for team forming, setting goals and rules with the team. Did you find this helpful at establishing a common plan for the project? If not, is there anything that you think could have been done to improve things? If it worked, what did you find most helpful?

Communication and Leadership:

Q5: Could you tell me about how your team communicated outside of lab, and how, if at all, you met up together?

Q6: Could you describe your team's leadership approach?

If the team was flagged through the TCRS and grades-based oracle:

Q7: Did you find the TCRS & followup from TAs in lab to be helpful? Did it help you and your teammates do a better job splitting up tasks, communicating among yourselves, or ensuring that work got done and according to your standards?

Q8: Is there anything that you would have liked us to do differently based on the issues we observed?

Q9: Would you have liked us to bring up the issues we observed more directly, and been more explicit about telling people on the team what to do and requiring followups? Or would you prefer having the teaching staff bring up issues but leaving it more hands-off on how to resolve them?

Q10: Would you have liked us to mention we saw issues from the TCRS, or in a different way, such as through Github contributions?

Q11: Is there anything that you wish had been done differently on your team in regards to how we responded?

Q12: More broadly, is there anything you wish *you* had done differently?

Figure 6.1 Interview Outline

If the team was flagged through the TCRS, but not the grades-based oracle:

Q13: We saw from some of the TCRS responses (remind student of context) that there were some issues your team faced, but your team did well in the end. If you remember, could you talk a bit more about what was going on? What do you think helped your team overcome issues like these?

Q14: Were there any other issues that you or your team experienced that we didn't see here?

Q15: Would you have liked any help from the teaching staff to help overcome them, and if so, what sort of help?

If the team was not flagged by the TCRS or grades-based oracle:

Q16: Reading through the TCRS responses you (and your team) submitted each week, we didn't notice any issues that needed to be addressed. However, we realise that these don't necessarily capture the entire story. Thinking back over the course of the project, were there any issues that you encountered communicating or collaborating with your team?

Q16a: ***(if yes)*** Would you have wanted help from the teaching staff? What questions could we have asked that would have helped uncover them?

Q16b: ***(if yes)*** What sort of followup would you want us to take?

Reflections on self-reflection:

Q17: Did you find the TCRS helpful for self-reflection during the project?

Q17a: ***(if yes)*** What questions did you think were particularly helpful? Are there any changes we could have made to help make them better?

Q17b: ***(if no)*** Can you think about what we could have done to make this more useful to you?

CHAPTER

7

SUMMARISING INDIVIDUAL CONTRIBUTIONS TO TEAM PROJECTS

This study explores the possibility of using automation techniques to summarise individual students' contributions to a larger team project. We show that using **software engineering automation** can help TAs grade projects more consistently than they can do otherwise, and provide students with more nuanced and more actionable feedback. Taken together, these results offer **improved student learning outcomes** from the improved feedback quality, and **early prediction of struggling teams** by more consistently identifying students who are not participating.

> Satisfies part of thesis: Using **software engineering automation** and survey techniques in **computer science education** results in **improved student learning outcomes**, **early prediction of struggling teams**, and more effective instructional materials.

7.1 Study Rationale

Most professional software engineers work in teams [Ric12; Lay00; Ram20], and to prepare students for this, team-based learning (TBL) features heavily in many computer science programs [Sim02; Iac20; Bat22]. TBL is an active learning approach grounded in constructivist theory, and helps students learn by bringing them together to discover knowledge. Prior work has demonstrated that TBL is an effective pedagogy that offers improved learning outcomes [Hry12; Aya15], but it is not without its challenges. Some teams lack the communication and project management abilities to effectively function together as a team [Tuc06; Abb17; Dzv18]. Other teams are hampered by the non-participation of one or more members, leaving more work for the remaining members. Peer evaluations are commonly used to discourage this, but they may be biased [Din14] and not effective for drawing comparisons across teams [Bru10].

Consequently, to encourage and reward participation, instructors need other ways of identifying the contributions of each student. The most common approach is outsourcing grading to TAs, who are responsible for reviewing individual contributions and providing grades and feedback. Unfortunately, TAs may struggle to provide students with consistent [Gla15; Hay03] and actionable feedback. While providing a grade deduction can serve as a motivator to do better, the most effective way for students to learn is providing formative feedback with specific comments on how to improve [Bla10; Wis20a]. For students to improve, they must receive consistent feedback [But95]. However, TAs may provide inconsistent grades and feedback that impedes learning.

Accurately identifying individual contributions can help instructors provide students with feedback of what is expected of them, thus improving learning outcomes. Many computer science courses use version control systems such as Git [Tor], where *commits* explicitly track the contributions of each student. However, just because the information is available through Git does not mean that it is in a format that facilitates identifying precisely what each student has contributed. Consequently, contributions may be overlooked, harming the fairness of the grading and providing inconsistent feedback. For female and minority students, who may be less assertive and less willing to contest feedback provided [Lea11; Par15], the effects may be particularly severe. Consequently, it is essential that the teaching staff can accurately identify individual contributions to both encourage and reward participation, and provide students with feedback on when contributions do not reach the expected mark. In this study, we aim to identify whether automated contributions summaries can help TAs with this.

More formally, we frame our work around the following research questions:

- **RQ16**: *Can automated summaries of student contributions enable faster grading by TAs?*
- **RQ17**: *Can automated summaries of student contributions enable more consistent grading by TAs?*
- **RQ18**: *Can automated summaries of student contributions enable less frustrating grading from the perspective of a TA?*
- **RQ19**: *How do automated summaries of student contributions enable better feedback?*

To answer these questions, we designed an algorithm to summarise individual students' code contributions to team assignments, and built a reference implementation, AutoVCS, for Java projects tracked through Github. Our algorithm uses AST-based differencing [Fei16] and thus can present potentially more valuable summaries than tallying lines of code (LOC). We conduct a study with 13 current or former TAs to understand how they grade assignments when using automated summaries. Our results show that automated summaries can help TAs grade more consistently and provide more nuanced and actionable feedback.

The contributions of this study are as follows:

- an algorithm for summarising what individual developers have contributed to collaborative software assignments,
- an implementation of our algorithm as a tool, AutoVCS, which works on Java projects,
- a demonstration that our algorithm, implemented in AutoVCS, helps TAs grade lab assignments more consistently and provide students with more useful feedback, and
- a demonstration that TAs prefer grading with summaries from our algorithm

7.2 Background

At NC State University, CS1.5 is a Java-based course taken by all CS majors and minors and is open to non-majors. CS1.5 typically has between 250 and 300 students a semester, and is taught by one PhD professor and 12-15 TAs, giving a student:teaching staff ratio of approximately 20:1. CS1.5 teaches fundamentals in object-oriented design and development, basic software engineering concepts, and linear data structures. CS1.5 has a companion lab, where students work in teams of three to apply concepts covered in lecture. Students work together on the same team, and use the same Github repository, for three or four weeks, at which point teams are scrambled for the next set of labs. Students complete a

total of 11 labs with three different teams over the semester. As described by Heckman and King [Hec18a], the course uses Jenkins to give students immediate feedback on their code and automate most grading; TAs are responsible only for grading Javadoc (to ensure it describes the code) and individual contributions (which are evaluated by checking commit history on Github). We have observed that grading individual contributions is a slow task that TAs dislike. Prior work has shown that even with rubrics, precisely evaluating contributions requires subjective judgement [Jon07]; thus TAs are unlikely to draw meaningful single-point distinctions. Consequently, TAs provide coarse grades and feedback, giving a 0 ("No contributions"), 5 ("Insufficient contributions") or 10 ("Sufficient contributions"), typically with no further elaboration. Prior work argues that students want more feedback on their work [Li14], and we aim to identify whether our algorithm can assist with this.

7.3 Contributions Algorithm & AutoVCS

To identify whether automated contributions summaries can support grading, we developed an algorithm that uses commit histories and program analysis to summarise individual students' code contributions to team-based assignments. We then built a reference implementation, AutoVCS, which operates on assignments hosted on Github and written in Java. Our algorithm features three main steps; a full implementation is available in our Github repository [Aut].

1. **Metadata Extraction:** Metadata is extracted for each repository, storing commit hashes, dates and times for each commit, commit author, and a list of the files changed on each commit. This step is performed on Line 2 of Algorithm 1.

2. **Change Extraction:** Similar to work done by Feist et al. [Fei16], this step extracts changes made on each commit. It does so by traversing Git history to identify changed files, and then building ASTs from adjacent file revisions and computing an edit script between them. This step is shown in Algorithm 1 from lines 6 to 20. AutoVCS uses our improved version of ChangeDistiller [Flu07] to build and difference ASTs for each file.

3. **Contributions Summaries:** Detailed edit scripts for each file on each commit are aggregated to present higher-level summaries for each user; this is shown in Algorithm 2, and the resulting summaries are shown in Figure 7.1. Summaries are computed with three levels of granularity: Ⓘ a weighted [Gal09] sum of all contributions; Ⓘⓘ a summary of changes made across all files; and Ⓘⓘⓘ a summary of changes made to each file. Additionally, to allow for grading non-code contributions, a full list of commits can be shown

Algorithm 1 Contribution Summary Algorithm

```
1: procedure SUMMARISECONTRIBUTIONS(repo,[...])
2:     r ←initRepository(repo)                                   ▷ Extract metadata
3:     R1 ←clone(repo)
4:     R2 ←clone(repo)
5:     ContribsByCommit ← {}
6:     for commit c in r do
7:         if c.parent is null or c.isMergeCommit then
8:             continue
9:         end if
10:        ContribsForCommit ← {}
11:        Check out R1 to c
12:        Check out R2 to c.parent
13:        for ChangedFile in c.ChangedFiles do
14:            AstNew ← buildAST(R1.ChangedFile)
15:            AstOld ← buildAST(R2.ChangedFile)
16:            ContribsForFile ← diff(AstNew, AstOld)▷ Compute an edit script between
    ASTs
17:            ContribsForCommit.insert(ContribsForFile)
18:        end for
19:        ContribsByCommit.insert(c, ContribsForCommit)  ▷ Map each commit to the
    changes made as part of it
20:    end for
21:    ByUser ← summarise(ContribsByCommit) ▷ Summarise changes per-user, to include
    multiple commits
22:    return ContribsByUser
23: end procedure
```

(Ⓘⓥ). For the Java code supported by AutoVCS, Ⓘⓘ and Ⓘⓘⓘ use a condensed version of the change types proposed by Gall et al. [Gal09]; Ⓘ is a weighted combination of these changes. Option Ⓘ also uses the weighted contribution scores to compute a percentage contribution for each member. Individual code changes are summarised into four categories: 1) changes to classes, 2) changes to methods, 3) changes to documentation, and 4) all other changes. These are shown in Ⓘⓘ in Figure 7.1.

Prior work has shown that LOC represents the best, although still not particularly good, predictor for team grades [Mie05]. We hypothesise that part of the problem is that not all LOC changes are equal: languages such as Java contain substantial "boilerplate" code that is often auto-generated. To address this, AutoVCS recognises four common boilerplate methods [Kam; Cha] in Java code: `hashCode()`, `equals()`, getters, and setters. Changes to these methods are skipped so that contributions are not artificially increased by autogenerated code. In our course context, GUI files are provided by the teaching staff, so AutoVCS has a toggleable option to skip them.

AutoVCS is a web application, and can be run in interactive mode and batch mode. In interactive mode, the user selects a single repository and (optionally) a time window and summaries are computed and displayed. Batch mode runs from a JSON configuration file, and produces a summary page for each repository specified.

7.4 Study

To answer our research questions, we designed and conducted a two-part controlled experimental study with 13 participants to evaluate whether our algorithm, as implemented in AutoVCS, can help TAs grade more effectively (RQ16 & RQ17), make grading less frustrating (RQ18), or provide better feedback (RQ19). This study was approved by the NCSU IRB Office as protocol #24701. The study outline and research questions answered in each part of the study are shown in Figure 7.2. We recruited participants from two groups: a) 44 students who have served as TAs for two team-based undergraduate CS courses (CS1.5 and a third-year software engineering course) within the past two years, and b) all CS PhD students at NC State University with TA experience. Nine students from group a) and four students from group b) participated. The participants had an average of 6.6 years of experience with Java (median: 6) and 4 semesters of TA experience (median: 4). Four participants identified as female, and five as members of a minority racial group.

This study was conducted in four, two-hour lab sessions, held physically in a computer lab. All sessions followed the same procedures, and participants attended only one session. A brief outline of the study is shown in Figure 7.2. In Part 1 (Section 7.4.2), participants

Algorithm 2 Summarise Changes By User Algorithm

1: **procedure** SUMMARISEBYUSER(ContribsByCommit) ▷ For a group of commits and associated fine-grained changes, presents a summary per user and a contribution score per user
2: $ContribsPerUser \leftarrow \{\}$
3: **for** `(commit, contribs) in ContribsByCommit` **do** ▷ For each user, combine contributions
4: **if** commit.author not in ContribsPerUser **then**
5: `ContribsPerUser.insert(commit.author, {})`
6: **end if**
7: `ContribsPerUser.insert(commit.author, contribs)`
8: **end for**
9: $SummarisedContribs \leftarrow \{\}$
10: **for** `user, contribs in ContribsPerUser` **do** ▷ Summarise and weight contributions
11: $UserContrib \leftarrow 0$
12: $UserContribSummary \leftarrow \{\}$
13: **for** `contrib in contribs` **do**
14: `UserContribSummary.insert(label(contrib.type), existingCount+1)` ▷ Summarises contributions
15: `UserContrib += weight(contrib.type)` ▷ Computes weighted score of all of a user's contributions
16: **end for**
17: `SummarisedContribs.insert(user, {UserContrib, UserContribSummary})`
18: **end for**
19: `return SummarisedContribs`
20: **end procedure**

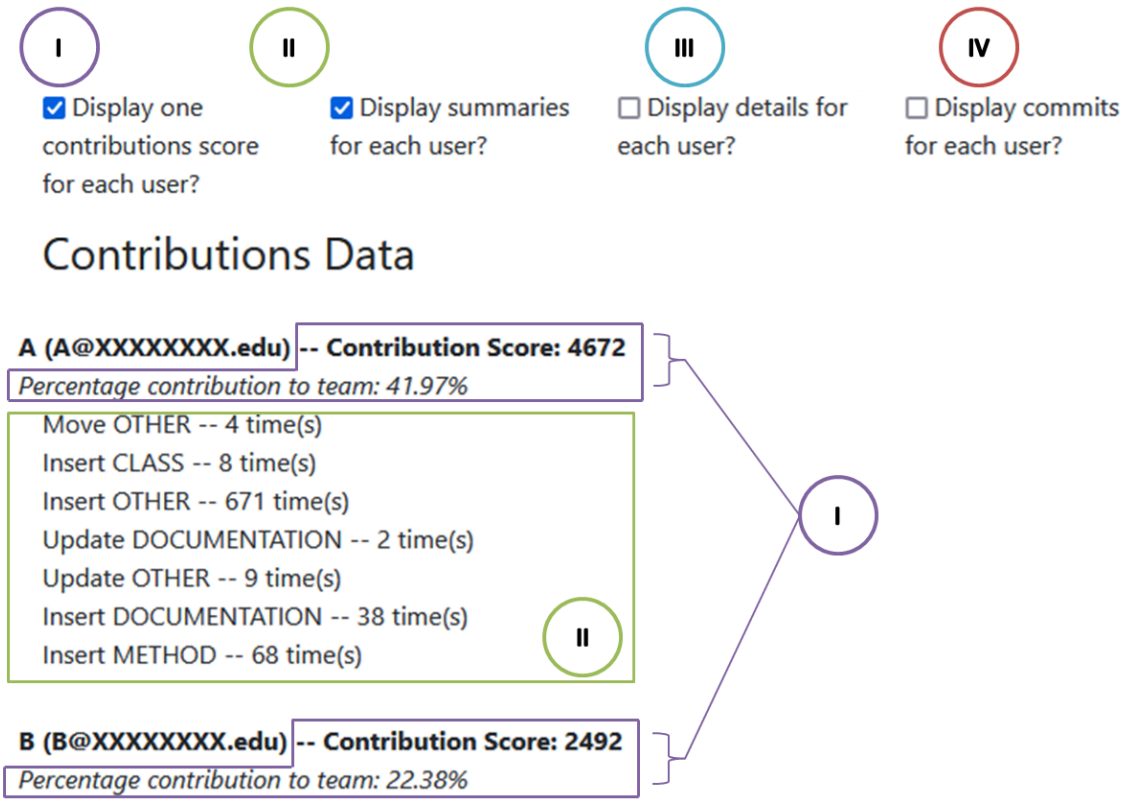


Figure 7.1 A trimmed contributions summary produced by AutoVCS. All four types of summaries can be toggled on and off independently; two are enabled. For brevity, details for student B and all contributions for student C are not shown.

Figure 7.2 Study Outline, showing the parts of the study, the approximate time spent on each part, and what RQs were answered by each.

graded lab assignments from a recent offering of our CS1.5 course. Some assignments were graded entirely by hand (the *control* group) and some with summaries from our algorithm (the *experimental* group). In Part 2 (Section 7.4.3), participants evaluated feedback from other participants in the study. Finally, participants completed a brief reflection (Section 7.4.4).

7.4.1 Terminology

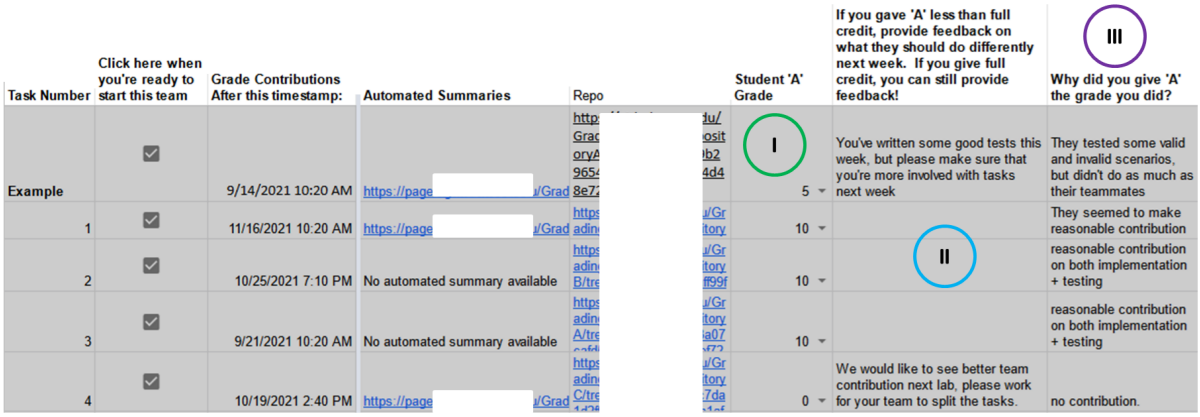
We refer to the individual students whose assignments were graded as *subjects*. We refer to the 13 TAs who participated in our study as *participants* or *raters*, depending on context. We refer to the grades assigned by *raters* to *subjects* as *ratings*. We refer to *rating subjects* when referring to individual students, or *grading assignments* when referring to the entire three-person team.

7.4.2 Part 1: Grading

In Part 1, participants were tasked with grading 17 lab assignments from a recent offering of our CS1.5 course. We provided each participant a Google Sheets spreadsheet, where each assignment to grade was on a separate row. A random subset of nine assignments had summaries from AutoVCS (the *experimental* group); the other eight had no summaries available (the *control* group). For each assignment, the spreadsheet contained a) a link to the Github repository with the code, b) a reminder of the time interval to grade, and c) where applicable, a link to the contributions summary from AutoVCS. The order of the 17 tasks was randomised for each participant. Figure 7.3 shows a example of the spreadsheet used, with grades and comments from one participant. For each subject on each team, raters provided a) a contribution score (0, 5, or 10, as discussed in Section 7.2), b) if the score was not a 10, a comment to the subject explaining what to do differently to receive more points, and c) a comment, not shared with the subject, giving the rationale for the score. We instrumented the spreadsheet to reveal tasks one at a time and capture start times and end times.

The assignments to grade were prepared by anonymising 18 weekly lab assignments from a recent offering of our CS1.5 course, replacing authors in git commits and code with pseudonyms.[1] These anonymised assignments were hosted on Github Enterprise to mirror normal grading practises. One assignment was used as an example to demonstrate the

[1] Rewriting files in git while keeping the history (commit author and timestamps) is not officially supported, and could not be performed on repositories where students introduced merge conflicts. From the most recent offering of our CS1.5 course, eighteen assignments could be completely anonymised.



Task Number	Click here when you're ready to start this team	Grade Contributions After this timestamp:	Automated Summaries	Repo	Student 'A' Grade	If you gave 'A' less than full credit, provide feedback on what they should do differently next week. If you give full credit, you can still provide feedback!	III Why did you give 'A' the grade you did?
Example	☑	9/14/2021 10:20 AM	https://page…/Grad	http…/Grac…ory…965…8e7…	I 5	You've written some good tests this week, but please make sure that you're more involved with tasks next week	They tested some valid and invalid scenarios, but didn't do as much as their teammates
1	☑	11/16/2021 10:20 AM	https://page…/Grad	https…/adin…B/tre…	10		They seemed to make reasonable contribution
2	☑	10/25/2021 7:10 PM	No automated summary available	https…/adin…B/tre…	10	II	reasonable contribution on both implementation + testing
3	☑	9/21/2021 10:20 AM	No automated summary available	https…/adin…A/tre…	10		reasonable contribution on both implementation + testing
4	☑	10/19/2021 2:40 PM	https://page…/Grad	https…/adin…C/tre…	0	We would like to see better team contribution next lab, please work for your team to split the tasks.	no contribution.

Figure 7.3 Excerpt from the spreadsheet used for Part 1, showing grades (Ⓘ), comments (ⒾⒾ), and rationale (ⅢⅢ). Feedback students B and C is not shown.

Grade 1	Comment 1		Grade 2	Comment 2		I would choose...
0	Try to contribute more by coordinating with your teammates and asking what help is needed.		5	Good job on fixing code and adding tests! Next time see if you can contribute more on the implementation side of things.		Either (no difference)
10	good job in FSM and code contribution.		10	Great work both on implementation and testing.		Either (no difference)
10	Good job in implementation and testing		10	There is some implementation and testing along with the javadocs. But it would be better if you did some more implementation.		Comment 2
0	fixing checkstyle is not enough contribution		5	you need to write more tests and implementations rather than fixing typos and generating javadocs		Comment 2

Figure 7.4 Excerpt from the spreadsheet used for Part 2, showing several pairs of comments alongside corresponding votes.

tasks and the contributions summaries.

7.4.3 Part 2: Evaluating Feedback

In Part 2, participants were asked to put themselves in the mindset of a student receiving feedback and evaluate its actionability. We provided participants with ten pairs of grades and comments from other participants in the study and asked them to choose which comment from each pair is "*more helpful in letting you know what to improve upon*", or *Either (no difference)* as appropriate. One grade and comment in each pair came from an assignment from the control group, and one came from the experimental group. This label was not shown, and the order of the two comments was randomised. An example of the spreadsheet used is shown in Figure 7.4.

7.4.4 Reflection Survey

Finally, we asked participants to complete a brief reflection on the grading experience. The reflection asked participants how they used the automated summaries; how helpful they found each of the main features (shown in Figure 7.1); how they would improve the summaries; and whether they would choose to use them again.

7.4.5 Data Description

7.4.5.1 Survey Responses

We converted the text Likert scale to numbers, where 1 maps to the lowest score, such as "Not at all helpful", and 5 maps to the highest score, such as "Extremely helpful". We treat them as interval-scaled data [Har15b]. All 13 participants completed the reflection survey.

7.4.5.2 Data Details

In Part 1, we tasked 13 participants with grading 17 assignments. Some participants graded more slowly than others, and consequently not all participants finished grading all assignments. The 13 participants in our study graded a total of 204 assignments, providing 610 ratings for individual students[2].

In Part 2, we provided each participant with ten pairs of comments from other participants, and asked them to choose which comment from each pair was more actionable. The 13 participants submitted 130 votes, choosing comments from the experimental group 60 times, comments from the control group 43 times, and expressing no preference 27 times.

7.4.5.3 Analysis

To answer RQ16 we compared grading times for assignments from the *control* and *experimental* groups. The distribution of the elapsed times is skewed right; consequently, we chose a Mann-Whitney U test, a non-parametric test.

To answer RQ17, we computed inter-rated reliability (IRR) for the *control* and *experimental* groups. Raters provided three grades for each assignment (for the three students on the team); thus each student is a unique subject rated by up to 13 raters (the participants in our study). We use Krippendorff's Alpha [Kri11] for computing IRR, as it handles a different

[2]Note this is not an exact multiple of 3, as two participants each missed rating one subject. If all participants had graded all assignments, there would have been 663 ratings (13 participants * 17 assignments * 3 students per assignment = 663 ratings).

Table 7.1 Grading time, in minutes, for assignments that were graded manually (*Control*) or with automated summaries (*Experimental*). Times are split into the first eight assignments (*first half*) and second nine assignments (*second half*) graded by each participant.

	First Half		Second Half		Overall	
	Mean	Median	Mean	Median	Mean	Median
Control	7.57	7.13	4.36	4.02	5.85	5.31
Experimental	7.88	6.79	4.66	4.10	6.35	5.44

number of ratings per subject. As suggested by Zapf et al. [Zap16] we calculated confidence intervals for both groups and identified the *q-value* where they are disjoint.

To answer RQ18, we read through responses to the reflection to understand how participants used the summaries.

We answered RQ19 with two different metrics. To identify if automated summaries help participants provide better feedback we performed Fisher's Exact Test on the preferences from Part 2. To understand if automated summaries help TAs see nuance, we performed a test of two proportions on the rate of partial credit for each group.

7.5 Results

In this section we present results for whether automated contributions summaries can enable faster (Section 7.5.1) or more consistent (Section 7.5.2) grading. We also consider impacts on the grading experience (Section 7.5.3) and feedback to students (Section 7.5.4).

7.5.1 RQ16: Grading Speed

We find no significant difference in grading speed from automated summaries. As shown in Table 7.1, participants graded assignments in the *control* group in an average of 5.85 minutes. Participants graded assignments in the *experimental* group in an average of 6.35 minutes. A Mann-Whitney U Test confirmed that the difference was not significant ($p = .677$).

We observe a learning curve as participants get more comfortable with, and consequently faster at, grading assignments. To evaluate a learning curve, we compared times taken to grade the *first half* and *second half* of the assignments within the *control* and *experimental* groups using Mann-Whitney U tests. We observed learning effects in both groups, showing that regardless of how participants graded assignments, they got faster over time ($p < .001$ for both groups).

We observe that grading may *feel* faster when using automated summaries. One participant reflected "*it definitely felt faster to grade*" with the automated summaries. While the numbers do not back this up, if the process feels faster, TAs may consider it less of a burden.

RQ16: *Automated summaries do not impact grading speed.*

7.5.2 RQ17: Grading Consistency

We find that automated summaries can help TAs grade more consistently. We calculated Krippendorff's Alpha (α) separately for the *control* and *experimental* groups to identify how consistently the raters of each subject agreed with each other. We find $\alpha = 0.286$ for the *control* group, and $\alpha = 0.609$ for the *experimental* group. At $q = .021$[3], the confidence intervals are disjoint, showing that automated summaries significantly improve grading consistency.

We note, however, that even though automated summaries help TAs grade more consistently, $\alpha = 0.609$ still indicates a relatively low level of agreement. Krippendorff argues that "*it is customary to require* $\alpha \geq .800$" [Kri04], which participants in our study did not meet. We discuss possible causes of this and implications in Section 7.6.1.

RQ17: *TAs grade assignments more consistently using automated summaries than without them.*

7.5.3 RQ18: Grading Preferences

We find that TAs have a strong preference for grading with automated summaries. All 13 participants said that they would prefer to use automated summaries for grading in the future, with 11 participants saying they would *strongly prefer* them. Participants gave the automated summaries an average rating of 4.85 out of 5. One participant said "*I think the tool was a huge help*" and rated it 5/5.

We find that participants find all of the features of the contributions summaries to be useful. As discussed in Section 7.4.5.1, we calculated the average score given to each feature. Participants found the *List of commits for each user* (Ⓘⓥ in Figure 7.1) to be the most useful feature, rating it 4.46/5. *Percentage Contribution to Team* (part of Ⓘ in Figure 7.1) was rated as the second most useful, with a score of 3.92/5. All features received a rating of 5 from at least one participant, and received an average rating of at least 3.4/5, approximately

[3]Krippendorff's α uses *q-values* as opposed to *p-values* as they provide improved resilience when performing multiple comparisons. *q-values* are interpreted the same way as *p-values* [Ben95; Sto03]

halfway between *Moderately Helpful* and *Very Helpful*. Our results show that both simple summaries of commit history and more advanced program analysis can assist with grading.

RQ18: *TAs strongly prefer grading with summaries from AutoVCS and find all of the features helpful.*

7.5.4 RQ19: Feedback Quality

TAs consider feedback from assignments graded with automated summaries to be more actionable than assignments graded manually. As described in Section 7.4.3, we asked participants to choose between pairs of comments to choose which one makes it clearer "*what to improve upon*". A Fisher Exact Test confirmed ($p = .0311$) a preference for comments from the *experimental* group, thus showing that TAs consider feedback from their peers more useful when it came from assignments graded with automated summaries.

We also observe that the quantity of feedback is impacted by automated summaries. We compared the rate at which partial credit was assigned in both groups, and find TAs award partial credit to 17.1% of subjects in the *control* group, and 24.9% of subjects in the *experimental* group. A test of two proportions shows that this difference is significant ($p = .018$). Consequently, the average grade given in the *control* group was higher than in the *experimental* group. TAs are expected to provide feedback on where to improve alongside partial credit, but are not required to do so for full credit. By giving more partial credit, this may help TAs provide students with more feedback and thus improve learning outcomes [Hat07; Wis20a].

One participant remarked in their reflection that the automated summaries encouraged them to be careful, saying that "*[I] was terrified at how much it made me reconsider some of my initial grading thoughts*". We thus see evidence that our contributions summary algorithm can help TAs grade more carefully.

RQ19: *TAs consider feedback from assignments graded with automated summaries to be more helpful than feedback from assignments graded without them, and automated summaries help TAs see nuance and provide partial credit more often.*

7.6 Discussion

Here, we probe grading inconsistencies (Section 7.6.1), discuss threats to validity (Section 7.6.2) and explore future work (Section 7.6.3).

7.6.1 Improving Grading Consistency

In Section 7.5.2, we found that automated summaries improve grading consistency, but that consistency remains an issue. To probe this further, we focused on the most extreme cases: subjects who were given full credit (10) and no credit (0) by different raters. We found seven of these subjects within each of the *control* and *experimental* groups. To understand these ratings, we read through the comments and rationale from each rater for these subjects.

We find that while the number of these disagreements does not differ across both groups, the causes do. In the *control* group, four of the seven disagreements came from issues *identifying* individual contributions. In two cases, a rater gave credit even when the student had no contributions. In an additional case, a rater gave credit for work done outside of the time window (work for a different lab, which used the same repository) and in a final case, a rater missed contributions that were made. By contrast, in the *experimental* group, we saw only two issues with *identifying* contributions. In one case, the rater gave credit for contributions outside the time window; in the second, the rater appeared to miss contributions within the time window. The remaining cases (3 from the *control* group, 5 from the *experimental* group) were caused by disagreements over what contributions were worthy of credit (such as system testing and documentation) and cases of pair programming. The sample size is small, but these results suggest that automated summaries may help TAs more accurately *identify* individual contributions; work remains to ensure that differences between types of contributions are considered when grading.

7.6.2 Threats to Validity

In this section, we discuss different types of threats to validity.

7.6.2.1 Conclusion

To combat any impacts of multiple study sessions, we used a script to introduce the study procedures.

Differences in elapsed times between groups were calculated with nonparametric tests to handle skewed data. IRR was calculated using Krippendorff's Alpha, which handles missing data [Kri11].

7.6.2.2 Internal

To counter learning effects, the order of tasks for each participant was randomised.

Participants knew that their behaviour was being studied, and thus may have graded more carefully. However, this applies to participants in both the *control* and *experimental* groups equally.

7.6.2.3 Construct

We measure consistency by calculating IRR, evaluating whether TAs' ratings agree with each other. We do not consider whether the ratings agree with an expert, such as a course instructor. Our evaluation matches typical grading practises.

7.6.2.4 External

We conducted Part 1 of study using Google Sheets, and participants graded student labs from a recent semester. Both the study tasks and format emulate the normal grading experience. However, all assignments came from a single semester of a single course. We suggest future work to consider other contexts.

All participants in the study were current or former TAs for team-based CS courses and have experience with evaluating individual contributions. As students, they are also familiar with interpreting feedback; however, they may do so differently from CS1.5 students.

7.6.3 Future Work

As discussed in Section 7.5.2, our results show that automated summaries can help TAs rate subjects significantly more consistently, but consistency is still relatively poor. Rubrics have been widely used to improve grading consistency and fairness [Fel18; Red10; Rag20]; however, to the best of our knowledge, no prior work has evaluated their impact on assessing individual contributions. We propose evaluating whether rubrics can be used in this context.

We find that participants particularly struggled with grading pair programming. While we instruct students to document pair programming via commit messages, contributions in Git appear only under the name of the student who committed the code. Work remains to be done to ensure that pair programming is graded fairly.

Much work remains to be done in account for non-code contributions. We found several participants who missed students' system testing contributions. Prior work suggests automatically crediting non-code contributions is an open problem, as evidenced by approaches such as All Contributors [You21], which sidesteps it by manually tracking them instead. We propose future work to support grading with automation for identifying non-code contributions.

In Section 7.5.4 we found that TAs consider feedback from assignments that were graded with automated summaries to be more actionable than feedback from manually-graded assignments. We suggest future work to evaluate learning gains by putting the feedback directly in front of students in the target course.

7.7 Conclusion

In this work, we developed an algorithm for summarising individual students' code contributions to team assignments. We built a tool, AutoVCS, that implements our algorithm, and evaluated it with 13 TAs, who graded some assignments with automated summaries and some assignments without it. We found that automated summaries help TAs grade assignments more consistently and provide students with more actionable feedback. Additionally, although automated summaries do not impact grading speed, TAs strongly prefer to grade assignments using them. We suggest future work to explore the use of rubrics for grading individual contributions and automated support for non-code contributions.

CHAPTER

8

CONCLUSION

8.1 Summary of Results

We began this dissertation with the following thesis statement:

> *Using software engineering automation and survey techniques in computer science education results in improved student learning outcomes, early prediction of struggling teams, and more effective instructional materials.*

In support of this thesis statement, we conducted five studies exploring how to use automation and surveys to further educational outcomes. In the remainder of this section, we briefly discuss the findings of this dissertation.

We began first in Chapters 3 and 4 exploring the use of software engineering automation in computer science education. In Chapter 3, we considered the issue of test flakiness, with the goal of identifying how to improve automated testing practises to provide students with more consistent feedback on their code. In Chapter 4, we considered the use of automated program repair in computer science education, seeking to identify whether automated repair can produce repairs of sufficient quality that they may have pedagogical benefits. In both studies, we demonstrated that software engineering automation can offer educational

benefits. In Chapter 3, we identified the impact of Selenium configuration and system configuration on test stability in the context of iTrust2 [Hec18b]. This provided a more stable application and more consistent feedback for teaching software engineering. In Chapter 4, we consider the types of mistakes that students make as they learn SQL, and develop a tool, SQLRepair, to perform automated repair on SQL queries. We find that students struggle with both the syntax and semantics of SQL, and that automated repair can fix mistakes from both categories. We also demonstrate that students find repairs sufficiently understandable they may be usable as an educational tool, in contexts such as intelligent tutoring systems. While the focus of these studies is different (automated testing in Chapter 3 and automated program repair in Chapter 4), both chapters demonstrate that software engineering automation can be beneficial in computer science education.

We next consider how to promote better teaming outcomes for undergraduate computer science students as they learn to collaborate on team projects. The first problem we considered is "How do we proactively identify teams that are struggling to collaborate effectively?" To this end, in the *collaboration reflection study* (Chapter 5) we developed a team collaboration reflection survey (TCRS), which asks students to briefly reflect on what they have accomplished over the past week and how effectively they think their team is working together. We matched teams flagged through the TCRS to teams that went on to perform poorly, and found a large majority of struggling teams (89%) could be proactively identified; additionally, a majority of students (64%) appreciated that the TCRS kept them on track or gave them a chance to reflect on the status of their project. However, an intervention we tried with student teams to promote better teaming outcomes was unsuccessful. To better understand the challenges that teams faced, and why the intervention may have failed, we conducted a followup *team challenges study* (Chapter 6). In this study, we interviewed students who had recently completed a team-based software engineering course. In this study, we sought to understand how students ran their teams, what if any challenges their teams faced, and if their team faced any challenges, how they tried to overcome them. This study demonstrated that major challenges included communicating effectively and setting and holding to deadlines, and that some, but not all, teams were able to overcome these challenges over the course of the project. Our results confirmed that most students found the TCRS helpful, and requested that TAs take a more active hand in navigating and resolving any challenges that the TCRS revealed. Finally, in our *contributions analysis study* (Chapter 7) we considered how to provide students with more effective feedback on their contributions to team-based projects. To do so, we developed an algorithm for summarising individual students' code contributions to team-based software projects, and implemented a tool based off of it. A user study demonstrated that our algorithm helps

teaching assistants provide students with more consistent and more actionable feedback, and that TAs strongly prefer grading with automated summaries than without them.

8.2 Implications

The clearest implication of this work is that much remains to be done in promoting positive teaming experiences for students in computer science courses. The oracle for *observed struggle* that we created in Chapter 5 demonstrated that approximately one in four teams experience substantial collaborative challenges that result in low peer evaluations or that cause the team to perform poorly on their project. When severe collaboration issues afflict one in four teams, computer science education researchers need to do more to understand the details of what is going on. Informed by this information, computer science educators can then help teams avoid or overcome these challenges. Furthermore, the TCRS we created suggests that up to twice as many teams face some sort of collaborative challenges than grades alone reveal. While this is in line with prior work [Tuc06; Mar16], it indicates that team challenges are widespread in software engineering courses. Some adversity can contribute to a positive learning experience, but too much adversity can have the opposite effect and instead demoralise students and impede learning outcomes [D'M14].

The TCRS provides instructors a way to *detect* team challenges, but the details of precisely what issues teams face in many ways still remain hidden. Teams do most of their work on their own, outside of class, and thus a majority of student interactions and their associated challenges are largely opaque to the teaching staff. Our followup work in Chapter 6 provides insights into these challenges, and shows that many of the issues teams face ultimately result from how they communicate and collaborate outside of lab. However, it is still unclear *why* some teams face these challenges in the first place. Why do some teams manage to communicate effectively, while others do not? Why do some teams successfully hold themselves to deadlines, while other teams continue to procrastinate? Our work reveals that these are common problems, but more remains to be done to understand them more fully. These collaboration experiences are ones that grades do not necessarily reveal, yet they can still cause substantial frustration and heartache for students. While the challenges we have observed do not differ drastically from those faced by students in other disciplines [Bur03; Pfa03; Owe15; ML17], their impact may be more severe. Negative experiences such as these may be demoralising for some students [Ban86], particularly in introductory classes. For female and minority students who are traditionally less assertive [Par15; Lea11], these negative experiences may poison their perspective of computer science programs. At a time when computer science educators are trying to make the field

more inclusive and welcoming towards a diverse body of students, it is essential that we identify how to support positive teaming experiences that will draw in students rather than pushing them away. There is much more work that computer science education researchers can and should do to gain insights into the types of challenges that student teams face so that we can support positive learning experiences.

Pursuing this same problem of team dysfunction from a different angle, in Chapter 7, we seek to improve how TAs provide students with feedback on their contributions to team assignments. In Chapter 6 we observe that most collaborative challenges were due to poor communication or procrastination, but some students reported being dissatisfied with the contributions of their teammates. It is possible that this is because TAs provide students with insufficient guidance on what was expected from them each week or failed to deduct sufficient points to motivate better contributions. Part of this may be due to how teams are evaluated by TAs. In Chapter 6, we observe that several students felt that the in-lab grading distracted from a back-and-forth dialogue about the team's progress. Our results in Chapter 7 support this, and suggest that TAs may simply not have enough time in lab to carefully review individual contributions and give on-the-spot grades and feedback to teams. We observe that TAs spent an average of six minutes grading individual contributions for three-person teams. While grading time may not be linear in the number of students on a team, this suggests that accurately grading contributions for five or six member teams may take upwards of ten minutes to do properly. When TAs spend approximately 15 minutes discussing with each team, it is plausible that they simply don't have enough time to evaluate individual contributions effectively [Vat21] and provide students with useful feedback on their contributions [Wis20b].

Overall, while the work in this dissertation enhances our understanding of how to help teams, it also shows that promoting positive and educational team-based learning is still an open problem for computer science educators.

8.3 Relation to Prior Work

In this section, we discuss how our results relate to prior work, considering prior work in teaming in higher education (Section 8.3.1) and teaming and educational theory (Section 8.3.2).

8.3.1 Teaming Education

The results that we have shown in Chapters 5 and 6 are broadly consistent with existing work in team-based learning. Tucker and Reynolds [Tuc06] report that approximately 40% of teams in a project-based studio design course are characterised by "conflict and selfish ambition". They report these teams fail to work together successfully over the course of the project, ultimately delivering "piecemeal design with little cohesion". We found a relatively consistent 25% of teams were flagged by the grades-based oracle established in Chapter 5.3, and Chapter 6.6.1 reveals that additional teams faced persistent issues working together over the course of the semester.

Our results are also broadly consistent with prior work studying teaming in software engineering education. Marques reports [Mar16] that software engineering students and recent graduates lack "teamwork skills and collaborative ability", which hamper their ability to work effectively in team projects. They report that among student software engineering teams that did not receive support from an external mentor, or "monitor", only 40% managed to produce a project that could be deployed successfully. While this is a higher rate of *observed struggle* than is revealed by our oracle (Chapters 5 and 6), it shows that challenges in software engineering teams are indeed widespread. Additionally, it matches our results in Chapter 6 that most teams face some sort of collaborative challenge that hinders their ability to work together successfully. Iacob and Faily [Iac20] similarly report that many undergraduate software engineering students struggle to collaborate effectively, and that mentorship improves students' perceptions of their teaming success (although not overall project grades). However, they caution that it is difficult to scale this approach to large classes. Our approach of using lightweight self-reflection surveys as a way to solicit feedback from students and better direct TA resources may offer improvements in helping teams, particularly at scale; our results in Chapter 6.6 echo this. Our results in Chapter 6.6 additionally suggest that self-reflection, as a key component of self-regulated learning, is sufficient for some teams to consider their challenges and overcome them. However, our results show that this is insufficient in the case of more severe team dysfunction and at motivating recalcitrant teammates. Thus, to help these teams overcome collaborative challenges, the course teaching staff may need to intervene and provide students with further guidance on how to work together effectively as a team.

We finally consider our results on team dysfunction in the context of broader engineering education, which has shown that collaborative difficulties are widespread [Bor13; Hal13]. Borrego et al. [Bor13] report that freeriding is the dominant team challenge in engineering education. They observe that freeriding may be particularly prevalent "when

individual contributions can[not] be identified" by instructors and when students perceive that projects have a low "inherent value". By contrast, we do not observe freeriding as a significant issue. In the upper-level software engineering course that we study, we observe that whole-team dysfunctions in communication, timeliness, and task planning are the predominant issues. Unfortunately, Borrego et al. do not break out observed challenges by class standing, so we are unable to compare our results specifically with upper-level courses. Compared to prior work [Hal13], we observe few cases of students who made negligible contributions to their team's overall effort. In all such cases, we observe that severe mental health challenges left students in a position where they were unable to engage effectively with their team.

Our work in Chapter 7.5.2 is consistent with work showing that TAs, and even instructors, may struggle to consistently evaluate students' contributions [Hay03; Gla15]. Hayes et al. [Hay03] report that TAs perform poorly at discerning what individual students have contributed to team-based projects. Even with Git and Github, which track exact contributions from each student on a team, we found that TAs struggle with this same problem. Our work in Chapter 7 provides TAs with summary information on each student's contributions. We demonstrate that this information helps them more consistently identify and grade the contributions from each student.

8.3.2 Teaming & Educational Theory

We consider our results on team challenges in the context of teaming theory and educational theory. Tuckman's theory of teaming [Tuc65] argues that teams progress through four (and later five [Tuc77]) stages, and that it takes time for members of a team to become acquainted, face and resolve challenges, and finally work together smoothly. We see evidence of this in Chapter 6.6.2. We observe that team challenges are widespread, and that it takes time for teams to "*norm*" and overcome their collaborative challenges. In the context of our six-week project, some teams are unable to ever overcome their collaborative challenges. This suggests that full-semester projects, such as described by Bates et al. [Bat22], may offer a better environment for students to both face and overcome collaborative challenges.

We consider our results on team formation in the context of teaming theory. In Chapter 6.7.2, we observe that the way a team is formed (whether around students who requested each other or not) has no consistent impact on its outcome, and the challenges, if any, that it faces. We observe several teams that were formed from students who mutually requested each other, and who ultimately collaborated effectively. However, we also observe a team that was formed from students who requested each other but ultimately struggled to work

effectively. Finally, we observe teams formed from members who did not request each other, yet still collaborated effectively. While our sample size is small, this suggests that teams can work well regardless of whether students know each other going into the project.

We briefly consider our results in the context of diversity in student identities. Prior work has shown that pairing together students from minority backgrounds can help form a sense of inclusion and consequently improve teaming outcomes [Tak14]. While the impact of team demographics on overall success was not our focus, our results support this. In Chapter 6.6.2.3, we observe a positive outcome for Team Bravo from pairing together students from a similar minority cultural background; this helped overcome a language barrier and helped one student work more effectively with the rest of their team. While we observed a benefit to paring students from a similar background together, Rienties et al. [Rie13] report that students form strong cross-cultural team relationships. By bringing together a diverse set of students, this may help improve learning outcomes by giving students a broader set of perspectives to learn from. Borrrego et al. [Bor13] report that groups that "typically value collective outcomes" may experience more positive teaming experiences, and point to female students and those of East Asian background as ones who may be particularly engaged team members. In Chapter 6.6, we observe that Team Hotel, which had many female students, acknowledged no collaborative challenges.

Finally, we consider how to effectively form groups from students of different skill levels, which may present a more difficult challenge. Prior work studying team-based learning has shown that having projects that are sufficiently complicated that each student has a unique task and role to play can make everyone feel involved with the team and reduce the risk of freeriding [Kar93; Kar95]. Conversely, teams that are formed from both high-performing and low-performing students can increase the risk of freeriding, as the lower-performing students are unable to meet the expectations of their higher-performing teammates, and consequently give up [Pie10]. At the same time, Bandura and Walters' Social Learning Theory [Ban63] argues that students learn by observing and modeling others, and that behaviour which is rewarded persists, and behaviour which is punished does not. From this angle, a team that features both higher performing students and lower performing students may improve learning for all, by giving each student examples of behaviour they may wish to emulate or avoid. Prior work by Hong and Page has shown that this intra-team diversity can improve not just learning over time, but also performance within a project, as students are able to draw upon a wider range of approaches and potential solutions [Hon04]. Dzvonyar et al. [Dzv18] apply this principle within software engineering, creating "balanced teams with regard to technical skills" so that more experienced students can serve as mentors to their less experienced teammmates. Our work has not considered

the optimal middle ground to maximise both participation and learning outcomes, but identifying it can support the best team learning environment.

8.4 Future Work

In this dissertation we have drawn upon software engineering automation and surveys to offer educational benefits for individual students and teams of students. There are several promising areas for future work based on the work in this dissertation. We consider future work including further use of automated program repair in computer science education (Section 8.4.1), using automation to gain further insights into teams and how they work together (Section 8.4.2) and helping teams overcome collaborative challenges (Section 8.4.3).

8.4.1 Automated Program Repair in CS Education

We have identified several promising advances of this research involving further use of automated program repair within the field of computer science education.

- In our *automated program repair study* (Chapter 4) we demonstrated that automated repair can be applied to special-purpose languages such as SQL, and that students find these repairs understandable. This suggests that automated program repair may be useful in Intelligent Tutoring Systems, offering students hints on what to try next based on the current status of their work [And85; Cro18]. So far, however, we have only given students a *complete* repair to consider. Better approaches may involve giving them *a part* of a repair, which could be sufficient to get them unstuck. We propose future work to evaluate this approach, comparing it to traditional ITS approaches that offer hints based off of other students' work [Bar10; Eag12; Pri18] to evaluate impacts on learning outcomes.

- In our *contributions analysis study* (Chapter 7) we proposed an algorithm for summarising individual students' code contributions to team-based projects. Our algorithm builds and differences abstract syntax trees (ASTs) to offer more information about the code contributed than just considering lines of code. However, compilers are unable to build ASTs from code that does not compile, which may result in contributions being missed if students regularly commit code with syntax errors. Prior attempts to automatically test code that students write has recognised that introductory students particularly struggle with getting syntax correct [Par17]. To

address this, prior work has considered using automated program repair on student submissions, to repair syntax errors students make so that automated tests can be run on their code [Par17; Bha16; Ahm21]. We propose future work using the BlueJ project's Blackbox dataset [Bro14b] to identify the proportion of compilation errors that are due to syntax mistakes. If students regularly write code that does not compile, and commit it to Git, we propose future work to identify whether automated repair can fix these syntax mistakes in a way that meaningful automated summaries can be created.

8.4.2 Automation and Teaming

In Chapter 5, we introduced lightweight automation for flagging struggling teams based on collaboration reflection survey responses, and made further improvements to it in Chapter 6. In Chapter 7, we used program analysis techniques to automatically summarise code contributions from individual students. Based on this work, we have identified several promising areas for further use of automation to assist with teaming in computer science education.

- In our *contributions analysis study* (Chapter 7), we demonstrated that automated contributions summaries can help TAs grade individual student contributions substantially more consistently than they do otherwise. We have not, however, considered whether they can grade more *accurately*, that is, whether they agree with expert raters such as instructors. We propose future work to identify how accurately TAs grade assignments when they have automated summaries to aide them, and if necessary consider future steps to help them more closely match against instructor grades. As part of this, we propose replicating the work from Chapter 7 with instructors as participants, to understand how consistently instructors grade assignments, and, as necessary, how to offer them support so that they can grade assignments more consistently.
- We additionally propose future work to evaluate the use of automated contributions summaries as a way to predict struggling teams. We suggest work to identify to what extent team challenges are a result of nonparticipation or last-minute contributions, versus broader interpersonal conflicts. From this, we propose evaluating whether (improved) contributions summaries can serve as a predictor of or proxy for other challenges teams face [Abb17; Dzv18; Mar16].

- Our work in the *contributions analysis study* (Chapter 7) summarises code contributions that students make, but is unable to identify other types of contributions, including documentation, design, and system testing. We propose future work to move beyond the All Contributors [You21] model and use natural language processing of semi-structured data such as Github Issues and Pull Requests to identify other types of contributions.

- The algorithm that we developed in the *contributions analysis study* (Chapter 7) can summarise individual code contributions in any language; however, the tool we implemented, AutoVCS, only works on Java projects hosted on Github. We propose future work to evaluate how to extend these techniques to other languages, including different paradigms such as functional programming. We propose future work to consider whether language-agnostic AST analysis can be used to efficiently scale this approach to arbitrary languages.

8.4.3 Helping Teams Overcome Challenges

Our work in Chapters 5 and 6 provided insights into the challenges that teams face in undergraduate software engineering courses. We propose future work to identify challenges students face in different contexts and helping them overcome these challenges.

- In our *team challenges study* (Chapter 6), we observed several students who expressed a preference for trying to solve team challenges within the team first, and then getting help from the course teaching staff if efforts were unsuccessful. To facilitate this, we propose a big-picture view of team challenges, allowing instructors to track issues that are observed through the TCRS, issues observed by TAs in lab, insufficient contributions, and more. We propose a user study to evaluate whether tracking these disparate types of issues can help instructors help teams function more effectively than is otherwise possible. As a part of this, we propose unified tooling to help instructors both track and visualise team challenges in their courses.

- In our *collaboration reflection study* (Chapter 5) we created a collaboration reflection survey and demonstrated that it can flag student software engineering teams that struggled to work together effectively. An intervention we tried did not work, and followup work in the Team Challenges Study (Chapter 6) revealed that teams need more of a guiding hand from the teaching staff. We propose future work based on these findings to conduct more hands-on interventions with teams. In particular, we propose evaluating how much intervention is necessary to help teams overcome

these challenges, while still giving them a chance to make mistakes and learn from their experiences.

- Our work in the *team challenges study* (Chapter 6) provided novel insights into the challenges that teams in undergraduate software engineering courses face. We propose future work to study team challenges in different contexts. In particular, we propose work to identify whether students on longer-running projects (for instance, an entire semester) manage to resolve the issues they are facing more effectively, or if they continue to struggle all semester. Additionally, we propose work to identify whether the challenges that teams face vary based on the level of the course, and whether more notice students face different challenges than more advanced students.

8.5 Final thoughts

In Summer 2020, I set out to try and use data from version control systems to predict teams that were failing to collaborate effectively. Nearly two years later, we're still not there, but we've learned much more about how teams function and how to support their learning experiences. I look forward to realising my original goal in the not too distant future.

BIBLIOGRAPHY

[Abb17] Abbasi, N. et al. “Conflict Resolution in Student Teams: An Exploration in the Context of Design Education”. 2017.

[Abe09] Abernethy, K. & Treu, K. “Teaching Computing Soft Skills: An Experiential Approach”. *J. Comput. Sci. Coll.* **25**.2 (2009), 178–186.

[Ada18] Adachi, C. et al. “Academics’ perceptions of the benefits and challenges of self and peer assessment in higher education”. *Assessment & Evaluation in Higher Education* **43**.2 (2018), pp. 294–306. eprint: `https://doi.org/10.1080/02602938.2017.1339775`.

[Aha16] Ahadi, A. et al. “Students’ Syntactic Mistakes in Writing Seven Different Types of SQL Queries and Its Application to Predicting Students’ Success”. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE ’16. Memphis, Tennessee, USA: ACM, 2016, pp. 401–406.

[Ahm21] Ahmed, T. et al. *SYNFIX: Automatically Fixing Syntax Errors using Compiler Diagnostics*. 2021.

[Ald19] Aldrich, J. & Le Goues, C. “Program Analysis” (2019).

[Alt15] Altadmri, A. & Brown, N. C. “37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data”. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE ’15. Kansas City, Missouri, USA: ACM, 2015, pp. 522–527.

[And85] Anderson, J. R. et al. “Intelligent tutoring systems”. *Science* **228**.4698 (1985), pp. 456–462.

[And10] Andrews, J. H. et al. *Comparing automated unit testing strategies*. Department of Computer Science, University of Western Ontario, 2010.

[Ani19] Aniche, M. et al. “Pragmatic Software Testing Education”. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. Minneapolis, MN, USA: ACM, 2019, pp. 414–420.

[Aut] *AutoVCS*.

[Aya15] Ayaz, M. & şekerci, H. “The Effects of the Constructivist Learning Approach on Student’s Academic Achievement: A Meta-Analysis Study”. *Turkish Online Journal of Educational Technology* **14** (2015).

[Aye08] Ayewah, N. et al. “Using static analysis to find bugs”. *IEEE software* **25**.5 (2008), pp. 22–29.

[Bal13] Balachandran, V. “Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation”. *2013*

35th International Conference on Software Engineering (ICSE). IEEE. 2013, pp. 931–940.

[Ban86] Bandura, A. *Social foundations of thought and action: A social cognitive theory*. Prentice-Hall, Inc, 1986.

[Ban63] Bandura, A. & Walters, R. H. "Social learning and personality development." (1963).

[Bao20] Bao, L. et al. "How does Working from Home Affect Developer Productivity? - A Case Study of Baidu During COVID-19 Pandemic". *ArXiv* **abs/2005.13167** (2020).

[Bar10] Barnes, T. & Stamper, J. C. "Automatic Hint Generation for Logic Proof Tutoring Using Historical Data". *J. Educ. Technol. Soc.* **13** (2010), pp. 3–12.

[Bat22] Bates, R. et al. "A Project-Based Curriculum for Computer Science Situated to Serve Underrepresented Populations". *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2022. Providence, RI, USA: Association for Computing Machinery, 2022, 585–591.

[Bau15] Baumer, B. "A Data Science Course for Undergraduates: Thinking With Data". *The American Statistician* **69**.4 (2015), pp. 334–342.

[Bel18] Bell, J. et al. "DeFlaker: Automatically detecting flaky tests". *International Conference on Software Engineering*. 2018.

[Ben95] Benjamini, Y. & Hochberg, Y. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing". *Journal of the Royal Statistical Society: Series B (Methodological)* **57**.1 (1995), pp. 289–300.

[Ber05a] Berglund, A. "Learning computer systems in a distributed project course : The what, why, how and where". PhD thesis. 2005.

[Ber05b] Berner, S. et al. "Observations and Lessons Learned from Automated Testing". *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: Association for Computing Machinery, 2005, 571–579.

[Bha16] Bhatia, S. & Singh, R. *Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks*. 2016.

[Bla10] Black, P. & Wiliam, D. "Inside the Black Box Raising Standards Through Classroom Assessment". **80** (2010).

[Bod04] Bodenreider, O. *Unified Medical Language System (UMLS)*. 2004.

[Bon10] Bonebright, D. A. “40 years of storming: a historical review of Tuckman’s model of small group development”. *Human Resource Development International* **13**.1 (2010), pp. 111–120. eprint: https://doi.org/10.1080/13678861003589099.

[Bor13] Borrego, M. et al. “Team Effectiveness Theory from Industrial and Organizational Psychology Applied to Engineering Student Project Teams: A Research Review”. *Journal of Engineering Education* **102**.4 (2013), pp. 472–512. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/jee.20023.

[Bra06] Brass, S. & Goldberg, C. “Semantic Errors in SQL Queries: A Quite Complete List”. *J. Syst. Softw.* **79**.5 (2006), pp. 630–644.

[Bro17] Brown, N. C. C. & Altadmri, A. “Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs”. *ACM Trans. Comput. Educ.* **17**.2 (2017), 7:1–7:21.

[Bro14a] Brown, N. C. & Altadmri, A. “Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data”. *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER ’14. Glasgow, Scotland, United Kingdom: ACM, 2014, pp. 43–50.

[Bro14b] Brown, N. C. C. et al. “Blackbox: A Large Scale Repository of Novice Programmers’ Activity”. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE ’14. Atlanta, Georgia, USA: Association for Computing Machinery, 2014, 223–228.

[Bru10] Brutus, S. & Donia, M. B. L. “Improving the Effectiveness of Students in Groups With a Centralized Peer Evaluation System”. *Academy of Management Learning & Education* **9**.4 (2010), pp. 652–662.

[Bur03] Burdett, J. “Making Groups Work: University Students’ Perceptions”. 2003.

[Seja] *Bureau of Labor Statistics 2000-2010 Employment Projections*. 2001.

[But95] Butler, D. L. & Winne, P. H. “Feedback and Self-Regulated Learning: A Theoretical Synthesis”. *Review of Edu. Rsch.* **65**.3 (1995), pp. 245–281.

[Buy00] Buy, U. et al. “Automated Testing of Classes”. *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’00. Portland, Oregon, USA: Association for Computing Machinery, 2000, 39–48.

[Cha06] Chakrabarti, A. & Godefroid, P. “Software Partitioning for Effective Automated Unit Testing”. *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. EMSOFT ’06. Seoul, Korea: Association for Computing Machinery, 2006, 262–271.

[Cha] Changyi. *Java Programming Skills – Boilerplate Code*.

[Cha14] Charmaz, K. *Constructing grounded theory*. sage, 2014.

[Che15] Chen, P. et al. “Impact of Collaborative Project-Based Learning on Self-Efficacy of Urban Minority Students in Engineering.” *Journal of Urban Learning, Teaching, and Research* **11** (2015), pp. 26–39.

[Che04] Chess, B. & McGraw, G. “Static analysis for security”. *IEEE Security Privacy* **2**.6 (2004), pp. 76–79.

[Chr19] Chren, S. et al. “Mistakes in UML Diagrams: Analysis of Student Projects in a Software Engineering Course”. *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET ’19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 100–109.

[Ijc] *Code completion: Intellij idea*. 2021.

[Cor19] Cordy, M. et al. “FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment. A Case Study on Mutation Testing and Program Repair”. *ArXiv* **abs/1912.03197** (2019).

[Abe] *Criteria for Accrediting Engineering Programs, 2021 – 2022*. 2020.

[Cro18] Crow, T. et al. “Intelligent tutoring systems for programming education: a systematic review”. *Proceedings of the 20th Australasian Computing Education Conference*. 2018, pp. 53–62.

[DD03] De Dreu, C. K. & Weingart, L. R. “Task versus relationship conflict, team performance, and team member satisfaction: a meta-analysis.” *Journal of applied Psychology* **88**.4 (2003), p. 741.

[DM08] De Moura, L. & Bjørner, N. “Z3: An Efficient SMT Solver”. *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’08/ETAPS’08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

[DW12] De Wit, F. R. et al. “The paradox of intragroup conflict: a meta-analysis.” *Journal of applied psychology* **97**.2 (2012), p. 360.

[Del16] Delić, H. & Bećirović, S. “Socratic method as an approach to teaching”. *European Researcher. Series A* 10 (2016), pp. 511–517.

[des04] desRivieres, J. & Wiegand, J. “Eclipse: A platform for integrating development tools”. *IBM Systems Journal* **43**.2 (2004), pp. 371–383.

[Din14] Dingel, M. & Wei, W. “Influences on peer evaluation in a group project: an exploration of leadership, demographics and course performance”. *Assessment & Evaluation in Higher Education* **39**.6 (2014), pp. 729–742. eprint: `https://doi.org/10.1080/02602938.2013.867477`.

[Dis] *Discord | Your Place to Talk and Hang Out*.

[D'M14] D'Mello, S. K. & Graesser, A. C. "Confusion" (2014).

[Dor12] Dorairaj, S. et al. "Understanding Team Dynamics in Distributed Agile Software Development". *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Wohlin, C. Springer Berlin Heidelberg, 2012, pp. 47–61.

[Dro20] Drosos, I. et al. "Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists". *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, 1–12.

[Duv07] Duvall, P. et al. *Continuous Integration: Improving Software Quality and Reducing Risk*. First. Addison-Wesley Professional, 2007.

[Dzv18] Dzvonyar, D. et al. "Team Composition in Software Engineering Project Courses". *2018 IEEE/ACM International Workshop on Software Engineering Education for Millennials (SEEM)*. 2018, pp. 16–23.

[Eag12] Eagle, M. et al. "Interaction Networks: Generating High Level Hints Based on Network Community Clustering." *International Educational Data Mining Society* (2012).

[Edw08] Edwards, S. H. & Perez-Quinones, M. A. "Web-CAT: Automatically Grading Programming Assignments". *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '08. Madrid, Spain: Association for Computing Machinery, 2008, p. 328.

[Fal14] Falleri, J.-R. et al. "Fine-grained and Accurate Source Code Differencing". *Proceedings of the International Conference on Automated Software Engineering*. 2014, pp. 313–324.

[Fei16] Feist, M. D. et al. "Visualizing Project Evolution through Abstract Syntax Tree Analysis". *2016 IEEE Working Conf. on SW. Visualization (VISSOFT)*. 2016, pp. 11–20.

[Fel18] Feldman, J. *Grading for Equity: What It Is, Why It Matters, and How It Can Transform Schools and Classrooms*. SAGE Publications, 2018.

[Flu07] Fluri, B. et al. "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction". *IEEE Transactions on Software Engineering* **33**.11 (2007), pp. 725–743.

[Fow11] Fowler, M. *Eradicating Non-Determinism in Tests*. 2011. URL: `https://martinfowler.com/articles/nonDeterminism.html`.

[Fra13] Fransen, J. et al. “Team Effectiveness and Team Development in CSCL”. *Educational Psychologist* **48**.1 (2013), pp. 9–24. eprint: https://doi.org/10.1080/00461520.2012.747947.

[Fry12] Fry, Z. P. et al. “A Human Study of Patch Maintainability”. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery, 2012, 177–187.

[Gal09] Gall, H. C. et al. “Change analysis with evolizer and ChangeDistiller”. *IEEE Software* **26**.1 (2009), pp. 26–33.

[Gaz17] Gazzola, L. et al. “Automatic Software Repair: A Survey”. *IEEE Transactions on Software Engineering* **PP** (2017), pp. 1–1.

[Gil13] Gilson, L. L. et al. “Virtual Team Effectiveness: An Experiential Activity”. *Small Group Research* **44**.4 (2013), pp. 412–427. eprint: https://doi.org/10.1177/1046496413488216.

[Git20] Gitinabard, N. et al. “Student Teamwork on Programming Projects. What can GitHub logs show us?” *Proceedings of the 13th International Conference on Educational Data Mining, EDM 2020, Fully virtual conference, July 10-13, 2020*. Ed. by Rafferty, A. N. et al. International Educational Data Mining Society, 2020.

[Gla02] Glass, R. L. *Software Engineering: Facts and Fallacies*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[Gla15] Glazer, N. “Formative Plus Summative Assessment in Large Undergraduate Courses: Why Both?” *The International Journal of Teaching and Learning in Higher Education* **2014** (2015), pp. 276–286.

[Gre18] Grech, N. et al. “Shooting from the Heap: Ultra-Scalable Static Analysis with Heap Snapshots”. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, 198–208.

[Gro19] Groeneveld, W. et al. “Software Engineering Education Beyond the Technical: A Systematic Literature Review”. *Proceedings of the 47th SEFI Conference 2019; 2019. (SEFI - European Society for Engineering Education)* **abs/1910.09865** (2019).

[Gul16] Gulwani, S. “Programming by Examples (and its Applications in Data Wrangling)”. *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016.

[Gul18] Gulwani, S. et al. “Automated Clustering and Program Repair for Introductory Programming Assignments”. *Proceedings of the 39th ACM SIGPLAN Conference*

on Programming Language Design and Implementation. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 465–480.

[Hal13] Hall, D. & Buzwell, S. "The problem of free-riding in group projects: Looking beyond social loafing as reason for non-contribution". *Active Learning in Higher Education* **14**.1 (2013), pp. 37–49.

[Han06] Hansen, R. S. "Benefits and Problems With Student Teams: Suggestions for Improving Team Projects". *Journal of Education for Business* **82**.1 (2006), pp. 11–19.

[Har15a] Hardin, J. et al. "Data Science in Statistics Curricula: Preparing Students to "Think with Data"". *The American Statistician* **69**.4 (2015), pp. 343–353.

[Har15b] Harpe, S. E. "How to analyze Likert and other rating scale data". *Currents in Pharmacy Teaching and Learning* **7**.6 (2015), pp. 836–850.

[Har] Harris, W. C. "Teaming Across the Computer Science Curriculum: Assembly Language, and Operating Systems" ().

[Hat07] Hattie, J. & Timperley, H. "The power of feedback". *Review of educational research* **77**.1 (2007), pp. 81–112.

[Hay03] Hayes, J. et al. "Evaluating individual contribution toward group software engineering projects". *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, pp. 622–627.

[Hec18a] Heckman, S. & King, J. "Developing Software Engineering Skills Using Real Tools for Automated Grading". *Technical Symposium on Computer Science Education*. SIGCSE '18. Baltimore, Maryland, USA: ACM, 2018, pp. 794–799.

[Hec18b] Heckman, S. et al. "10+ Years of Teaching Software Engineering with Itrust: The Good, the Bad, and the Ugly". *International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET '18. Gothenburg, Sweden: ACM, 2018, pp. 1–4.

[Her15] Herzig, K. & Nagappan, N. "Empirically Detecting False Test Alarms Using Association Rules". *International Conference on Software Engineering*. Vol. 2. 2015, pp. 39–48.

[Hog08] Hogarth, A. "Introducing a collaborative technology strategy for higher education students: Recommendations and the way forward". *Education and Information Technologies* **13** (2008), pp. 259–273.

[Hon04] Hong, L. & Page, S. E. "Groups of diverse problem solvers can outperform groups of high-ability problem solvers". *Proceedings of the National Academy*

of Sciences **101**.46 (2004), pp. 16385–16389. eprint: https://www.pnas.org/content/101/46/16385.full.pdf.

[Hry12] Hrynchak, P. & Batty, H. "The educational theory basis of team-based learning". *Medical Teacher* **34**.10 (2012). PMID: 22646301, pp. 796–801. eprint: https://doi.org/10.3109/0142159X.2012.687120.

[Hun21] Hundhausen, C. et al. "Evaluating Commit, Issue and Product Quality in Team Software Development Projects". *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2021, 108–114.

[Hun18] Hunt, D. *Introduction to Selenium*. 2018. URL: https://www.seleniumhq.org/docs/03_webdriver.jsp.

[Iac20] Iacob, C. & Faily, S. "The Impact of Undergraduate Mentorship on Student Satisfaction and Engagement, Teamwork Performance, and Team Dysfunction in a Software Engineering Group Project". *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Portland, OR, USA: Association for Computing Machinery, 2020, 128–134.

[JAC88] JACKSON, E. A. "Marking Reliability in B.Sc. Engineering Examinations". *European Journal of Engineering Education* **13**.4 (1988), pp. 487–494. eprint: https://doi.org/10.1080/03043798808939448.

[Jia18] Jiang, J. et al. "Shaping Program Repair Space with Existing Patches and Similar Code". *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, pp. 298–309.

[Joh13] Johnson, B. et al. "Why don't software developers use static analysis tools to find bugs?" *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681.

[Jon07] Jonsson, A. & Svingby, G. "The use of scoring rubrics: Reliability, validity and educational consequences". *Educational Research Review* **2**.2 (2007), pp. 130–144.

[Kam] Kamalizade, A. *How to Reduce Java Boilerplate Code With Lombok*.

[Kar93] Karau, S. J. & Williams, K. D. "Social loafing: A meta-analytic review and theoretical integration." *Journal of personality and social psychology* **65**.4 (1993), p. 681.

[Kar95] Karau, S. J. & Williams, K. D. "Social loafing: Research findings, implications, and future directions". *Current Directions in Psychological Science* **4**.5 (1995), pp. 134–140.

[Ke15] Ke, Y. et al. “Repairing Programs with Semantic Code Search (T)”. *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 295–306.

[Ker18] Kery, M. B. et al. “The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool”. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI ’18. Montreal QC, Canada: Association for Computing Machinery, 2018, 1–11.

[Kha20] Khakurel, J. & Porras, J. “The Effect of Real-World Capstone Project in an Acquisition of Soft Skills among Software Engineering Students”. *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE T)*. 2020, pp. 1–9.

[Kha09] Khandkar, S. H. “Open coding”. *University of Calgary* **23** (2009), p. 2009.

[Kim13a] Kim, D. et al. “Automatic patch generation learned from human-written patches”. *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 802–811.

[Kim13b] Kim, Y. et al. “Automated unit testing of large industrial embedded software using concolic testing”. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 519–528.

[Kin12] Kinder, J. “Towards static analysis of virtualization-obfuscated binaries”. *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 61–70.

[Kin08] Kinder, J. & Veith, H. “Jakstab: A static analysis platform for binaries”. *International Conference on Computer Aided Verification*. Springer. 2008, pp. 423–427.

[Kri04] Krippendorff, K. *Content analysis: An introduction to its methodology*. SAGE, 2004.

[Kri11] Krippendorff, K. “Computing Krippendorff’s alpha-reliability” (2011).

[Kuu16] Kuutila, M. et al. “Benchmarking Web-testing-Selenium versus Watir and the Choice of Programming Language and Browser”. *arXiv preprint arXiv:1611.00578* (2016).

[LÏ1] Lämmel, R. et al. “Large-Scale, AST-Based API-Usage Analysis of Open-Source Java Projects”. *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC ’11. TaiChung, Taiwan: Association for Computing Machinery, 2011, 1317–1324.

[Lay00] Layzell, P. et al. “Supporting collaboration in distributed software engineering teams”. *Proceedings Seventh Asia-Pacific Software Engineering Conference. APSEC 2000*. 2000, pp. 38–45.

[LG12] Le Goues, C. et al. “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each”. *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 3–13.

[Le 15] Le Goues, C. et al. “The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs”. *IEEE Transactions on Software Engineering (TSE)* **41**.12 (2015). DOI: 10.1109/TSE.2015.2454513, pp. 1236–1256.

[Lea11] Leaper, C. & Robnett, R. D. “Women Are More Likely Than Men to Use Tentative Language, Aren’t They? A Meta-Analysis Testing for Gender Differences and Moderators”. *Psychology of Women Quarterly* **35**.1 (2011), pp. 129–142. eprint: `https://doi.org/10.1177/0361684310392728`.

[Lea] Learning, L. *Principles of Management*.

[Leo13] Leotta, M. et al. “Repairing Selenium Test Cases: An Industrial Case Study about Web Page Element Localization”. *Int’l Conf. on Software Testing, Verification and Validation*. 2013, pp. 487–488.

[Leo14] Leotta, M. et al. “Reducing Web Test Cases Aging by Means of Robust XPath Locators”. *Software Reliability Engineering Workshops*. 2014, pp. 449–454.

[Li20] Li, J. “Vulnerabilities mapping based on OWASP-SANS: a survey for static application security testing (SAST)”. *Annals of Emerging Technologies in Computing (AETiC), Print ISSN* (2020), pp. 2516–0281.

[Li14] Li, J. & Luca, R. D. “Review of assessment feedback”. *Studies in Higher Education* **39**.2 (2014), pp. 378–393.

[Lin21] Lin, X. et al. “How Do Students Collaborate? Analyzing Group Choice in a Collaborative Learning Environment”. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2021, 212–218.

[Liv05] Livshits, V. B. & Lam, M. S. “Finding Security Vulnerabilities in Java Applications with Static Analysis.” *USENIX security symposium*. Vol. 14. 2005, pp. 18–18.

[Lon16] Long, F. & Rinard, M. “Automatic Patch Generation by Learning Correct Code”. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’16. St. Petersburg, FL, USA: ACM, 2016, pp. 298–312.

[Lou20] Lou, Y. et al. “Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach”. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, 75–87.

[Lou14] Loughry, M. L. et al. “Assessing Teamwork Skills for Assurance of Learning Using CATME Team Tools”. *Journal of Marketing Education* **36**.1 (2014), pp. 5–19. eprint: `https://doi.org/10.1177/0273475313499023`.

[Lu93] Lu, H. et al. “A Survey on Usage of SQL”. *SIGMOD Rec.* **22**.4 (1993), pp. 60–65.

[Luk05] Luk, C.-K. et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. *SIGPLAN Not.* **40**.6 (2005), 190–200.

[Luo14] Luo, Q. et al. “An Empirical Analysis of Flaky Tests”. *International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 643–653.

[Mag19] Maguire, J. et al. “Mentoring Mentors in Cooperative Software Engineering Education Programmes”. *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ICER ’19. Toronto ON, Canada: Association for Computing Machinery, 2019, p. 307.

[Mao16] Mao, K. et al. “Sapienz: Multi-Objective Automated Testing for Android Applications”. *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, 94–105.

[Mar19] Marginean, A. et al. “SapFix: Automated End-to-End Repair at Scale”. *Proceedings of the 41st International Conference on Software Engineering*. ICSE ’19. IEEE Computer Society, 2019.

[Mar01] Marinov, D. & Khurshid, S. “TestEra: a novel framework for automated testing of Java programs”. *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. 2001, pp. 22–31.

[Mar16] Marques, M. R. “Monitoring: An Intervention to Improve Team Results in Software Engineering Education”. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE ’16. Memphis, Tennessee, USA: Association for Computing Machinery, 2016, p. 724.

[Mar13] Martinez, M. et al. “Automatically Extracting Instances of Code Change Patterns with AST Analysis”. *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 388–391.

[Mec16] Mechtaev, S. et al. “Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis”. *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 691–701.

[ML17] Mendo-Lázaro, S. et al. “Construction and Validation of a Measurement Instrument for Attitudes towards Teamwork”. *Frontiers in Psychology* **8** (2017), p. 1009.

[Meq20] Meqdadi, O. & Aljawarneh, S. “A study of code change patterns for adaptive maintenance with AST analysis”. *International Journal of Electrical and Computer Engineering* **10**.3 (2020), p. 2719.

[Mic21] Microsoft. *Intellisense in visual studio code*. 2021.

[Mie05] Mierle, K. et al. “Mining Student CVS Repositories for Performance Indicators”. *Proceedings of the 2005 Intl. Workshop on MSR*. MSR ’05. St. Louis, Missouri: ACM, 2005, 1–5.

[Mig20] Migler, A. & Dekhtyar, A. “Mapping the SQL Learning Process in Introductory Database Courses”. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE ’20. Portland, OR, USA: Association for Computing Machinery, 2020, 619–625.

[Mil21] Miller, C. et al. “How Was Your Weekend? Software Development Teams Working From Home During COVID-19”. *Proceedings of the 43rd International Conference on Software Engineering*. IEEE Press, 2021, 624–636.

[Nag05] Nagappan, N. & Ball, T. “Static analysis tools as early indicators of pre-release defect density”. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 2005, pp. 580–586.

[Nam08] Namey, E. et al. “Data reduction techniques for large qualitative data sets”. 2008, pp. 137–162.

[Nel80] Nelson, L. “The socratic method”. *Thinking: The Journal of Philosophy for Children* **2**.2 (1980), pp. 34–38.

[Net07] Nethercote, N. & Seward, J. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: Association for Computing Machinery, 2007, 89–100.

[Neu05] Neufeld, D. J. & Fang, Y. “Individual, social and situational determinants of telecommuter productivity”. *Information & Management* **42**.7 (2005), pp. 1037–1049.

[Ngu14] Nguyen, B. N. et al. “GUITAR: an innovative tool for automated testing of GUI-driven software”. *Automated Software Engineering* **21**.1 (2014), pp. 65–105.

[Ngu13] Nguyen, H. D. T. et al. “SemFix: Program repair via semantic analysis”. *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 772–781.

[Ngu19] Nguyen, S. et al. “Combining Program Analysis and Statistical Language Model for Code Statement Completion”. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 710–721.

[Nlt] *NLTK: Sample usage for sentiment*. 2021.

[Nod20] Noda, K. et al. “Experience Report: How Effective is Automated Program Repair for Industrial Software?” *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, pp. 612–616.

[Oak04] Oakley, B. et al. “Turning student groups into effective teams”. *Journal of Student Centered Learning* **2** (2004).

[Owe15] Owens, J. P. “Student Satisfaction with Group Work: Perceptions and Attitudes”. *Proceedings of the 2007 Academy of Marketing Science (AMS) Annual Conference*. Ed. by Sharma, D. & Borna, S. Springer International Publishing, 2015, pp. 67–73.

[Oye18] Oyetoyan, T. D. et al. “Myths and facts about static application security testing tools: an action research at Telenor digital”. *International Conference on Agile Software Development*. Springer, Cham. 2018, pp. 86–103.

[Pan17] Panadero, E. “A Review of Self-regulated Learning: Six Models and Four Directions for Research”. *Frontiers in Psychology* **8** (2017), p. 422.

[Par15] Parham, J. B. et al. “Influences on assertiveness: gender, national culture, and ethnicity”. *Journal of Management Development* **34**.4 (2015), pp. 421–439.

[Par17] Parihar, S. et al. “Automatic Grading and Feedback Using Program Repair for Introductory Programming Courses”. *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’17. Bologna, Italy: Association for Computing Machinery, 2017, 92–97.

[Par18] Parizi, R. M. et al. “Measuring Team Members’ Contributions in Software Engineering Projects using Git-driven Technology”. *2018 IEEE Frontiers in Education Conference (FIE)*. 2018, pp. 1–5.

[Pat13] Patitsas, E. et al. “Comparing and Contrasting Different Algorithms Leads to Increased Student Learning”. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. ICER ’13.

San Diego, San California, USA: Association for Computing Machinery, 2013, 145–152.

[Pat12] Patterson Kerry, & (Firm), P. *Crucial conversations : tools for talking when stakes are high*. English. New York ; London : McGraw-Hill, [2012], 2012.

[Pau06] Paulus, T. M. et al. "'Isn't It Just Like Our Situation?' Engagement and Learning in an Online Story-Based Environment". *Educational Technology Research and Development* **54**.4 (2006), pp. 355–385.

[Paz15] Pazos, P. & Magpili, N. "Facilitating team Processes in virtual team projects through a web-based collaboration tool and instructional scaffolds". *2015 ASEE Annual Conference & Exposition*. 2015, pp. 26–754.

[PÓ20] Pérez, B. & Rubio, A. L. "A Project-Based Learning Approach for Enhancing Learning Skills and Motivation in Software Engineering". *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Portland, OR, USA: Association for Computing Machinery, 2020, 309–315.

[Pfa03] Pfaff, E. & Huddleston, P. "Does It Matter if I Hate Teamwork? What Impacts Student Attitudes toward Teamwork". *Journal of Marketing Education - J Market Educ* **25** (2003), pp. 37–45.

[Pie10] Pieterse, V. & Thompson, L. "Academic alignment to reduce the presence of 'social loafers' and 'diligent isolates' in student teams". *Teaching in Higher Education* **15**.4 (2010), pp. 355–367.

[Pin06] Pineda, R. & Lerner, L. "Goal attainment, satisfaction and learning from teamwork". *Team Performance Management* **12** (2006), pp. 182–191.

[PM19] Presler-Marshall, K. et al. "Wait, Wait. No, Tell Me. Analyzing Selenium Configuration Effects on Test Flakiness". *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*. 2019, pp. 7–13.

[PM21] Presler-Marshall, K. et al. "SQLRepair: Identifying and Repairing Mistakes in Student-Authored SQL Queries". *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 2021, pp. 199–210.

[PM22a] Presler-Marshall, K. et al. "Identifying Struggling Teams in Software Engineering Courses Through Weekly Surveys". *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. SIGCSE '22 (2022).

[PM22b] Presler-Marshall, K. et al. "What Makes Team[s] Work? A Study of Team Characteristics in Software Engineering Projects". *Proceedings of the 2022 ACM Conference on International Computing Education Research*. ICER '22 (2022).

[Pri18] Price, T. et al. “The impact of data quantity and source on the quality of data-driven hints for programming”. *Proceedings of the International Conference on Artificial Intelligence in Education*. 2018, pp. 476–490.

[Qi14] Qi, Y. et al. “The Strength of Random Search on Automated Program Repair”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 254–265.

[Raf13] Rafferty, P. D. “Group Work in the MBA Classroom: Improving Pedagogical Practice and Maximizing Positive Outcomes With Part-Time MBA Students”. *Journal of Management Education* **37**.5 (2013), pp. 623–650. eprint: `https://doi.org/10.1177/1052562912458644`.

[Rag20] Ragupathi, K. & Lee, A. “Beyond fairness and consistency in grading: The role of rubrics in higher education”. *Diversity and inclusion in global higher education*. Palgrave Macmillan, Singapore, 2020, pp. 73–95.

[Ram20] Ramin, F. et al. “More than Code: Contributions in Scrum Software Engineering Teams”. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ICSEW’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 137–140.

[Rap07] Rapp, T. & Mathieu, J. “Evaluating an Individually Self-Administered Generic Teamwork Skills Training Program Across Time and Levels”. *Small Group Research - SMALL GROUP RES* **38** (2007), pp. 532–555.

[Rav15] Ravnås, O. A. V. *Frida: The engineering behind the reverse-engineering*. 2015.

[Red10] Reddy, Y. M. & Andrade, H. “A review of rubric use in higher education”. *Assessment & evaluation in higher education* **35**.4 (2010), pp. 435–448.

[Ric12] Richardson, I. et al. “A Process Framework for Global Software Engineering Teams”. *Information and Software Technology* **54**.11 (2012), pp. 1175 –1191.

[Rie13] Rienties, B. et al. “The Role of Cultural Background and Team Divisions in Developing Social Learning Relations in the Classroom”. *Journal of Studies in International Education* **17**.4 (2013), pp. 332–353. eprint: `https://doi.org/10.1177/1028315312463826`.

[Rob08] Robbes, R. & Lanza, M. “How Program History Can Improve Code Completion”. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 2008, pp. 317–326.

[Rob17] Roberts, C. et al. “Peer assessment of professional behaviours in problem-based learning groups”. *Medical Education* **51**.4 (2017), pp. 390–400. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/medu.13151`.

[Rud17] Rudawska, A. “Students’ Team Project Experiences and Their Attitudes Towards Teamwork”. *Journal of Management and Business Administration. Central Europe* **25** (2017), pp. 78–97.

[Sah17] Saha, R. K. et al. “Elixir: Effective object-oriented program repair”. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 648–659.

[Sn09] Saldaña, J. *The coding manual for qualitative researchers*. English. Includes bibliographical references (pages [210]-219) and index. London: London ; Thousand Oaks, Calif. : Sage, 2009., 2009.

[Seo14] Seo, H. et al. “Programmers’ Build Errors: A Case Study (at Google)”. *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 724–734.

[Sim02] Sims-Knight, J. E. et al. “Teams in software engineering education”. *32nd Annual Frontiers in Education*. Vol. 3. 2002, S3G–S3G.

[Sin17] Singh, D. et al. “Evaluating how static analysis tools can reduce code review effort”. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2017, pp. 101–105.

[Sit04] Sitthiworachart, J. & Joy, M. “Effective Peer Assessment for Learning Computer Programming”. *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’04. Leeds, United Kingdom: Association for Computing Machinery, 2004, 122–126.

[Sejb] *Software developers, Quality Assurance Analysts, and testers : Occupational outlook handbook*. 2021.

[Sos] *Stack Overflow Developer Survey 2019*. 2019.

[Sto03] Storey, J. D. “The positive false discovery rate: a Bayesian interpretation and the q-value”. *The Annals of Statistics* **31**.6 (2003), pp. 2013 –2035.

[Stu02] Stuyf, R. R. V. D. “Scaffolding as a Teaching Strategy”. 2002.

[Sul16] Sulír, M. & Porubän, J. “A Quantitative Study of Java Software Buildability”. *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, 17–25.

[Swe88] Sweller, J. “Cognitive Load During Problem Solving: Effects on Learning”. *Cognitive Science* **12**.2 (1988), pp. 257–285. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1202_4`.

[Taf16] Tafliovich, A. et al. “Evaluating Student Teams: Do Educators Know What Students Think?” *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE ’16. Memphis, Tennessee, USA: Association for Computing Machinery, 2016, 181–186.

[Tai19] Taipalus, T. & Perälä, P. “What to Expect and What to Focus on in SQL Query Teaching”. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. Minneapolis, MN, USA: ACM, 2019, pp. 198–203.

[Tai18] Taipalus, T. et al. “Errors and Complications in SQL Query Formulation”. *ACM Trans. Comput. Educ.* **18**.3 (2018).

[Tak14] Takeda, S. & Homberg, F. “The effects of gender on group work process and achievement: an analysis through self- and peer-assessment”. *British Educational Research Journal* **40**.2 (2014), pp. 373–396. eprint: `https://berajournals.onlinelibrary.wiley.com/doi/pdf/10.1002/berj.3088`.

[Tor] Torvalds, L. *Git*.

[Tra20] Trautsch, A. et al. “A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in apache open source projects”. *Empirical Software Engineering* **25**.6 (2020), pp. 5137–5192.

[Tuc06] Tucker, R. & Reynolds, C. “The Impact of Teaching Models, Group Structures and Assessment Modes on Cooperative Learning in the Student Design Studio”. *Journal for Education in the Built Environment* **1**.2 (2006), pp. 39–56. eprint: `https://doi.org/10.11120/jebe.2006.01020039`.

[Tuc65] Tuckman, B. W. “Developmental sequence in small groups.” *Psychological Bulletin* **63**.6 (1965), pp. 384–399.

[Tuc77] Tuckman, B. W. & Jensen, M. A. C. “Stages of Small-Group Development Revisited”. *Group & Organization Studies* **2**.4 (1977), pp. 419–427. eprint: `https://doi.org/10.1177/105960117700200404`.

[Vas15] Vasilescu, B. et al. “Quality and Productivity Outcomes Relating to Continuous Integration in GitHub”. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, 805–816.

[Vat21] Vattøy, K.-D. et al. “Examining students’ feedback engagement and assessment experiences: a mixed study”. *Studies in Higher Education* **46**.11 (2021), pp. 2325–2337. eprint: `https://doi.org/10.1080/03075079.2020.1723523`.

[Vei21] Veikkola, A. “Selenium ja Playwright-testiautomaatiotyökalujen vertailu web-automaatiokäytössä” (2021).

[Vil17] Vila, E. et al. “Automation Testing Framework for Web Applications with Selenium WebDriver: Opportunities and Threats”. *International Conference on Advances in Image Processing*. ICAIP 2017. Bangkok, Thailand, 2017, pp. 144–150.

[Wal21] Walsh, A. R. et al. “Exploring the Team Dynamics of Undergraduate Engineering Virtual Teams During the Rapid Transition Online Due to COVID-19”. *2021 ASEE Virtual Annual Conference Content Access*. 2021.

[Wan17] Wang, C. et al. “Synthesizing Highly Expressive SQL Queries from Input-output Examples”. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 452–466.

[Wei09] Weimer, W. et al. “Automatically Finding Patches Using Genetic Programming”. *Proceedings of the 31st International Conference on Software Engineering*. ICSE ’09. IEEE Computer Society, 2009, pp. 364–374.

[Wil00] Williams, L. A. & Kessler, R. R. “The effects of "pair-pressure" and "pair-learning" on software engineering education”. *Thirteenth Conference on Software Engineering Education and Training*. 2000, pp. 59–65.

[Wil01] Williams, L. A. & Kessler, R. R. “Experiments with Industry’s “Pair-Programming” Model in the Computer Science Classroom”. *Computer Science Education* **11**.1 (2001), pp. 7–20. eprint: `https://doi.org/10.1076/csed.11.1.7.3846`.

[Wil20] Wills, C. E. *Analysis of current and future computer science needs via advertised faculty searches for 2021*. 2020.

[Wil13] Wilson, S. G. “The Flipped Class: A Method to Address the Challenges of an Undergraduate Statistics Course”. *Teaching of Psychology* **40**.3 (2013), pp. 193–199. eprint: `https://doi.org/10.1177/0098628313487461`.

[Wis20a] Wisniewski, B. et al. “The Power of Feedback Revisited: A Meta-Analysis of Educational Feedback Research”. *Front. in Psych.* **10** (2020).

[Wis20b] Wisniewski, B. et al. “The Power of Feedback Revisited: A Meta-Analysis of Educational Feedback Research”. *Frontiers in Psychology* **10** (2020).

[Wög05] Wögerer, W. *A survey of static program analysis techniques*. Tech. rep. Citeseer, 2005.

[Woh12] Wohlin, C. et al. *Experimentation in Software Engineering*. English. Berlin, Heidelberg: Berlin, Heidelberg : Springer Berlin Heidelberg : Imprint: Springer, 2012., 2012.

[Xua16] Xuan, J. et al. “Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs”. *IEEE Transactions on Software Engineering* (2016).

[Yi17] Yi, J. et al. “A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments”. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 740–751.

[You21] Young, J.-G. et al. “Which contributions count? Analysis of attribution in open source”. *2021 IEEE/ACM MSR*. 2021, pp. 242–253.

[Zam17] Zampetti, F. et al. “How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines”. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, pp. 334–344.

[Zap16] Zapf, A. et al. “Measuring inter-rater reliability for nominal data – which coefficients and confidence intervals are appropriate?” *BMC Med. Rsrch. Methodology* **16**.1 (2016), p. 93.

[Zwe21] Zweben, S. & Bizot, B. *2020 Taulbee survey: Bachelor’s and doctoral degree production growth continues but new student enrollment shows declines*. 2021.