From Intuition to Evidence: A Data-Driven Approach to Transforming CS Education

Anthony J. Allevato

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Stephen H. Edwards, Chair
Manuel A. Pérez-Quiñones
Naren Ramakrishnan
Deborah G. Tatar
Carlos Evia Puerto

July 16, 2012
Blacksburg, Virginia

Keywords: Computer science education, automated grading, assessment, evaluation, intuition, evidence, Web-CAT, Eclipse, BIRT, reporting, data collection, student behavior, time management, procrastination, student performance, extra credit, C++, Dereferee, memory management, pointers, item response theory, reference tests, JUnit, difficulty, discriminating ability

Copyright 2012, Anthony J. Allevato

From Intuition to Evidence: A Data-Driven Approach to Transforming CS Education

Anthony J. Allevato

(ABSTRACT)

Educators in many disciplines are too often forced to rely on intuition about how students learn and the effectiveness of teaching to guide changes and improvements to their curricula. In computer science, systems that perform automated collection and assessment of programming assignments are seeing increased adoption, and these systems generate a great deal of meaningful intermediate data and statistics during the grading process. Continuous collection of these data and long-term retention of collected data present educators with a new resource to assess both learning (how well students understand a topic or how they behave on assignments) and teaching (how effective a response, intervention, or assessment instrument was in evaluating knowledge or changing behavior), by basing their decisions on evidence rather than intuition. It is only possible to achieve these goals, however, if such data are easily accessible.

I present an infrastructure that has been added to one such automated grading system, Web-CAT, in order to facilitate routine data collection and access while requiring very little added effort by instructors. Using this infrastructure, I present three case studies that serve as representative examples of educational questions that can be explored thoroughly using pre-existing data from required student work. The first case study examines student time management habits and finds that students perform better when they start earlier but that offering extra credit for finishing earlier did not encourage them to do so. The second case study evaluates a tool used to improve student understanding of manual memory management and finds that students made fewer errors when using the tool. The third case study evaluates the reference tests used to grade student code on a selected assignment and confirms that the tests are a suitable instrument for assessing student ability. In each case study, I use a data-driven, evidence-based approach spanning multiple semesters and students, allowing me to answer each question in greater detail than was possible using previous methods and giving me significantly increased confidence in my conclusions.

This work is supported in part by the National Science Foundation under grants DUE-0618663 and DUE-0633594. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Acknowledgments

First, I would like to thank my committee chair, Dr. Stephen Edwards, who is by far one of the most talented computer scientists and software engineers I've ever known. Your ability to zero in on the most elegant solutions to design and implementation problems continues to astound me, and I've learned more from my years of working with you than I think I could have anywhere else. I also thank the rest of my committee, Dr. Manuel Pérez-Quiñones, Dr. Naren Ramakrishnan, Dr. Deborah Tatar, and Dr. Carlos Evia, for their support and valuable feedback at various stages throughout my research, and on other collaborative work that we have done together at the university.

My fiancée Sarah has been incredibly patient during all the evenings and weekends that I had to spend working on this dissertation, developing teaching resources, writing publications, grant proposals, and all of the other projects that have cut into our quality time. Thanks for only posting *one* embarrassing photo of me falling asleep at my computer on Facebook. I love you and I couldn't have done it without you!

To my parents, Richard and Susan Allowatt, thanks for all the support—emotionally and sometimes financially—during my time at Virginia Tech (and, well, my whole life). And thanks further for respecting my wishes when I put a moratorium on people asking me the question, "When are you going to finish your Ph.D.?"

To my fellow grads and undergrads occupying the CS education research nook in Torgersen Hall—Jason Snyder, Kevin Buffardi, Ellen Boyd, Mohammed Al-Kandari, and Mike Woods—thanks for all the entertaining conversations and "business" lunches. Having you all around made me enjoy coming into the office and getting work done (or getting distracted from it). I also have to thank Jackie Falatko for pointing me toward some great jazz and hard rock to listen to and help me focus during my final couple months of writing.

To Dr. Calvin Ribbens and Dr. Barbara Ryder, thanks for many semesters of GTA support during my time as a graduate student, and for being flexible and willing to carve out a full-time position in the department for me to continue my passion for teaching for the past year as I got my motivation back to work on this dissertation. And to so many other faculty members in the department—Dr. Sean Arthur, Dr. Godmar Back, Dr. Osman Balci, Mr. Dwight Barnette, Dr. Roger Ehrich, Dr. John Lewis, Mr. William McQuain, Dr. Cliff

Shaffer, and Dr. Eli Tilevich—thanks for all of the valuable guidance and insight that you provided me with as I have taken classes from you, worked with you, collaborated with you, or simply chatted with you about our respective work.

So many of the CS department's staff, both current and former, have been warm and accommodating during my time here. Terry Arthur, Libby Bradford, and Frenda Wall, without your expertise and assistance I would not have survived my few months filling in as an undergraduate academic advisor. So many others—Chris Arnold, Ryan Chase, Ginger Clayton, Melanie Darden, Jessie Eaves, Jody Humphreys, Sharon Kinder-Potter, Julie King, Carol Roop, and Emily Simpson—have been incredibly helpful with all the administrative tasks that I've come to them with, both as a student and as a teacher.

I've made so many other friends during my time at Virginia Tech—students in classes alongside me, former students of my own, and folks in or around Blacksburg—who have enriched my life in various ways that it would be impossible to name them all, but I'll list a number of them and hopefully the ones I forgot won't read this: Amine, Ben, Erika, Heather, Jaime, Jocelyn, Kaslin, Laurian, Michael, Priyanka, Ricardo, Sameer, Sara, Shvetha, Tara...

Contents

List of Figures

List of Tables

1
Introduction

Educators are continually trying to improve their understanding of the behaviors of their students, both in order to help those students succeed in class and also to refine their own teaching methods. This is true across every discipline; in the case of computer science education, understanding how students learn is of paramount importance during introductory classes such as CS1 and CS2 in order to improve retention—especially for minority groups and at-risk students—and ensure that students will succeed as computer scientists when they graduate.

Too often, however, educators are left relying on intuition to drive their teaching. Consider—as one example—the problem of encouraging students to exercise good time management skills on programming projects. It is unlikely that students or other teachers would disagree with an instructor who asserts that students who begin working earlier on a project are likely to perform better, even if that assertion is based solely on intuition. An instructor may even offer extra credit to students who complete their work early in order to encourage the time management behavior that they would like to see.

Educators wish to know not only how well their students are learning, but wish to understand the quality of their teaching and the artifacts they use for teaching as well. The work produced by students and the grades that they receive result from the interaction of a vast number of variables. Some of these variables are student-oriented: possession of innate skills, a student's ability to take in new knowledge and concepts, and his or her ability to apply those concepts and synthesize new knowledge derived from them. Other variables relate to the instructor: how effective is a particular teaching method, either in general or for a particular group of students?

In his article "The Naughties in CSEd Research: A Retrospective" [Lis10], Raymond Lister draws an analogy to the development of modern medicine:

> If two patients in adjacent beds, with the same complaint, were treated by different doctors, and one patient lived while the other died, then there was a basis for deciding which treatment was better.

Lister advocates for automatic and routine data collection to strengthen quantitative research in computer science education, arguing that educators need these data to systematically study why some students succeed and other students fail.

The sheer number of factors involved in assessing student work make it extremely difficult to evaluate each factor individually when presented with students' final work products. The traditional approach used by educators is a cycle much like the following:

1. Collect student work.
2. Assess the collected work and quantify it with a broad measure, such as composite final score.
3. Intuitively devise a response or intervention based on the broad measure.
4. Collect further student work.
5. Assess the new collected work and quantify it as before.
6. Intuitively decide whether or not the response was effective based on a comparison to the same broad measure from the previous iteration; accept/reject the intervention or adjust and repeat.

One of the weak links in this process are the reliance on human intuition in steps 3 and 6. What is missing is *evidence* that drives both the initial incentive response and the continued offering of that incentive. Continuing the example from earlier, is the instructor's intuition even true that students who start earlier perform better on the assignments? If not, then there is no compelling reason to offer the incentive in the first place. On the other hand, even if the intuition happens to be correct, then we must still determine whether the incentive, once applied, had the desired effect of improving these behaviors in students. Furthermore, if the success of an intervention can only be evaluated based on differences in broad measures before and after the intervention, it is difficult to state categorically that only the intervention and no other factors caused the change.

A certain degree of intuition is typically involved in steps 2 and 5 as well. Consider, for example, that the score for a programming assignment in an introductory course might include an assessment of the student's class design, quality of documentation, and code style. Graders often have an intuitive grasp of which of these components are problem areas for students and this can guide them in both assigning those scores and in understanding which students excel in those areas and which have deficiencies. The assignment of these

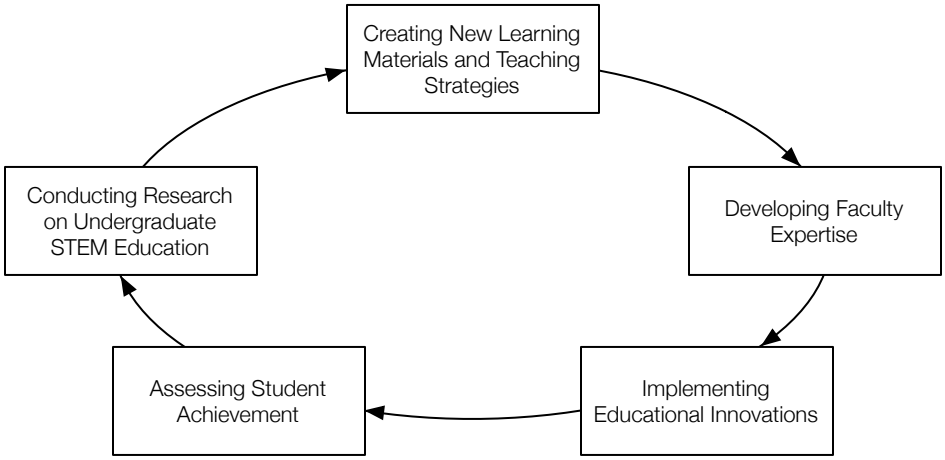

Figure 1.1: The cyclic model depicting the relationship between knowledge production and improvement of practice in undergraduate STEM education, upon which the NSF CCLI program was based.

scores, however, often still relies on intuition to decide how to map a by-hand inspection of the student's code onto a numeric scale.

The critical need for data-driven assessment practices is recognized by agencies and other organizations who provide support for researchers in education. The National Science Foundation's Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics (TUES) program is a source of funding for educators in science, technology, engineering, and mathematics (STEM) to research, develop, and evaluate new teaching techniques and materials. In their 2007 program solicitation [CCL07] under the program's former name, "Course, Curriculum, and Laboratory Improvement (CCLI)", the program directors depict a cyclic model that relates knowledge production to improvement of practice in undergraduate STEM education, with assessment of student achievement and research on assessment results influencing the designs of the other components in the model (see Figure 1.1).

It is clear that educational assessment in STEM fields—such as computer science education, which is the primary focus of the work presented here—must be regarded with the same degree of rigor as experimentation in other scientific disciplines. If we do this, as Lister recommends, then we can strengthen the educational assessment cycle that we saw previously:

1. Collect student work.
2. Assess the collected work and quantify it with as many fine-grained measures as are practical.
3. Examine each measure and the interactions between them and devise an *evidence-based* response or intervention.

4. Collect further student work.
5. Assess the new collected work and quantify it as before.
6. Examine the measures and interactions between them and decide. *based on new evidence* whether or not the response was effective; accept/reject the intervention or adjust and repeat.

By replacing intuition with real evidence, educators can better serve their students by adjusting more quickly and in a more targeted manner to deficiencies in either student learning or effectiveness of teaching. However, setting up experiments to collect the necessary data for educators to perform these evaluations can be a significant burden, especially if the range of questions of interest is very broad. In order to achieve this goal, we must have infrastructure available that makes automatic and routine data collection as effortless as possible for both students and instructors alike.

1.1 Current State of the Art

Techniques used by computer science instructors to assess the code submitted by their students vary greatly. Some instructors rely on entirely manual evaluation, where the grader is a human being who executes the code to gauge its correctness and then inspects the code for adherence to style guidelines. Others use some degree of automatic evaluation, where the grader is a computer program that performs the correctness tests and style checks, as well as computes other metrics that would be intractable to determine by hand such as the percentage of code covered during execution.

Automated assessment systems have long been in widespread use due to the advantages that they provide for both the instructor and the student. Instructors benefit from the ability to better distribute their time and resources, or those of their teaching assistants, by relying on the computer to carry out the majority of the grading. More importantly, students can receive immediate feedback on the correctness of their code, which permits scenarios that allow students multiple attempts on an assignment, using that feedback to refine their solution and resubmit. As we can see, data collection can also play a crucial role in the students' development process; if we collect more data about the students' work as they are developing their solutions, and we use those data to reason about their ongoing performance and synthesize useful feedback for them, then they can be more productive and have a greater chance of success.

Web-CAT is the most widely adopted of these automated grading systems, currently in use at over 70 institutions with more than 10,000 users. The currently running instance of Web-CAT at Virginia Tech has been in use since January 2007 and provides automatic grading support for 18 institutions; many other institutions maintain their own Web-CAT servers.

Web-CAT makes use of an extensible plug-in-based architecture so that different programming languages and evaluation techniques can be used without requiring changes to the server itself. For two of the most commonly used languages in early CS education—Java and C++—Web-CAT and its grading plug-ins collect a wide range of data about work that students submit, including:

- The time of the submission and its index among all submissions made by that student to the assignment
- The number of lines of code written, distinguished by commented vs. non-commented lines and also by test code vs. non-test code
- The amount of code coverage (statement, method, or decision) achieved by executing the student's own tests
- The ratio of the instructor's reference tests that passed/failed against the student's solution, the individual results of these tests, and descriptions of any reasons for failure

To see how data such as this plays an important role in studying computer science education, consider the following three questions which were alluded to previously:

1. How do students' time management skills affect their score on a programming assignment?
2. Is an educational tool being effective in helping students identify and diagnose problems with their code?
3. What is the quality of the reference test suite that is being using to assess the correctness of students' solutions?

None of these questions can be answered in a satisfying way by using only the final snapshot of a student's work and his or her resulting score. The first question requires information about when a student started as well as when he or she finished the assignment. The second question depends on the nature of the change that the instructor wishes to see in student behavior or performance and how that change is measured. That change might be reflected in the student's final score, but a number of other factors are also used to compute that value, then using the combined score would not be appropriate to measure a shift in a single factor. The third question requires fine-grained information about student performance on individual items used to assess correctness, so again using the combined score would be inappropriate.

Semester	Submissions	Students	Institutions	Assignments	Course Offerings	Avg. Subs./Student[1]	Avg. Subs./Day[2]	Highest Daily Subs.
Spring 2007	36974	636	4	144	25	6.52	244.86	1234
Fall 2007	34450	491	5	127	19	6.50	271.26	1239
Spring 2008	39443	840	7	227	32	4.98	116.70	935
Fall 2008	37564	742	6	234	25	4.85	274.19	1236
Spring 2009	46641	856	7	241	21	4.60	220.00	1219
Fall 2009	32066	713	8	158	32	3.53	123.81	1188
Spring 2010	34340	560	5	144	23	5.34	268.28	1083
Fall 2010	37845	437	6	125	17	6.48	326.25	1609
Spring 2011	54775	884	5	195	25	5.22	273.88	1681
Fall 2011	56814	730	6	194	23	5.93	188.13	1838
Spring 2012	81718	1359	6	208	33	6.42	480.69	2419
Combined	492630	6374	14	1481	275	5.42	344.26	2419

Table 1.1: Usage statistics for all institutions hosted on the Virginia Tech Web-CAT server from Spring 2007 to Spring 2012.

1.2 Incorporating Data into CS Education Research

Chapter 2 describes a number of approaches that CS educators have taken to increase the amount of useful data available about student programming habits. Some of these approaches collect data based on the results of automated assessment of work that students submit to a grading system. Other approaches collect fine-grained data during students' actual programming sessions in order to quantify the time they spend on assignments.

In the case of Web-CAT, most programming courses that use the system for grading allow students to make an unlimited number of submissions until the due date for an assignment, and for each submission the system computes and preserves the metrics and statistics mentioned previously. This gives instructors access to a data store of unprecedented magnitude that represents not only the behaviors of their students but also of their progress toward a working solution with respect to correctness.

Table 1.1 presents usage statistics for Virginia Tech's primary Web-CAT server, which has

[1] Average (mean) number of submissions by students on each assignment.
[2] Average (mean) number of submissions per day, excluding days where no submissions were made.

Semester	Submissions	Students	Assignments	Course Offerings	Avg. Subs./Student	Avg. Subs./Day	Highest Daily Subs.
Spring 2007	23984	371	80	15	5.85	224.15	1007
Fall 2007	26914	382	80	13	6.26	240.30	1099
Spring 2008	12778	263	88	13	3.90	79.86	636
Fall 2008	16318	341	110	13	4.77	158.43	693
Spring 2009	13773	260	79	8	4.07	136.37	1165
Fall 2009	17341	405	87	15	2.76	137.63	775
Spring 2010	22226	355	73	15	5.22	185.22	1019
Fall 2010	24774	227	56	9	7.19	258.06	1444
Spring 2011	33653	507	72	17	5.44	308.74	1206
Fall 2011	41512	509	75	14	6.41	274.91	1755
Spring 2012	49315	708	106	15	6.38	436.42	1495
Combined	282588	2861	750	147	5.35	235.88	1755

Table 1.2: Web-CAT usage statistics for Virginia Tech students only from Spring 2007 to Spring 2012.

been in service since Spring 2007. Previous semesters (from Spring 2004 through Fall 2006) are backed up in separate databases not on the current production server. Between Spring 2007 and Spring 2012 (excluding summer semesters), the current server has collected and/or graded nearly half a million submissions from over 6,000 students at 14 universities. On average, the server has processed 344 submissions per day, with a maximum of 2,419 submissions occurring on a single day during Spring 2012. Table 1.2 shows the same statistics when I limit consideration to only Virginia Tech courses and students.

The greatest advantage to this data store is that it is derived naturally from work that is required for students to complete the course. Each submission attempt that they make to an assignment is processed, analyzed, and stored without requiring any extra effort on the part of the instructor. Likewise, the work discussed in Chapter 3 greatly reduces the effort required by the instructor to extract the desired data. Through normal use of this system at Virginia Tech, we have already constructed a number of reports than can be reused by other instructors who want to begin examining data from their courses right away. In the event that the existing reports are insufficient, they can be easily modified or new ones can be created using the tool described in Chapter 3. This ease of access opens up great potential

for post-hoc analyses using a large number of factors that previously would have required extensive prior preparation otherwise.

Furthermore, this data store continues to grow over time because data from previous semesters and courses is automatically retained. This gives instructors the ability to conduct studies on a much larger scale than was previously possible. Studies that involve data from multiple students, assignments, courses, and semesters, and that make use of appropriate groupings and within-subjects designs, provide instructors with much stronger results than those that focus on a smaller data set. For this reason, they should use as much of the data store as is relevant to answering their research questions. As an example, the case study in Chapter 4 explored ten semesters worth of data from 2004 to 2008; this data set is currently the largest publicly available data set of statistics and results about student-written programming work, and we are planning to expand it forward to 2011 as well by including new data collected since the initial IRB protocol.

1.3 Research Goal

With such a vast data store available to Web-CAT users, this dissertation addresses the following question:

> How can computer science educators make use of pre-existing data from required student work both to assess how students are learning and to assess the quality of their teaching?

By "assessing learning," I am referring to the process of evaluating what students know (have they mastered a concept or can they apply various programming techniques to produce bug-free solutions?) and how they behave (do they wait until the last minute to start an assignment? Do they test incrementally, or all at once at the end?), using appropriate performance measures.

On the other side of the issue, the process of "assessing teaching" is the evaluation of teaching techniques, methods, and knowledge representations that instructors might use in the classroom to communicate new knowledge to students. In this dissertation, however, I broaden that definition to include many other types of artifacts that the instructor provides to students to enhance learning and policies that he or she puts into place with regards to the classroom experience or completion of assignments. This includes educational tools that are intended to reinforce concepts or diagnosing deficiencies in knowledge, assessment instruments that evaluate the proficiency of students, and interventions that are intended to effect positive change in students' behaviors.

The ease with which these data can be collected allows educators to embark on the cyclical data-driven exploration process that was described at the beginning of this introduction. As

Web-CAT gathers a wealth of data from students working on their required assignments, an instructor can analyze those data immediately after the assignment is complete, or even during the course of the assignment. After testing whichever hypothesis he or she originally formed, a response or intervention can be attempted, perhaps as soon as the next assignment in the course. Then, the cycle repeats as new data are collected in the same manner as before. The effects of that intervention can be evaluated using the full accumulated data set, and the quality of the intervention is either confirmed or new changes can be explored.

1.4 Broader Assessment

The assessment of teaching that is made possible by this work has value as an accreditation tool as well [WP05] [WP06]. Under the "EC 2000" requirements from the Accreditation Board for Engineering and Technology (ABET), accreditation is to be treated as a continuous process rather than an occasional exercise performed during visits to departments. In order to boost the quality of instruction from year to year, it is incumbent upon educators to employ routine, continuous data collection in their courses and have convenient access to the collected data for analysis. In the same sense that instructors should no longer rely on hunches or intuition in order to improve the quality of their teaching as it affects the students directly in the classroom, they should likewise be able to use evidence of student learning based on data collected across multiple semesters to make a strong case to the accreditation board that learning objectives are being properly met.

1.5 Structure of This Document

Chapter 2 presents a review of existing literature in the area of data collection from student-submitted programming assignments, as well as studies that have been done by analyzing such data. Chapter 3 describes the design and implementation of the reporting tool that was integrated into Web-CAT to enable large-scale data extraction and discusses some of the major issues that arose during development.

The three chapters that follow present case studies that serve to illustrate the breadth of possible questions that can be explored using different projections of the data store, and the depth to which both *learning* and *teaching* can be assessed using a data-driven approach.

Chapter 4 directly examines *student behavior.* I evaluate learning with an exploration of how students' time management on programming assignments positively or negatively impacts their results by isolating students who performed well on some assignments and poorly on others and then looking for differences in when they started working and finished working in each case. I assess teaching by analyzing an extra-credit incentive that was offered in

an effort to improve students' time management habits and evaluating whether or not the incentive had the desired effect.

Chapter 5 describes the effects of an *educational tool* that students used in their work on programming assignments—a small toolkit intended to improve students' understanding of pointers and manual memory management in C++, a frequent area of difficulty for many novice programmers. This case study evaluates learning by examining feedback from the tool to identify the specific types of problems that students encountered most frequently. At the same time, I evaluate a teaching tool by determining whether it was effective in improving the overall performance of students who used it compared to those who chose to opt out.

Chapter 6 uses fine-grained data from execution of student code against the instructor's *reference test suite.* I treat the reference test suite as a collection of "true/false" questions and apply concepts from item response theory to gain more understanding of the items. I evaluate learning by examining which parts of an assignment seemed to cause the most difficulty for students. Then, I assess a teaching artifact—the reference test suite—that itself is used to assess the proficiency of students and determine if it serves as a good indicator of said proficiency.

Finally, Chapter 7 presents my conclusions, lists some published contributions that the work has already produced prior to this dissertation, and describes some areas of future work.

2
Literature Review

Due to the disparate nature of the three case studies presented in this dissertation, this literature review has been divided into separate sections for each study.

2.1 Case Study: Evaluating Student Behavior

As described in the introduction, existing automated assessment systems that support data extraction or analysis and have generated publications were designed with a very specific goal in mind, rather than generality. Furthermore, a great majority of published and ongoing work in data-driven CS education research deals not with large submissions of code from students, but rather with computer-based test-taking and intelligent tutoring systems. Recent years have brought more in-depth research into student programming habits on large-scale coding projects, however.

2.1.1 Treating Each Attempt as a Snapshot for Analysis

Since Web-CAT independently stores each submission attempt that a student makes to an assignment, its users have access to the entire code history of each student's progress on an assignment. Other experimenters have used CVS repositories as project snapshots in a similar fashion in order to analyze their students' work.

[MLRW05] describes an experiment that tracks the evolution of students' code through the CVS repositories in which it was stored. The analysis centers around quantifying students' work habits through metrics such as the amount of code written, the amount of revisions between commits, time between commits, and so forth, and the authors attempt to find a correlation between these features and the performance of each student on the project. The

features that the authors extracted can be grouped into three categories: CVS repository data (such as the number of revisions per file and the time of the revision), source code characteristics (number of while loops, comments, and certain formatting habits), and higher-level PMD rule violations. Furthermore, student academic data were added to the master data set and students were clustered into groups according to their rank. For quantitative and comparative purposes, each of these features was normalized to have zero mean and unit variance.

The results of this experiment showed little correlation between the habit-related data that they collected and student scores. The authors themselves describe their result as "surprising": "Of the 166 features we examined, only 3 had significant correlation with grade. Of these, the most significant was the total number of lines of code written." While this is somewhat grim in the sense that it implies that much of the extra data that is available does not provide a significant amount of useful information, their findings still gives us some insight into similar types of data that we may find useful in our analysis, and how to make it easily accessible to our users.

One notable obstacle that was addressed by Mierle et al. was identifying CVS commits that occurred as part of the same request by the user. Since the repository itself does not keep track of which transactions were made as part of a single command by the client, the authors used an approach that grouped transactions from the same user with the same comment string that occurred within a fixed, small number of seconds of each other. With a Web-CAT submission, the opposite problem emerges: the entire transaction occurs on the project files as a whole, even those that were not changed between attempts. If an instructor was interested in identifying only those files that were modified, he or she could compute a hash for each one and compare these hashes between neighboring submissions.

While Web-CAT does not represent a strict CVS repository, it can be considered to be an "ad hoc" repository in the form of the submission set for each student. By mapping the concept of a CVS "commit" operation to a submission attempt on Web-CAT, its users can adopt some of the aforementioned authors' methodologies quite easily. Ideally, in a scenario where an unlimited number of submission attempts is permitted for an assignment, students would be encouraged—to paraphrase former Chicago mayor Richard Daley—to "submit early and submit often." At the very least, even a small number of submissions provides us with a handful of "anchor" points from which to analyze their code and habits versus the score that they received on that submission or the project as a whole.

2.1.2 Larger-Scale Analysis and Predictions

Moving up from performance on individual projects to performance across multiple courses, [Cha06] discusses how student performance in lower-level computer science courses may be used as a predictor of performance in later semesters. The author performed an experiment using all of the required computer science courses at the then-CSAB-accredited (now ABET)

program at the United States Air Force Academy and builds predictive models for what he terms as "assessments"; that is, programming assignments, exams, and the overall course grade represented both as a percentage and as the letter grade encoded using standard GPA values.

Chamillard builds these predictive models as linear regressions, using all of the assessments above as the dependent variables (predicted values) and the course grades from all previously completed courses as the independent variables (predictor values). Incomplete data precluded his original intent of using all previous assessments as the independent variables. The author's results showed that the strongest pairwise correlations occurred between courses that were taken during the same semester, implying that students tend to either have a good semester or a bad semester across the board. One of the weakest correlations involved a course sequence in which half of the student's grade is determined by group, rather than individual, work. This seems to imply that the collaborative nature of the work neutralizes the effects of each individual student's prior performance on the grade achieved in that course sequence.

As of the time of this writing, data backups from Virginia Tech's Web-CAT server—including the database and entire code submission history for each student—are available for each semester dating back to Spring 2004, which makes available a large sample population of current and former students from which interested users can look for trends extending beyond performance on single assignments. As Chamillard discovered, however, I would also encounter the problem of incomplete data, due to curriculum changes that have occurred during that time (such as the restructuring of a course's material or the addition of a lab section to a course), or courses taught by instructors who used Web-CAT only as an electronic drop box (with no automated grading) or who used another grading system entirely.

If I were to restrict some of my analyses to the subset of students for which complete data exists, and if this sample is significantly large, then I could extend the methodology of Chamillard beyond his definitions of "assessments," thanks to the more comprehensive data that Web-CAT collects. This would make it possible to identify correlations that might make good long-term predictors based on other metrics available in the Web-CAT data store, such as unit-testing statistics.

2.1.3 Collecting Real-Time Data on Student Behaviors

Hackystat [JKA+04] is a framework that allows users to collect real-time behavioral data about the software development process. The framework is implemented via plug-ins for many popular IDEs, including Eclipse and Microsoft Visual Studio, which provide "sensors" that collect data about the user's programming habits by polling the state of the editor at 30-second intervals to measure the time a user spends performing certain tasks, such as editing a file. These data are then transmitted to a server at a user-configurable interval to be processed, where a timeline is constructed that describes the user's workflow and focus

of attention. Hackystat also supports some of the same metrics that Web-CAT computes, such as unit testing results and coverage for Java code, to generate statistics that allow a user to examine the results of the time spent on those tasks. These sensors must be configured by the user, however—an impediment to quick adoption by instructors teaching novice programmers.

BlueJ is a simplified IDE targeted at novice programmers in introductory Java courses that has seen widespread adoption in many computer science programs. BlueJ provides a powerful extension model that allows developers to add new features and inject extra functionality into existing features. Due to its customizability and popularity, many computer science education researchers have used it as a basis for their own work.

Matthew Jadud [Jad05] developed a BlueJ extension that allowed him to track the compilation behavior of his students in order to perform a quantitative, empirical analysis of those behaviors. He tracks the time between compilation events, the specific error that caused compilation to fail (for pedagogical reasons, BlueJ only displays one error at a time to the user), and changes in success/failure between pairs of compilation attempts (did code that failed to compile before properly compile later, or vice versa, or did it fail/succeed both times). Jadud's work only focuses on single BlueJ sessions and does not track across multiple sessions.

ClockIt [NBFJ+08] is another BlueJ extension that expands this idea by tracking not only compilation events, but other types of events as well. It tracks ten types of events: *project-open, project-close, package-open, package-close, compilation-success, compilation-error, compilation-warning, invocation-event, file-change,* and *file-delete.* "Invocation events" refer to instantiating objects on the interactive BlueJ object bench and invoking methods on those objects, which indicate when students are performing testing of their code. Unlike Jadud's work, ClockIt submits its event log to an online service upon exit from BlueJ so that data can be tracked across multiple sessions for the same project. The extension also provides visualizations for both instructors and students to examine the data collected on their projects.

Retina [MKLH09] is a similar system developed to track events during student development, but which integrates with BlueJ, Eclipse, and the command-line `javac` compiler. In addition to tracking compilation events and errors, it also tracks uncaught runtime exceptions that occur when the student executes his or her code and provides enhanced diagnostic feedback. Retina distinguishes itself from Jadud's work and from ClockIt by providing recommendations to students based on the behavioral patterns that the tool observes. Using common instant messaging protocols, the system can communicate to the user recommendations based on high error rates per compilation, excessive time spent on an assignment, and making the same error multiple times.

During a special session at SIGCSE 2012, the lead developers of BlueJ discussed a project called Blackbox, which would instrument their IDE with the ability to collect large-scale data that would be of interest to computer science education researchers [KU12]. Their goal is

to create a centralized repository of data about development work to which BlueJ users can opt to contribute. These data would include information about development habits similar to other tools discussed previously: edit events, file saving events, compilation events (both successful and with details about errors when they occur), and program execution events. Educators would be given open access to these (anonymized) data in order to carry out their own investigations about the behavior of novice programmers.

While this project is still in the very early stages of development, Kölling and Utting leave a number of important questions unanswered. BlueJ is merely an IDE, so by itself it cannot provide any data about student program performance as evaluated by an automated assessment system like Web-CAT. Understanding how students develop programs only addresses half of the problem for many studies; investigators also wish to know how those behaviors influence their performance. The authors currently do not address this beyond stating that they will provide hooks for users to inject their own data into the data store, but how they will go about this is unclear.

Another important question is how the anonymization of the data—necessary when the repository will be centralized—will be performed. Many investigators would like to be able to track students across multiple assignments or even multiple courses in their studies, which would require that some globally unique identifying information still be available. The authors do not appear to have a system in place that would ensure that each user would be able to be consistently identified in the data store regardless of the class they are enrolled in or computer they are working on; it is also not clear that they even wish to be able to do this. A significant number of studies are impossible without some form of "study code" to identify individual users.

For some types of studies, having such a large data set to explore is certainly a boon to researches. However, I feel that many studies, such as those presented in this dissertation, would benefit from and be more tractable using localized data collection, where a system like Web-CAT can provide more local control and ease of access to the data set. Section 7.2 on future work discusses some ways that researches are planning to extend Web-CAT to achieve these goals.

2.2 Case Study: Evaluating an Educational Tool

The second case study examines the types of memory-related errors that beginning programmers make in C++ and the effectiveness of a tool designed to assist students in mastering that concept.

For novice programmers, the proper use of pointers and memory management are frequently found to be a source of extreme difficulty. A survey conducted by Milne and Rowe [MR02] asked 26 students to rank 28 programming concepts on a 7-point Likert scale where 1 indi-

cated "very easy to learn" and 7 indicated "very hard." The survey also collected responses from 40 teachers who ranked the concepts based on their perception of students' difficulties.

Of those 28 concepts, the students ranked "dynamic allocation of memory (with `malloc`)" as the 4th most difficult ($N = 26, \mu = 4.192$) and "pointers" as the 5th most difficult ($N = 26, \mu = 4.154$). Likewise, teachers ranked "pointers" as the most difficult ($N = 37, \mu = 6.054$) and "dynamic allocation of memory (with `malloc`)" as the 3rd most difficult ($N = 34, \mu = 5.882$).

Lahtinen, et al. [LAMJ05] conducted a survey similar to Milne and Rowe, but collected a much larger number of responses. They asked 559 students and 34 teachers to rank 12 programming concepts on a 5-point Likert scale from where 1 indicated "very easy to learn" and 5 indicated "very difficult."

Of those 12 concepts, students ranked "pointers and references" as the most difficult on average ($N = 518, \mu = 3.59, \sigma = 1.04$). Teachers also ranked this concept as the most difficult ($N = 32, \mu = 4.44, \sigma = 0.56$). The standard deviation among teachers was the lowest across all of the concepts surveyed, indicating that many instructors seem to agree that this concept in particular is one that causes great difficulty for students.

Further complicating matters is the apparent fact that novice programmers lack the debugging skills needed to successfully narrow down and diagnose problems in their code [GO86]. Even advanced students, who may be highly proficient at programming (that is, specifically writing working code), do not have appreciably better debugging skills than those novice students [AEH05].

The tool that we developed to address these concerns, *Dereferee* [AEPQ09], was inspired by a similar library called *Checkmate* [PWH00]. Checkmate is specifically intended for students who are learning C++ pointers and memory management for the first time, but its implementation is overly restrictive and contains design decisions that severely limit its usability (lack of support for arrays or parameterized constructors, for example). The core idea of instrumenting C++ pointers with minimal interference to a student's development process through templates and macros was sound, however, so we developed Dereferee to address these issues.

Pike et al. did not specifically track the number and types of errors that students were encountering in their own development. Instead, their formal evaluation was carried out by dividing students into two groups and evaluating their ability to find bugs in provided faulty code. One group used Checkmate while the other group used raw C++ pointers. The authors claim that the differences in time-to-completion between the two groups was statistically significant; unfortunately, the data behind this claim is not provided, preventing further insight.

Other researchers have approached memory diagnostics from the compiler side rather than the library side. *Fail-Safe C* [Oiw09] is a memory-safe implementation of the C, rather than C++, programming language. Oiwa achieves this by implementing his own C com-

piler and runtime library in order to provide memory checking in native C pointers, rather than requiring that developers explicitly modify their code to insert such instrumentation. *PAriCheck* [YPC+10] and *MemSafe* [SB12] are two similar approaches to this problem that modify the compiler in some way to track memory usage.

While modifying the compiler is the most transparent approach because it allows students to write "true" C++ code (that is, code without explicit instrumentation directives), it is also a significant amount of work—both in terms of implementing the modifications themselves and ensuring that the modifications can run cleanly on students' computers across multiple platforms. None of these projects are geared toward the beginning user, nor are they amenable to the pursuing collection of data about student errors that we would like to use to study student mastery of these concepts.

2.3 Case Study: Evaluating an Assessment Instrument

Item response theory (IRT) [Bak85] is a statistical paradigm for the design and analysis of testing instruments that measure the abilities of members of a population. In its most basic forms, it can be used to determine the quality of tests given to students based on the difficulty of items on those tests and on the ability of those items to discriminate between students with higher ability levels and those with lower ability levels. Open source content management systems that incorporate testing facilities, such as Moodle, now frequently provide such difficulty and discrimination statistics to their users for multiple choice test items. Sudol and Studer [SS10] provide a short but pointed discussion of how IRT can be applied in the classroom to explore the quality of test items.

More advanced research involving item response theory that has been done in the area of computer science education has been on the development of intelligent tutoring systems and other computer-adaptive testing (CAT) systems that offer personalized learning experiences for their users. The GRE and GMAT exams are two of the most high-profile exams that make use of IRT to adapt the exam content to the ability level of the user as the test progresses [CLC05].

Item response theory has found other applications in computer science education as well, such as to facilitate the accreditation process as discussed in Section 1.4 [WP05] [WP06]. The authors developed a tool that allowed them to routinely collect data from student responses to questions in various courses, and the faculty teaching those courses mapped the questions to specific program objectives as defined by ABET. By using IRT techniques, they were able to evaluate student performance across various objectives and provide faculty with valuable feedback about how student learning may have improved or worsened from semester to semester.

The third case study is based on the fact that, when evaluating programming assignments

using Web-CAT, instructors have the option of verifying the correctness of their students' solutions by executing a suite of "reference" tests against those solutions. Each of these tests has the property that it either passes or fails when executed against a student's solution, so IRT principles can be used to analyze these test items as well—either to assess the parts of an assignment that cause difficulty for students, or to determine the quality of the unit tests themselves. However, after an extensive search it appears that there is no literature that uses IRT to analyze student-written programming assignments in this way.

3
Design, Implementation, and Usage

3.1 Motivation

Web-CAT collects a large amount of data that is of interest to educators wishing to compile reports about their assignments or the performance of their students. These data, however, are stored in a variety of formats and storage mechanisms:

1. Descriptive information stored in a MySQL database includes information about courses (names, offerings, and student enrollments) and assignments (names, due dates, scoring guidelines, and automated grading policies).

2. Statistics computed during the evaluation of a student's submission are written to a Java `grading.properties` file, which is a flat text file of key-value pairs. These statistics include code size (number of lines of code), code coverage information (total number of statements/methods/decisions and number executed), execution time, which instructor reference tests passed or failed, and others. Some of these statistics are known to Web-CAT and are imported directly into the database as well; others might be left in the flat file or pulled into the database using a generic result schema, and instructors can add their own (see Section 3.3 on extensibility).

3. Other result files in various formats (HTML, XML, and proprietary) are generated by tools used during the grading process, such as reports from static analysis tools like Checkstyle and PMD.

Given this disparity in data storage, the traditional approach of generating reports by connecting directly to the Web-CAT database would not be adequate because much of the information of interest is not stored there. I also did not wish to "pollute" the database by

pulling in all of the data of types 2 and 3 above. While Web-CAT does pull information about a submission's score from the `grading.properties` file in to make it quickly accessible for queries to the application's user interface, pulling every possible value into the database would complicate the schema and make extending the system with custom grading plug-ins less approachable for other users.

Since data of interest could come from any of the sources above, it was essential that I provide a uniform method to query data from Web-CAT. The user designing or requesting a report should not have to be concerned about the actual storage location or file format of the data.

Furthermore, since Web-CAT's plug-in-based architecture allows instructors to write their own plug-ins that compute and store any additional data that they choose, the method should make access to this custom data no more difficult than access to built-in information used by Web-CAT itself.

3.2 Choosing a Reporting Engine

In order to expose the data stored on Web-CAT for generating reports, I wished to integrate an existing report generation software package into Web-CAT. The initial set of requirements for the reporting package that I would use were as follows:

- It is open-source software distributed under a license compatible with the license under which Web-CAT is distributed (AGPL).
- It is implemented in or has appropriate interfaces for Java, because Web-CAT is implemented in Java.
- It is extensible to support communicating with proprietary data sources, since the disparity of data storage described above prevents mapping the reporter to commonly used database access architectures such as JDBC.
- It should provide not only advanced content layout and tabulation, but also a large variety of charts and graphs similar to those from Microsoft Excel and other statistics packages, so that they may visualize their data directly without the need to import it into an external application.
- Reports should ideally be renderable in common document formats (such as PDF), but at the very least HTML support should be provided for inline viewing on Web-CAT itself.
- In order to encourage adoption and facilitate ease of use for instructors wishing to use these features, reports should be easy to write—for example by using a WYSIWYG report design tool.

- Even if a report is designed to involve rich tables and charts, it should be possible to extract the raw data from a report so that it can be analyzed externally if desired.

I investigated three report generation packages in my initial design process: JFreeReport (now part of the Pentaho Reporting Project) [Pen12], Jasper Reports [Jas12], and Eclipse Business Intelligence Reporting Tools (BIRT) [BIR12]. JFreeReport was the barest of the three, offering little functionality beyond a simple in-memory table model for data and very basic layout capabilities. It did, however, offer adequate charting (bar, pie, scatter, and line charts, among others) by way of its companion project, JFreeChart, which was already being used in other parts of Web-CAT to display scoring charts to students. At the time of my initial investigation, there was no tool to ease the task of designing reports.

Jasper Reports provided a richer data access model and layout capabilities than the JFree projects, but did not offer any charting of its own; it supports using JFreeChart for this purpose. Jasper Reports also offers a report design tool, but at the time of my investigation its usability left much to be desired.

BIRT (Business Intelligence and Reporting Tools), a project developed for the Eclipse IDE, showed the most promise and is ultimately the one that I chose. BIRT can be used to generate reports either through the WYSIWYG designer implemented in the Eclipse IDE or as a standalone runtime that can be integrated into a larger system. Its charting and layout capabilities are unmatched by the other packages that I examined, even offering interactive charts rendered in SVG format in supported web browsers, and it provides JavaScript-based event handlers to allow users to customize much of the report generation and rendering process at runtime.

Furthermore, prior experience in developing for the Eclipse IDE and its runtime architecture significantly reduced the learning curve that would be required to adopt BIRT for use in the system. BIRT uses a plug-in-based architecture to support various kinds of "data sources" and "data sets", such as direct SQL database access or file-based data sources. Even though Web-CAT stores data in each of these formats, any one of these data sources on its own would not be sufficient.

Instead, I implemented a new plug-in that acts as a "bridge" between BIRT and Web-CAT. Web-CAT is built on top of Apple WebObjects, a Java-based web application framework. WebObjects provides a fairly sophisticated object-relational mapping, so each table in Web-CAT's database is shadowed by a Java class with accessor and mutator methods corresponding to the columns in the table. These classes can also be extended with methods that provide dynamically computed properties as well.

The object graph inside Web-CAT can be navigated using a "key-value coding" notation, which uses dot-separated key paths to evaluate properties, similar to how JavaBeans properties are denoted. For example, `submission.user.userName` would call the `getUser()` method on the `submission` variable, and then the `getUserName()` method is called on that result. By adopting this notation when defining reports, I simplified the way that data are

captured and reap the benefits from the WebObjects model. This unifies the data in disparate storage locations and varying formats because a key path such as the one above can resolve to either a database-backed property or to a dynamic method that pulls data from an external file or aggregates several fields together, thus the end-user does not need to be aware of the true format of the data.

3.3 Extensibility

Another major advantage of the system's design is the considerable extensibility attained by using Web-CAT's existing object model. The plug-in-based architecture used by the Web-CAT grading subsystem allows instructors to configure assignments so that any arbitrary scripts are executed on the files that students submit. These scripts can compute any statistics or generate any data for which the instructor has appropriate tool support. For example, if the code coverage and LOC metrics are insufficient for a study, then an instructor can write a small script to execute a tool that computes additional metrics and add that script to the plug-in chain used for grading.

Typically, any additional data generated by plug-ins would be written to the properties file for that submission. These properties would then be available to reports using the same dotted key path notation that is used to access built-in values. For example, if a plug-in writes a value to the properties file with the key `foo`, then it could be extracted in a report using the key path `result.properties.foo` (assuming that the receiver of this key path expression is a student's submission).

Furthermore, by adding the prefix `saved` (followed by a period) to the property name, Web-CAT will import the value of that property into the database using a generic `ResultOutcome` type. Properties that are saved by one execution of the plug-in on a student's submission are then automatically made available to the next execution of the plug-in, so that scripts can easily make use of historical data to track student behavior and results as they work on the assignment. Similarly, importing such properties into the database allows a user designing a report to write queries based on this fine-grained data instead of merely querying on submissions themselves.

3.4 Definitions

Before I present a walkthrough of the process of designing and executing a report, I define some terms that BIRT and Web-CAT use in reporting.

The **data source** acts like a "database connection" and manages one or more data sets (described below). BIRT provides built-in support for a number of types of data sources,

such as connections to databases, web services, and flat files. When I integrated BIRT into Web-CAT, I created a custom "Web-CAT Data Source" specifically for connecting to Web-CAT in order to unify access to the data that are stored in various formats and locations on the system.

Another type of data source built in to BIRT, the "Scripted Data Source," is used to manage "Scripted Data Sets" that allow for advanced transformations of data retrieved from other sources, without requiring that I re-implement those transformations in my own Web-CAT data bridge. This powerful feature of BIRT allows the report designer to write arbitrary Javascript code that post-processes the data retrieved from Web-CAT and this processed data can be used instead of the original data to generate tables and charts.

Typically, a report will only have one Web-CAT Data Source, and possibly a Scripted Data Source if more transformative power is desired. Neither type of data source requires any configuration. Unlike a traditional JDBC connection that would include the URL of the database to access, the Web-CAT Data Source simply uses data from the instance of the Web-CAT application to which the report is eventually uploaded.

A **data set** represents a single query to one of the above data sources and its associated results when the query is complete. Just as I extended BIRT with a Web-CAT Data Source, I did the same with a Web-CAT Data Set, which is described with a schema that includes the type of objects from which the data will be extracted, the data values that will be pulled from those objects, and the data types for those values. In a particular Web-CAT Data Set, each row represents a single object in Web-CAT, and each column along that row is the value of an attribute reachable from that object via a key path.

The type of object corresponds directly to a data model object (and a database table) on Web-CAT. The most common object used for generating reports is `Submission`, which represents a single submission that a student has made to an assignment. Other objects that are useful as rows in a data set are `ResultOutcome` objects, which represent a more fine-grained result of a submission (such as a single JUnit test result), or `SurveyResponse` objects, which contain statistical and free-response feedback from students who completed an optional survey about an assignment.

A **report element** is a visual element in a report template that contains data extracted from a data set. BIRT provides report elements such as tables, a wide variety of charts and plots, and other basic data fields that can be included in a report.

3.5 Creating Report Templates in the BIRT Designer

This section provides a brief walkthrough of the process of designing a new report template in the Eclipse BIRT WYSIWYG report designer.

Eclipse is a project-oriented development environment, meaning that any document that it manages must belong to a specific project structure. Therefore, the first step to creating a report template is to create a new project (of any type, such as "General > Project") to hold the report templates. Multiple report templates can be located in the same project, or the user can choose to organize them into multiple projects as he or she sees fit.

3.5.1 Creating a New Report Template

Once a project exists, a new report template can be created. To do so, right-click on the project, choose the "New > Other..." action, and then select "BIRT > Report Design[1]." The next step of the wizard will ask for the filename of the report; it should end with the extension `.rptdesign`.

Once the report is created, it will be opened in a tabbed layout similar to that shown in Figure 3.1. The first tab is the "Overview" tab, which is added by the Web-CAT extensions to BIRT. This view allows the user to edit extended metadata that Web-CAT uses to manage the report, such as the title and brief description that will be shown in the Web-CAT report template library that is described in a later section. The "Layout" tab shows the WYSIWYG view of the report template and allows the user to drag and drop tables, charts, and other report elements into the template. Lastly, the "Outline" view on the far left presents a categorical tree view of all the data sources, data sets, report elements, and scripts used in the template.

3.5.2 Defining a Web-CAT Data Set

In order to generate a report containing data from Web-CAT, the user must define a data source and data set that describes the nature of the data to be extracted. First, right-click the "Data Sources" folder in the "Outline" view and choose "New Data Source..." In the wizard that appears, indicate that you want to create a "Web-CAT Data Source" and give it a name (such as "Web-CAT", since there will only be one in the report). Then click "Next" until the wizard is dismissed.

Now that the report template contains a data source connected to Web-CAT, a data set must be created to represent a result set—the properties to be extracted from a batch of objects in the data model. Unlike the data source, a report template can contain multiple data sets. To define a new data set, right-click the "Data Sets" folder in the "Outline" view and choose "New Data Set..." In the wizard that appears (Figure 3.2), select the data source

[1] Due to the history of the development of the Web-CAT reporting engine, what I refer to as a "report template" is called a "report design" by BIRT, and the term "template" refers to a different concept. In this document I continue to use the term "report template" except in cases where I must refer to "report designs" specifically in the BIRT user interface.

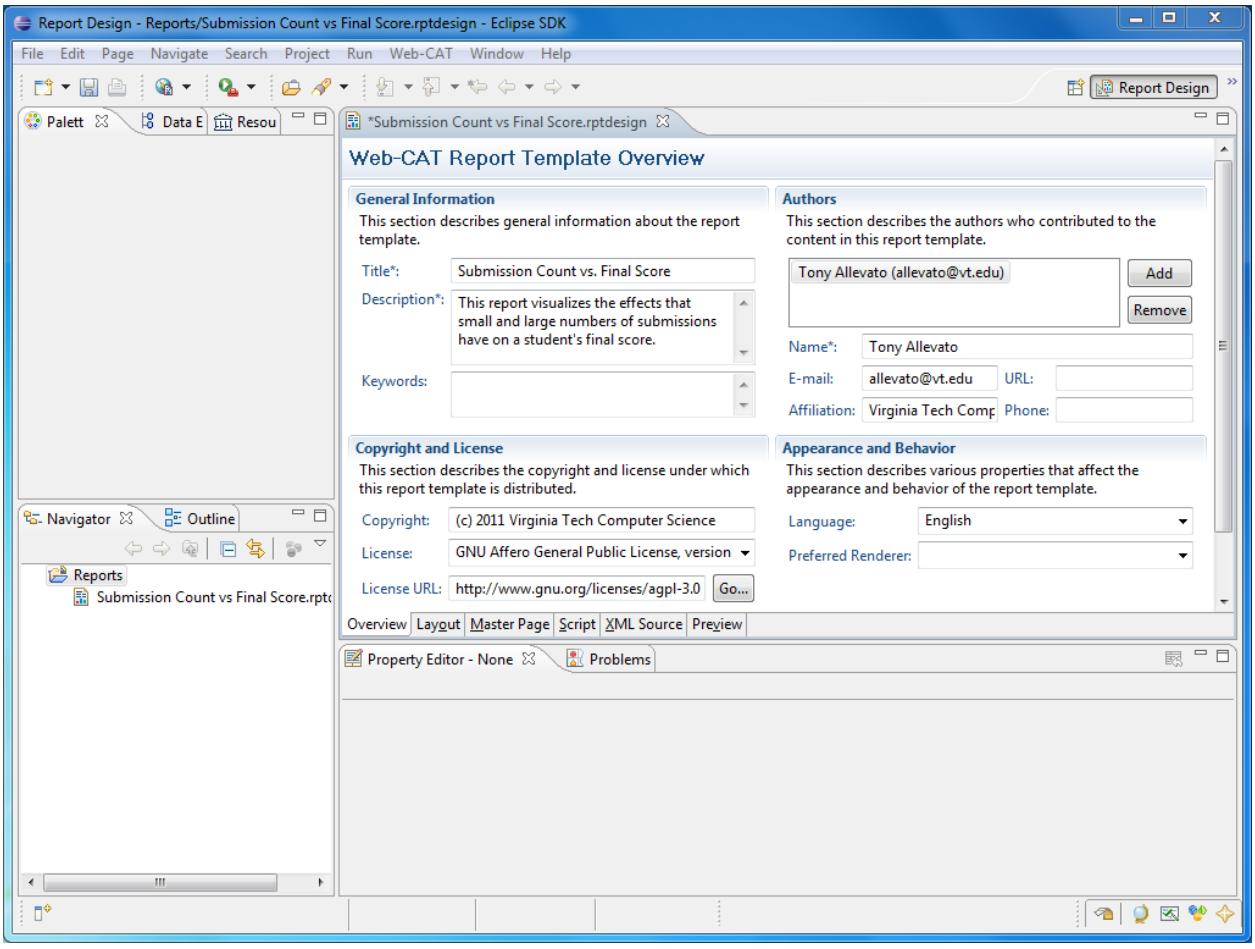

Figure 3.1: A newly created report template showing the report metadata editor.

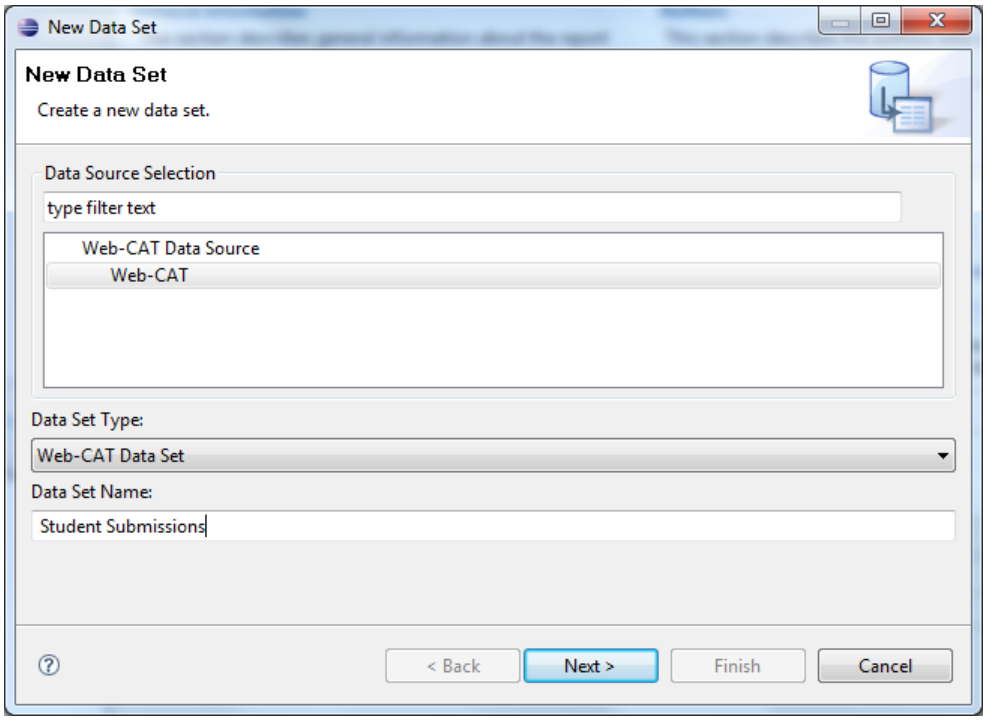

Figure 3.2: Creating a new Web-CAT data set.

named "Web-CAT" that was created in the previous step. Under "Data Set Type," select that you want to create a "Web-CAT Data Set" and give it a meaningful name. In this example, the data set will process student submissions to an assignment, so I named it "Student Submissions."

Clicking "Next" will present the data set schema editor (Figure 3.3). The "Entity type" at the top of the editor indicates that kind of entity, or object, in the Web-CAT data model that should be included in the data set. Web-CAT contains a wide variety of objects in its data model that describe details about courses, assignments, users, and work submitted to the system. Since most educators are interested in information derived from submitted work, the most common entity to use here is `Submission`, which represents a single submission attempt by a student in an assignment.

Each row in the table below the entity type defines a column in the result set. In this sense, the model is very SQL-like; the column expressions are similar to column names in an SQL `SELECT` statement. The "Column Name" field is a report-internal name that is used to refer to the column inside tables, charts, and scripts. The "Expression" field indicates how to navigate the Web-CAT object graph, starting from the desired entity type, in order to get to the desired value.

The simplest kind of expression is a dot-delimited key path, as described previously. For

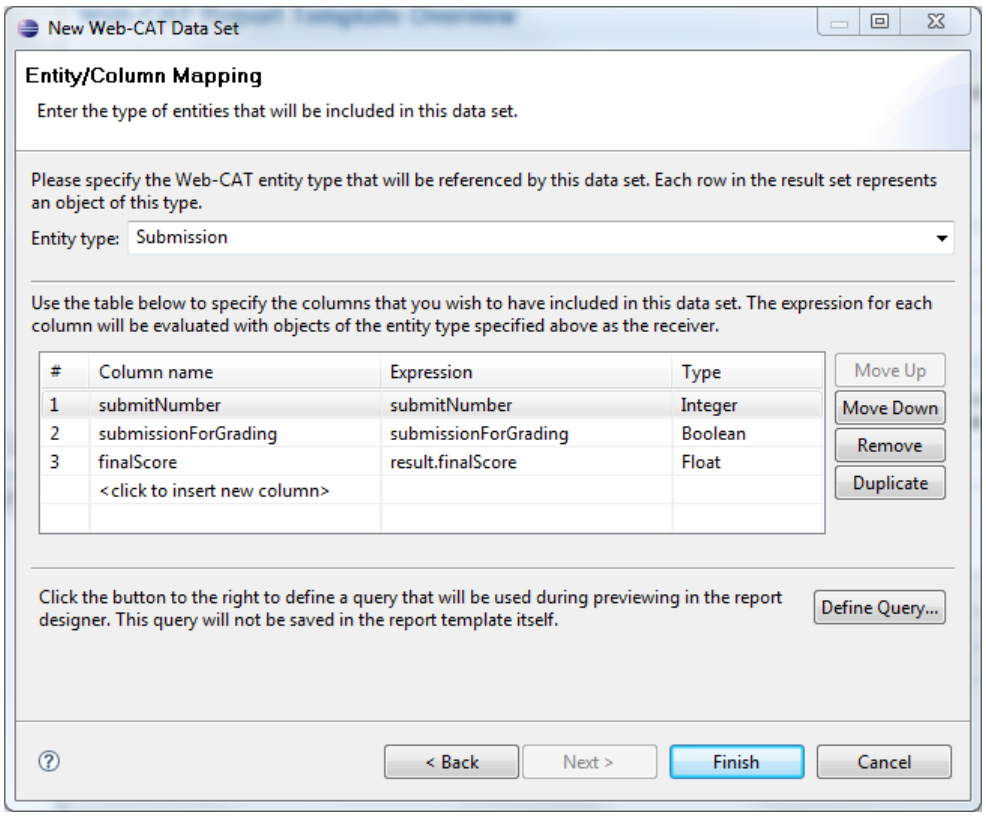

Figure 3.3: Defining the schema for a Web-CAT data set.

example, if the entity type for a data set is `Submission`, then the `submitNumber` key path would query that property on each `Submission` object to generate that column in the result set. Similarly, the expression `result.finalScore` would query the `result` property, which returns another object (of type `SubmissionResult`), and then subsequently query the `finalScore` property of that object. The data type of the expression's result is indicated in the "Type" field. The standard set of data types—strings, integers, floating-point numbers, Boolean values, and timestamps—are supported.

The expressions supported by the Web-CAT reporting engine are more powerful than simple key paths, however. I also provided support OGNL (Object Graph Navigation Language) [OGN12], which is a superset of dotted key path notation that adds list selection, projection ("map" in many functional languages), and pseudo-lambda expressions (they are not true lambda expressions because OGNL lacks closures—all variables have global scope). OGNL's selection and projection features are most useful for computing summary statistics about a set of objects that are reachable from the primary objects in a query result. For example, I have used the following expression in report templates that I have written to compute the total number of lines of code, excluding test cases, in a student's submission:

```
#ENV.accumulate(
  result.submissionFileStats
    .{? #this.className != null && !#this.className.contains("Test")}
    .{#this.loc})
```

The special `#ENV` variable is part of the custom Web-CAT data bridge and provides convenience functions that OGNL does not natively provide. In this case, `accumulate` is the summation operation. The `result.submissionFileStats` property returns an array of `SubmissionFileStats` objects; each object is a parsed and processed source file that was part of the student's submission. After that, the `.{?` *`expr`* `}` operator is a selection operation that includes only the elements from the array that have a class name that does not contain the string `Test`; this is a crude way (and imperfect, but sufficient for our purposes) of determining whether the class is a test case or not. The special variable `#this` allows us to access each element in the pseudo-closure. Finally, the `.{` *`expr`* `}` operator is the projection operation that returns the `loc` (lines of code) value for each of the `SubmissionFileStats` that was selected previously. Finally, the top-level `#ENV.accumulate` function sums the results and returns a single integer representing the number of lines of code in all non-test classes found in the student's submission.

OGNL expressions such as these can become unwieldy in complex situations, however. In situations where the derived value is not directly obtainable through a named property, these expressions provide a great deal of power to extract a wide variety of data from the system. The expression editor in the Web-CAT Data Set wizard provides some basic assistance with constructing these and similar aggregating expressions.

When generating a report (see Section 3.6.2), the user will be asked to build a query for each data set that determines which objects on Web-CAT will be used to generate the result set. In other words, the data set is a schema that is encoded into the report template itself. The query that determines which objects are used to generate the result set, however, is not part of the report. This design decision allows the same report template to be used for a wide variety of queries. For example, a single report template could be used to generate reports that include submissions from a single student across multiple semesters or submissions from all students on all assignments in a single course. It is then the responsibility of the report template to perform any grouping that is necessary on the objects being used as source data for the report, since there is no guarantee that any grouping occurred on the Web-CAT server.

3.5.3 Adding Report Elements

Once the data set has been defined, visual report elements can be added to the template. Adding tables to a report is easy; simply drag a data set onto the page in the "Layout" tab and BIRT will automatically create a table with the appropriate columns for that data set. Then, any desired formatting and adjustments can be applied.

Adding charts to a report template requires slightly more effort. With the "Layout" tab still selected, find the "Chart" item in the "Report Items" section of the "Palette" view on the far left and drag it to the desired location on the page. This will cause the New Chart wizard to be displayed, as shown in Figure 3.4. BIRT supports many of the standard charts and plots that users of software such as Microsoft Excel are used to seeing—bar charts, line charts, pie charts, scatter plots, and many others. On this first page of the wizard, select the type of chart that you wish to add.

Clicking "Next" will move to the "Select Data" page (Figure 3.5), which lets you specify the data set that should be used to generate the chart in drop-down list next to the "Use Data from" control. Once a data set is selected, the "Data Preview" table is populated with a small sample of data from the data set. Columns from this data set can be mapped to series in the chart by dragging the header of one of the columns up to the desired series. More complex expressions can be written for these series as well; however, these are Javascript expressions evaluated by the BIRT runtime and not OGNL expressions executed on the Web-CAT data store.

Clicking "Next" again will move to the "Format Chart" page (Figure 3.6), where you can control the appearance of the chart. The options here are similar to those found in may other charting applications: axis titles, tick marks, and labels; markers and line styles for data displayed on the chart; control over fonts and colors for various chart elements; and a legend.

When the chart has been completely configured, click "Finish" to return to the report de-

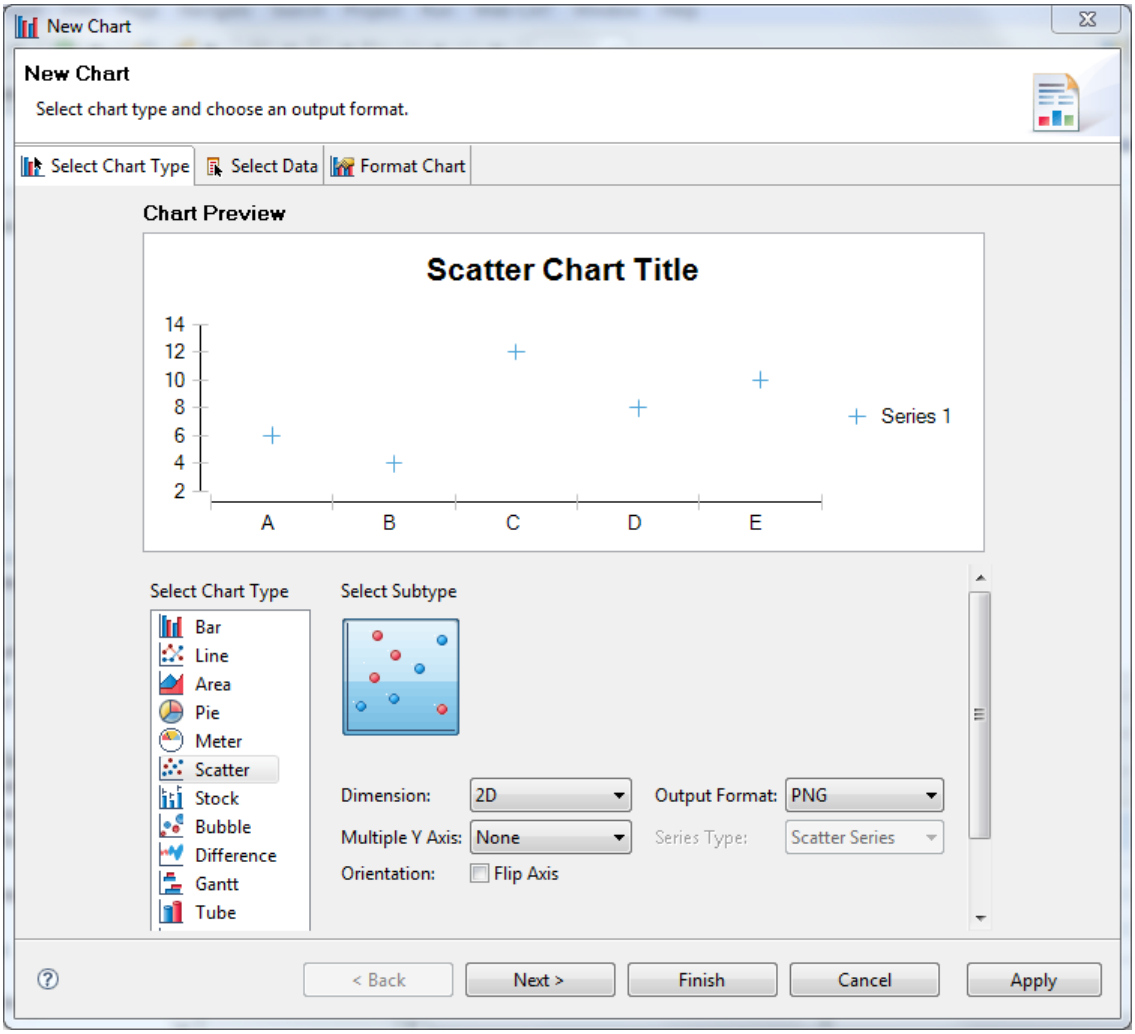

Figure 3.4: Creating a new scatter chart in a report template.

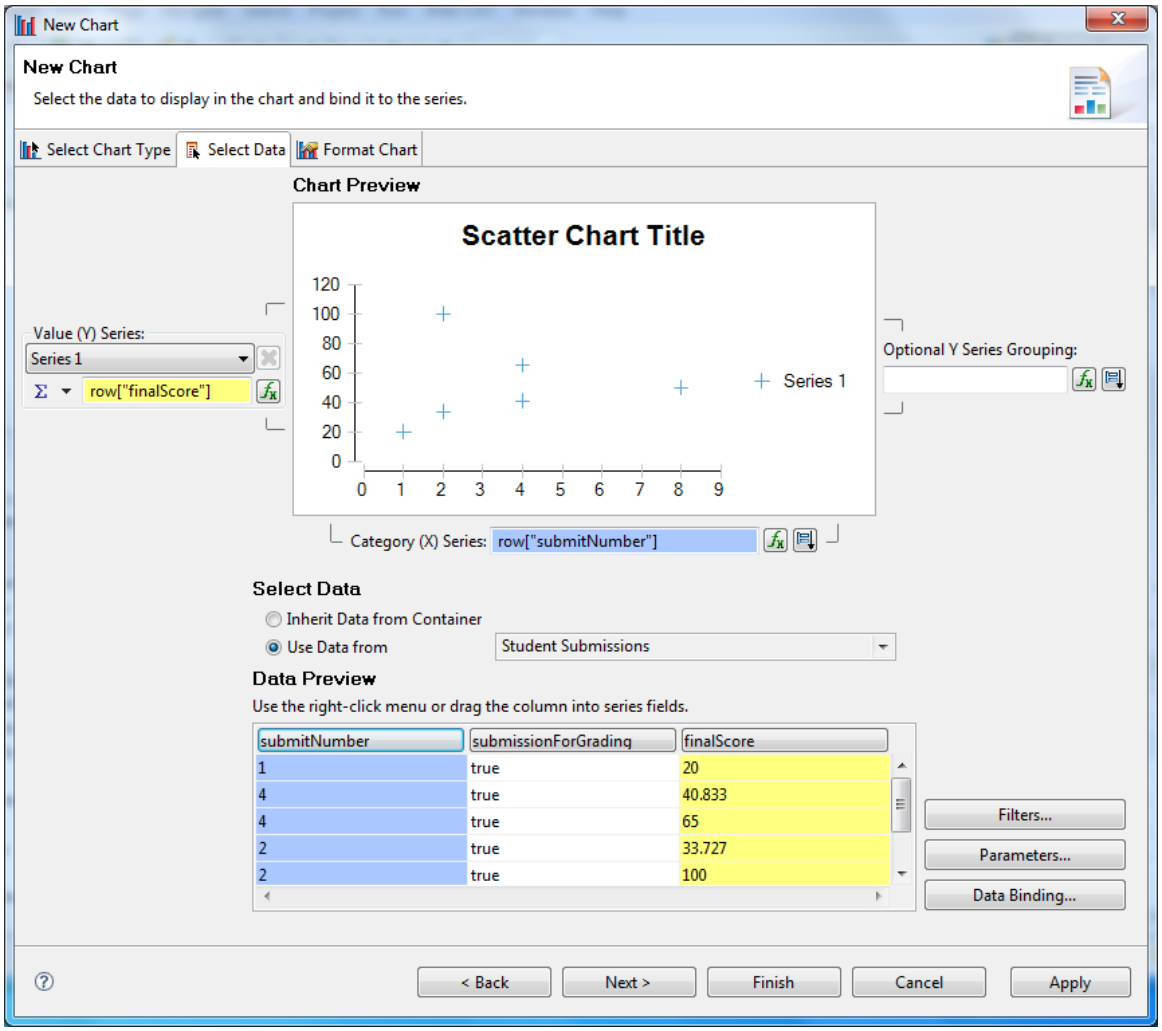

Figure 3.5: Specifying the data that will be used to render the chart.

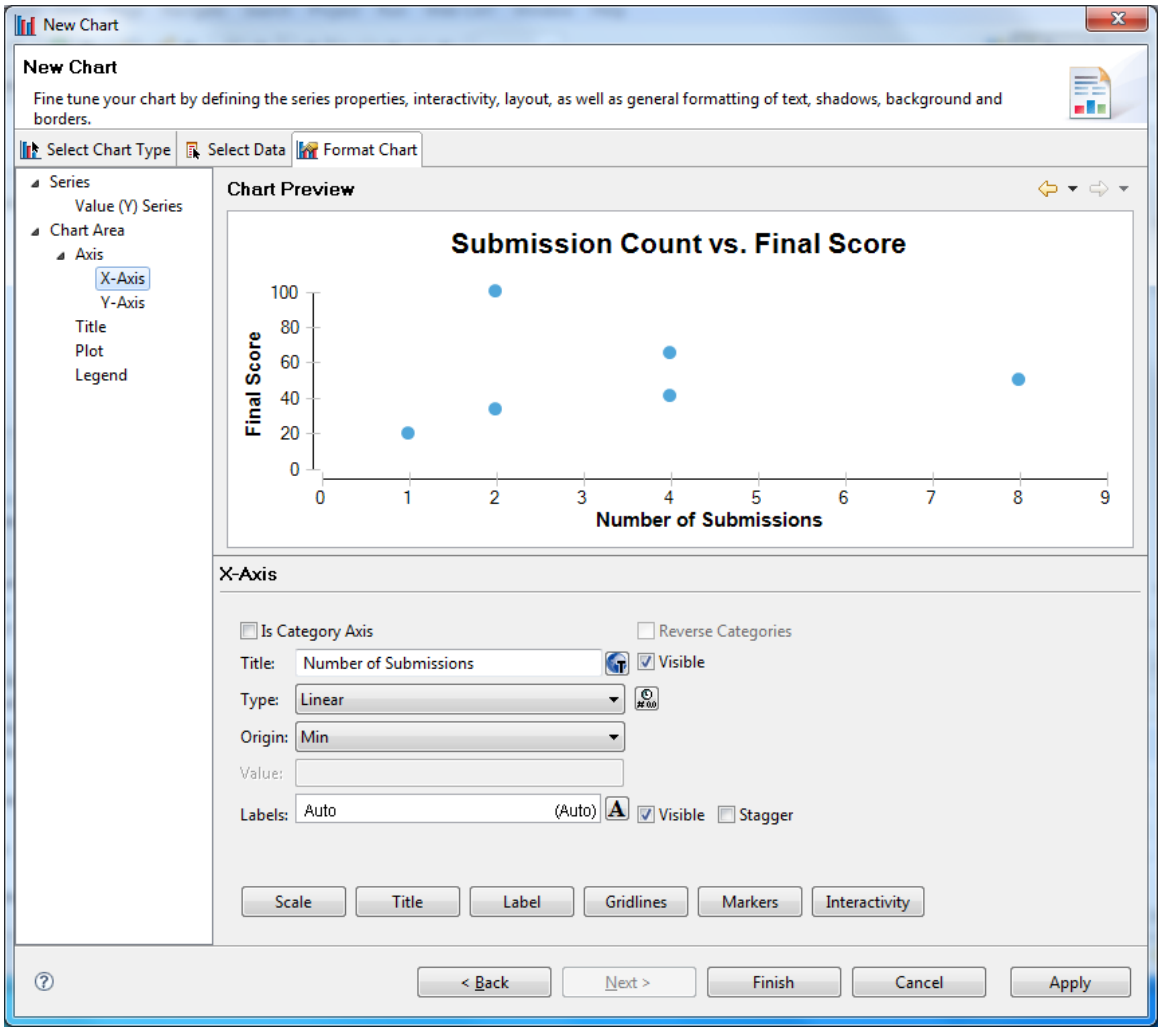

Figure 3.6: Applying rich formatting to the chart.

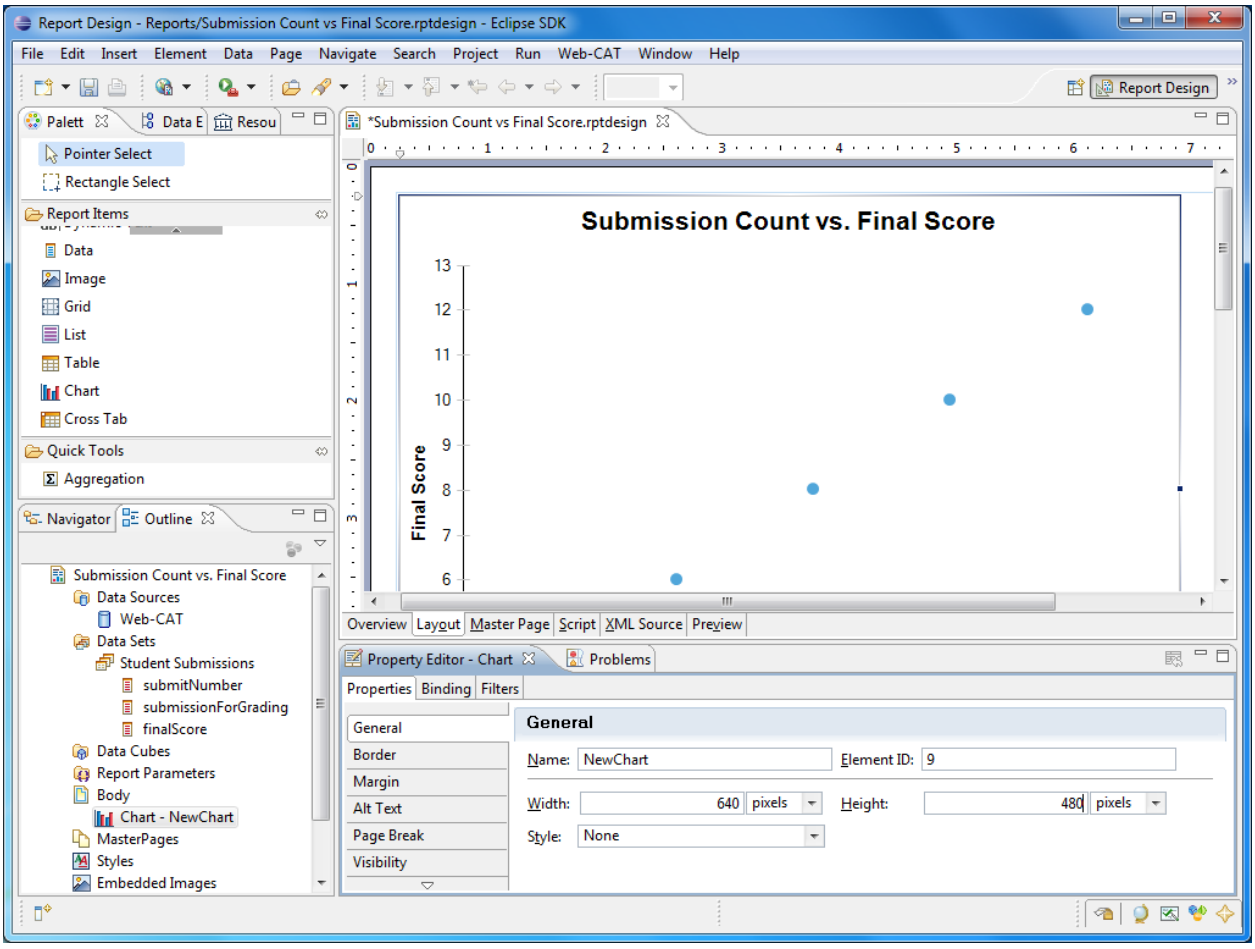

Figure 3.7: A completed report template in the BIRT editor.

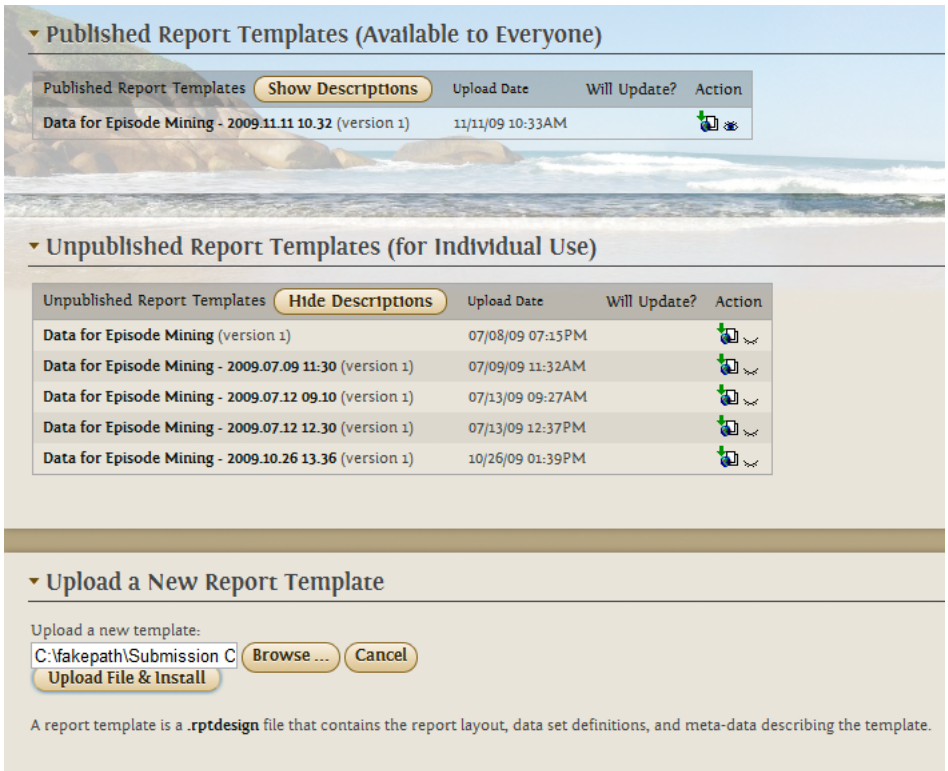

Figure 3.8: Using the report template library to manage available templates.

signer. A preview of the chart will be displayed on the page in the "Layout" tab (Figure 3.7). This report template is now complete and can now be saved in order to use it to generate reports based on live data from Web-CAT.

3.6 Using Reports on Web-CAT

3.6.1 Uploading and Managing Reports

Web-CAT provides a "report template library" (Figure 3.8) that lets instructors manage the templates that are available for data extraction in a similar fashion to the library that manages grading plug-ins. Once an instructor has created a report template by following the steps in Section 3.5, he or she can access this library by choosing the "Reports > Templates" menu on Web-CAT and upload the `.rptdesign` file to the server.

When a new report template is uploaded to Web-CAT, it initially appears in the section labeled "Unpublished Report Templates (for Individual Use)." This area contains private report templates that only the user who uploaded them (and the system administrator) can

Figure 3.9: Choosing the report template to generate.

see and run. If instructors wish to make one of their report templates available for anyone using this Web-CAT instance, then they can use the show/hide action to move that report into the "Published Report Templates (Available to Everyone)" area.

3.6.2 Generating a Report

Once the desired report template is uploaded, the user can begin using it to generate reports. If he or she chooses the "Reports > Generate" menu on Web-CAT, the screen shown in Figure 3.9 will appear. This screen lists all of the report templates that are available to the current user; that is, his or her own personal templates and any that have been made public by someone else. To generate one of the reports in the list, the user should click its title. This will present a second screen that asks him or her to define the query or queries that will be executed to determine the data that will go into the report. Each Web-CAT data set defined in the report template is mapped to a query that the user must build when generating the report.

When I designed the reporter, I wanted to make it approachable for Web-CAT's end users (instructors who may not be deeply familiar with Web-CAT's data model) but still be advanced enough to support my own studies and studies by other members of the Web-CAT team who know the system much more thoroughly. I achieved this by providing extensible "query assistants" that let the end user define a query in a variety of ways.

Web-CAT currently provides two query assistants. The first is an interface that supports common usage scenarios for most instructors, and is shown in Figure 3.10. This assistant displays a list of semesters and courses in the left pane and a list of assignments in the right pane. The instructor builds the query by checking off the courses and assignments that contain the data that he or she wishes to see. Multiple courses and assignments can be selected to generate broad reports if needed. Some miscellaneous options below those panes allow the instructor to choose whether data about *all* submissions by a student on

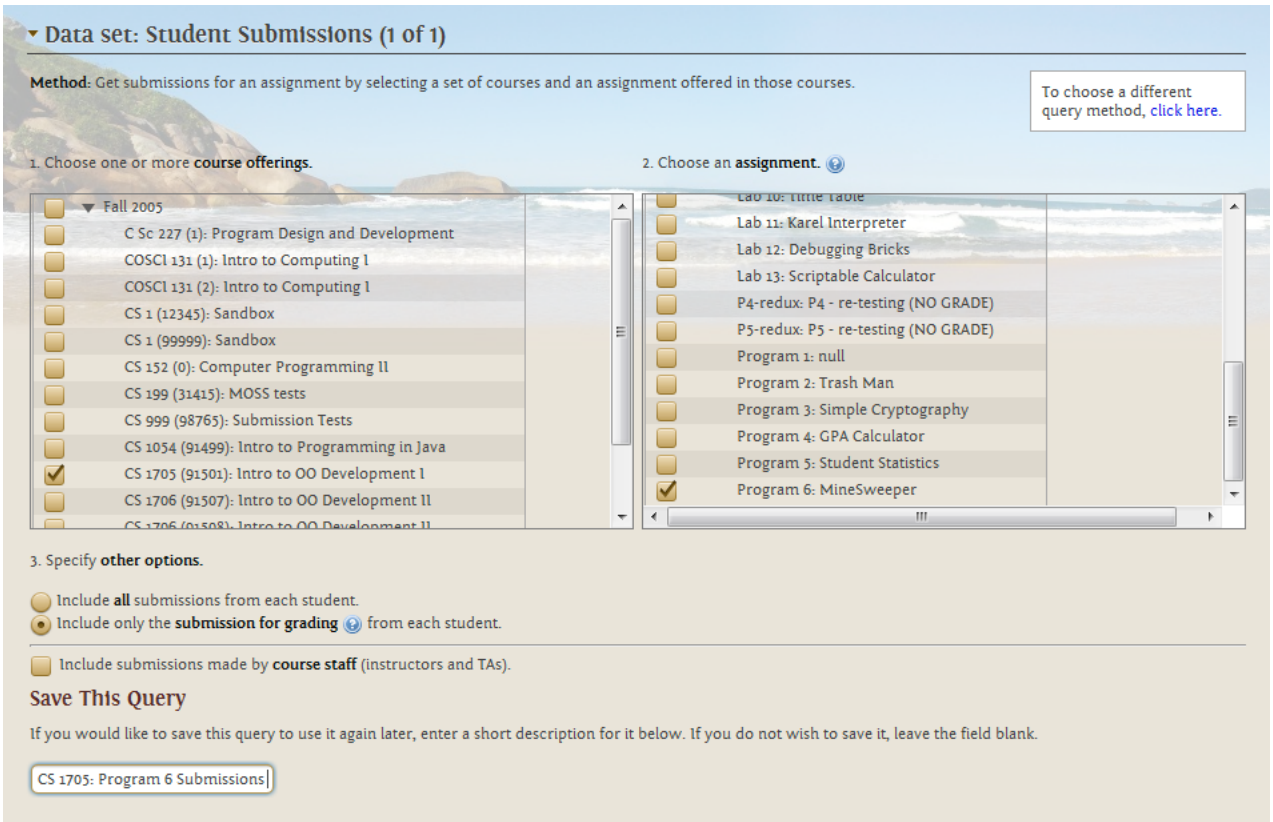

Figure 3.10: Choosing the semesters, courses, and assignments from which the report should include submissions.

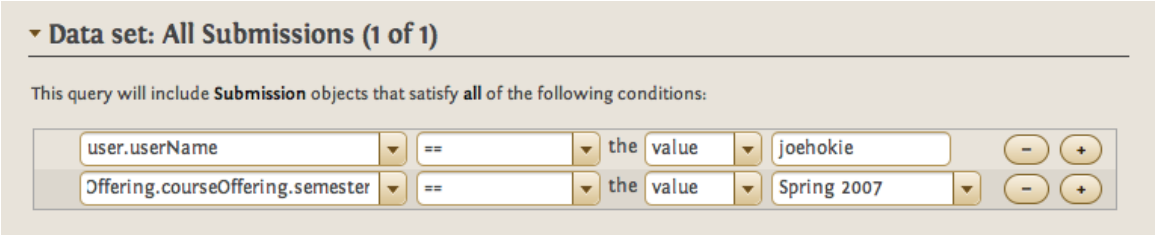

Figure 3.11: Specifying an advanced query as the conjunction of multiple criteria, using direct expressions on the Web-CAT object model.

each assignment should be included or if only the *submission for grading* (typically the final submission, but whichever submission is being used to count toward the student's overall course score) should be included in the result set.

A second query assistant is an "advanced" assistant that lets the user write the query as the conjunction of a set of filtering criteria, or *qualifiers*. These qualifiers use the same key path notation to access the Web-CAT data model that is used when defining data sets in the BIRT report designer. This allows much more complex queries to be written that perform filtering on criteria other than just course and assignment. With this power comes somewhat increased difficulty in usage, but to mitigate this the query assistant provides type-aware features found in many modern development environments—key paths are auto-completed as the user types them in, and the operators and filtering values used to build the query are intelligently determined based on the type of data represented by the key path. For example, an instructor who wants to collect data about all submissions made by a particular user in any of their courses that semester could specify the criteria shown in Figure 3.11.

In this scenario, the `==` operator is chosen from a drop-down menu that lists only the valid operators for the `user.userName` property. The username would be entered in a plain text field but the `Spring 2012` semester would be chosen from another list containing only the valid semesters found on that Web-CAT instance, relieving the user of the need to worry about the internal representation of those entities.

As a final note, if the instructor anticipates reusing this query to generate future reports, he or she can give it a descriptive name so that it can be selected quickly in the future instead of being rebuilt from scratch.

Once the query is built, the user clicks the "Generate Report" button and waits for the data to be retrieved, processed, and rendered. A progress bar displays the current status of the report as it is being generated. The time required to generate one of these reports will vary depending on the size of the result set and the nature of the data that are being collected. If the result set is small (a single assignment, for example) and all of the desired data are stored in the database, then the process may only take a few seconds on modern hardware. If the result set spans multiple assignments and/or courses, or contains data that must be

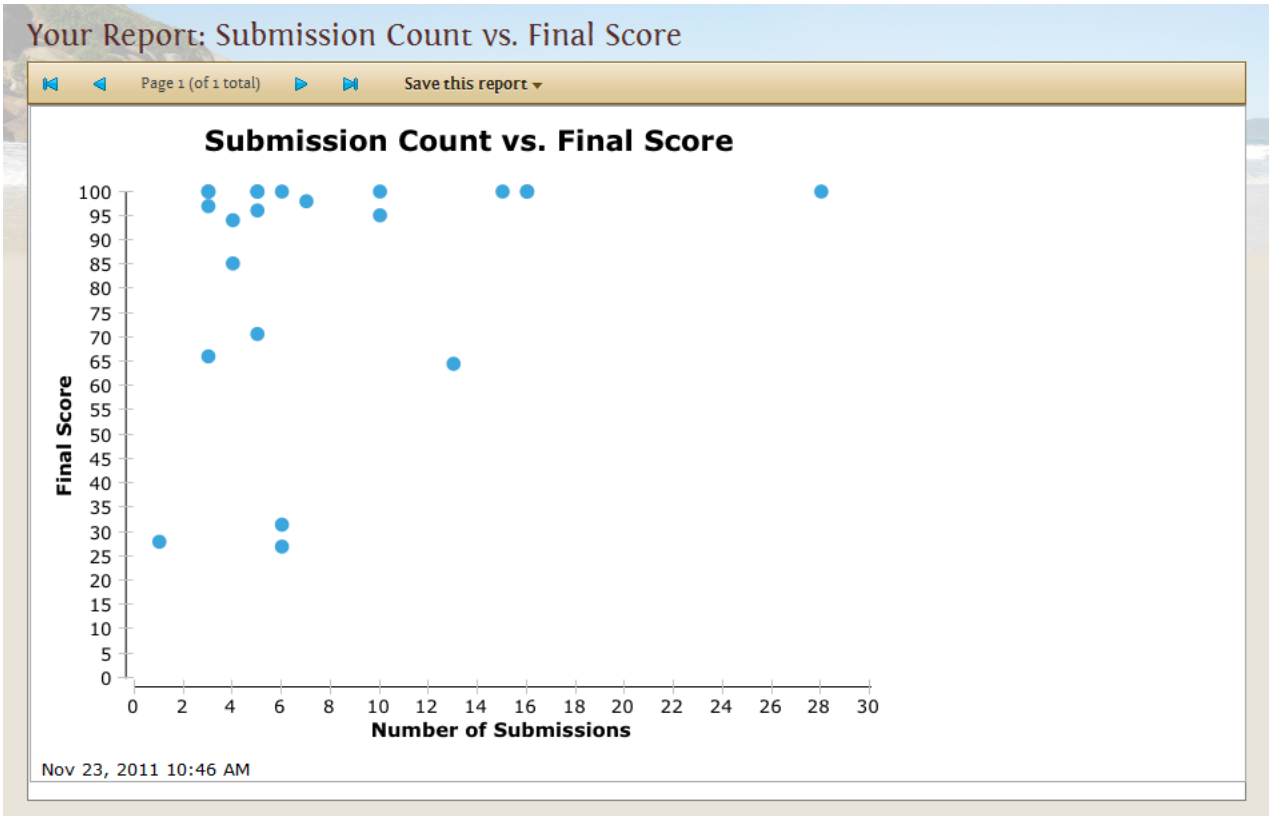

Figure 3.12: Viewing the final generated report.

loaded from the file system, then more time may be needed to generate the report. For this reason, the reporter does not lock the user in to wait for the report to become ready. The user is free to navigate to other parts of the Web-CAT system and perform other tasks, or even close their browser window entirely, and return later (see Section 3.6.3) to view the completed report. An e-mail notification is also sent to the user when the report is ready.

Once the report has been successfully generated, it will be displayed as illustrated in Figure 3.12. The report in this example consists of one chart that plots each student's final score against the number of submissions that he or she made to an assignment. For reasons of efficiency, reports containing long tables or multiple charts will be paginated. The navigation controls in the toolbar above the report allow the user to view it a page at a time.

Also in the toolbar above the rendered report is the "Save this report" button that allows the user to save the report or its data in a more permanent format. The fully rendered report—including tables, charts, and other rich formatting—can be saved as a PDF file so that it can viewed offline or shared with others. The raw data from the report can also be extracted in CSV format. This gives instructors the ability to further analyze the data in any report, perhaps in a third-party statistical software package.

3.6.3 Viewing Old Reports

Once a report has been generated, Web-CAT preserves the results in their original state so that they can be viewed and the data re-extracted at a later time. To do so, an instructor chooses the "Reports > View" option from the navigation menu. From that screen, they can view any previous reports that they have generated as well as delete ones that they no longer wish to keep. Users should keep in mind that the old reports represent a *snapshot* of the requested data *at the time that the report was generated.* If a report was generated using data from an assignment that was in-progress, and more submissions have been made since the time that the report was generated, viewing the old report will *not* update it with the new data. To refresh the report with new data, the user will need to generate it again through the "Reports > Generate" page.

4
Case Study: Evaluating Student Behavior

Every semester, computer science educators encounter students who may be highly interested in computing and capable of doing the work but who still perform poorly in the course. Educators frequently cite poor time management skills as one of the leading causes of lower scores on programming assignments; students either don't start early enough or don't give themselves enough time to complete the work. This case study explores the time management issue, by mining data about student submission patterns from Web-CAT.

4.1 Data Used in This Case Study

The data set used in this case study consists of work from three CS courses that made up the introductory programming and software design sequence in past years at Virginia Tech: CS 1705, CS 1706, and CS 2605. Ten semesters worth of work, from Spring 2004 through Fall 2008, were used. Summer semesters were omitted because the accelerated nature of those courses often requires other changes to the pacing of the course and the structure of assignments.

Across these ten semesters and three courses, Web-CAT has collected work from 1,011 individual students on 105 programming assignments. These assignments only include individual programming projects; in-lab assignments have been excluded because they are short, targeted two-hour assignments, and because they typically involve pair programming and I wish to analyze only individual behaviors.

The 1,011 students in this data set made a total of 89,879 submissions to Web-CAT, 6,001 of which were "scoring submissions." In the introductory programming courses at Virginia

Tech, students are typically permitted to make an unlimited number of submissions to Web-CAT before the due date (and possibly after the due date, with an appropriate penalty levied). Each submission provides the student with useful feedback that allows her to refine her solution, but only one submission—usually the final one—will be used to assign her a score.

This case study explores the following questions:

- When do students who perform well on programming assignments start working compared to students who perform poorly?
- Similarly, when do students who perform well finish vs. those who perform poorly?
- Do those two groups of students spend a different amount of time on the assignment overall?

First, I must codify what it means to perform "well" on an assignment and what it means to perform "poorly." In this study, I use a partition point of 80% on the score of a submission to separate these two groups. This aligns nicely with the traditional US letter grade scale, where a score in the A/B range (above 80%) can be considered performing "well" and a score in the C/D/F range (below 80%) can be considered performing "poorly."

I do note that the categorization above does *not* align perfectly with the grade requirements necessary to advance to the next course in computer science at Virginia Tech. In our program, a student must can move on to the next course if he or she achieves a C or better. However, students who consistently turn in C-level work are certainly at risk and would be considered by their instructors to have a questionable mastery of the subject matter, so I have grouped the C-level students into the "poorly" performing group rather than the group performing "well."

For the remainder of this discussion, I will refer to these two groups as the A/B group and the C/D/F group. Note that this is a partitioning of *submissions*, not of *students*. It is certainly possible for a student to have turned in work that falls into both groups at some point during their time in our program.

Table 4.1 provides statistics about the scoring submissions in the data set after partitioning them into the A/B and C/D/F groups. Again, these statistics are computed over *submissions*, so the same student might fall into both groups.

4.2 Behavior vs. Innate Ability

One concern that I must address in this study is the fact that some students will consistently perform well or consistently perform poorly for factors other than those that I am exploring.

Statistic	A/B Group	C/D/F Group
# Submissions	3989	2012
# Students	869	785
Maximum	100.00	79.95
75% quartile	99.00	71.69
Median	95.03	57.01
25% quartile	90.00	35.00
Minimum	80.00	0.00
Mean	94.03	50.75
Std. Dev.	6.02	24.17

Table 4.1: Population statistics for the entire data set, partitioned into A/B and C/D/F groups.

An argument could easily be made that extremely talented students might succeed on an assignment no matter when they start working on it, or that there are students who simply do not "get" programming and will struggle no matter how much time they spend trying to solve the problem. If I look at differences between better- and worse-performing students and the correlations between behaviors and outcomes, how can I be assured that the differences are due to changes in those behaviors and not some other innate quality of the student?

In order to address this, I identified students who made submissions that only fell into the A/B group and students who made submissions that only fell into the C/D/F group. These students were then removed from the data set, leaving only those who had at least one submission in the A/B group and one submission in the C/D/F group. Figure 4.1 shows the breakdown of submissions based on which group the submitting student fell into. Doing this allowed me to perform within-subjects comparisons, giving me more confidence that differences in performance were *caused* by differences in behavior that were found between those groups.

Table 4.2 shows the statistics for the reduced data set after the consistent A/B and C/D/F performers were removed. Overall, 623 students making 3,437 scoring submissions remained after this reduction.

4.3 Results

The results presented here are a refinement of results that were published in [ESA+09]. The original paper examined a number of student behaviors—number of submissions, times of submissions, and amount of code written. For the majority of this case study, I will focus solely on the time-oriented behaviors and the associated effects on performance. I will

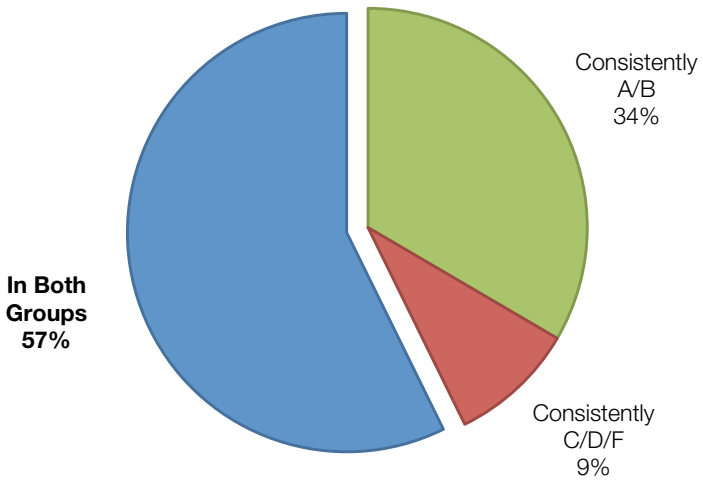

Figure 4.1: Student submissions categorized by the group of the student making the submission.

Statistic	A/B Group	C/D/F Group
# Submissions	1983	1454
# Students	623	623
Maximum	100.00	79.95
75% quartile	98.00	72.92
Median	94.00	60.81
25% quartile	88.95	40.00
Minimum	80.00	0.00
Mean	93.17	53.64
Std. Dev.	6.18	23.09

Table 4.2: Population statistics for the data set after removing students who fell consistently in the A/B group and those consistently in the C/D/F group.

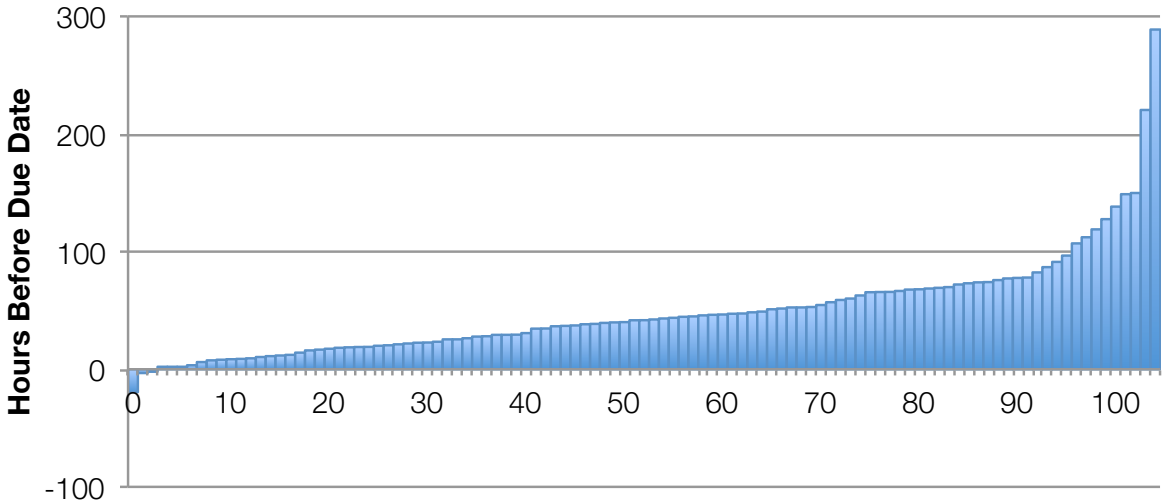

Figure 4.2: Mean time of first submissions on each assignment. Each bar represents one assignment.

examine the amount of code as it relates to difficulty and effort for an assignment later when I discuss interventions to improve time management.

4.3.1 Time of First Submission

Figure 4.2 shows the mean times of the first submissions to each assignment, in hours before the due date. These means range from 288.8 hours (just over 12 days) on the high end, to −19.4 hours on the low end. Negative numbers are possible because at Virginia Tech, many courses allow students to submit an assignment for a limited time after the due date, with an associated penalty.

Figure 4.3 also shows mean first submission times, but grouped by whether the submission was in the A/B group or the C/D/F group. One can see that in most cases, the mean over the C/D/F group indicates that these students began later than the students in the A/B group.

One important note about these measures is that I do not have information about precisely when a student *starts* working on the assignment. These measures are the best estimation of that based on the time of his or her first voluntary submission to Web-CAT. Other studies that have been done with this data set indicate that by the time students make their initial submissions to be graded, they have already written a significant amount of their solution code. The start times reported here are therefore a lower bound on the time that students began work. However, my findings using "first submission time" as an approximation of

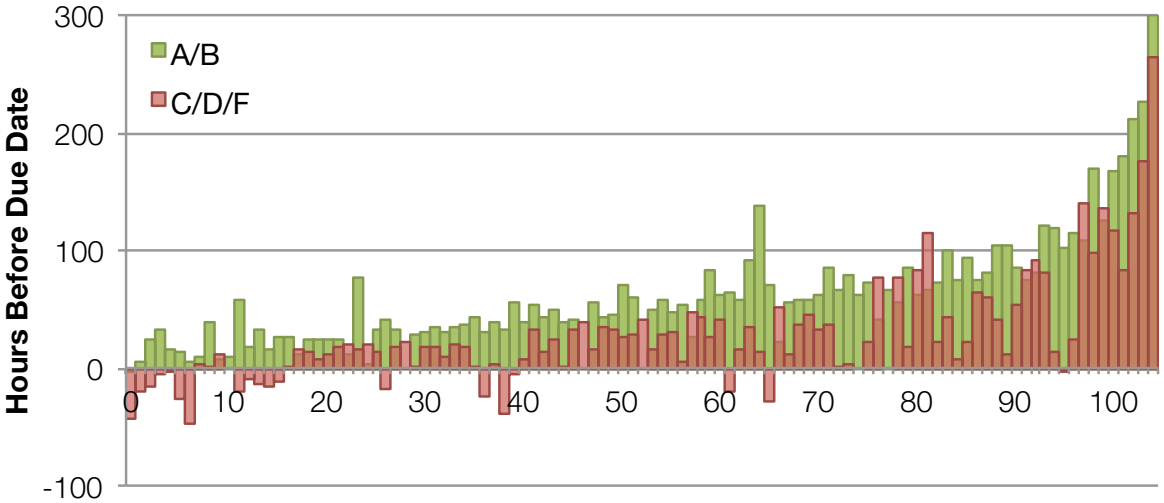

Figure 4.3: Mean time of first submissions on each assignment, distinguished by group. Each bar represents one assignment.

start time show the same trends as those reported by researchers who performed studies that tracked student development outside of an automated grading system to measure actual start time.

Figure 4.4 shows the distribution of first submission times by a student on any assignment, where submissions have been grouped into 24-hour buckets, and positive numbers represent days before the due date. As the reader can see from the figure, there is a statistically significant difference in the first submission times between the two groups ($N = 3437$, $F(1,1) = 227.2$, $p < 0.0001$). The mean first submission time for the A/B group is 64.9 hours ($\sigma = 72.0$), or approximately 2.7 days. The mean for the C/D/F group is 29.6 hours ($\sigma = 61.6$), or just over one day.

From the figure, the reader can also see that two-thirds of students in the A/B group made their first submission to Web-CAT at least one day in advance of the due date. Two-thirds of students in the C/D/F group, however, did not make their first submission until the day the project was due or later.

Caution must be exercised when examining absolute submission times, however. Different assignments may have different windows of time during which they are available for students to work and make submissions. Earlier CS courses typically have more assignments that are shorter in duration, since they are introducing students with less experience to introductory topics. Likewise, later courses may have students working on larger projects that involve more work, so those courses may have fewer assignments that are available for a longer period of time. Large variations in the amount of time available to work on an assignment across the entire data set make it difficult to compare assignments directly.

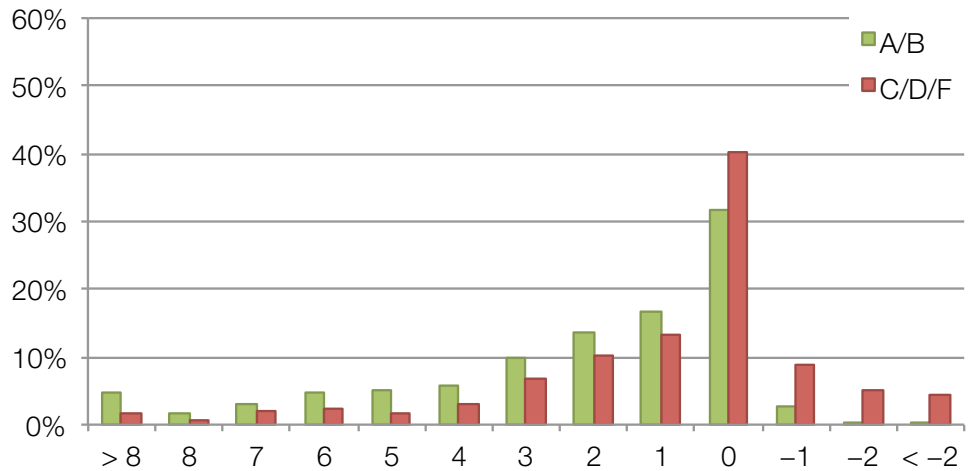

Figure 4.4: Distribution of times of first submission, by group. The horizontal axis represents days before the assignment's due date. Negative numbers represent first submissions that were made after the assignment was due.

Further complicating this is that I cannot perform a within-subjects comparison using the assignment as an independent variable, because I only have a single data point—the scoring submission—by each student for each assignment.

In order to address this, I normalized the submission times for each assignment based on the time of the earliest submission attempt by any student on that assignment. Figure 4.5 shows the distribution of earliest submission times for each assignment. For purposes of comparison, Figure 4.6 shows the distribution with submissions in the A/B group distinguished from those in the C/D/F group. Ignoring groups, the earliest submission times range from 492 hours (approximately three weeks) on the high end to 7 hours on the low end.

For each assignment in the data set, I used the earliest submission time as the base value to normalize all other submission times for that assignment. This yields a relative scale where 0.0 still represents the due date, and 1.0 represents the time of the earliest submission on that assignment.

Figure 4.7 shows the distribution of normalized first submission times. Even though the time scales are different, the distribution is similar to the unnormalized times in Figure 4.4.

The overall result is the same for the normalized times. I observed a statistically significant difference between the A/B group and the C/D/F group ($N = 3437$, $F(1,1) = 123.7$, $p < 0.0001$). After normalization, the A/B group had a mean first submission time of 0.37 ($\sigma = 0.37$), and the C/D/F group had a mean first submission time of 0.13 ($\sigma = 0.85$). Students who received A/B scores started working on the assignment between two and three times as early as students who received C/D/F scores. These results are consistent with

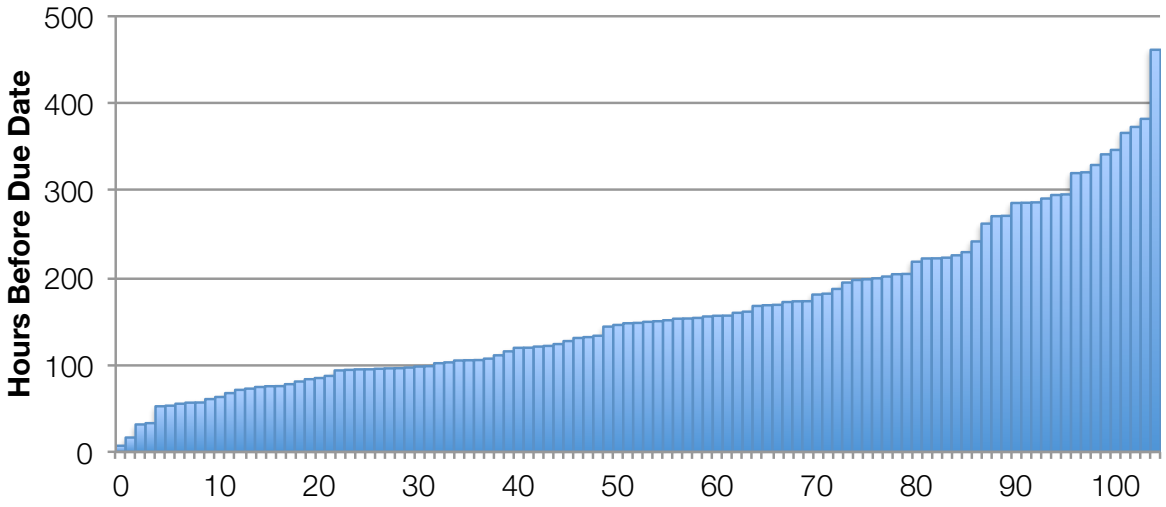

Figure 4.5: Earliest time of first submission on each assignment. Each bar represents one assignment.

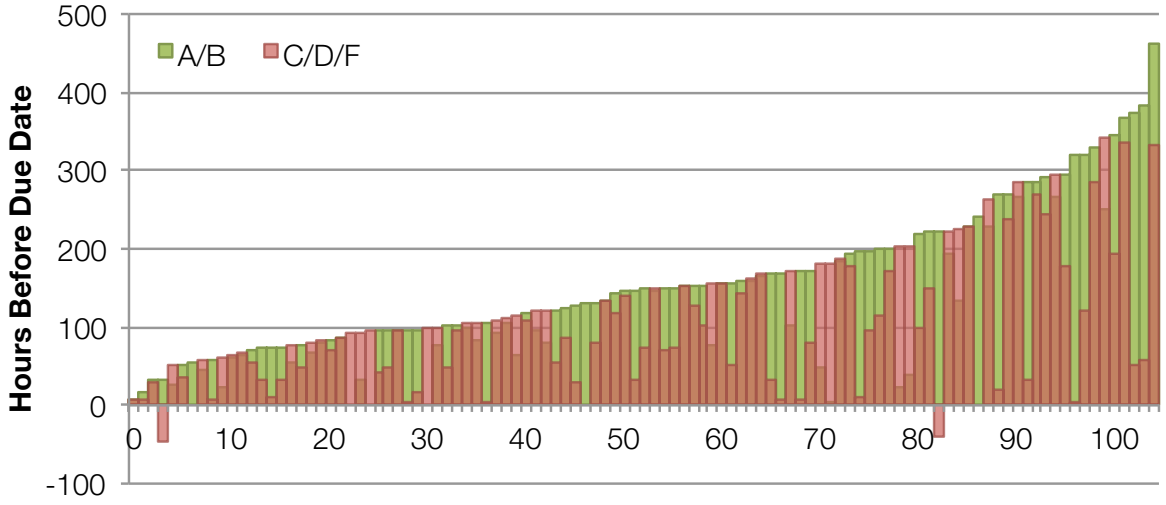

Figure 4.6: Earliest time of first submissions on each assignment, distinguished by group. Each bar represents one assignment.

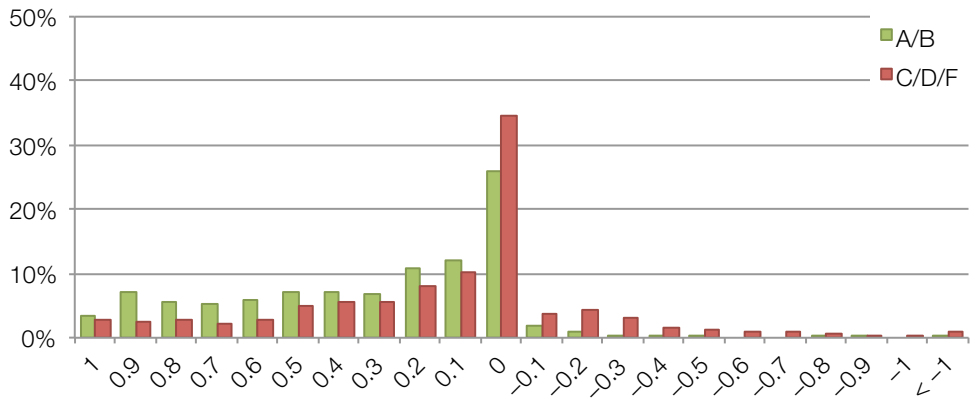

Figure 4.7: Distribution of normalized times of first submission, by group.

the unnormalized times, showing that the variability in the times that the assignments were made available did not contribute much of a blurring effect on the absolute data.

4.3.2 Time of Final Submission

Figure 4.8 shows the distribution of absolute final (scoring) submission times by a student on any assignment, in a parallel fashion to the first submission times in Figure 4.4. As with the first submission times, 24-hour buckets were used, so this distribution represents the number of days before the due date that students finished working on an assignment.

Figure 4.8 shows that, like with first submission times, there is a statistically significant difference in the time of final submissions between the two groups ($N = 3437$, $F(1,1) = 516.2$, $p < 0.0001$). The mean final submission time for students in the A/B group was 29.3 hours ($\sigma = 54.8$), while the mean final submission time for students in the C/D/F group was -11.4 hours ($\sigma = 47.7$). One-third of students who received A/B scores on the assignment finished it at least one day in advance of the due date. On the other hand, half of the students who received C/D/F scores did not finish until after the due date.

As with the first submission times, I normalized the final submission times to remove any possible blurring effects. In order to keep the first submission and final submission times on the same scale, I used the same base for normalization—a normalized time of 1.0 is equal to the time of the earliest *first* submission. Figure 4.9 shows the distribution of normalized final submission times.

As before, the overall result is the same; there is a statistically significant difference between the groups ($N = 3437$, $F(1,1) = 149.7$, $p < 0.0001$). The mean normalized final submission time for students in the A/B group was 0.14 ($\sigma = 0.39$), while the mean normalized final submission time for students in the C/D/F group was -0.19 ($\sigma = 1.12$).

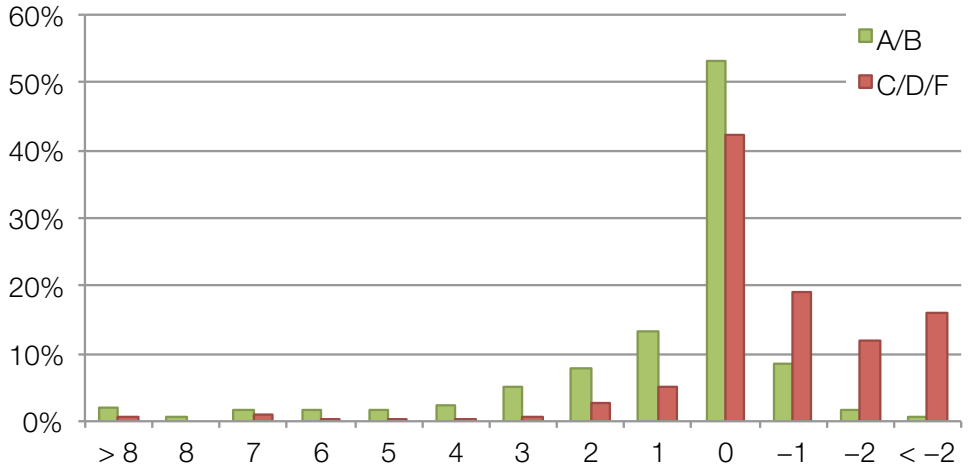

Figure 4.8: Distribution of times of final submission, by group.

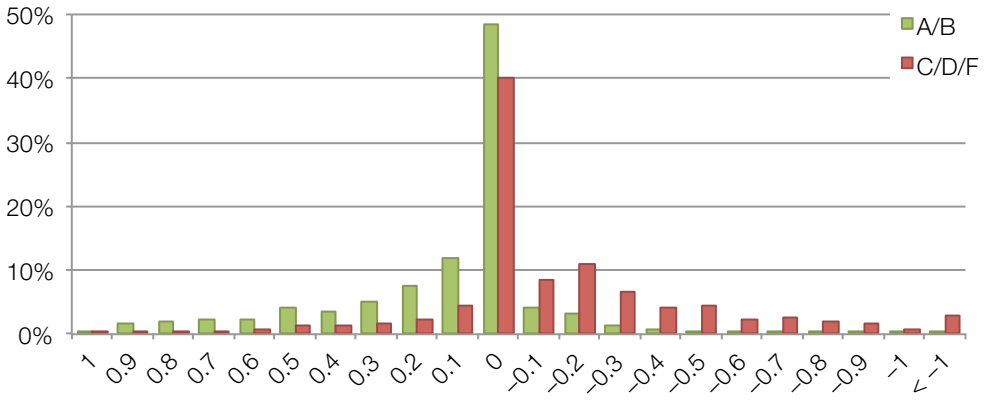

Figure 4.9: Distribution of normalized times of final submission, by group.

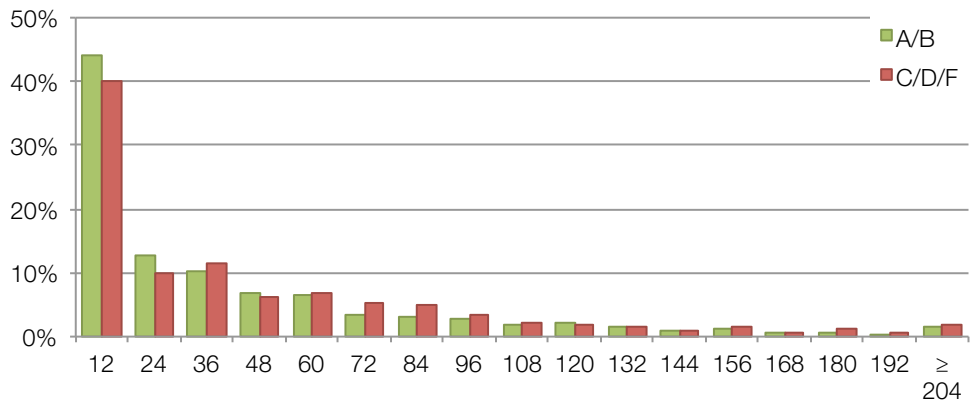

Figure 4.10: Distribution of elapsed time in hours, by group.

A very interesting observation can be made by comparing the start and finish times of the two groups. The mean final submission time for the A/B group (29.3 hours, 0.14 normalized) is nearly the same as the mean first submission time for the C/D/F group (29.6 hours, 0.13 normalized). In other words, students who receive A/B scores on an assignment are *finishing* their work at almost exactly the same time that students who receive C/D/F scores are just *starting*.

4.3.3 Time Spent on an Assignment

I also investigated the total time that elapsed between a student's first and final submissions to an assignment. I emphasize again that this does not exactly reflect the amount of time that the student spent working on the assignment. As discussed previously, students develop a considerable portion of their solutions even before making their first submissions to Web-CAT. Similarly, submissions are mere snapshots into a student's development process. Knowing the times of submissions N and $N + 1$ yields no information about how much of that time period was spent actually working on the assignment.

Figure 4.10 shows the distributions for each group of elapsed time between the first and final submissions to an assignment. In this case I used 12-hour rather than 24-hour buckets, because nearly half of the students in the data set had elapsed times of less than one day.

An analysis of variance indicates that there was a statistically significant, though small, difference between the two groups ($N = 3437$, $F(1, 1) = 9.94$, $p < 0.0016$). The mean elapsed time for students in the A/B group was 35.6 hours ($\sigma = 48.9$), while the mean elapsed time for students in the C/D/F group was 41.0 hours ($\sigma = 51.2$). The difference in the means is so small when compared to the standard deviation, however, that it is not

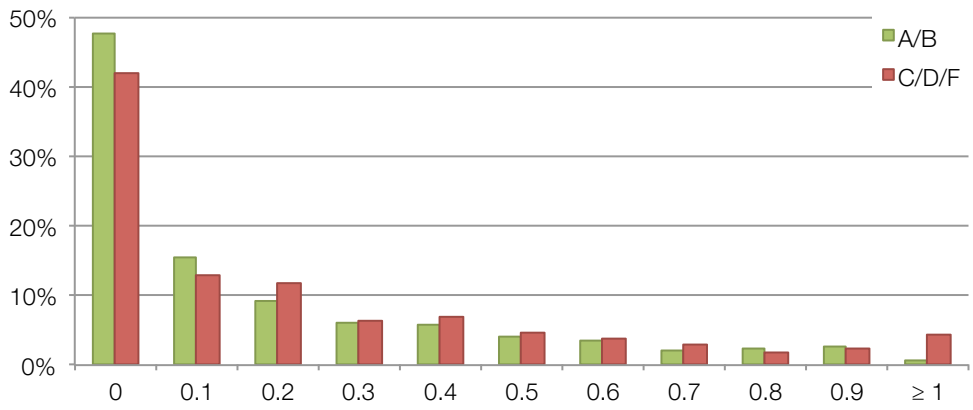

Figure 4.11: Distribution of normalized elapsed time, by group.

convincing evidence that students who perform better on the assignments also converge more quickly on a working solution.

As with the first submission and final submission times, I also explored normalized time spent on the assignment. Figure 4.11 shows the distributions for each group of normalized elapsed time between the first and final submissions. The normalization effectively magnifies the difference between the two groups, and an ANOVA reveals a statistically significant difference ($N = 3437$, $F(1,1) = 19.7$, $p < 0.0001$). The mean normalized elapsed time for students in the A/B group is 0.23 ($\sigma = 0.37$), while the mean normalized elapsed time for students in the C/D/F group is 0.32 ($\sigma = 0.85$).

4.3.4 Comparison to the Full Data Set

After having pruned the data set by removing the students consistently in the A/B group and the students consistently in the C/D/F group, I also wished to step back and compare the behavioral patterns of the students in each of those groups to the students in the reduced data set. My findings were consistent with the observations above. When students in the reduced data set performed well on an assignment, they behaved in a similar fashion to those who consistently fell into the A/B group. Likewise, when students in the reduced data set performed poorly, they were found to have behaved similarly to those who consistently fell into the C/D/F group.

4.4 Intervening to Improve Time Management

Armed with specific evidence about how a student's performance on an assignment is affected by his or her time management habits, how might an instructor seek to improve those behaviors for at-risk students? What incentives might be offered to encourage students to start or finish their work earlier?

Several approaches to improving student time management have been explored by educators. Students can be directed to academic enrichment resources at their university; many schools provide short seminars or courses intended to help students learn to manage their time and workload in multiple classes. Data like the results in Section 4.3 can be shown to students, although the mere act of presenting statistics as proof to students is not likely to motivate them. Dividing large projects into multiple staged deliverables [HHMWW06] can be useful in preventing students from waiting until the last minute to start, as can requiring them to keep detailed records of the time that they spent working on the assignment [SE11].

Extra credit opportunities are another commonly used method of rewarding students who participate in contests or go above and beyond the requirements of an assignment [Rob00], and also a method of attempting to affect changes in student behaviors. In computer science courses at Virginia Tech for example, instructors have sometimes given additional points to students who finish their work ahead of the deadline. Through Web-CAT, such a bonus can be automatically applied to each student's submission.

James Stodder raises some issues about the ethics of classroom-based incentives, especially those that affect grades in the class [Sto98]. Although his teaching and research are in the area of economics, his concerns about grading students on behavior that might be driven by luck are valid across many disciplines. Extra credit opportunities ideally should be crafted so that they reward explicit behavioral changes or additional effort of some kind that would have been unlikely to occur if the extra credit had not been offered.

Educators must also be careful when offering extra credit to ensure that the opportunity can be—and has a good chance of being—attempted by students at all ability levels in the class. If only the most advanced students will be able to earn the extra credit, then we only stretch the gap between them and the weaker students further. Extra credit offered for work beyond what was required on the assignment might only be earned by students who were already advanced and/or creative. On the other hand, a bonus for completing the assignment sufficiently far in advance is something that students at all ability levels have a chance to earn—they need only imagine that the extra credit deadline is the true deadline and shift their work habits accordingly.

Intuitively, I assumed that offering students extra credit to finish earlier would result in a positive shift in the starting and finishing times of students on the assignments, as well as a positive shift in performance because the data above showed that students perform better

when they start and finish earlier. I would like to use a data-driven approach to see if this holds.

Unfortunately, due to the age of the data set used above, information about which assignments offered extra credit and the exact nature of those opportunities would be difficult to obtain. Instead, I explore this question with new data from a course where this question was specifically raised. Data collection was carried out in the Spring 2012 offering of CS 1044, a C++ programming course for non-CS majors at Virginia Tech. There were four programming projects assigned throughout the semester. The amount of time that students were able to work on each assignment, from its original posting to its deadline, was about two weeks on average—with the exception of the second project, which was available for approximately four weeks due to a spring break period.

The four projects in the course were, in order: writing a perpetual calendar (Calendar), performing basic image manipulation (Images), computing the odds of a series of lottery games (Lottery1), and extending the lottery game to use more advanced data structures and algorithms (Lottery2).

The Calendar and Images projects had no extra credit opportunities. After they were completed, I extended the following extra credit opportunity to the class for Lottery1 and Lottery2: Submissions made at least three days before the deadline were granted an extra 10 points. (Scores that would exceed 100% were not capped.)

I had to make two decisions regarding the nature of this bonus. First, how far in advance of the deadline should the bonus be applied? The data above indicate that students perform better if they make their first submission even just one day in advance of the assignment's deadline. However, I did not simply want to reward behavior that fell right on the threshold of what the previous data indicate, but to affect a much larger change in their habits. For this reason, I chose **three days before the deadline** as the cutoff for the bonus.

Second, how big should the bonus be? A bonus that is too small might be ignored by students, whereas a bonus that is too large might have an undesirable effect on the grade distribution in the course. Programming projects in this course comprise 50% of the student's overall score. Earning the 10% bonus on both programming assignments for which it applied thus corresponds to a 2.5% increase in the overall score—a potentially large change. Students were not made aware of this fact explicitly, however; only those who made the effort to do the calculations themselves ahead of time would see this.

4.4.1 Exploring the Data

Before evaluating the extra credit opportunity, I wished to see if the trends in the larger data set in Section 4.3 also held for the students in this course. The complete CS 1044 data set includes 1,083 scoring submissions by 285 students. Using the same 80% split point to partition students into A/B and C/D/F groups, I removed the students who were

consistently in either of those groups. This left 202 scoring submissions from 55 students, which was a bit lower than expected. This appears to be the result of some degree of "grade inflation" on the programming assignments in the course, so 80% might not be the ideal split point for this data set. However, the notion of a student "doing well" is typically not relative to the other students in the course, but rather it depends only on that particular student's score, so I will continue to use the same 80% point here.

In this reduced data set, I did not find a statistically significant difference in the starting time between groups ($N = 202$, $F(1,1) = 0.349$, $p = 0.556$), in the finishing times ($N = 202$, $F(1,1) = 2.286$, $p = 0.132$), or in time spent ($N = 202$, $F(1,1) = 0.975$, $p = 0.326$). There was also no significant difference found between groups for the normalized times. This is likely due to the much smaller size of this data set compared to the one in Section 4.3.

Among the entire set of 1,084 scoring submissions, the only statistically significant difference between A/B submissions and C/D/F submissions was in the normalized finish time ($N = 1083$, $F(1,1) = 4.070$, $p = 0.0439$). The mean normalized finish time for A/B submissions was 0.22 ($\sigma = 0.28$), while the mean for C/D/F submissions was 0.06 ($\sigma = 0.22$).

Of the 285 students in the class, 56 (19.6%) started all four projects at least 3 days before the deadline, and 39 (13.7%) of them finished all four before that threshold. 122 students (42.8%) earned extra credit by finishing at least one of the final two assignments before the threshold. 107 of them, however, had also completed at least one of the first two assignments before the three-day threshold as well. On the other end of the spectrum, there were 74 students (26.0%) who finished all four projects fewer than three days before the deadline or after it had passed.

4.4.2 Behavior With and Without Extra Credit

I examined the effect of the extra credit opportunity on students' starting and finishing times by looking for differences between the assignments that had extra credit and those that did not. There was a statistically significant difference in the absolute start times between the two groups of assignments ($N = 1083$, $F(1,1) = 115.77$, $p < 0.0001$).

An unexpected result emerged when I examined the means themselves, however. The mean start time for the assignments with no extra credit was 5.67 days ($\sigma = 5.80$); for the assignments with extra credit, the mean was 3.25 days ($\sigma = 3.85$). In other words, students started earlier on average on the assignments where extra credit was *not* offered. The difference is significant, but in the reverse direction to what I had anticipated.

I also found that there was a significant difference in the absolute finish times ($N = 1083$, $F(1,1) = 104.83$, $p < 0.0001$). The same relationship between the means holds here as well. The mean finish time for the assignments with no extra credit was 4.54 days ($\sigma = 5.30$); for the assignments with extra credit, the mean was 2.38 days ($\sigma = 3.39$). As before, students on average finished earlier on the assignments where extra credit was not offered.

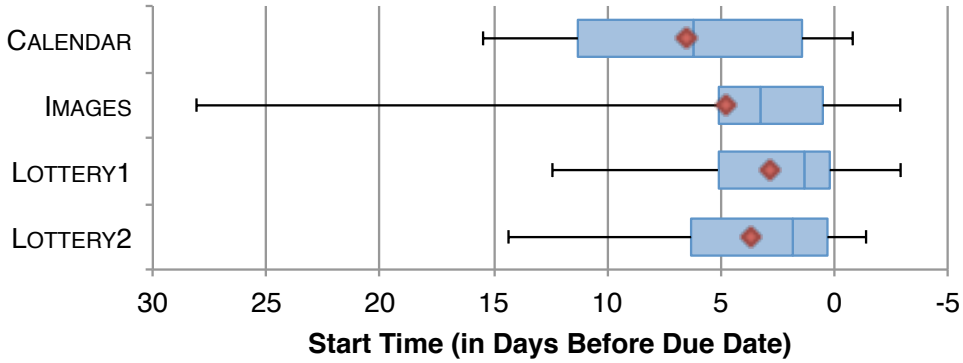

Figure 4.12: Box-and-whisker plot showing distributions of start times (in days) for each of the four programming assignments.

Seeing that there is a difference of roughly one day between the mean start and finish times encouraged me to look into the time spent on the assignments as well. I found no significant difference in time spent between the two groups ($N = 1083$, $F(1,1) = 3.029$, $p = 0.0821$). The mean time spent on the assignments without extra credit was 1.12 days ($\sigma = 2.85$); on assignments with extra credit, the mean time spent was 0.87 days ($\sigma = 1.97$). There were 348 scoring submissions to the extra credit assignments where the bonus was not earned because the student continued (or started) working into the three day threshold. 75% of those submissions were by students who spent less than a day between their first submission and final submission to that assignment. Given the fact that the assignments were available for two weeks or more, many of these students probably could have shifted forward the time that they worked on the assignments and earned the bonus. This indicates that the extra credit did not provide the motivation I had hoped.

4.4.3 Other Factors

Since the extra credit opportunity does not appear to be a significant factor in the way students managed their time on these assignments, what other factors might have come into play?

Figure 4.12 shows the distributions of start times (in days) for each of the four programming assignments. There is a significant difference among these absolute start times for each assignment ($N = 1083$, $F(3,3) = 55.00$, $p < 0.0001$). Tukey's HSD test indicates that CALENDAR and IMAGES differ significantly from everything else; LOTTERY1 and LOTTERY2 differ significantly from the former two but not from each other. From the figure was can see that, essentially, students started earlier on work assigned earlier in the course and later on work that was assigned later.

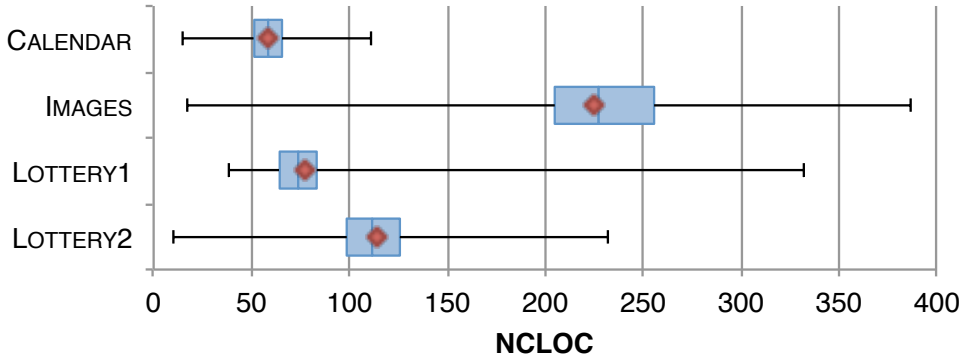

Figure 4.13: Box-and-whisker plot showing NCLOC distributions for each of the four programming assignments.

The assignment with mean start time furthest ahead of the due date was the first one in the course, CALENDAR ($\mu = 6.54$ days, $\sigma = 5.29$). This is perhaps not a surprising result; students might have had more flexibility at the beginning of the semester to manage their time between this and other classes they were taking. They may have also been less able to accurately estimate the amount of time that such an assignment would take (many of these students were first-time programmers) and erred on the side of caution. Then, as the semester continued, deadline and workload pressures in other classes began to have an effect, causing them to start later than they did on the previous assignments. Since the assignments with extra credit were given in the later half of the course, those pressures may have provided enough inertia to cancel out the incentive.

Another possibility is that the difficulty of the assignments, or the amount of effort involved, outweighed any effects that the extra credit opportunity might have had. My intuition is that the first assignment, CALENDAR, was the easiest, and that the second assignment, IMAGES, might have been a bit too ambitious and was, in retrospect, the most difficult of the four. LOTTERY1 and LOTTERY2 fell in the middle, as a way of pulling back slightly when students experienced significant difficulty with IMAGES.

To quantify this notion of difficulty and effort, I examined the number of non-commented lines of code (NCLOC) that students wrote for each assignment. Presumably, more difficult assignments require more effort to complete, and that effort is directly related to the number of lines of code that students were required to write.

I found that the four assignments did have significantly different NCLOC ($N = 1081$, $F(3,3) = 1516.33$, $p < 0.0001$). Tukey's HSD test indicates that each of the four assignments is significantly different from the others. In increasing NCLOC order, CALENDAR had a mean NCLOC of 58.6 ($\sigma = 12.1$), LOTTERY1 had a mean NCLOC of 76.9 ($\sigma = 24.7$), LOTTERY2 had a mean NCLOC of 113.6 ($\sigma = 26.7$), and IMAGES had a mean NCLOC of

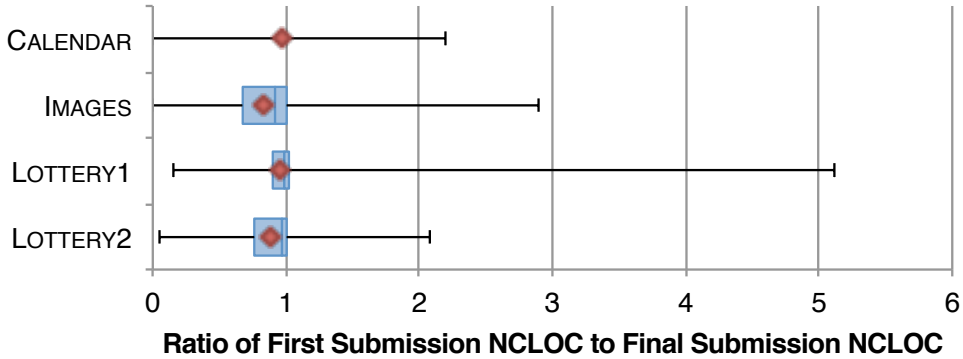

Figure 4.14: Box-and-whisker plot showing distributions of the first/final NCLOC ratios for each of the four programming assignments.

224.9 ($\sigma = 54.0$). These results are consistent with my intuition of the relative difficulty of the assignments. Figure 4.13 shows the distribution for each assignment in more detail.

Let us again consider the time that students started on the assignment. Recall that the time of the first submission does not represent the true time that the student started on the project. One way to estimate the amount of effort that a student has put into an assignment before their first submission is to compute the ratio of NCLOC in his or her first submission to NCLOC in his or her final submission.

In programming courses where students are expected to test their code using a prescribed methodology (such as writing JUnit test cases), they are able to get significant feedback during local development. Due to this, students in such courses typically have a considerable portion of their solution already written when they being submitting to Web-CAT and this first/final ratio is very close to 1. The remaining development time between first and final submissions is then devoted to debugging rather than writing large amounts of new code.

In the CS 1044 course, students were not required to follow any particular testing methodology. The only feedback they received outside of Web-CAT was from executing the program in its entirety with different inputs. However, the first/final NCLOC ratios for the assignments in this course do not appear to reflect a difference in that regard. Figure 4.14 shows the distribution of first/final NCLOC for students in the class, grouped by assignment.

I found a significant difference between these ratios across the four assignments ($N = 1082$, $F(3, 3) = 14.465$, $p < 0.0001$). Tukey's HSD test indicates that CALENDAR and LOTTERY1 are significantly different from IMAGES and LOTTERY2. More importantly, however, is the observation that the 25th percentile for any of the assignments is never less than 0.67. In other words, a vast majority of students in the class already had written at least 2/3 of the code that makes up their final submission by the time they started submitting anything to

Web-CAT. This indicates that students did spend an appreciable amount of time working on the assignments before the time of their first submission. This is not a surprising result, due to the small size of these assignments. The CALENDAR and LOTTERY1 assignments averaged less than 100 non-commented lines of code—in other words, roughly one "page." If that is the typical amount of code required for a behaviorally correct solution to the assignment, I would expect students to have most of it already written before they sought out feedback or evaluation from Web-CAT, at which point they would spend their remaining time debugging rather than writing new code.

Despite these findings, there was no significant difference in first/final NCLOC between the assignments with extra credit and those without ($N = 1082$, $F(1,1) = 0.0668$, $p = 0.796$). The mean first/final NCLOC for submissions to assignments without extra credit was 0.907 ($\sigma = 0.291$); the mean for assignments with extra credit is almost identical, at 0.912 ($\sigma = 0.299$). Thus, it does not appear that students made a larger relative effort before their first submission on the extra credit assignments than they did on the ones without, so I have no evidence that they *started* significantly earlier on them.

Out of the 285 students in the class where the extra credit was offered, 99 (34.7%) did not finish the first two assignments until within three days of the deadline (or after the deadline). Of those 99 students, only 11 finished one of the final two assignments before the extra credit deadline, and four finished both before the extra credit deadline. Likewise, 105 students (36.8%) finished one of the first two assignments at least three days before the deadline; 18 of them went on to finish both of the final two assignments before the extra credit deadline. In other words, there were 204 students (71.6%) in the class whose behavior could have strictly improved as a result of the incentive, but only 33 of them showed such an improvement. On the other end of the spectrum, there were 81 students (28.4%) who finished the first two assignments more than three days before the deadline, but 42 of them regressed on the final two assignments; 25 of them finished only one assignment before the extra credit deadline, and 17 of them did not finish either assignment before the extra credit deadline.

4.5 Summary

The results above provide substantial evidence to educators that students perform better on programming assignments when they **start earlier** and **finish earlier**. By eliminating all the consistently strong and consistently weak students, I was able to do within-subjects comparisons to strengthen the conclusions. Interestingly, students who performed poorly were starting on the assignments around the same time that students who performed well were finishing. However, it did not appear that students who performed well and those who performed poorly spent a different amount of time on the assignments.

When I examined the same trends in the entire data set by including the consistently strong

and consistently weak students, I found that students with mixed performance behaved similarly to the consistently good group when they did well, and behaved similarly to the consistently weak group when they performed poorly.

Knowing that time management plays such a strong role in student performance, I then attempted an intervention in a different class by offering extra credit to students who completed the assignments sufficiently far in advance of the deadline. The first two assignments did not have extra credit opportunities, but the last two offered 10 additional points to anyone who finished at least 3 days ahead of the deadline. Surprisingly, it did not appear that the offered bonus had any positive effect on student behavior. Students did not start significantly earlier on the assignments where extra credit was offered, even though those assignments required less effort to complete than one of the assignments that came before them.

From these findings I can conclude that the incentive power of the extra credit opportunity was most likely overshadowed by other factors that I have not examined here. Given that students were not taking this course in isolation, I must consider the possibility that outside forces, such as deadline pressures in other courses, had a greater impact on their time management in my course.

5
Case Study: Evaluating an Educational Tool

The second case study examines how data from Web-CAT have been used to evaluate another educational tool in order to determine whether it effectively met its designers' goals of improving student learning.

Many computer science programs include C++ as a core part of their curricula. In our courses at Virginia Tech, we have consistently seen that memory management techniques in C++ are one of the most, if not the most, frequent sources of difficulty for our students. This observation is consistent with the experiences of others, as described in Chapter 2.

Unfortunately, the tools typically used to teach C++, such as Microsoft Visual C++ or the GNU gcc compiler, are professional tools that do not provide the most appropriate feedback for educational purposes, especially for students at an introductory level. We have also observed that the complexity of the debugging tools available in those environments acts as a barrier to student use; instead, they prefer "caveman debugging" such as inserting output statements to trace execution. A better strategy might be to "push" diagnostic information to the student, rather than requiring them to actively "pull" these data.

We developed a toolkit named *Dereferee* [AEPQ09] to address these issues. By using this toolkit, students receive detailed diagnostics when they make memory-related errors in their codeerrors that normally would cause a program to crash, or worse, would silently lead to undefined or incorrect behavior. The toolkit detects and diagnoses 40 unique kinds of pointer mismanagement errors ranging from dereferencing null, uninitialized, or dangling pointers, to more advanced problems such as faulty pointer arithmetic or accessing arrays out of bounds. In addition, memory leaks are detected at the precise moment that access to a live block of memory is lost.

5.1 Data Used in This Case Study

At Virginia Tech, we used Dereferee in our introductory C++ course (our students third course, after two semesters of Java) from 2005 to 2008. (The department has since switched to Java for all three courses.)

To investigate whether students benefited from using Dereferee, I examined solutions that students submitted for four programming assignments during the Fall 2006 semester: a linked double-ended queue (LinkedDeque); a program that searches for "word ladders," chains of words from a dictionary where each word differs from its predecessor by one letter (WordLadders); a map interface backed by a binary search tree (MapBST); and a general tree using the left-child/right-sibling implementation scheme (GeneralTree). Each of these assignments required students to apply a clear understanding of pointers and memory management concepts, and they proceed from less difficult to more difficult in the order described above.

Students were permitted to make an unlimited number of submissions for each assignment during the time which the assignment is open, and submissions were made to an automated grading system for assessment against a set of instructor-written reference tests.

In this offering of the course, students were given the option of using Dereferee; its use was encouraged and reinforced by lab exercises but not required for the four large programming assignments. Out of 67 students in the course, 54 students (81%) used Dereferee on at least one submission of some assignments.

As I began my analysis, I was concerned that there might be some bias among the students who chose to use the tool; that is, more advanced students might see the value of Dereferee as a debugging aid and be more likely to use it. To check for the presence of such a bias, I compared final exam scores between students who elected to use Dereferee on any submission and those who did not, and found no statistically significant difference in exam scores between the groups ($F(1, 65) = 0.22$). In other words, I was unable to find evidence that students who chose to use Dereferee were academically stronger (or weaker) in the course, providing some confidence that this issue was not an important factor in the results.

Student submissions were executed against a set of instructor-provided reference tests using the CxxTest unit-testing framework which has been modified to provide additional protection against signals raised during test execution. Thus, if an instructor writes 30 tests and a student's code causes a segmentation fault in a test case, he or she can still get feedback from the remaining tests instead of crashing the test harness completely.

During execution of the reference tests, each test may produce one of four possible results: *success*, *assertion failure* (a logical error that produced an incorrect result), *abnormal termination* (a segmentation fault or some other kind of program crash), or *unbounded recursion* (stack overflow). A fifth result, *timeout*, occurs when the student's solution exceeds the

maximum amount of time allowed; unlike the other results, a timeout terminates the entire test harness, preventing any of the subsequent tests from running.

Without Dereferee, many pointer-related errors would manifest themselves as abnormal terminations, or worse, silently succeed and allow execution to continue, potentially with corrupted data. When using Dereferee, these types of errors are caught and identified with a specific error code. Many of these codes are highly detailed, describing not only the cause of the error but the specific operator that was used (for example, "dereferenced a null pointer using the * operator"). Since I was only concerned with the underlying cause of the errors in my evaluation, I folded them into broader categories: uninitialized pointer errors, dangling pointer errors, null pointer errors, and errors involving faulty pointer arithmetic. Furthermore, since none of the programs that I examined involved usage of arrays or pointer arithmetic, I ignored the last category and focused my evaluation entirely on uninitialized, dangling, and null pointers.

5.2 Process

Students in this case study used a pre-release version of the Dereferee toolkit that differed greatly from the current release that is now publicly available. The behavior of the original version is much like that of the latest version but lacked some of the stricter checks, such as the use of uninitialized pointers in relational expressions. Some of the functionality regarding pointer arithmetic has been improved as well. Not only was I interested in the types of errors made by students who did not use Dereferee, but I was also interested in what types of errors may have been missed by the early version of the toolkit.

For each of the four assignments, I executed every submission made by each student under the original testing conditions in order to collect the results that the students saw during their own development cycle. I then instrumented the code in each submission using an automated process. If the submission did not use Dereferee, its source was modified to inject checked pointer declarations and then it was re-executed to collect the more specific error information that the toolkit provides. If the submission already used Dereferee, I simply re-executed it with the newer version of the toolkit.

5.3 Results

5.3.1 Types of Errors Encountered

First I looked at a simple distribution of the errors that were discovered in student code. Figure 5.1 shows that across all submissions to all assignments, 60% of the errors were

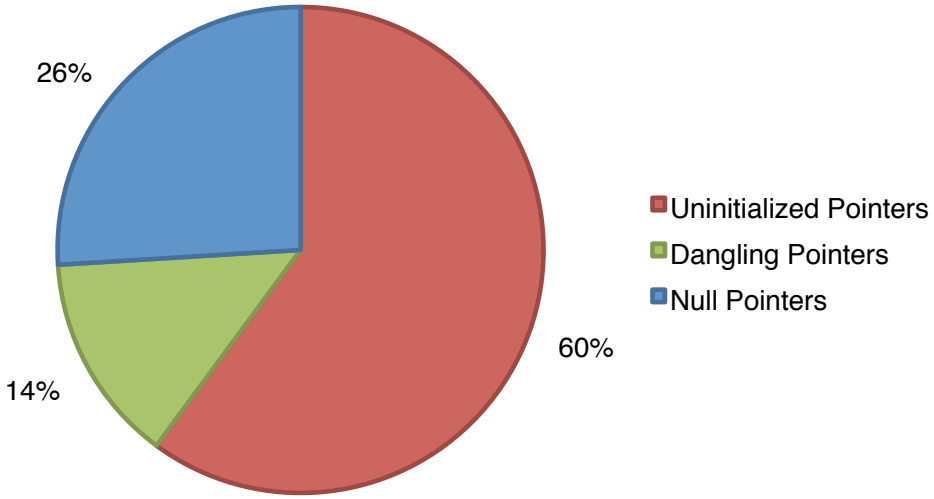

Figure 5.1: The distribution of memory-related errors in students' submissions. Each category describes the misuse of a particular type of pointer, usually by dereferencing.

caused by improperly using an uninitialized pointer (dereferencing it, comparing it to another pointer, deleting it, and so forth). In addition, 14% of the errors involved dangling pointers (using a pointer to memory that had already been freed), and 26% of the errors were improper uses of null pointers.

It was somewhat surprising that over half of the errors discovered involved using uninitialized pointers, because my intuition is that students cause more errors by failing to perform null checks properly or by trying to access a pointer to freed memory. The explanation for this may be two-fold. On the teaching side, if my (or another instructor's) intuition is that students struggle more with null or dangling pointers, perhaps the fewer occurrences of those errors point to successful teaching on our part, but also to a deficiency in how we teach initialization. In our curriculum in particular, being one that (at the time of these assignments) transitioned students from Java to C++, perhaps this is an indication that students do not fully understand that in C++ variables are not zeroed/nulled out when they are declared, as they are in Java. On the tool side, the stricter safeguards against using uninitialized pointers in an expression. Even operations that do not dereference a pointer, such as equality tests, will raise an error if one of the pointers is uninitialized. Therefore it is possible that a student could write code that compares an uninitialized pointer to an active pointer and it would yield correct results when executed without Dereferee but would fail when using it.

I also looked at the errors that existed in just the final submission by each student to each

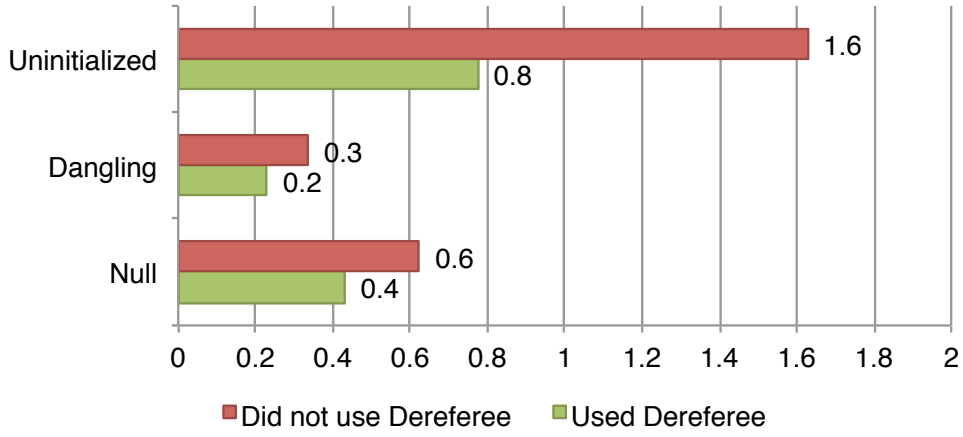

Figure 5.2: Average number of errors of each type across all submissions for students who did and did not use Dereferee.

assignment; in other words, errors that the student was never able to fix by the end of his or her development cycle. The distribution is nearly the same: 58% uninitialized, 13% dangling, and 29% null.

5.3.2 Frequency of Errors

By examining the overall frequency of errors in all submissions (3777 in total), a difference emerged between students who did use Dereferee versus those who did not, as summarized in Figure 5.2. Across all submissions where students did not use Dereferee, each submission experienced an average of 2.6 pointer errors ($\sigma = 5.7$) when tested against the reference test set. For submissions where students did elect to use Dereferee, however, the average was 1.4 ($\sigma = 3.6$), a 45% reduction. By considering both the choice to use Dereferee and the assignment as factors, while also separating out any interaction effects, an analysis of variance indicates that this difference is statistically significant ($F(3, 3770) = 56.9, p < 0.0001$). However, Cohen's d coefficient is 0.24, indicating that Dereferee only accounted for a difference of roughly a quarter of a standard deviation—a small effect size. This trend was true for each individual error category as well. Submissions without Dereferee averaged: 1.6 ($\sigma = 5.3$) uninitialized pointer references, compared to 0.8 ($\sigma = 3.4$) with Dereferee ($F(1, 3770) = 13.9, p < 0.0005, d = 0.19$); 0.3 ($\sigma = 1.5$) dangling references, compared to 0.2 ($\sigma = 0.9$) with Dereferee ($F(1, 3770) = 17.9, p < 0.0001, d = 0.08$); and 0.6 ($\sigma = 2.1$) null pointer references, compared to 0.4 ($\sigma = 1.3$) with Dereferee ($F(1, 3770) = 5.6, p < 0.02, d = 0.11$).

Looking at errors in the final submissions produced by students (225 altogether) revealed

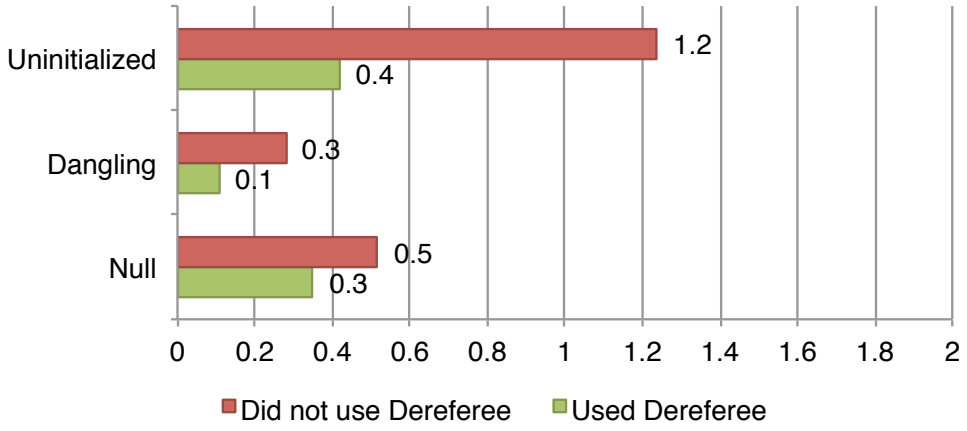

Figure 5.3: Average number of errors of each type across only final submissions for students who did and did not use Dereferee.

similar trends in the means, as shown in Figure 5.3. Students who did not use Dereferee produced final work with an average of 2.0 errors overall: 1.2 uninitialized pointer errors, 0.3 dangling references, and 0.5 null pointer errors. Students who did use Dereferee averaged 0.9 errors overall: 0.4 uninitialized pointer errors, 0.1 dangling references, and 0.3 null pointer errors. These differences in final submission errors between the two groups were not significant, however, because of the smaller number of data values and the fact that the final submissions had the fewest errors, making the differences between the means small compared to the standard deviation.

5.3.3 Students Affected by Errors

Figure 5.4 shows the percentage of students who were affected by each of the three types of errors being investigated. The first chart shows the percentage of students in the course who had *any instance* of each type of error detected by the reference tests during their development cycle.

One might expect that the number of students who *ever* see an error at any point in their development cycle would be much higher, but these statistics only provide a lower bound because they include only snapshots of that cycle that are captured when students submit solutions to the automated grading server. Between each of these snapshots, students are doing their own testing as well, and if their own testing is reasonably thorough then they should catch some portion of the errors on their own and fix them before submitting their work.

These results show that for a particular type of error and assignment, anywhere from 10%

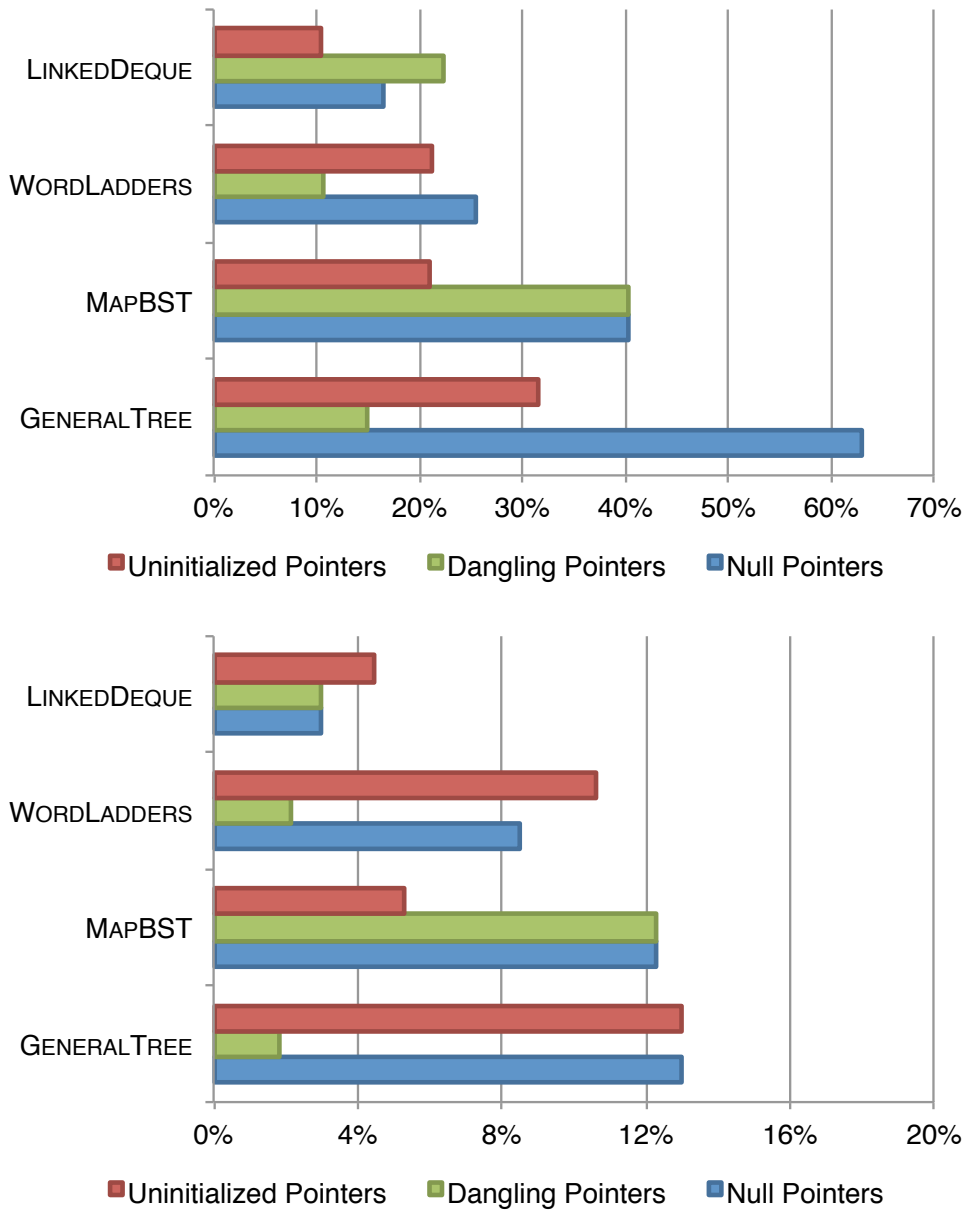

Figure 5.4: Percentages of students affected by each type of memory-related error broken down by assignment, on any submission and on final submissions.

to over 60% of students had errors in their solutions. If I ignore the type of error or the assignment on which it occurred, however, 84% of students overall had at least one error in a submission that they made during the course.

The second chart in Figure 5.4 shows data for only the final submission made by each student, indicating the percentage of students who were unable to resolve all occurrences of each type of error by the end of their development cycle. In this case, fewer than 13% of students were unable to resolve any particular type of error on each assignment. This is clearly an improvement, but if I again ignore the error type and assignment, the results are more sobering: 51% of students still had at least one unresolved pointer error on some assignment during the course.

5.3.4 Latent Errors

The errors represented in Figure 5.4 are those that were present in students' solutions whether they were properly identified as pointer errors during testing, or they resulted in a generic abnormal termination, or they silently executed "successfully." I also wanted to look at "latent" pointer defects: pointer errors that were executed but were not originally identified as such. In other words, latent pointer defects are uninitialized, dangling, or null pointer errors in students' final submissions that were detected after instrumentation, but that were not identified as pointer errors before instrumentation, where they instead resulted in abnormal termination, assertion failure, or even successful execution.

I partitioned the students' submissions on each assignment into those that used Dereferee and those that did not use Dereferee. This partitioning is mutually exclusive for submissions, but across students it is possible for there to be overlap if a student began the assignment without using Dereferee and then decided to use it later, or vice versa. I would expect the students who used Dereferee to have fewer latent errors, because those errors would be more likely to have been properly identified by the toolkit during testing. The instances where I might expect this not to be the case are those where students had errors that the new version of the library better detects.

For each of the three error types of interest, Figure 5.5 shows the percentage of students who had latent defects of that type in their final solutions. Of the students who did not use Dereferee, at least one student had latent pointer defects of each type on each assignment, where these defects would have been properly identified if they had been using the toolkit. Overall, 39% of the students who did not use Dereferee on their final submissions still had latent pointer defects of some kind.

Of the students who did use Dereferee on their final submissions, I found that for some error types on some assignments, no latent defects at all were found for any student. This is presumably because Dereferee properly identified the cause of pointer errors in these situations, allowing students to correct their work. However, after instrumenting their code

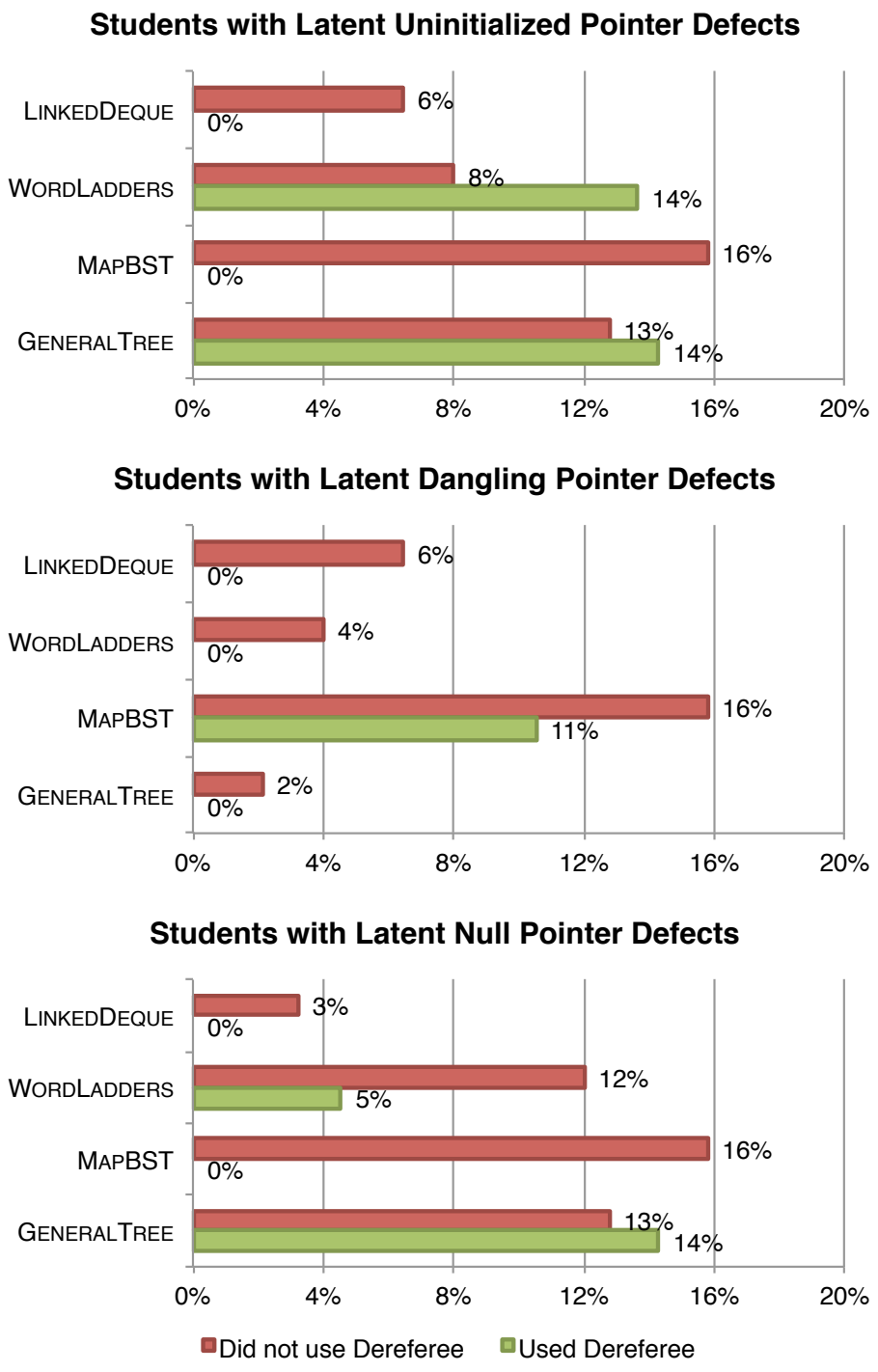

Figure 5.5: Percentages of students who had latent defects in their final submissions, broken down by assignment.

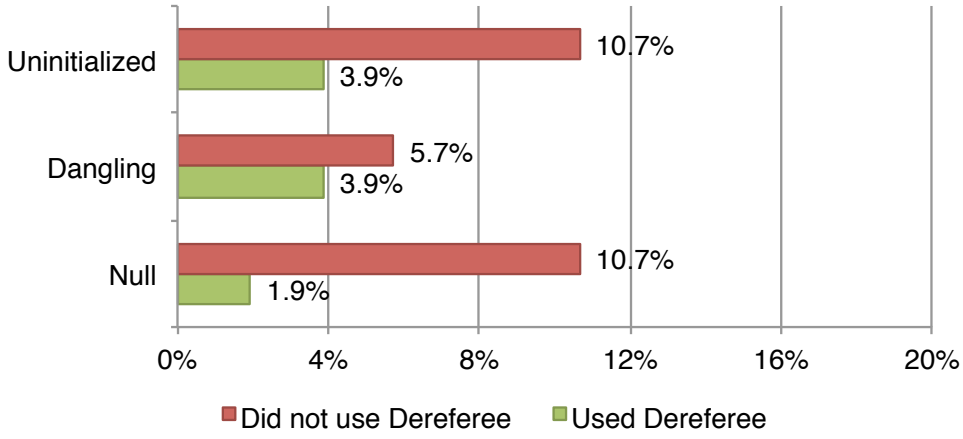

Figure 5.6: The proportion of final submissions across all projects that contained latent pointer errors.

with the updated version of the toolkit, some latent defects were still revealed—overall, 19% of students were affected.

Figure 5.6 summarizes the total latent pointer defects of each type across all assignments for those students who did use Dereferee and those who did not. Of the 225 final submissions across all four assignments, only 40 (17.8%) had any latent errors: 10 from students who used Dereferee (9.7%), and 30 from students who did not (24.6%). A χ-squared test shows that students who did not use Dereferee were more likely to have at least one latent pointer defect than students who did use it ($\chi^2(1, 225) = 8.32, p = 0.0039$). Using Cramér's ϕ statistic for the effect size of a χ^2 test I found that $\phi = 0.192$, indicating that the association is not terribly strong and that I should consider a larger population size to strengthen these findings. Indeed, this trend was also true for each category of pointer defect. 13 final submissions (10.7%) by students who did not use Dereferee had latent defects involving uninitialized pointers, compared to only four (3.9%) by those who used Dereferee ($\chi^2(1, 225) = 5.26, p = 0.02, \phi = 0.153$). Seven final submissions (5.7%) by students who did not use Dereferee had latent defects involving dangling pointers, compared to only four (3.9%) by those who used Dereferee, a difference that is not statistically significant ($\chi^2(1, 225) = 3.52, p = 0.06, \phi = 0.125$). 13 final submissions (10.7%) by students who did not use Dereferee had latent defects involving null pointers, compared to only two (1.9%) by those who used Dereferee ($\chi^2(1, 225) = 5.99, p < 0.015, \phi = 0.163$). While the ϕ coefficients for these tests do not give conclusive evidence, these results do indicate that using Dereferee helped students produce fewer defects during development and led to fewer latent pointer defects once their final version was completed, and it gives me a reason to explore the issue further in a larger population.

Seeing that a number of Dereferee users still had latent defects in their final solution for

at least one assignment in the course compelled me to investigate this subset further. Of the defects themselves, 60% were caused by uninitialized pointers. This is a result that I expected, because I greatly tightened the error checking in this area between the version used by my original population of students and the current release. In other words, the version of Dereferee used by the students themselves was not able to diagnose and report such errors in a number of cases. Dangling pointers caused 16% of the latent defects and null pointers caused 24%. These two numbers were surprising; they may point to enhancements that were made between releases, or to bugs that existed in the original version, either of which could have resulted in false positives.

5.4 Summary

This case study provides better insight into what students have learned and what they understand about dynamic memory allocation in a language such as C++. I also applied an intervention—the introduction of the Dereferee tool into the course material—in order to ease many of the difficulties of this topic, and saw positive results from that intervention.

Specifically, the data from this case study showed that students who used Dereferee on programming assignments did experience fewer errors *in their submissions* than non-users of the toolkit, but small effect sizes encourage further investigation with a larger sample size. I do not claim necessarily that the students who used Dereferee experienced fewer errors during *development*, however. Dereferee only provides additional diagnostic information about the errors that exist in code; it does not make any effort to correct or recover from those errors. The benefit provided by Dereferee is that students who had superior diagnostics about errors in their code—especially errors that might have executed silently without failing—were able to correct those errors locally before submitting their code to Web-CAT for grading.

By receiving this enhanced feedback during development, students who used Dereferee were able to converge on error-reduced solutions more quickly than students who did not use it. Students who used the toolkit seemed to have statistically significantly fewer "latent defects" in their code, meaning that nearly all of the errors that executed in their code were identified specifically as a type of memory error, rather than as a generic program crash or silent execution. However, the association between using Dereferee and lacking latent defects is not terribly strong. Again, in order to become more confident about this result, the question should be explored using a larger population. While the change in our curriculum from C++ to Java since the time of the original experiment makes it more difficult for me to do so, the advantage of the data collection infrastructure in Web-CAT is that it would be possible to seek out other instructors who use Web-CAT for C++ and collect data from them without intruding considerably on their classroom process.

6
Case Study: Evaluating an Assessment Instrument

To verify the correctness of a student's solution on a programming assignment submitted to an automated grading system, the instructor typically implements a "test harness" that repeatedly executes the student's code with different inputs, or interacts with different components of the solution to verify that it is behaviorally correct.

For Java assignments graded on Web-CAT at Virginia Tech, we typically express these tests using the JUnit unit testing framework, which allows an instructor to construct a set of tests to be run on each student's solution. The tests are independent of each other, as long as the student doesn't use any unexpected implementation strategies that cause state to persist between test cases (singletons have been known to cause problems in this way). Students' scores are based partly on the percentage of reference tests that pass when run against their solutions.

Composing a good reference test suite that adequately evaluates the student's understanding of an assignment's proficiency can be challenging for instructors. After the assignment is complete, the instructor can view the overall grades for each student, but that only describes how well the student performed on the assignment as a whole. More work would be required to obtain the results for each individual test case for each student.

Having access to the individual test case results would give an instructor the ability to treat the entire reference test suite as a *test* in the item response theory sense, where each test case is an *item* and the student's *response* for each item is whether or not the test passed when executed against his or her solution. Since "test" has conflicting meanings when we discuss these two concepts interchangeably, Table 6.1 concisely shows the mapping between the JUnit terminology and the equivalent item response theory concepts.

Thus, by applying methods from item response theory, an instructor can assess the *difficulty*

JUnit term	Item response theory term
test, test case	item
test suite	test

Table 6.1: Mapping between terminology used in JUnit and that used in item response theory.

and *discriminating ability* of each of the reference test cases. The benefits are two-fold; the instructor can use these parameters to evaluate the quality of his or her own reference test suite and also to evaluate his or her students' *demonstrated proficiency* on an assignment in perhaps a more accurate way than merely the percentage of test cases correct. In this case study, I focused primarily on the former scenario, saving discussion about the latter scenario for the end.

6.1 Data Used in This Case Study

For this case study, I isolated a single programming assignment in CS 2114 (Software Design and Data Structures). I pulled scoring submissions to that assignment for each student over the past three semesters (Fall 2010 to Fall 2011). From each of these submissions, Web-CAT's data model allowed me to extract the results for each individual reference test that was run against the student's solution—a value of 0 if the test failed, or 1 if the test passed. Using these data, I built a response matrix containing each students' results across the entire test suite. Overall, there were 334 scoring submissions, each executing 82 test cases.

The particular assignment used in this analysis was chosen because of the special nature of its reference test suite. On this assignment, students were asked to develop two implementations of a standard *queue* data structure—one that was backed by a circular array and another that was backed by a linked list. When a student's solution was graded, exactly the same set of reference test cases was run against both queue implementations (41 on the array queue, and the same 41 on the linked queue). This situation gave me the unique opportunity to not only assess the quality of the reference test suite, but also to explore which of the two data structure implementations caused more difficulties for students.

6.2 Process

Most discussions of item response theory center around three models, with differing numbers of parameters. The most general model is the three-parameter model (3PL), which incorporates parameters a_i denoting the discriminating ability of the ith item, b_i denoting the

item difficulty, and c_i denoting a value frequently called a "guessing" parameter, but which is more generally the probability with which students at an infinitely low proficiency level can still correctly respond to the item. The two-parameter (2PL) model assumes that the effects of guessing are negligible and fixes c_i at 0. Similarly, the one-parameter (1PL) model assumes that all items have equal discriminating ability and fixes a_i at a constant a for all items.

Since I was interested in both the difficulty and discriminating ability of the individual items, I eliminated the 1PL model from consideration. I then ran both the 2PL and 3PL models on my response matrix in order to investigate whether the additional c_i parameter had any influence. I discovered that many test items had significant c_i values in the range $0 \leq c_i < 0.2$, and one had a value greater than 0.5.

As noted previously, c_i is referred to as a "guessing" parameter in most IRT literature that deals with traditional multiple-choice exams. In my case, it is clear that it would not be possible for a student to "guess"—in an active sense—the correct behavior of the solution to a programming assignment. Unlike a multiple-choice exam, students are not presented with a finite set possible answers from which to choose their responses. However, there may still be some amount of chance involved in a student's solution producing the correct results. Therefore, I chose to continue using the 3PL for this analysis but resisted the temptation to assign an intuitive meaning to the c_i parameter and merely refer to it as the asymptotic lower bound on the probability of a correct response.

I also note that the parameter estimates, as well as the θ scores, will be different depending on which logistic model is used. This makes intuitive sense; under the 2PL model that ignores chance, item difficulties must be lower in order for students with lower proficiency to be able to answer them. Likewise, the proficiency estimates of those students might be marginally higher. When chance is factored into the 3PL model, this "loosens" up the estimates to a certain degree; the difficulty of some items will be higher and students with extremely low proficiency might have a lower θ than under the 2PL model because any correct responses would be attributed to chance.

With the help of the `ltm` (latent trait model) library available for the R statistical software package, I used the response matrix to calculate the three parameters for each item. This process also produced a *demonstrated proficiency level* θ_j for the student jth student in the data set (or simply θ when the context is clear), based on his or her "response" to each item. For brevity, this θ value will be referred to simply as *proficiency*.

6.3 Results

The demonstrated proficiency levels are different from students' scores on an assignment. A student's correctness score is determined in part by the simple percentage of reference test

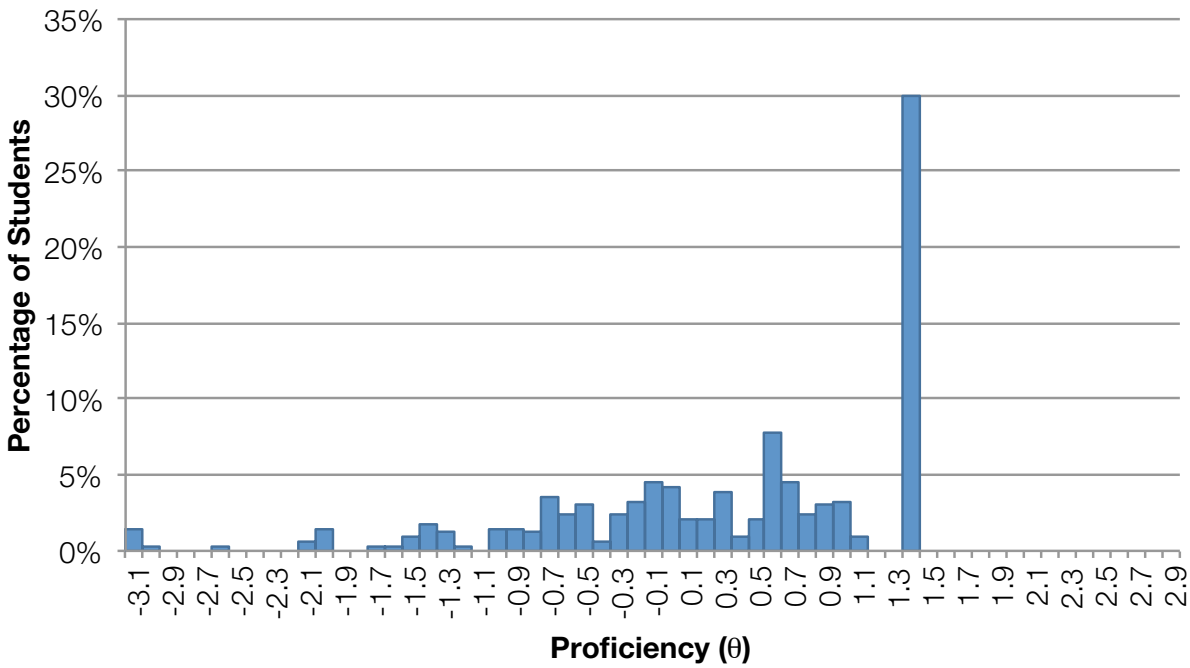

Figure 6.1: Distribution of students' demonstrated proficiencies.

cases that pass on that student's solution. The θ_j values, on the other hand, incorporate the discrimination and difficulty parameters of the items as well. Rather than falling on a $0 - 100\%$ scale, the θ_j values fall on a scale where 0 represents the mean proficiency of all students, and proficiency can theoretically range from $-\infty$ to $+\infty$. The practical range of these values, however, is typically around -3 to $+3$.

Figure 6.1 shows the distribution of proficiency levels for the students who worked on the assignment. In this example, θ ranges from a low point of -3.059 to a high point of 1.435, with a mean of 0 and standard deviation of 1.

One can see that this distribution does not have the bell shape that might be expected based on IRT literature. This can be explained by noting that I am evaluating scoring submissions that represent the end product of a long incremental development process. The responses generated by the student's solution on a reference test suite do not represent a mere snapshot of student knowledge as a traditional multiple-choice test would, but rather the culmination of that development process that was supplemented by feedback and hints from the automated grader. I should expect that students would receive high scores after several days of working on an assignment, thus biasing the distribution.

Figure 6.2 plots each student's correctness score on the horizontal axis against his or her proficiency level θ on the vertical axis. There is an extremely high positive correlation

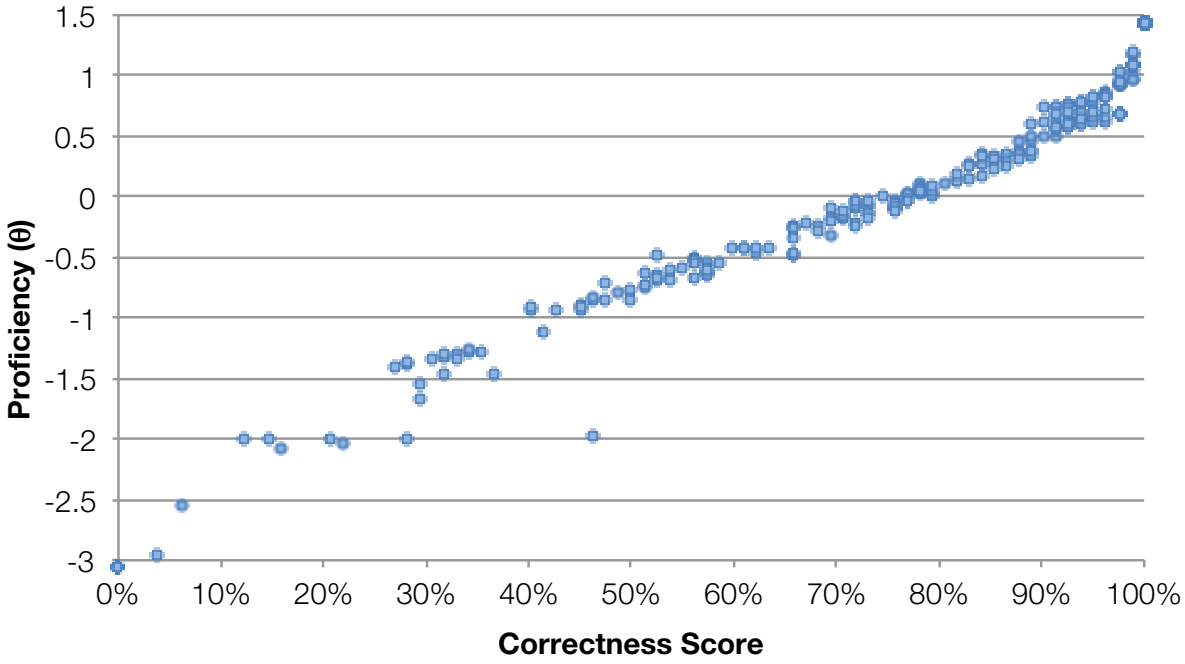

Figure 6.2: Students' correctness scores plotted against their proficiency.

($r = 0.973$) between the two scores. Recall that proficiency takes into account the difficulty and discrimination of the items, while the correctness score is the simple mean of dichotomous responses. Therefore, students with higher scores will have passed many of the same test cases, and the variability of difficulty and discrimination parameters for the test cases that they failed will be small, causing those students to be clustered together. For students with lower scores, the tests that they passed or failed may have considerably more variation in their item parameters, causing their proficiency levels to vary more as well.

6.3.1 Response Patterns

Knowing the distribution of individual response patterns can provide insight into how well the reference test suite is evaluating students. Are there far fewer response patterns than students? This would be an indication that many students are having difficulty on the same set of test cases, which would call into question the independence of those items and the strength of that reference test suite (alternatively, this could indicate possible plagiarism). Or are there many differing response patterns and each student is experiencing his or her own unique difficulties on the assignment?

Investigation into response patterns could be performed on the response matrix even before

any IRT analysis is performed, but I mention it here because the method used by `ltm` to extract the θ scores brings this information to the forefront. Students with the same response patterns will have the same θ score as a result of the estimation process; this is clear because θ is calculated based on the parameters of the items. When the `ltm` library presents these scores, it does not present scores for individual examinees; instead, it presents the scores and number of occurrences of each unique response pattern.

Out of the 334 students, there were 211 unique response patterns among their final scoring submissions. 100 students (29.9%) had an all-1s response pattern; that is, they passed every test case in the reference test suite. Five students (1.5%) had an all-0s response pattern; they failed every test case. Among the remaining 229 students, there were 209 unique response patterns. Eight response patterns were encountered by two students, one by three students, one by five students, and one by seven students. Thus, there were 198 response patterns that were only encountered by one student each in the class. This is a strong indication that most students ran into their own sets of problems with the assignment as opposed to groups of students all sharing the same problems.

6.3.2 Difficulty of Test Cases

The first parameter that I investigated is the difficulty (b_i) of the test cases in the reference test suite. The difficulty parameter represents the likelihood of a correct response on the item. In the 1PL and 2PL models, b_i is the location on the proficiency scale at which the probability of producing the correct response to the item is 0.5. This intuitive interpretation of b_i is made more complicated in the 3PL model, however, due to the c_i parameter. In the 3PL model, a student with $\theta_j = b_i$ should produce the correct response with probability $\frac{(1+c_i)}{2}$. As with the proficiency levels, the theoretical range of b_i is $-\infty$ to $+\infty$, but values in the range -2 to $+2$ are desirable so that items are not too easy or too hard for a majority of the population.

Figure 6.3 shows the difficulty parameter values for each test case in the reference test suite, sorted from highest to lowest difficulty. Since each test case is executed against two queue implementations, they are treated as separate test cases and there is a separate bar for each of these two executions.

An ideal test would have items with difficulties distributed evenly across the proficiency sale, so that there are items that function among students with low proficiency, students with average proficiency, and students with high proficiency. The reference test suite in Figure 6.3 has what I consider to be a satisfying distribution. While there are not many test cases with higher difficulties (due to the shape of the proficiency distribution discussed above), the distribution is very close to being linear. Large "clumps" of test cases at a particular proficiency level should be avoided, because they would act like a barrier where a student near that proficiency level could only pass all or none of those test cases with a certain probability.

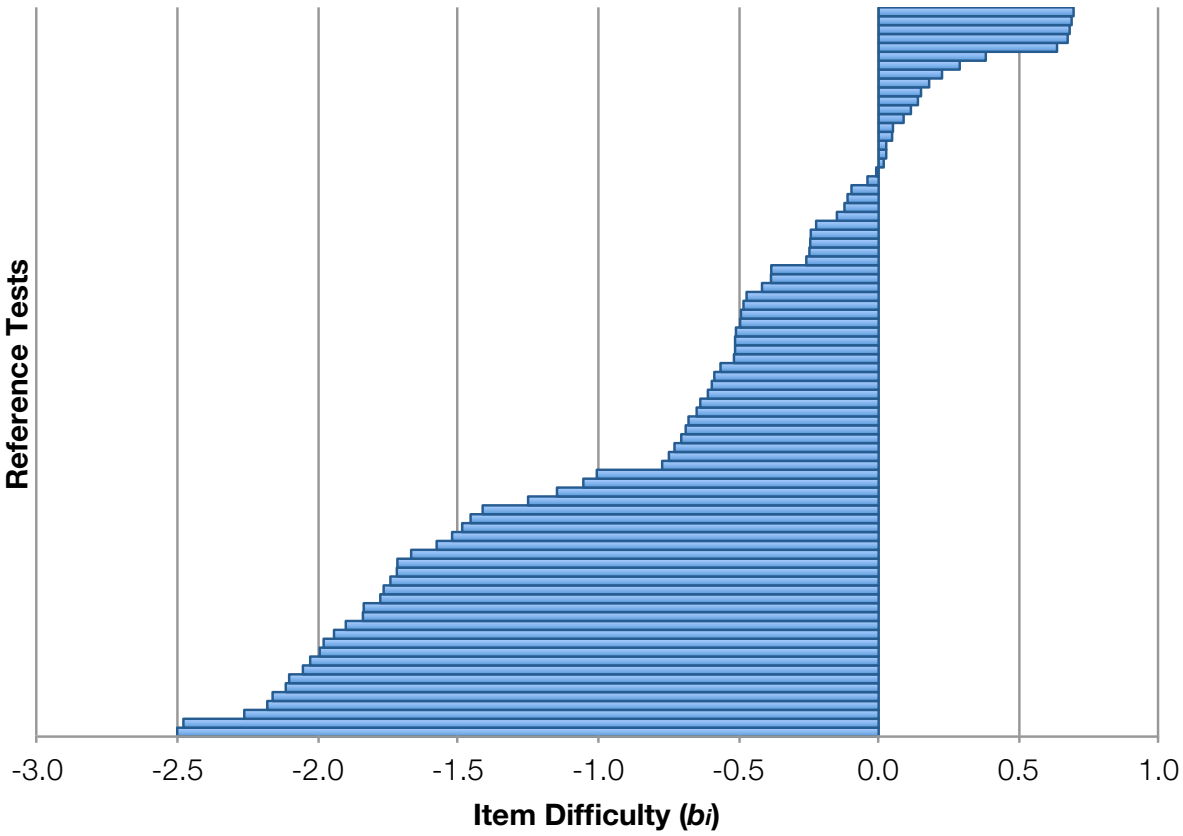

Figure 6.3: Computed difficulty (b_i) parameters for each test case in the instructor's reference test suite. Each bar represents one test case.

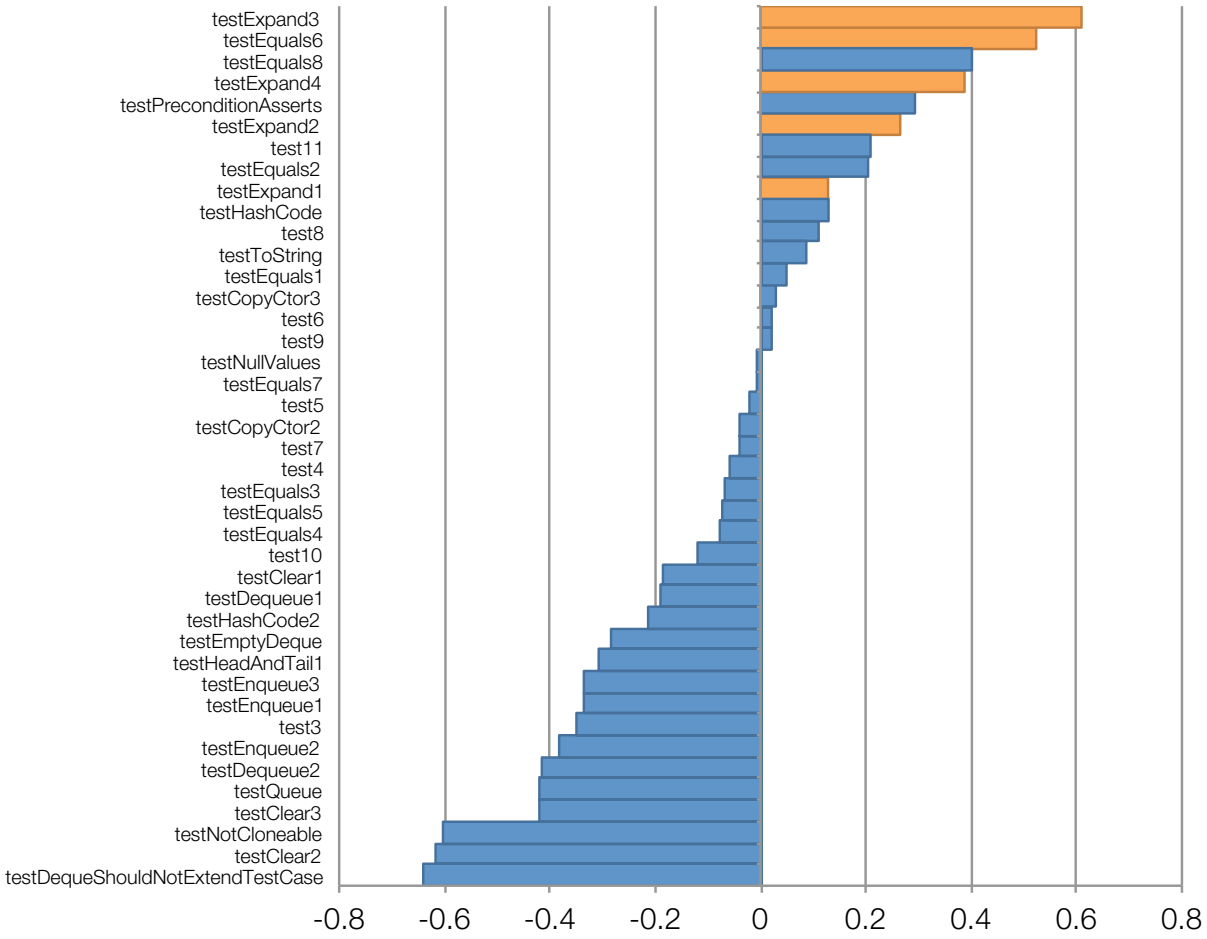

Figure 6.4: Difference between the difficulty of a test case on the array queue and the difficulty of the same test case on the linked queue. Positive numbers indicate higher difficulty on the array-based implementation. Items in orange specifically test the expansion or wraparound behavior of the array.

Recall that the reason that the queue assignment was chosen for this study was the unique nature of its reference test suite—the same set of unit tests was executed against two distinct implementations of the same data structure. Before I move on to discuss the other parameters, I will take a brief aside and explore which of the two implementation strategies students found to be more difficult.

My intuition about this assignment, based on interactions with students during office hours and my own understanding of how the data structures are implemented, is that students find the array-based implementation of the queue more difficult than a linked-list-based implementation. In the array-based implementation, they have to confront unique issues such as expanding the capacity of the array and properly managing wraparound conditions of the circular array, which are not present in a traditional linked-list-based implementation.

Figure 6.4 includes a bar for each test case in the reference test suite, but no longer distinguished by queue implementation. The length (and direction) of the bar indicates the difference in difficulty between the test case when run on the array-based queue versus the same test case run on the linked-list-based queue. Positive numbers indicate that the test had a higher difficulty on the array-based queue.

A few different observations can be made based on this figure. Of the 41 test cases, 16 had higher difficulty when run against the array-based queue; the other 25 test cases had higher difficulty when run against the linked-list-based queue. This could be interpreted to mean that the linked-list-based queue caused more complications for students, which would contradict the anecdotal claim above. One possible explanation for this is that the linked-list-based queue might be more "structurally sensitive" than the array-based version; that is, if a student made a small mistake in the references that link the nodes in the data structure, that would be more likely to cause exceptions and other failures across a wider range of test cases than a small mistake in the array-based queue. Overall, however, one implementation does not strongly stand out as being more difficult than the other.

As a final comparison of the two queue implementations, I manually examined the source code for the reference test suite and identified the unit tests that were written to specifically test the expansion or wraparound behavior of the array-based queue. These tests were still executed on both queues, but for the linked-list-based version, I would not expect them to behave in a significantly different manner to the other test cases.

The orange bars in Figure 6.4 represent these test cases. As I expected, they only appear on the positive side of the split, indicating that those specific test cases were more difficult for students on the array-based queue. This reinforces the original motivation for using IRT; I would expect these tests to be more difficult on the array-based queue because they are written to test boundary conditions that only appear in that implementation. Seeing these results reassures me that those test cases are meeting the desired objectives.

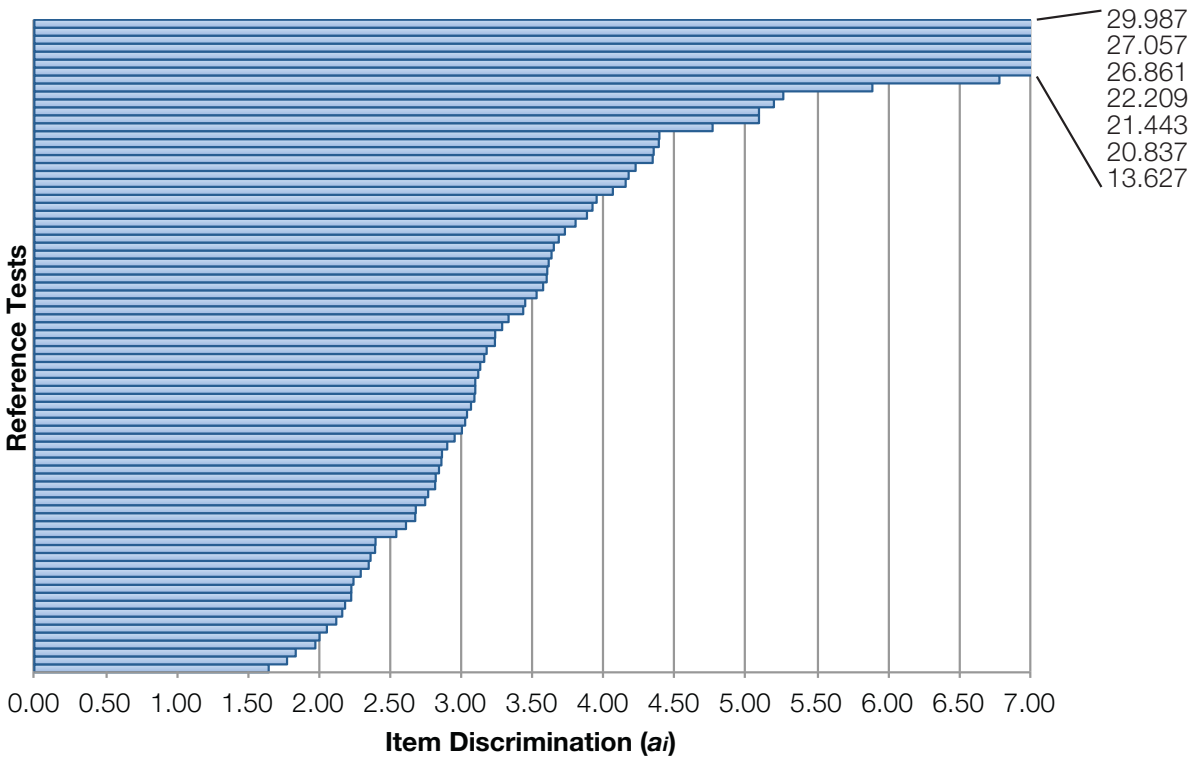

Figure 6.5: Computed discrimination (a_i) parameters for each test case in the instructor's reference test suite. Each bar represents one test case.

6.3.3 Discriminating Ability of Test Cases

The second parameter that I investigated is the discriminating ability (a_i) of the test cases in the reference test suite. The discriminating ability of an item is an indicator of how well the item distinguishes subjects with proficiency levels lower than the item's difficulty (b_i) from those with proficiency levels higher than b_i. According to [Bak85], the theoretical range for this parameter is $-4 \leq a_i \leq +4$ with values seen in practice typically within $-2.8 \leq a_i \leq +2.8$; however, it would also seem that items can considerably exceed this range, as I will show below.

Figure 6.5 shows the discrimination parameter values for each test case in the reference test suite, sorted from highest to lowest discrimination. As with the similar graph showing item difficulties, each test case is executed against two queue implementations and there is a separate bar for each of these two executions.

Constructing a test (or in this case, a test suite) primarily using items with high discriminating ability seems like a good idea in theory, but the implication in doing so is that students

Label	Range of Values
None	0
Very low	$0.01 - 0.34$
Low	$0.35 - 0.64$
Moderate	$0.65 - 1.35$
High	$1.35 - 1.69$
Very high	> 1.70
Perfect	$+\infty$

Table 6.2: Suggested labels for ranges of discrimination parameters (under a logistic model).

with proficiency levels below the minimum b_i would have extreme difficulty succeeding on *any* of the test items. Therefore, it is useful to have items with a wider range of discriminating abilities in the suite, just as it is to have a wide range of difficulty levels.

How does one interpret the value of the discrimination parameter? Unlike the difficulty parameter, discrimination is not measured on the same scale as proficiency. Guidelines can help us categorize the a_i values using meaningful labels. In [Bak85], the labels shown in Table 6.2 are suggested.

However, as I will show below when I examine individual test items in greater detail, these rigid categories may not always be suitable. Determining whether an item is a low or high discriminator may be better done in the context of the entire suite.

6.3.4 The Item Characteristic Function

The notions of item difficulty and discrimination were presented earlier in layperson's terms without discussion of the underlying theory. Item response theory dictates that one can use the estimated a_i, b_i, and c_i parameters as parameters to a mathematical function that, given a proficiency level θ, predicts the probability that a student with that proficiency level will produce a correct response on the item. This function is called the *item characteristic function* (or *item characteristic curve*).

The preferred model for the item characteristic function is the logistic function, an S-shaped (sigmoid) curve that has applications in a number of scientific fields ranging from biology to statistics. (Technically, the introduction of the c_i parameter eliminates some of the mathematical properties of the one- and two-parameter logistic models, but 3PL is still referred to as a logistical model for historical reasons.)

For a particular item i in a test, the probability of a student with proficiency θ producing

the correct response can be computed by

$$P_i(\theta) = c_i + \frac{1 - c_i}{1 + e^{-a_i(\theta - b_i)}}$$

where a_i, b_i, c_i are the discrimination, difficulty, and minimum probability parameters of the item, respectively.

From this equation, one can make the following observations. The difficulty parameter b_i corresponds to the shift of the curve along the θ (horizontal) axis. For items with positive discrimination, the function yields low probability at lower proficiency levels and high probability at higher proficiency levels. The discrimination factor a_i represents the slope of this curve as proficiency increases. If $a_i = 0$, then $P(\theta) = \frac{1+c}{2}$ for all θ; in other words, the item has no discriminating ability because subjects have an equal chance of producing the correct response or not, regardless of their proficiency. For extremely low a_i, the curve has a more linear shape, and for higher a_i, the curve remains essentially flat and asymptotically close to a lower bound of c_i until θ approaches b_i, at which point it begins climbing before flattening again asymptotically close to 1.

Figure 6.6 shows the item characteristic curves for three selected test cases in the reference test suite. Each curve is plotted against the observed student performance on the test case; the θ scale has been divided into buckets with widths of 0.5, and each bar represents the mean of all results on that test case by students in that θ range, where each result is either 0 (test case failed) or 1 (test case passed). In each case, one can see that the actual performance correlates well with the probability estimated by the item characteristic function.

The first test case in Figure 6.6 is one with both high difficulty ($b = 0.681$) and high discrimination ($a = 5.198$). The probability of students with extremely low proficiency producing the correct response is low ($c = 0.017$). This test case was one that tested students' abilities to properly store and retrieve null values in their linked-list-based queue. As I expected, it was primarily the students with high proficiency levels who managed to address this scenario and handle it properly without producing incorrect results or `NullPointerException`s that would cause the test to fail. There were no students with $\theta < -0.5$ who successfully passed this test case.

The second test case in Figure 6.6 has extremely low difficulty ($b = -2.18$) and extremely high discrimination ($a = 13.627$). The probability of producing the correct result for this test case is almost 100% for every student except for those with extremely low θ scores; only 8 out of the 334 students failed this test case, and all but one of those students had $\theta < -2.5$. This test case was one that tested the correctness of the student's `clear()` method on the array-based queue. The item characteristic curve and the actual performance results on this item indicate that most students found it very easy to achieve the correct behavior for this scenario. It is likely that the students who failed this test case were those who had major deficits in their understanding of the assignment, or who had severe enough implementation flaws that the test was not even able to execute properly.

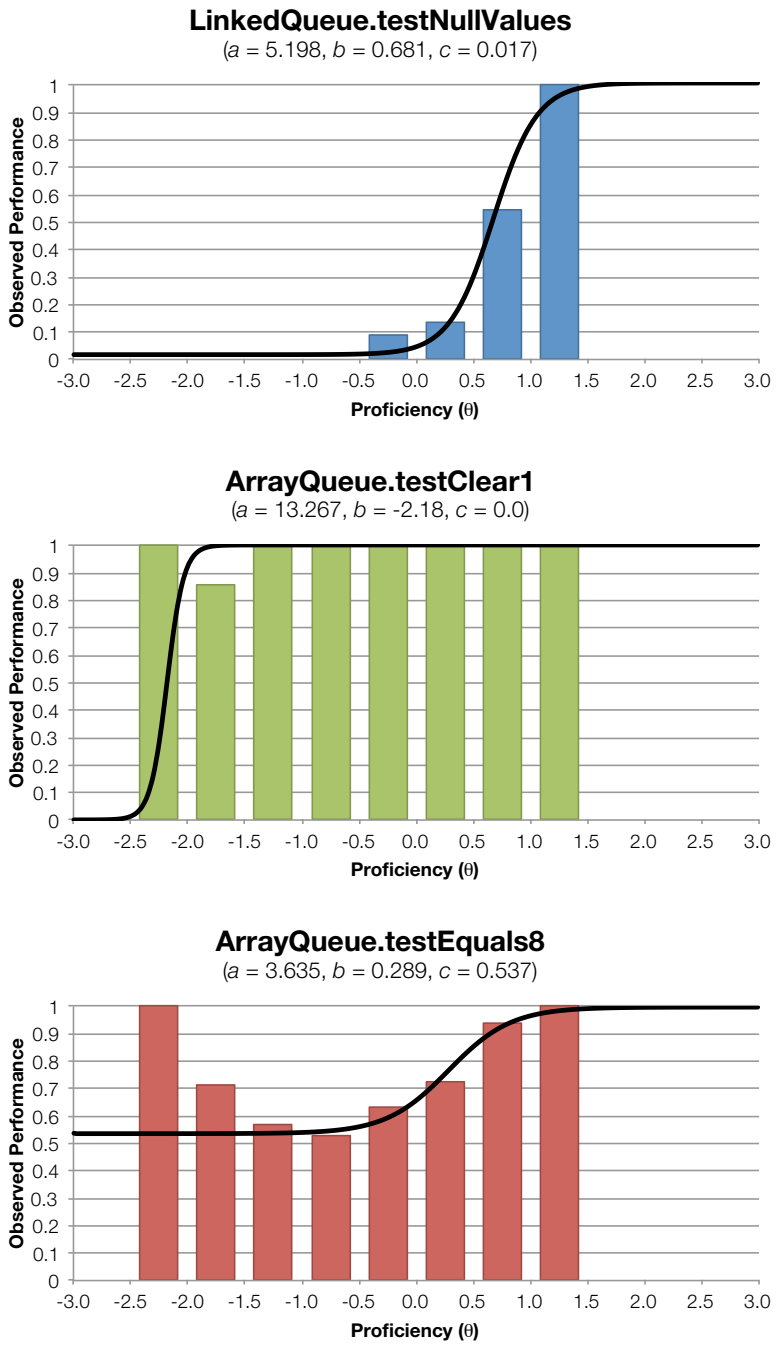

Figure 6.6: Characteristic curves of selected test cases in the reference test suite.

The third test case in Figure 6.6 was estimated to have slightly greater than average difficulty ($b = 0.289$) and average discrimination ($a = 3.635$). This test case is notable because of its considerably high asymptote ($c = 0.537$), which indicates that even students with infinitely low proficiency would have a greater than half probability of passing the test case, and the observed performance exhibits a peculiar inverse bell shape where students at both the lowest and highest proficiency levels performed well but those in the middle performed more poorly.

Notice that the discrimination value of the last test case is one that [Bak85] would have labeled as "very high" according to Table 6.2. However, the fluctuating nature of the actual performance bars leads me to claim that the discriminating ability of this test case is much lower than that, and the shape of the curve agrees with that claim. One can see the reason for this odd behavior by examining the actual test case more thoroughly, which tested the inequality of two queues; that is, the test succeeds if the student's `equals()` method returns `false` when called using two unequal queues. In Java, if the `equals()` method has not been overridden (falling back on the implementation in the `Object` class), then calling it with two unequal queues will return `false` by default. In other words, a student who chooses to not override `equals()` at all will still pass this particular test purely by chance. Similarly, due to the way by which `equals()` is implemented for collections in Java, it is much more likely that students will generate false negatives (equal queues returning that they are unequal) than it is to generate false positives (unequal queues returning that they are equal)—again, purely by chance. This likelihood is captured in the test case's extremely high c_i parameter value.

The logistic model for the item characteristic curve also demonstrates what would happen if an item had a *negative* discrimination parameter. In this case, the probability would be close to 1 for lower proficiency levels and then drop to c_i for higher proficiency levels. The reference test suite that I examined in the case study did not contain any such items, although it could be argued that the `ArrayQueue.testEquals8` item could have been modeled just as accurately with a negative discrimination factor as it was with a positive one.

There are two likely explanations of how a true negative discriminator might be discovered in a reference test suite. First, it might indicate the presence of a bug in the test case (a Boolean inversion causing an assertion to fail when it should pass, or vice versa). Alternatively, the test case might rely on program behavior that was not made explicit in the assignment specification, leaving the expected behavior up to interpretation.

The graph in Figure 6.7 shows the item characteristic curves for every test case in the full reference test suite. While the number of test cases in the suite makes it difficult to identify test cases individually, a "big picture" visualization of the suite such as this would still allow an instructor to quickly observe the existence of test cases that are problematic, such as those with extremely low or negative discriminating ability, or those with very high lower asymptotes.

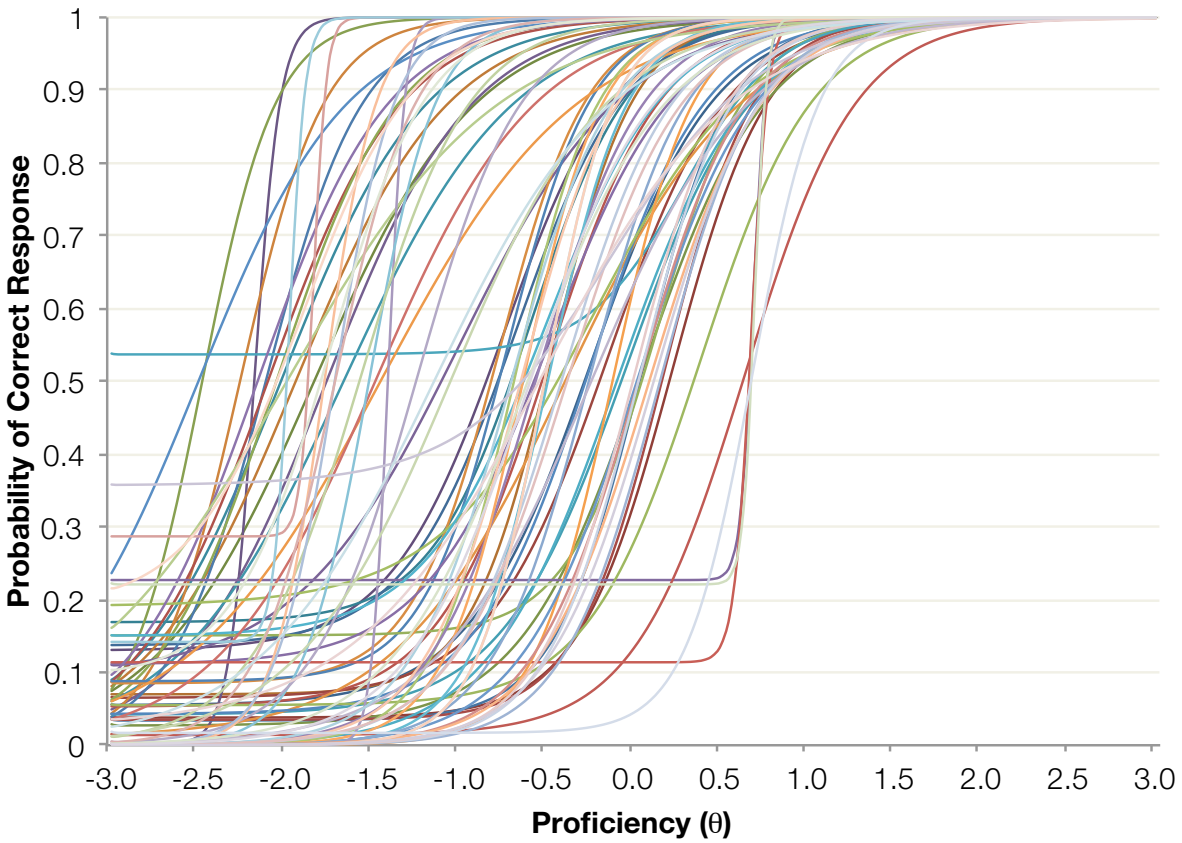

Figure 6.7: Characteristic curves for each item in the reference test suite.

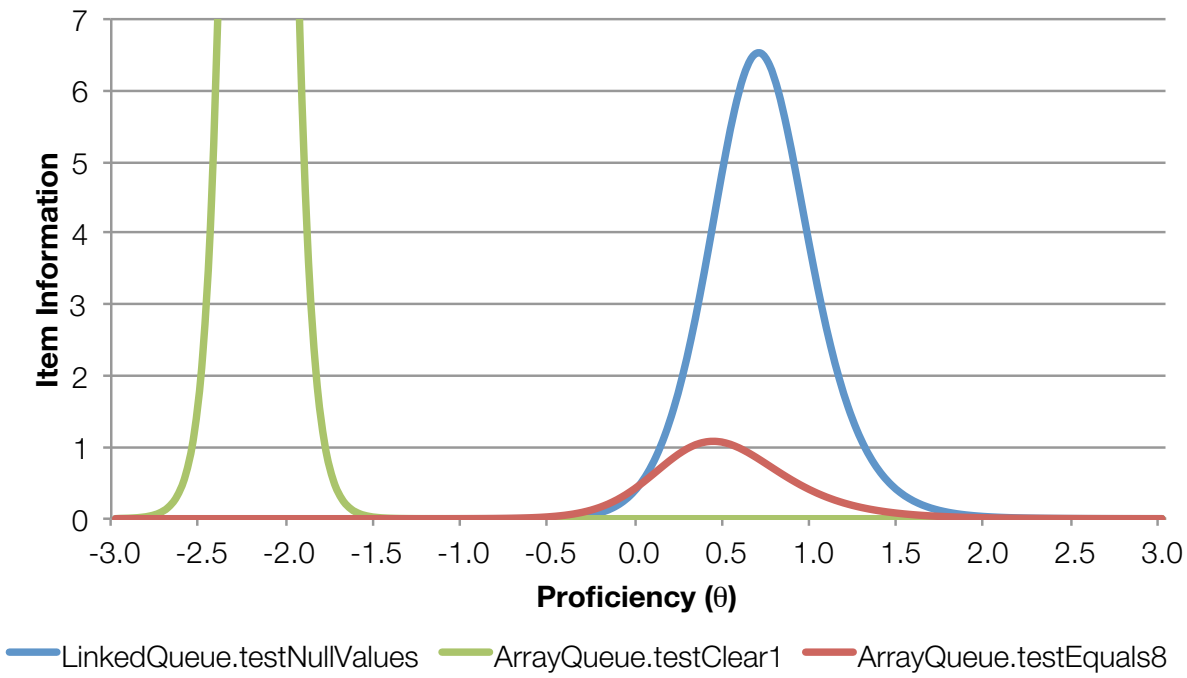

Figure 6.8: Item information curves for the three test cases in Figure 6.6.

6.3.5 Information Provided by Each Test Case

Item response theory also lets me investigate the amount of *information* provided by the items in the reference test suite. The statistical notion of information, credited to R. A. Fisher, is inversely related to the variability of a measurement. In item response theory, each item in a test measures an underlying trait (in my case, proficiency) of the examinee. Items with larger amounts of information measure that trait with a higher degree of accuracy, and items with smaller amounts of information measure it with less accuracy.

This is an extension of the concept of *reliability* from classical test theory. Unlike CTT, where reliability is a single index for each item, in IRT one can speak of an information function $I_i(\theta)$ that measures the amount of information provided by the item for students at different proficiency levels.

For the three-parameter model, the *information function* for an item is a bell-shaped curve defined by

$$I_i(\theta) = a_i^2 \frac{(P_i(\theta) - c_i)^2}{(1 - c_i)^2} \frac{1 - P_i(\theta)}{P_i(\theta)}$$

where $P_i(\theta)$ is the item characteristic function defined earlier. Under the 2PL model, maximum information occurs at a proficiency level equal to the item's difficulty, which can be seen

by substituting $c_i = 0$ into the equation above. Under the 3PL model used here, information is maximized slightly to the right of b_i, depending on the magnitude of c_i. Additionally, if a student has a greater probability of successfully answering an item by chance or by guessing, then the information provided by that item is reduced. In other words, $I_i(\theta)$ decreases as c_i increases.

Figure 6.8 shows the information curves that correspond to the test cases in Figure 6.6 (the information curve for `ArrayQueue.testClear1` has a maximum value around 45; the plot has been cut off in order to better show the shape of all three curves). These examples show how the discrimination parameter affects the shape of the curve. Items with high discrimination provide a large amount of information over a narrow range of proficiency levels, so they measure student proficiency with considerable accuracy near the difficulty level of the item but with much less accuracy further away from it. Likewise, items with low discrimination provide a small amount of information over a wider range of proficiency levels, so they less accurately measure student proficiency across that range. In the case of `ArrayQueue.testEquals8`, one can see that it provides little information over a small range of proficiency levels, which is another indicator of its weakness as a test item.

6.3.6 Information Provided by the Entire Test Suite

Since the general shape of the item information function $I_i(\theta)$ is somewhat easily predicted from the characteristic function $P_i(\theta)$, working with the item information functions individually is not terribly useful. Instead, users of IRT are typically more interested in the amount of information that the entire test provides. In item response theory, the *test information function* is the sum of the full set of item information functions,

$$I(\theta) = \sum_i I_i(\theta).$$

In statistics, standard error is the inverse square-root of information,

$$\mathrm{SE}(\theta) = \frac{1}{\sqrt{I(\theta)}},$$

so the amount of information provided by the test at a particular proficiency level also indicates the accuracy with which the test estimates that proficiency. Having more information reduces the error of the estimate.

The simple summation defined by $I(\theta)$ provides a theoretical basis for a common sense idea: The more items a test has, the more accurately it can measure a student's proficiency. I can apply the same logic to unit testing. If I add more test cases that cover more decision paths and equivalence classes of input values to my test suite and those tests pass, then my confidence increases that a solution is behaviorally correct. Note that this does *not* imply

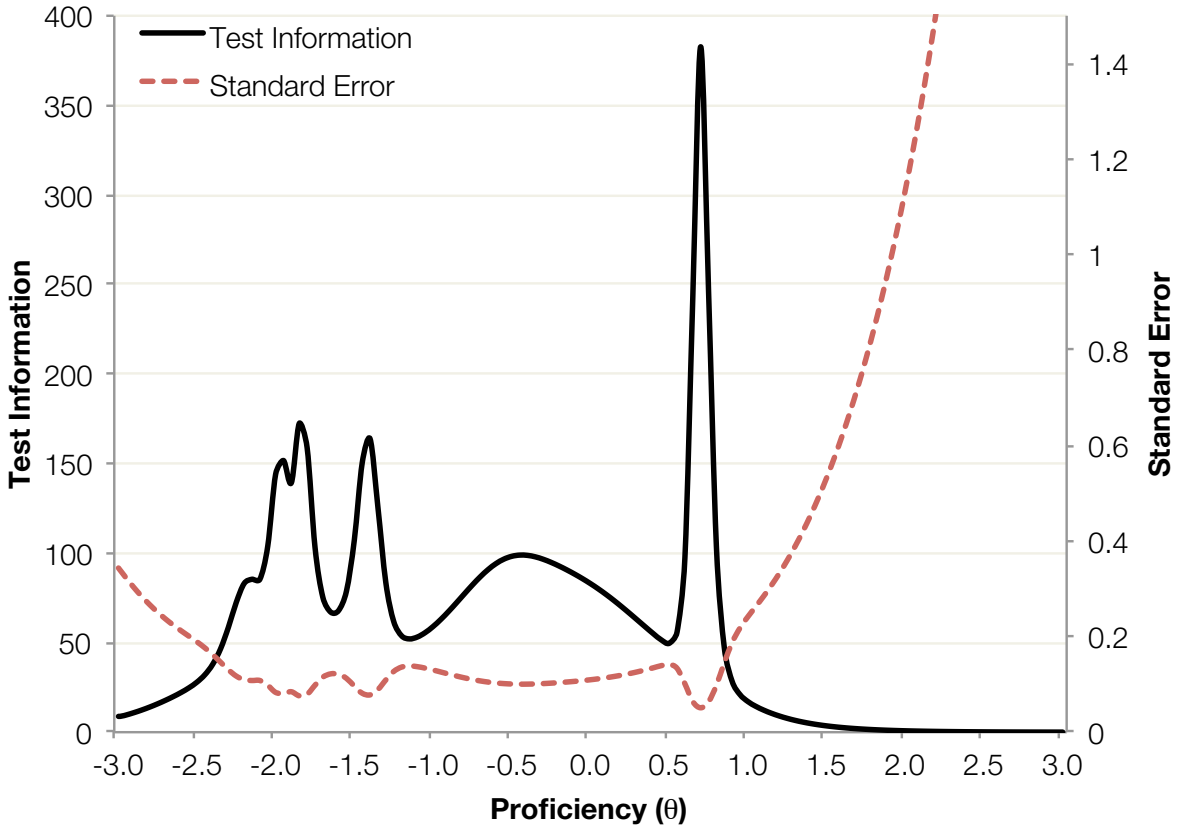

Figure 6.9: The test information curve and corresponding standard error of proficiency estimation for the reference test suite.

that I can repeat the same test case multiple times to increase its accuracy. IRT assumes that the items in the test are locally independent. From my experimentation, after duplicating an item a sufficient number of times, an equilibrium state is reached where the discriminating ability of each of the instances is equal and reduced by an amount proportional to the number of duplicates, and the test information function remains unchanged.

The definition of $I(\theta)$ also shows that the shape of the curve can be controlled by cherry-picking test items from a larger set when the individual item information is known. The shape of an ideal test information function depends on the purpose and goals of the test. If the test is intended to simply measure a student's understanding of subject material, then the ideal TIF would be a straight line that provides the same amount of information across all proficiency levels. For a higher-stakes test whose purpose is, for example, to bestow an award on students who perform at or above a certain level, the test writer can choose items that maximize the test's information at or around the target θ value.

The test information curve for the two-queues reference test suite is shown in Figure 6.9, along with the corresponding standard error of the estimate. At the extrema, the accuracy with which the test suite measures the student's proficiency level decreases. This particular information function has multiple peaks corresponding to the test cases with very large discrimination parameters when compared to the rest of the suite. Overall, however, one can see that the standard error of the proficiency estimates is fairly low across most of the θ range where students actually fell. While improvements could be made to individual items, the information and standard error curves here indicate that the reference test suite as a whole appears to be a suitable instrument for assessing the proficiency of students on this assignment.

6.4 Summary

I have applied techniques from item response theory to analyze the reference test suite for a programming assignment both to understand what parts of the assignment caused more or less difficulty for students, as well as to understand the quality of the test suite itself. In one situation where a test case showed particularly unusual behavior—success by students with low and high proficiency but poorer performance in the middle—the results gave me the impetus to take a closer look at the test case and lead me to understand how and why that behavior manifested itself.

A separate question also arises from using item response theory in this way on student programming work: Should we use IRT to change the way we score students on these assignments? It would seem natural to claim that we can use a student's demonstrated proficiency (θ) as a basis for assigning a grade that would be more accurate than a simple percentage, since this estimate takes into account the difficulty and discrimination factors of each reference test case. This is one of the advantages of using IRT over simple scoring for traditional test instruments. There are some problems with this approach, however, when the test instrument is a large-scale programming assignment. First, it would require the instructor to defer assigning scores until all work on the assignment had ended. This would cause a disparity between the feedback that a student receives from Web-CAT—the percentage of reference tests passed, with hints about the ones that did not pass—and the final score that he or she will eventually receive. While Figure 6.2 did show that the correlation between θ score and traditional score was very high, it is possible that a student's θ-derived final score could be lower than their traditional score. Students with lower proficiency would be unduly affected by this.

Perhaps more importantly, using the θ scores for assigning grades would run counter to the way that students understand the grading process, especially on programming assignments. Students view programming as a constructive activity, where the only factors at play are the quality of their solutions as assessed by a reference test suite. Using the θ scores would factor

not only the quality of the student's own solution into grading, but the quality of every other student's solution, since the difficulty and discrimination parameters are computed from the entire response matrix. This would make students feel that they are "in competition" with the other students in the course for their grade, instead of it being based on the quality of only their own work. For these reasons, I do not recommend that θ scores be used to assign grades on these types of assignments.

Instructors might also find it tempting to use the results from IRT analysis to make changes to their reference test suites. Whether this is advisable depends on the nature of the changes they wish to make. Certainly, if a test case is an extremely low (or negative) discriminator, then this indicates a problem with the test case and it should be addressed.

One might naturally continue down this path and determine that test cases with extremely low difficulty or high lower asymptotes provide little value as well. For example, one of the reference test cases with lowest difficulty and highest discriminating ability was one that did not test behavioral correctness of an operation on the queue data structure, but merely acted as a consistency check to verify that the student did not extend an inappropriate base class. Passing this test is essentially a requirement for passing any of the other tests, and most students did pass it. Those who did not either had code that failed to compile (in which case it would pass no tests), or code that was broken in some other severe manner that caused the test case to fail.

An instructor should exercise caution before eliminating test cases such as these. While it is true that they provide a minimal estimation of a student's ability, if any at all, they have a greater purpose in the reference test suite—to provide feedback to the student. For example, a student who has an otherwise behaviorally correct solution might have simply forgotten to implement a required interface. In this situation, giving the student that feedback is crucial to helping them converge on the correct solution, while eliminating this test case would deny the student that feedback. A better option for an instructor to consider would be to leave such test cases in the reference test suite but to give them scoring weights of zero.

7
Conclusions

In this dissertation, I have described an infrastructure that was added to Web-CAT, an automated grading system for computer science programming assignments, to provide access to data derived from existing required student work. A key advantage of this system is that no extra effort was required on the part of the course instructor to modify the way that assignments were specified in order to collect the assignments or the data derived from them because Web-CAT already collects a wide range of code metrics and performance statistics from student-submitted work. Furthermore, the extensibility of the system enables instructors to supplement the data already collected by Web-CAT with any additional data of their choosing by writing a custom plug-in that will be executed as part of the grading process.

I also showed how, with little effort, these data could be extracted and analyzed externally using existing statistical tools that are familiar to many educators and researchers. Once educators have access to this substantial data store, they are able to look for trends and correlations in their data with both a degree of breadth (across multiple assignments, courses, and semesters) and a degree of depth (using fine-grained metrics) that was previously intractable.

I presented three case studies to illustrate this breadth and depth and to back up hypotheses and intuition previously formulated in the classroom by actual evidence. The first case study explored time management habits of students. As computer science educators, we instinctively believe that starting earlier, as opposed to waiting until the last minute, is a more likely path to success on large programming assignments, and we constantly encourage our students to do so. By isolating students who performed well on some assignments and poorly on others, I sought out changes in their time management habits that might account for those differences. I found that when those students started even one day before the deadline or earlier, they had a two-thirds chance of achieving a grade of B or higher on the assignment. Conversely, when those students started on the due date or later, they had a two-thirds chance of achieving a C or lower. I saw similar trends when I examined the

time that students finished the assignments. Since these students were able to achieve good grades on some assignments, it seems clearer that they have the ability to do so and that their weaker performance on other assignments is due to poor time management rather than a lack of ability.

Given these results, how might instructors seek to effect positive changes in the time management habits of these students? One incentive that I tried was to offer extra credit to students who finished an assignment significantly earlier than the due date. After analyzing these results, however, I found that the extra credit opportunity did not provide the desired results; very few students started earlier on the assignments where extra credit was offered, compared to those where it was not. In fact, students started later on average on the assignments where extra credit was offered than on those where it was not. This led me to conclude that the promise of additional points was not enough to counter other outside factors that might delay their work on an assignment.

The second case study involved an analysis of a tool called Dereferee, which is intended to help students better understand how to use pointers and manually manage dynamic memory in C++. Students use the tool by adding directives to their source code where pointers are used, which injects extra code that performs much stricter checks on their pointer usage and produces improved and more easily understood diagnostics when an error would have resulted from improper, unsafe, or undefined access. When assignments were graded on Web-CAT, statistics were retained about the types and frequencies of errors that occurred, and I investigated these data to better understand where students had the most difficulty. The results perhaps countered my initial intuition about the kinds of errors that students would be most likely to encounter.

Then, since Dereferee was optional for students, I compared the performance of students who used it to those who did not. I found that, at first blush, there was a statistically significant difference between the two populations with regard to the number of errors that they had in their submitted code, and also with regard to the presence of any "latent defects" (errors that existed in their code but were never identified as such when executed). However, the effect sizes for these tests did not show a strong association, indicating that I would need to perform the analysis again with a larger sample size in order to increase my confidence.

Finally, my third case study used information derived by executing a reference test suite against student-submitted code and applied techniques from item response theory to determine which parts of the assignment students found to be less or more difficult. The assignment that I evaluated had a unique property—it involved writing the same abstract data type (a queue) twice, using two different implementation strategies. This allowed me to compare the two strategies and quantitatively determine which one students found to be more difficult. My intuition was that the circular array-based implementation, with its special cases involving wrap-around behavior, would be the more difficult strategy for students to implement, but the data did not back that up conclusively.

Using the same results from IRT analysis, I explored the quality of the individual items

in the reference test suite to determine if it was a satisfactory instrument for evaluating student proficiency on the assignment. I used both item difficulty and discriminating ability to answer this question, which led into an analysis of the amount of information provided by the item, but more importantly, the information provided by the suite as a whole. I found that, with few exceptions, the tests cases in the reference test suite for this assignment did adequately separate students with lower estimated proficiency levels from those with higher proficiency. Furthermore, across nearly the entire observed range of student proficiency, the reference test suite was able to measure that proficiency to a very accurate degree.

In each of the case studies, I discovered results that confirmed some of my intuitions regarding how students learn and how effectively we teach them. At the same time, however, I also obtained evidence that contradicted other intuitions that I had develped over multiple years of teaching. The latter is far more important than the former. When an educator has his or her intuition confirmed, it may be the case that no further action need to be taken for a particular scenario. On the other hand, every intuition that is contradicted provides the opportunity for vital adjustments and course corrections that could only be discovered using this data-driven approach. Furthermore, it is this same approach that will permit educators to re-evaluate those adjustments until the desired results are achieved.

In his column "What If We Approached Teaching Like Software Engineering?" [Lis11], Raymond Lister cites the work in our paper on student behavior [ESA+09] as one of "many examples of work that sets the target to which computing departments should aspire" regarding the use of data collection to influence teaching. This work has become the model for nearly all of the studies that I, and others on my team, have since conducted in computer science education. By making use of as much fine-grained relevant factors as possible and incorporating historical data dating back several years across hundreds of course offerings and assignments, we are able to evaluate how our students learn and understand the quality of our teaching to a finer degree and with a greater confidence than has previously been possible.

7.1 Existing Contributions

The work presented in this dissertation has already contributed a number of publications that have been accepted at conferences and in peer-reviewed journals. This work includes:

- An early discussion of the data collection issues associated with Web-CAT and a brief survey of problems that can be explored using the reporting tool [ATEPQ08].
- Exploring differences in behavior between students when they perform well and when they do poorly on programming assignments [ESA+09], a precursor to the case study in Chapter 4.

- Evaluating the effectiveness of the Dereferee tool [AEPQ09], a precursor to the case study in Chapter 5.
- Discovering common patterns in student behavior and resulting performance using "frequent episode mining" techniques [AE10].
- Collecting broad code coverage and correctness measures across multiple semesters to determine the effects of changing the switching teaching platforms [AE12].
- Determining student adherence to test-driven development practices and the effects of that behavior [BE12], and unique approaches to test-driven development [ESCS12].

7.2 Future Work

A number of work items remain that would make the Web-CAT reporting tool more approachable to educators. With Web-CAT in use at 70 institutions and rising, giving those users easier access to the data at their institutions will increase their ability to answer a wealth of questions about their students' behaviors and teaching practices, such as those described in our case studies.

7.2.1 Collecting Data Outside of Explicit Submissions

One feature that Web-CAT currently lacks is an ability to track student behavior outside of the discrete voluntary submissions made by students. Recall, for example, that the case study about time-based behavior did not track when students actually started working, but only when they made their first submission, and that their first submissions to Web-CAT already contained a large portion of their final solution.

In order to better assess the effects of certain student behaviors, Web-CAT needs the ability to track their activity outside of the snapshots that they submit for grading. As discussed in Chapter 2, there are already a number of plug-ins for development environments used by students that provide more fine-grained tracking of events including when they start working on a project, when they make changes to source files, when they compile their code, and so forth. We are currently planning to add similar infrastructure to Web-CAT. Instead of asking students to explicitly submit snapshots of their work, we would store their work in Git repositories and use IDE plug-ins to aggressively and frequently commit their edits during development. By strictly controlling the project creation process by having students "clone" an empty repository for the assignment and confining their work to that repository, we will finally have access to the student's complete development cycle and can study it in great detail.

7.2.2 Automating the Design of Reports

When the reporter was first designed, we imagined that one of its major usage scenarios would involve the live generation of charts and graphs from data on the system so that educators could quickly visualize information about their students' performance. Under this supposition, we designed the workflow in which the user designs a report using the external BIRT report designer, then uploads it to Web-CAT in order to generate reports from that template.

However, as we made more extensive use of the system in order to collect data for our own research, we found that the most common usage scenario by far was to use the reporter to simply extract a large table of student data that would then be analyzed in an external application such as JMP or Microsoft Excel.

Based on this experience, a planned future extension is to allow the user to design a simple table-based report entirely through the Web-CAT user interface, without requiring them to use the BIRT report designer. BIRT exposes its design API that would allow report templates to be generated on the fly without the need for the WYSIWYG tool, based on inputs from the user about the nature of the data to be included. This feature would make the Web-CAT reporting infrastructure significantly more accessible to most users by doing away with the external tool requirement entirely.

Through this interface, users would define the data columns that they wish to extract or compute in the same way that they do in the BIRT designer, and the corresponding table in the report template would be automatically generated, along with other required metadata. Once this report template is created, it would be stored in the template library for later reuse, just as uploaded templates are stored.

Furthermore, shortcuts could be provided for access to frequent kinds of data, or data that might require lengthy expressions to calculate. For instance, Web-CAT's data model contains a number of statistics related to the amount of code coverage achieved in a student's submission. Rather than require the user to specify each of these key paths or complex OGNL expressions manually, he or she should be able to simply choose a "Code Coverage" option that would automatically import all of the related data.

7.2.3 Automatically Scheduling Reports

Another way to encourage and facilitate data collection by instructors would be to support scheduling repeated generation of reports. The current system requires that the instructor individually request the generation of each report and specify the query to select the appropriate submissions or other entities of interest for each one. In cases where an instructor wants to respond as quickly as possible to potential difficulties encountered by students, it would save time and effort if he or she could specify which report should be generated when

the assignment is initially configured along with a schedule that defines its frequency (daily or twice daily, for example) and its duration (for X number of days, or until the assignment closes). Then, using the existing system that notifies users when a long-running report is ready, the requesting user could be e-mailed each instance of the report, effectively creating a convenient "push" model for educational data collection.

Bibliography

[AE10] Anthony Allevato and Stephen H. Edwards. Discovering patterns in student activity on programming assignments. In *2010 ASEE Southeastern Section Annual Conference and Meeting*, April, 2010 2010.

[AE12] Anthony Allevato and Stephen H. Edwards. Robolift: engaging cs2 students with testable, automatically evaluated android applications. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 547–552, New York, NY, USA, 2012. ACM.

[AEH05] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '05, pages 84–88, New York, NY, USA, 2005. ACM.

[AEPQ09] Anthony Allevato, Stephen H. Edwards, and Manuel A. Pérez-Quiñones. Dereferee: Exploring pointer mismanagement in student code. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, page 173–177, New York, NY, USA, 2009. ACM, ACM.

[ATEPQ08] A. Allevato, M. Thornton, S. H. Edwards, and Manuel A. Pérez-Quiñones. Mining data from an automated grading and testing system by adding rich reporting capabilities. In *Proceedings of the First International Conference on Educational Data Mining*, pages 167–176, Montreal, Canada, 06/2008 2008.

[Bak85] Frank B. Baker. *The Basics of Item Response Theory*. Heinemann, 1985.

[BE12] Kevin Buffardi and S. H. Edwards. Exploring influences on student adherence to test-driven development. In *Proceedings of the 17th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '12, page TBD, New York, NY, USA, 2012. ACM.

[BIR12] Eclipse birt home. http://www.eclipse.org/birt/phoenix/, 2012.

[CCL07] Course, curriculum, and laboratory improvement (ccli) program solicitation. http://www.nsf.gov/pubs/2007/nsf07543/nsf07543.pdf, 2007.

[Cha06] A. T. Chamillard. Using student performance predictions in a computer science curriculum. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, ITICSE '06, pages 260–264, New York, NY, USA, 2006. ACM.

[CLC05] Chih-Ming Chen, Hahn-Ming Lee, and Ya-Hui Chen. Personalized e-learning system using item response theory. *Computers amp; Education*, 44(3):237 – 255, 2005.

[ESA+09] Stephen H. Edwards, Jason Snyder, Anthony Allevato, Manuel A. Pérez-Quiñones, D. Kim, and Betsy Tretola. Comparing effective and ineffective behaviors of student programmers. In *Fifth International Workshop on Computing Education Research - ICER '09*, page 3, Berkeley, CA, USA, 2009. ACM Press, ACM Press.

[ESCS12] Stephen H. Edwards, Zalia Shams, Michael Cogswell, and Robert C. Senkbeil. Running students' software tests against each others' code: new life for an old "gimmick". In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 221–226, New York, NY, USA, 2012. ACM.

[GO86] L. Gugerty and G. Olson. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '86, pages 171–174, New York, NY, USA, 1986. ACM.

[HHMWW06] Darren Hayes, Jonathan Hill, Anne Mannette-Wright, and Henry Wong. Team project patterns for college students. In *Proceedings of the 2006 conference on Pattern languages of programs*, PLoP '06, pages 3:1–3:6, New York, NY, USA, 2006. ACM.

[Jad05] Matthew C. Jadud. A first look at novice compilation behavior using bluej. In *In Proc. 16th Workshop of the Psychology of Programming Interest Group*, 2005.

[Jas12] Jasperreports: Jasperforge. http://jasperforge.org/projects/jasperreports, 2012.

[JKA+04] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Qin Zhang, Aaron Kagawa, and Takuya Yamashita. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from hackystat-uh. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE '04, pages 136–144, Washington, DC, USA, 2004. IEEE Computer Society.

[KU12] Michael Kölling and Ian Utting. Building an open, large-scale research data repository of initial programming student behaviour. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 323–324, New York, NY, USA, 2012. ACM.

[LAMJ05] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '05, pages 14–18, New York, NY, USA, 2005. ACM.

[Lis10] Raymond Lister. Cs education research: The naughties in csed research: a retrospective. *ACM Inroads*, 1(1):22–24, March 2010.

[Lis11] Raymond Lister. What if we approached teaching like software engineering? *ACM Inroads*, 2(1):17–18, February 2011.

[MKLH09] Christian Murphy, Gail Kaiser, Kristin Loveland, and Sahar Hasan. Retina: helping students and instructors based on observed programming activities. In *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, pages 178–182, New York, NY, USA, 2009. ACM.

[MLRW05] Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson. Mining student cvs repositories for performance indicators. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[MR02] Iain Milne and Glenn Rowe. Difficulties in learning and teaching programmingviews of students and tutors. *Education and Information Technologies*, 7(1):55–66, March 2002.

[NBFJ+08] Cindy Norris, Frank Barry, James B. Fenwick Jr., Kathryn Reid, and Josh Rountree. Clockit: collecting quantitative data on how beginning software developers really work. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, ITiCSE '08, pages 37–41, New York, NY, USA, 2008. ACM.

[OGN12] Apache commons ognl—object graph navigation library. http://commons.apache.org/ognl/, 2012.

[Oiw09] Yutaka Oiwa. Implementation of the memory-safe full ansi-c compiler. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 259–269, New York, NY, USA, 2009. ACM.

[Pen12] Pentaho reporting project. http://reporting.pentaho.com, 2012.

[PWH00] Scott M. Pike, Bruce W. Weide, and Joseph E. Hollingsworth. Checkmate: cornering c++ dynamic memory errors with checked pointers. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, SIGCSE '00, pages 352–356, New York, NY, USA, 2000. ACM.

[Rob00] Eric Roberts. Strategies for encouraging individual achievement in introductory computer science courses. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, SIGCSE '00, pages 295–299, New York, NY, USA, 2000. ACM.

[SB12] Matthew S. Simpson and Rajeev K. Barua. Memsafe: ensuring the spatial and temporal memory safety of cat runtime. *Software: Practice and Experience*, pages n/a–n/a, 2012.

[SE11] Clifford A. Shaffer and Stephen H. Edwards. Scheduling and student performance. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 331–331, New York, NY, USA, 2011. ACM.

[SS10] Leigh Ann Sudol and Cassandra Studer. Analyzing test items: using item response theory to validate assessments. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 436–440, New York, NY, USA, 2010. ACM.

[Sto98] James Stodder. Experimental moralities: Ethics in classroom experiments. *The Journal of Economic Education*, 29(2):127–138, 1998.

[WP05] Titus Winters and Tom Payne. What do students know?: an outcomes-based assessment system. In *Proceedings of the first international workshop on Computing education research*, ICER '05, pages 165–172, New York, NY, USA, 2005. ACM.

[WP06] Titus Winters and Tom Payne. Closing the loop on test creation: a question assessment mechanism for instructors. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, SIGCSE '06, pages 169–170, New York, NY, USA, 2006. ACM.

[YPC+10] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 145–156, New York, NY, USA, 2010. ACM.