ABSTRACT

Title of dissertation: MARMOSET: A PROGRAMMING PROJECT ASSIGNMENT FRAMEWORK TO IMPROVE THE FEEDBACK CYCLE FOR STUDENTS, FACULTY AND RESEARCHERS

Jaime W. Spacco, Doctor of Philosophy, 2006

Dissertation directed by: Professor William W. Pugh
Department of Computer Science

We developed Marmoset, a system that improves the feedback cycle on programming assignments for students, faculty and researchers alike.

Using automation, Marmoset substantially lowers the burden on faculty for grading programming assignments, allowing faculty to give students more rapid feedback on their assignments.

To further improve the feedback cycle, Marmoset provides students with limited access to the results of the instructor's private test cases *before* the submission deadline using a novel token-based incentive system. This both encourages students to start their work early and to think critically about their work. Because students submit early, instructors can monitor all students' progress on test cases and identify where in projects students are having problems in order to update the project requirements in a timely fashion and make the best use of time in lectures, discussion sections, and office hours.

To study in more detail the development process of students, Marmoset can be configured to transparently capture snapshots to a central repository every-time

students save their files. These detailed development histories offer a unique, detailed perspective of each student's progress on a programming assignment, from the first line of code written and saved all the way through the final edit before the final submission. This type of data has proved extremely valuable for many uses, such as mining new bug patterns and evaluating existing bug-finding tools.

MARMOSET: A PROGRAMMING PROJECT ASSIGNMENT FRAMEWORK TO IMPROVE THE FEEDBACK CYCLE FOR STUDENTS, FACULTY AND RESEARCHERS

by

Jaime W. Spacco

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:

Professor William W. Pugh, Chair/Advisor
Professor Adam Porter
Professor Jeffrey K. Hollingsworth
Professor Atif M. Memon
Professor Kent Norman

This dissertation is dedicated to the memory of Frank “Dudley” Wallen.

ACKNOWLEDGMENTS

I want to thank Bill for having patience with me back when I didn't know how to do research, and then once I started to figure it out, for pushing me to do better work than I thought I would ever be capable of doing.

I want to thank both sets of parents for their unconditional love, support and inspiration, even if they still have no clue what it is that I actually do for a living.

Thanks to my current and former roommates at Geek House for ice cream, video games, cookouts, super bowl parties, and board games.

Thanks to everyone I've played on an intra-mural team with over the last seven years—I'll never forget stepping onto the turf at night and playing under the lights. That all meant much more to me than it probably should have... Good times, good times.

Overall, I want to thank all of my friends, here at Maryland and everywhere else, for being wonderful. All I need to say about my friends is that if I ever get to the gates of heaven and have to pick the criteria they'll use to judge me, I would say "Judge me by the quality of the people who have chosen me as a friend".

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Feedback is an extremely important part of the educational process: Students want feedback on how well they are learning the course material; instructors want feedback on how well their students are learning the course material as well as how effectively they are teaching the material; and researchers want to use feedback about the actions and interactions of both students and instructors to better understand effective ways for instructors to teach and for students to learn.

Many computer science courses have a strong programming component, where completing one or more programming assignments is an important means for students to demonstrate mastery of the course material. However, most programming assignments are evaluated using an outdated model where instructors provide feedback and a final grade only on each student's final submission, and only after the submission deadline. Because their work has already graded and the grades cannot be improved, students have little incentive to carefully review the feedback they've received on a programming assignment. Furthermore, the timing often works out such that students receive feedback from a previous assignment while already hard at work on the next assignment, giving them even less incentive to review the feedback. Finally, the final submissions are produced through an opaque process that captures very little detailed data about students' development process.

To improve the feedback provided to students, instructors and researchers, we built Marmoset, an automated snapshot, submission and testing system. Marmoset[1] improves upon the state of the art by doing three things well:

- Lower the overhead for grading by taking advantage of automation whenever possible
- Provide feedback to students *before* the submission deadline *without* simply giving students all of the test cases ahead of time (since that practice may encourage students to blindly "code to the tests")
- Transparently and unobtrusively capture as much "naturally-occurring" data about student programming practices as possible

1.1 Improving Feedback to Students

Traditionally, students receive feedback about their programming assignments through a variety of channels, ranging from informal feedback such as advice, hints or clarifications provided during lecture, office hours, lab sessions, or exchanges on a course newsgroup or wiki, to more formal feedback such as the final grade, results of running the program against test cases, or an evaluation of the programming style used in the program.

The standard practice when grading programming assignments is to give students a project description or specification and a deadline by which students must

[1]We wanted to name the project after a small, cute animal, so we picked "Marmoset" somewhat randomly. The name is not an acronym for anything and has no special significance.

complete the assignment, then collect the programs at the deadline, evaluate them, and provide grades and other feedback to the students.

The grading process usually has a high overhead, and can take anywhere from two days to two weeks. Students often do not receive their formal grade and feedback about a programming assignment until they are hard at work on the next assignment, and therefore have less incentive to review carefully the feedback from a previous assignment because it cannot change their grade.

Marmoset automates the collection and grading of submissions, an extremely important step that frees instructors from the burden of writing scripts to manage submissions, run submissions against test cases, record the results, compute final grades based on these results, and return the results to the students. The time saved can then be spent helping students with the aspects of programming with a higher cognitive load, such as program design, efficiency, or programming style.

Many systems exist that help automate the grading process to various degrees [54, 15, 24, 12, 25]; however, none of these systems address the issue of providing feedback to students *before* the submission deadline, when students can best use that feedback to improve the quality of their programs as well as their grade on the assignment.

Providing feedback to students *before* the submission deadline is crucial—because students can still improve their grade, they have incentive to pay careful attention to the feedback. On the other hand, we believe (and all other educators we've spoken to agree with us) that simply giving students the complete set of test data their programs will be expected to handle encourages students to "code to the

tests". One particularly bad habit students often adopt when given the complete set of test inputs is programming by "Brownian motion", where students make a series of small, seemingly random changes to the code in the hopes of making their program pass the next test case.

To solve this problem, we have devised a feedback system called "Release Testing" that provides feedback about the outcomes of executing student programs against instructor-supplied test cases. Release testing is limited in that it is subject to a token-based incentive system that rewards students for beginning work early and for writing their own test cases.

1.2 Improving Feedback to Instructors

Instructors want to know how well students are learning the material, and how well they themselves are teaching the material. In the context of programming assignments, instructors would like to be able to ask questions such as:

- How many students have passed at least half of the test cases?
- The project is due in three days; how many students have not started?
- Are there any test cases that very few or no students are passing?
- Are there test cases that test parts of the specification that students seem to be misinterpreting?

However, the typical feedback mechanisms available to instructors, such as questions raised during lecture, office hours, lab sessions, or on a course newsgroup

or wiki, final submissions and final grades, and course evaluations, are too coarse-grained, incomplete and inadequate to answer any of the questions listed above in any meaningful way. Furthermore, these feedback mechanisms are subject to bias by a vocal minority of students whose concerns do not necessarily represent the concerns of the rest of the class.

We have designed Marmoset such that students are encouraged to submit early and often. In addition, we store all submissions as well as the outcomes of running each submission against the instructor provided test suite. Instructors can access this rich data source to get overviews of the progress of the entire class that answer questions such as those listed above. Instructors can then use this information to identify difficult test cases, and adjust lecture or lab sessions as appropriate to cover the material students are struggling with.

1.3 Improving Feedback for Researchers

Recently, working groups in the computer science education community have begun rigorous studies of novice programmers in order to assess what novices are learning and ultimately to determine the best way to teach novices. Unfortunately, these studies have been expensive to conduct, requiring either carefully crafted questions and a standardized marking system so the questions can be administered at several institutions (see McCracken at al [38]), or time-consuming "think aloud" studies (see Lister et al [33]) where students are encouraged to talk aloud while working on problems so that their vocalizations can be recorded, transcribed and later analyzed.

In addition, these studies require students to work in a closed environment such as a timed lab session or examination, and therefore gather little data about how students work independently, in an untimed fashion and with access to whatever resources they normally use for programming, such as textbooks, online APIs, or Google.

To learn about how students work "in the wild", we must unobtrusively collect data from students as they are working on programming assignments. The data available to researchers under the traditional model of assigning and grading programming assignments consists mainly of the feedback mechanisms available to instructors, such as questions raised during lectures or office hours, course evaluations, and final submissions. This data is useful, but ultimately inadequate: Final submissions are too coarse-grained, and contain too little information about the development process, to enable studies of novice programming practices.

To remedy this situation, we have built the Course Project Manager [46], a plugin for the popular Eclipse Integrated Development Environment (IDE) that captures to a central repository regular snapshots of student code every time students save their files. These fine-grained snapshots grant researchers a view with an unprecedented level of detail at the development history of hundreds of students implementing dozens of projects.

The resulting dataset has proved useful for a variety of purposes, such as mining new bug patterns and evaluating and tuning selected bug detectors in FindBugs [16], the popular open-source bug checker; the work evaluating FindBugs represents one of the first comprehensive analyses of false negatives as well as false positive in the static error detection community.

1.4 Thesis

The thesis for this research is as follows:

> We can greatly improve the feedback provided to students, instructors, and researchers by building and deploying a system that takes advantage of automation whenever possible, provides feedback to students *before* the deadline, and transparently collects students' "naturally-occurring" data.

1.5 Contributions

This research makes the following contributions:

1. We describe building and deploying Marmoset, a system for automated testing of student programming assignments and fine-grained data collection of students' development process.

2. We conduct a multi-institution survey of computer science educators' grading practices for programming assignments in order to shed light on how assignments are assessed and how the graded parts of an assignment are weighted.

3. We introduce "release testing", a token-based incentive system that provides students with limited feedback of the outcomes of running their submissions against the instructors' test suite.

4. We describe the Course Project Manager, a plugin that transparently collects students' "naturally-occurring" data.

5. We evaluate the precision and recall of select bug detectors in the open-source static checker FindBugs [16] using the Marmoset dataset.

6. We study programming assignments designed to encourage students to adopt Test-Driven Development (TDD), and report on our experiences.

Chapter 2

Design and Implementation

In this chapter, we peek behind the curtain at the underlying design and architecture of Marmoset. We specifically discuss how the design of Marmoset addresses security, robustness and scalability.

2.1 Overview

We designed Marmoset to test student submissions in a fully automated and robust manner. We had the following design goals in mind when building and improving the system:

- No human intervention is required: Once the system is configured, Marmoset tests student submissions, whenever they are submitted by students, fully automatically.

- Poorly written student code should not cause any disruptions to the system: Code with infinite loops or deadlocks cannot hang the system, and buggy code should not crash the system.

- The system should not require regular downtime for maintenance and/or reboots. Once the system is up and running, it should continue running, without any human intervention, unless a bug is discovered and a patch needs to be

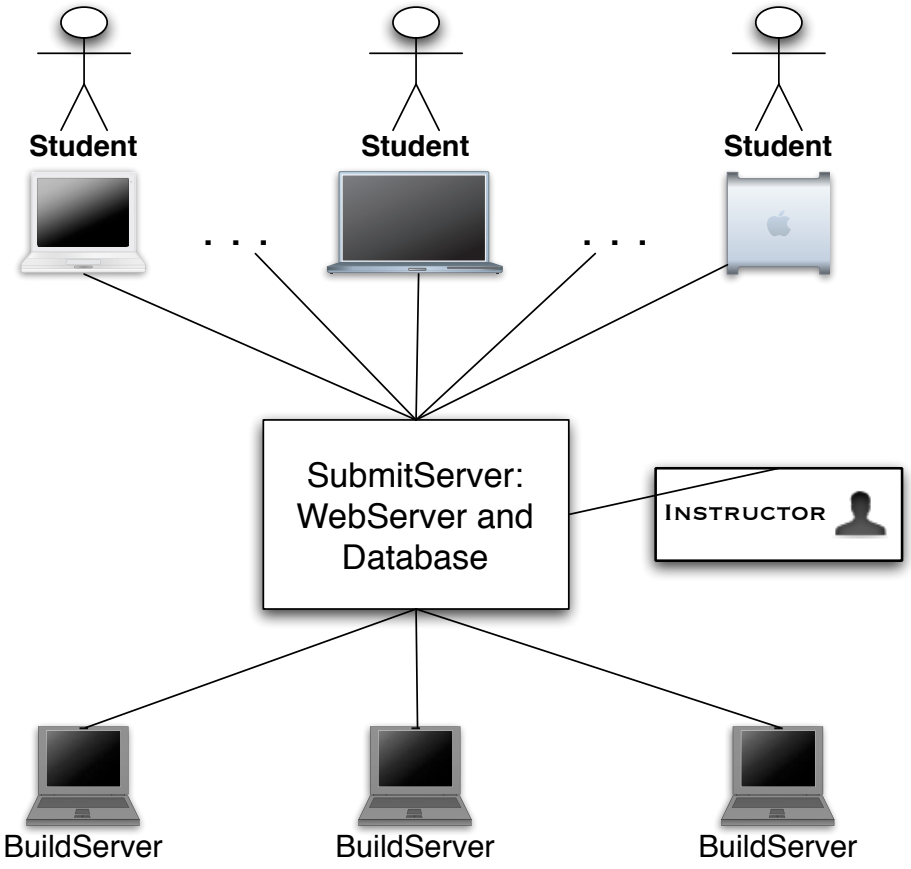


Figure 2.1: Overview of the design of Marmoset

rolled out.

- Whenever possible, the system should detect and correct errors due to hardware failure or other system-level issues.

- Even if there is a bug in Marmoset or in an instructor's test suite, Marmoset should do something sensible from the perspective of the students.

Our goals are for Marmoset to be able to test any (potentially buggy) student code at 3:00AM without requiring any human intervention or other type of maintenance. This turns out to be a non-trivial engineering task that required a careful

design. Figure 2.1 shows the overall design of the Marmoset system, which consists of a web front-end called the *SubmitServer* and a database both running on one host, and a series of distributed *BuildServers* running on one or more other hosts. The design of the SubmitServer is described in Section 2.2 and the BuildServer is described in Section 2.3.

2.2 SubmitServer

The SubmitServer is a web application that runs on top of a *servlet container*. The codebase is targeted at Apache Tomcat [2], but other servlet containers, such as Resin [6], can be used with minimal configuration changes.

The SubmitServer essentially displays views of the data stored in a database. The database is normally located on the same physical host machine as Tomcat, providing additional security as the database can then be configured to only accept connections from localhost (particularly useful for versions of MySQL [41] prior to 5.0, as they do not support SSL by default).

2.3 Distributed BuildServers

A BuildServer is a daemon process that periodically connects to the SubmitServer, requests a submission to run against a set of test cases, performs the necessary testing, and then returns the results to the SubmitServer.

A BuildServer does not maintain any state between testing one submission and testing the next one; therefore, many BuildServers can connect to the same

SubmitServer to distribute work more efficiently.

If a machine running a BuildServer goes down, new BuildServers can be quickly brought online on a different machine so that students do not experience any down-time. Similarly, a single machine can host multiple BuildServers in order to better harness multiple CPUs. Similarly, if there is a backlog of new submissions to be tested (such as close to a project deadline, the re-test of many submissions triggered by an instructor uploading a new test-setup, or at the end of a semester when all CVS snapshots are dumped into a research database) then dozens or hundreds of BuildServers can be started, for example by using a cluster, to quickly and efficiently test many submissions in parallel.

If a BuildServer or its host machine crashes while testing a submission, the SubmitServer will eventually "time out" that submission and send it to a different BuildServer to be re-tested. In this manner, Marmoset automatically detects and adapts to changing conditions in the network.

2.3.1 Handling Inconsistent Test Cases

Marmoset is a production system, used by students, TAs and faculty at all hours of the day and night. The system tests submissions fully automatically, without oversight or supervision. We have tried to design the system to detect potential problems and bring them to the attention of a Marmoset administrator.

Hardware failures, such as network outages or machine crashes, are inevitable in any production system like Marmoset, and should be anticipated and handled gracefully whenever possible. If a major hardware or network failure takes place,

for example if the machine hosting the SubmitServer web application crashes or the network path leading to this machine is disrupted, the system becomes very clearly unusable, and very little can be done to recover from this sort of error short of restoring the SubmitServer database from backups and reinstalling the SubmitServer web-application onto another machine. On the other hand, if a machine hosting a BuildServer crashes or otherwise becomes unusable, other BuildServers can be quickly and easily brought online on other machines to handle the work.

A more difficult error to detect and correct occurs when one or more test cases return incorrect outcomes for a student submission. These type of intermittent problems typically manifest as a test case failing that should have passed, although it is possible for a test case to pass that should have failed. This can happen for two main reasons:

- Network or hardware failure: Some test cases rely on external resources such as an external website, or files located on the filesystem, and can occasionally fail due to an intermittent network outage, a momentary spike in the load on a machine, and so on.

- A bad test case that returns inconsistent results: Instructors rarely design inconsistent test cases for single-threaded projects, but it does occassionally happen in practice. Furthermore, testing multi-threaded programming assignments often necessitates inconsistent test cases.

Incorrect test outcomes due to intermittent reliability issues are less likely to plague instructors who are writing and running their own grading scripts because

they will likely notice any such errors immediately and either choose another machine that does not exhibit the problem, or re-run the test cases to ensure their correctness.

Because Marmoset is constantly testing submissions without human oversight, it is more likely to experience an intermittent hardware problem, such as a router briefly going offline and dropping packets, or a spike in the load on a machine making processor or filesystem resources to temporarily appear unavailable. It is difficult to predict when these types of errors are going to occur, and they are unlikely to occur on multiple runs of the same code. Thus, Marmoset runs each test case multiple times to ensure the accuracy of the results.

In order to detect incorrect test outcomes, Marmoset constantly performs *background retesting* by re-testing an old submission when there are no new submissions to be tested. If the results of a retest are the same as the original, Marmoset notes that there has been one consistent re-test; if the results are different, the system keeps the set of new results and notes that there has been one inconsistent background re-test. After a submission has been retested 5 times, we stop testing that submission. Instructors can then look for evidence of inconsistent test outcomes by for example listing all the submissions that have returned inconsistent results, or all test cases that have re-tested inconsistently. Thus far, the vast majority of inconsistent test results have been due to test cases that use threads (common in some junior and senior-level courses that use Marmoset), bugs in Marmoset itself, or hardware/network outages.

Currently, we do not attempt to perform any sort of automatic corrections based on the results of inconsistent retests—we simply point the instructor to any

inconsistent retests. In this manner, background retesting is a simple means of establishing *confidence* in test outcomes, something that, like the errors in instructor test suites discussed in Section 4.2, we had never considered prior to developing Marmoset.

The background retests help Marmoset behave more like a fault-tolerant production system and less like ad-hoc grading scripts. Other automated grading systems, such as Steve Edwards' WebCAT [12], focus on the pedagogical and/or labor-saving benefits of such systems, and make no mention of detecting intermittent errors and scant mention of designing and operating a robust system.

2.4 Security

When designing a system with security in mind, we are really managing risk, as no system is ever 100% secure. We took as many reasonable precautions regarding security when designing Marmoset, such as:

- Security Manager: Student-written Java code runs in a security manager that prevents potentially insecure operations such as opening sockets, executing shell commands, or reading/writing files, except where the instructor grants specific permissions required by test cases

- Unprivileged account: Student-written code in other languages, such as C or Ruby, runs under an unprivileged account can only write in the directory where it will run student code, but cannot read the parent directory and therefore cannot navigate into other directories. This approximates the behavior granted

by the 'chroot' program (which makes the current directory appear to be '/') but avoids the complications of compiling and linking against standard libraries in `/usr/include` or `/usr/lib`.

- Student code is executed on a physically separate machine from the webserver and database: The distributed design of the BuildServers means that student-written code is always executed on a physically separate machine from the machine hosting the production SubmitServer web application and the SubmitServer database. Thus, if students submit malicious or broken code, the worst that can happen is that one of the BuildServer machines will crash or be compromised; student data such as grades and test outcomes are not at risk. Furthermore, because every single student submission is stored in the SubmitServer's database, we have a record of any malicious[1] code that is submitted and can use log messages to quickly identify the perpetrator.

- Use Secure Sockets Layer (SSL) to encrypt all traffic to the SubmitServer web application and between the SubmitServer and the BuildServers. This means that all passwords are encrypted, which is particularly important when students and faculty are working in a wireless environment.

- Regularly run static analysis tools such as FindBugs [16] or Fortify [17] on the Marmoset codebase to look for vulnerabilities. Fortify is particularly helpful for this purposes as it scans for vulnerabilities that are specific to servlets and SQL databases.

[1]Thus far, we have not seen any evidence of malicious submissions.

After four semesters of use (Fall 2004 through Spring 2006, plus summer and winters sessions), we have not encountered any malicious code submitted by students. The security of Marmoset could of course be improved, but at this point we have taken all of the reasonable precautions we could think of, and would need to perform a cost-benefit analysis to justify further efforts in this area.

Chapter 3

Release Testing

In this chapter, we discuss the motivation for and implementation of *release testing*, a novel incentive-based feedback system for programming assignments.

3.1 Motivation

Instructors typically evaluate *functional correctness*[1] (i.e. does the program do what the specification requires it do, and not do what the specifications requires it doesn't do?) of student programs by executing them against a series of test cases[2].

When grading programming assignments for functional correctness, one major question always arises: How much feedback, if any, should students receive about their work *before* the assignment deadline? At one extreme, students can receive no feedback at all, and at the other extreme, students can see the results of running all of the instructor's test cases against their submission. The standard compromise is to partition the test cases into *public* test cases that are given to students before the deadline and demonstrate the types of inputs that student programs are expected to pass, and *secret*[3] test cases that exercise the code more rigorously and are not

[1]Student programs are also evaluated along other dimensions, such as *style*, but these other grading concerns are orthogonal to this discussion, and for the purposes of this chapter, "grading" specifically refers to the evaluation of functional correctness.

[2]In this document, *test cases* will be used interchangeably with *test data* and *test inputs*.

[3]At other institutions, secret tests have been referred to as private tests, instructor tests or

given to students until after the deadline.

We want to provide students with feedback *before* the assignment deadline regarding how close their work is to passing all of the secret test cases (and therefore fulfilling all of the functional correctness requirements) without simply giving all of the test cases to students [4].

While the public/secret partition works reasonably well in practice, there are still some drawbacks. First, instructors often write the secret tests after the project deadline passes because there is no incentive to do so any sooner, which can lead to instructors assigning features that will be difficult to evaluate. Also, because the public tests are distributed to the students, instructors have no direct way of accessing the outcomes of executing student code against the public tests and therefore cannot easily find places where the specification is ambiguous and modify the public tests accordingly. Next, instructors may make the public tests the "easy" tests and keep the complicated or "tricky" test cases secret. This can surprise students, as they may pass all of the public tests and submit their work confident that they've done well, then find out a week later that they have failed all of the secret tests. Students can become especially frustrated when post-deadline feedback from secret tests reveals a simple misconception, misinterpretation, or error that could have been fixed quickly or easily, but the feedback has come too late to make

grading tests.

[4] Virtually all educators agree that giving students unrestricted access to *all* the test cases before the deadline encourages "coding to the tests", whereby student programs handle the specific test inputs but few other inputs. Isaacson [24] describes this problem, although it is undoubtedly discussed in many other places and other contexts.

a difference in the student's grade.

In an ideal world, students receive their grades and post-deadline feedback and use this information to learn from their mistakes. In reality, both students and instructors have moved on to the next project by the time students receive post-deadline feedback, and, with the exception of grading mistakes, students find little incentive to focus on an old project whose grade cannot be changed when there are fresh points still to be earned implementing the next project.

In the next section, we describe our mechanism to providing students with limited feedback to secret test cases, and discuss some of the implications.

3.2 The Release Testing Mechanism

In addition to the *public* and *secret* tests, Marmoset provides a new category of test case called a *release test*[5]. Release tests differ from public tests in that they are stored on a central server and are never distributed to the students; release tests differ from secret tests in that the results of release tests are selectively made available to students according to a *release policy*.

To be eligible for release testing, a student's submission must first pass all of the public tests. Because public tests are distributed to students, they should already know whether or not their submission passes all of the public tests before uploading their work to the server.

[5]The term *release test* comes from industry, where development teams need to decide when software is ready for release, either to the Quality Assurance team, to the alpha testers, to the public, or wherever.

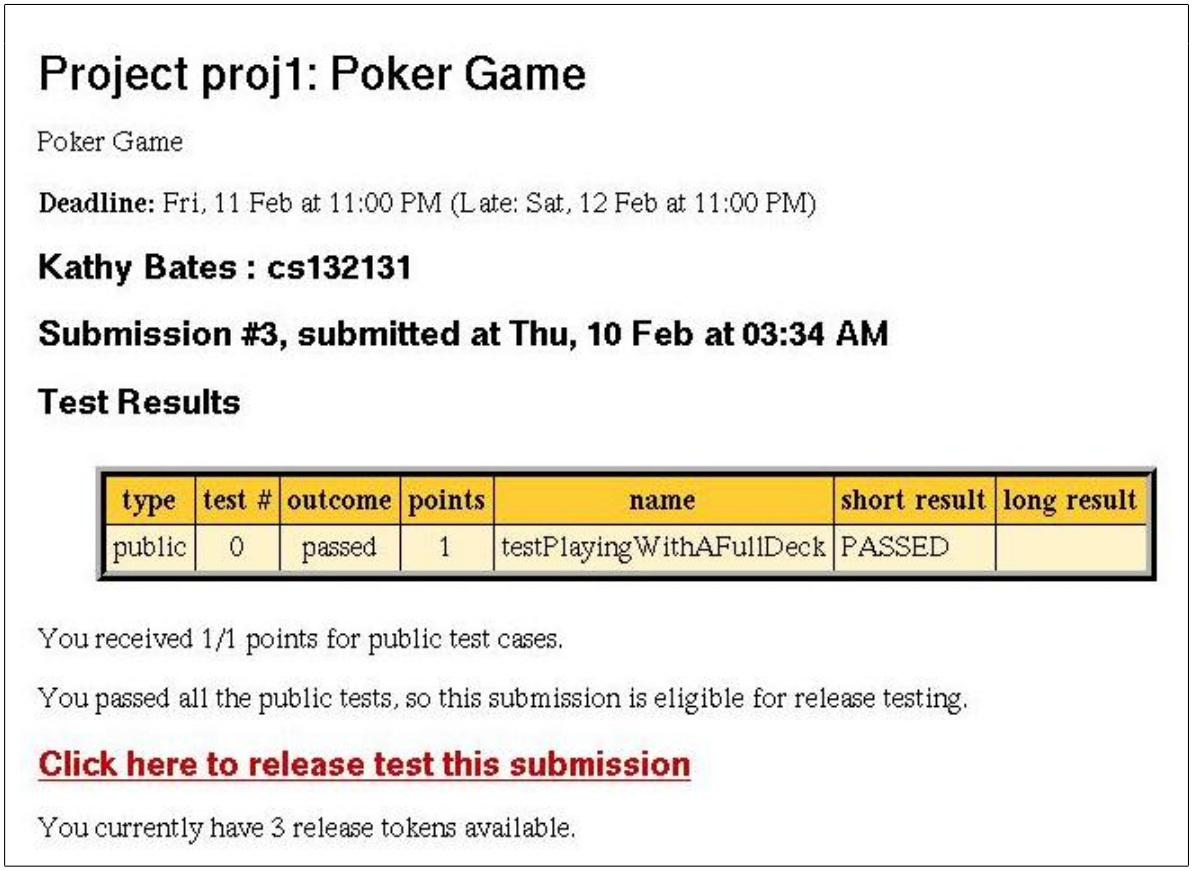
When a student is eligible for and requests release testing for a particular submission, the Marmoset system reveals to the student the *number* of release tests the submission passes and fails and the *names only* of the first two failed release tests (if any). For example, for a Poker hand evaluator project, students might be told that their submission passed 6 release tests, failed 4 release tests, and that the names of the first two failed test cases are `testFlush` and `testFourOfAKind`[6].

In addition, students are limited in how often they can request release testing. Each release test consumes a *release token*; students receive a fixed number of tokens, and each token regenerates after a fixed period of time. For example, the default configuration grants students 3 release tokens, each of which regenerates 24 hours after use.

All of these parameters can be configured on a per-project basis; we chose to reveal the names of two release tests and to give students three release tokens arbitrarily, and have not conducted any studies to determine whether these parameters are optimal.

Figure 3.1 is a screenshot of Marmoset's display of the results of the Poker hand evaluator project before a student requests release testing, while Figure 3.2 is a screenshot of what is displayed to the student after requesting release testing. Note that after requesting release testing, the student only has 2 release tokens remaining, as the token just spent requires 24 hours to regenerate.

[6]Currently Marmoset displays only the names of the test cases. However, it would not be difficult to to display the name of the test and an instructor- supplied "hint".



Project proj1: Poker Game

Poker Game

Deadline: Fri, 11 Feb at 11:00 PM (Late: Sat, 12 Feb at 11:00 PM)

Kathy Bates : cs132131

Submission #3, submitted at Thu, 10 Feb at 03:34 AM

Test Results

type	test #	outcome	points	name	short result	long result
public	0	passed	1	testPlayingWithAFullDeck	PASSED	

You received 1/1 points for public test cases.

You passed all the public tests, so this submission is eligible for release testing.

Click here to release test this submission

You currently have 3 release tokens available.

Table 3.1: Screenshot of Marmoset's display of a release-eligible submission *before* requesting release testing.

Project proj1: Poker Game

Poker Game

Deadline: Fri, 11 Feb at 11:00 PM (Late: Sat, 12 Feb at 11:00 PM)

Kathy Bates : cs132131

Submission #3, submitted at Thu, 10 Feb at 03:34 AM

Test Results

type	test #	outcome	points	name	short result	long result
public	0	passed	1	testPlayingWithAFullDeck	PASSED	
release	1	failed	1	testFlush		
release	2	failed	1	testFourOfAKind		
release	?	failed	?	?		
release	?	failed	?	?		
release	?	failed	?	?		

You received 1/1 points for public test cases.

You received 4/9 points for release tests.

You currently have 2 release tokens available.

Release token(s) will regenerate at:

- Thu, 22 Jun at 11:27 PM

Table 3.2: Screenshot of Marmoset's display of a release-eligible submission after requesting release testing.

3.3 Goals of Release Testing

The token-limited restrictions imply that if students wait until the day before a project is due to begin working, they can only use 3 release tokens. This is a very concrete incentive for students to begin working early because they can utilize more release tokens, receive more feedback, and ultimately earn better grades on programming assignments.

Release testing also has two goals that are more subtle than the obvious incentive to start early: To encourage students to think critically about their code, and encourage students to write their own test cases.

Even a student who begins every assignment early will eventually run into the following situation: "...Only one precious release token remaining... No release tokens will regenerate for many hours... Am I really ready to use spend this final token?". Students are regularly required to ask themselves: Have I really done *everything* I can to debug my code such that burning this final release token will reveal new information? This situation causes students a good deal of anxiety; however, it also encourages students to think carefully and critically about their code before spending a release token. We hope that students will learn to think carefully about their programs as a matter of course, and begin to see critical thinking as a normal part of the software development cycle.

Encouraging students to write their own test cases is the other major goal of the release testing process. A release test merely gives students the name of the test, such as `testFlush`; it does not give concrete information such as a stack trace or

the line number where the error happens. We tell students (and instruct our TAs to tell students) that the best way to figure out why their code doesn't pass `testFlush` is to write their own test case that checks for a flush. This requires that students understand at a high level the expectation of the test case [7], and then figure out how to validate their own code against these expectations—in short, students are encouraged to learn to write good test cases on their own.

Students can benefit from learning to write test cases in many ways; in Section 8.5, we review recent literature on the benefits of introducing testing into the computer science curriculum. Edwards [14] provides perhaps the most eloquent and succinct description of the student's primary benefit from learning to test their own code thoroughly: it helps them change their development methodology from "trial-and-error" to "reflection-in-action".

Writing good test cases is empowering for students because it provides them with a rigorous methodology for debugging, as Edwards discusses [14]. Furthermore, the mentality encouraged by considering method pre-conditions and post-conditions and the circumstances under which they may be violated is similar to constructing a type of empirical proof that their program is correct. To push this analogy farther, if a program is a proof that solves an interesting problem, then a thorough test suite is a meta-proof that the first proof is indeed correct.

In Chapter 6, we discuss in detail additional strategies for motivating students to write their own test cases.

[7] Marmoset is dependent upon instructors to come up with names for each test case that are descriptive of the test at a high-level.

The release testing system provides feedback about the instructor's secret tests to students *before* the project deadline, when the students can best use feedback to learn from their mistakes and improve their programs. Furthermore, release tests are a very concrete incentive for students to start working on their programming assignments early; the earlier they start, the more release tokens they can use! Release tests also mean that students who finish the project early and are satisfied with their score are rewarded for their diligence with the peace of mind that they are done. Finally, because all tests are run on the server for every submission, release tests provide feedback to the instructional staff about the progress students are making on each test case, which in turn helps instructors identify difficult test cases in time to adjust lectures or lab sessions to cover the material covered by the difficult test cases (More information about feedback to instructors can be found in Chapter 4).

3.4 Student Experiences with Release Testing

Directly assessing the pedagogical impact of Release Testing is a challenge. One possibility is to perform a controlled experiment: Split the class into two groups, one of which can use release testing for project 1 while the other can use it for project 2. However, this system is not fair to the students if the difficulty level of the projects is not the same. Since the difficulty level for projects generally increases as the semester progresses and we are unlikely to find projects of the same level of difficulty, we have not felt it was ethical to perform this type of controlled study.

Question	1	2	3	4	5	NA
Is your overall impression positive? (1=negative, 5=positive)	0	5	12	32	21	0
Do you prefer release testing over traditional post-deadline testing? (1=post-deadline, 5=release)	5	3	9	14	39	0
Were you able to make good use of feedback from release tests? (1=no, 5=yes)	4	13	6	31	16	0
Did release testing encourage you to start projects earlier that you might have otherwise? (1=no, 5=yes)	3	10	8	34	15	0
Did release testing make you feel more relaxed and confident (or, conversely, more tense and unsure)? (1=tense and unsure, 5=relaxed and confident)	6	20	20	19	5	0
For projects with secret test cases, did you keep working after you had passed all of the release tests? (1=no, 5=yes)	10	7	10	17	25	1

Table 3.3: Student Survey Results, CS-2 Fall 2005, 48 respondents out of 105 students

Another possibility would be to compare different semesters of the same course before and after the introduction of Marmoset. We were not able to perform such a comparison because the initial deployment of Marmoset at the University of Maryland coincided with a major restructuring of the introductory curriculum that changed the language used in the first two semesters from C/C++ to Java, and introduced an entirely new sequence of projects.[8]

Although a direct assessment of the impact of release testing on student achievement has not been possible so far, we did want to gain some understanding of how students perceived the system, and whether or not they felt it enhanced their experience or detracted from it. To this end, we conducted a survey; the questions and responses are in Figure 3.3.

[8]In the future, we hope to make such a comparison by using Marmoset in a different course without making other simultaneous curriculum changes.

The students who took the survey had used Marmoset in Object-Oriented Programming I, and were currently enrolled in Object-Oriented Programming II. We solicited responses using a Likert scale from 1 to 5, where 1 is the least positive outcome, 3 is neutral, and 5 is the most positive outcome.

In general, students had a positive impression of the system. According to the student responses, the feedback from the release tests was useful, and were a motivation to start work early. Surprisingly, even given the positive reaction to the system overall, the students were split evenly on the question of whether or not release tests helped them feel relaxed and confident. We speculate that some students found themselves in a situation where their project failed one or more release tests, but they either did not understand the reason for the failure, or did not understand how to fix the problem. Using a traditional program-grading workflow, there would be a lag time of a week or two between a submitting a programming assignment and receiving the grade; Marmoset, however, supplies the grade *before* the deadline, meaning that the anxiety and lack of confidence students feel is very likely a reaction to their poor grade rather than to the release test mechanism itself.

Chapter 4

Instructor Feedback

In this chapter, we discuss the feedback Marmoset provides to instructors. Marmoset provides instructors with a breakdown of how many students have passed each test case for several days leading up to the deadline, which helps instructors plan lectures and lab sessions to cover the material students are finding the most difficult. In addition, Marmoset is designed to detect mistakes in instructor-provided test suites and handle these mistakes gracefully.

4.1 Viewing Student Data

Marmoset provides instructors with a variety of views of student data. Instructors can peruse aggregate data illustrating the progress of the entire class on each test case, or a breakdown of how the best submission for each student performs on each test case. In addition, instructors can drill down for a particular student and view that student's submission history.

This section describes each of these views in more detail.

4.1.1 Aggregate Data for an Entire Class

Figure 4.1 is a screenshot of a table available through Marmoset illustrating how many students in the entire class have passed each test case for the days leading up

as of 11:00 PM	Feb 12	Feb 11	Feb 10	Feb 09	Feb 08
PlayingWithAFullDeck	126	125	99	81	60
IsDeckShuffled	126	125	98	81	60
Flush	120	119	89	69	52
FourOfAKind	119	118	88	71	55
ThreeOfAKind	116	115	85	68	52
Pair	119	117	83	66	48
TwoPair	114	113	86	68	49
FullHouse	117	114	88	67	50
Straight	108	105	77	65	43
StraightFlush	115	113	86	71	45

Figure 4.1: Overall progress for the entire class for a CS-2 poker hand evaluator project from Spring 2005. The submission deadline was February 11 and the late deadline February 12.

to the deadline. The deadline was February 11 and the late deadline was February 12 for this project, which was a poker hand evaluator. There were 138 students registered for the course when this project was assigned, although several students did not complete the project and some students did not finish the course.

The data contained in this screenshot is useful for noting broad trends. For example, the test for Straight caused students the most difficulty, as only 108 students passed this test by the late deadline, the fewest number of students for any of the test cases by about 4.5%. We could predict that this test would be one of the most difficult test cases several days in advance because on February 8—a full 72 hours before the on-time deadline—test for Straight was also one of the test cases that students had the most trouble passing.

This type of high-level overview feedback reveals which test cases are giving students the most trouble several days before a project deadline, allowing instructors to devote part of lecture, lab time, or recitation sections to the concepts or issues covered by the more difficult test cases.

Note that on February 9—a full 48 hours before the submission deadline—over half of the class had passed all of the test cases and were therefore done, which is indirect evidence that at least half of the students are done with their programming assignments well before the deadline. This of course does not prove that Marmoset was the deciding factor that encouraged students to begin their assignments early, and we do not have data showing when at least half the class had finished a programming assignment for courses not using Marmoset. However, we believe Marmoset encourages students to begin working earlier, and we hope to

show evidence for this in the future.

4.2 Fixing Instructor Test Suites

The initial version of Marmoset deployed in the Fall 2004 semester naïvely assumed that the instructor's test suite was correct and would never need to be changed. Unfortunately, five of the eight projects assigned in CS-2 that semester had errors in the initial instructor's test suite that required a series of hacks to update. These errors come in two basic flavors:

- Ambiguous specification: The project specification is ambiguous. This usually becomes obvious when part of the class interprets the specification differently from the rest of the class. One can either clear up the confusing part of the specification, or strengthen or relax the requirements for any test cases that exercise the poorly-specified functionality.

- Bad test case: Occasionally, an instructor will write a bad test case, such as a test case that returns inconsistent results or that can fail on correct student code.

Many of these errors were pointed out by the detailed feedback about student progress Marmoset provides to instructors.

We quickly realized that instructor mistakes were not rare, isolated events but rather part of the normal working pattern for grading. Thus, re-designing Marmoset between the Fall 2004 and Spring 2005 semesters, we added a workflow for updating instructor test suites and automatically re-testing student submissions against

semester	# projects	# projects updated	% changed
Spring 2005	23	9	39%
Fall 2005	39	13	33%
Spring 2006	37	18	49%
Total	99	40	40%

Table 4.1: Test Suites changed by the instructor *after* students began submitting.

the latest test suite. We also began requiring instructors to submit a *reference* or *canonical* implementation of each project, and began storing *all* the test outcomes for every test run of a submission against each version of the test suite. The additional data helps us track when test suites are changed and the impact of a new test suite on student performance (which is helpful for explaining to students how test suites evolve in response to specification ambiguities and resolving grade disputes).

Since adding the workflow for updating test suites after the project has been posted, we can more accurately track which test suites need to be updated after being posted. The breakdown of test suite modification per semester is given in Figure 4.1. This Figure shows that between 1/3 and 1/2 of test suites need to be updated *after* students begin submitting, often as a direct response to feedback provided by Marmoset.

After all, test cases are code, and instructors, like everyone else, make mistakes when writing code. Using a traditional automated testing workflow (i.e. where instructors or TAs build or modify ad-hoc scripts for each project), one of the reasons that grading takes so long is that test suites often reveal weaknesses in the project description, or bugs in the testing code itself. Under this traditional workflow, the feedback unfortunately comes too late to amend the specification or

otherwise give students more useful feedback. In fact, it is very likely that bugs in the test suite would be missed using a traditional workflow due to a number of factors, such as the time pressure to finish the grading and the lack of the high-level views of student performance on each test case to reveal patterns. The burden of discovering problems in the test suite is shifted onto the students' shoulders. However, since the grading for a project is typically performed *after* the deadline passes but *before* the submission deadline for the next project, students are likely to focus most of their attention on the points still available on the current project rather than the points lost on the previous project. Thus, students are less likely to focus careful enough attention on the outcome of their test results to find any but the most obvious mistakes in the test suite.

On the other hand, the feedback Marmoset provides helps reveal problems in a timely fashion both to instructors (who can easily isolate a test case that too few or too many students are passing) and to students (who have incentive to figure out why a test case is failing and are more likely to discover a faulty test case or ambiguous specification), so that instructors can fix problems quickly and provide students with feedback early enough for students to make use of it.

Although it was not an initial design goal of the Marmoset system, one of the strengths of Marmoset is that instructor mistakes are anticipated and handled in a graceful way. We are unaware of any other automated grading systems that address this issue.

Chapter 5

Data Collection with Marmoset

In this chapter, we detail our motivation for collecting fine-grained research data, and use the Marmoset dataset to mine new bug patterns and evaluate the precision and recall of the FindBugs [16] open-source static checker.

5.1 Motivation

Virtually all instructors have strongly-held opinions about the "right" way to teach computer science. A recent overview by Kim Bruce [5] of a thread on the SIGCSE members mailing list about the advantages and disadvantages of teaching introductory computer science using an "Objects First" approach is a prime example of the spirited debate inspired by discussions of effective teaching methods. Unfortunately, these debates too often rely on "folk wisdom" and anecdotal evidence rather than rigorous scientific studies and strong research. Part of this is likely because, as instructors, we have a very indirect and limited view of how well students are learning to program. Our standard sources of feedback—questions in class or office hours, posts to a class newsgroup, project grades, course and instructor evaluations, students' logfiles and reports from group projects, and the experiences of instructors and TAs during office hours, to name a few—are rather crude, coarse-grained mechanisms from which it is often difficult to draw meaningful conclusions.

Recently, the computer science education community has formed a series of working groups to better assess students' mastery of basic programming concepts, with the ultimate goal of improving pedagogy techniques in computer science. The results of these initial studies have not been encouraging. The "McCracken Report" [38], a multi-institution study at eight universities in five countries, revealed in 2001 (to the dismay of many educators) that in general students could not program as well as their instructors expected upon completion of the introductory programming sequence. Further multi-institutional work by Lister et al [33] demonstrated a similar lack of proficiency in students' ability to read and trace code. These studies, as well as continuing studies by ITiCSE working groups, have served as a powerful wake-up call to CS educators that many students are not learning to program, and that pedagogical methods in our field drastically need improvement.

Unfortunately, multi-institution studies of this variety are expensive and time-consuming to conduct. The McCracken report required standardized questions and grading scales to be developed before the semester, administered at many universities in different countries, then evaluated, coded and analyzed according to the same criteria. The work on program tracing and understanding done by Ray Lister's working group relied in part on "think-aloud" sessions, where students are encouraged to speak their thoughts out loud as they answer the questions. Their vocalizations are recorded, then later transcribed and analyzed; this is a very time-consuming process that limits the amount of data that can easily be collected.

In addition to the overhead, collecting this type of data requires that students work in class on prepared questions much like an examination, or in closed lab

sessions. Students typically do not have access to the same types of resources they may rely on in their natural programming environment, such as their textbook or Google.

These studies fail to capture a much cheaper source of naturally occurring data—snapshots of code written by students. Student code snapshots are orthogonal to the kind of data collected by the ITiCSE working groups, and shine light on what students do when working on their own, without strict time constraints and with access to whatever outside resources they typically use when programming.

Studying students' code artifacts, while unlikely to improve our understanding of *how* students learn introductory computer science concepts or *why* students make certain mistakes, nonetheless gives us an excellent window into exactly *what* students are doing, such as the types of mistakes they are making, the amount of time they are spending on various tasks, when they are writing the majority of their code, and so on. By collecting detailed snapshot-based development histories and automatically testing these snapshots, Marmoset provides us with an excellent platform for evaluating tools, such as debuggers or static bug finders, aimed at shifting students' cognitive load away from tricky language features and towards problem solving.

5.2 The Course Project Manager Plugin

In order to collect regular snapshots of student code cheaply and easily, we have built a plugin called the Course Project Manager (CPM) [46] for the popular Eclipse [11]

Integrated Development Environment (IDE). The CPM transparently commits a student's files to a central repository every time a file is added, removed or saved. This mechanism provides the image of a wide-area file system for students in that they can access their files from their laptop, desktop, cluster or any other networked computer. The CPM also provides students with a detailed backup history of their files in case they delete an important file or their laptop hard drive crashes; these are extremely frustrating experiences that we can hopefully prevent with the CPM plugin. Finally, Eclipse performs compilation in the background and underlines syntax errors in red, meaning that students do *not* need to save to find their syntax errors. One effect is that about 70 % of these snapshots are compilable.

Data captured at the granularity of every save operation is fine-grained but reveals little information about *why* the student is saving: A student may save their files because they've just completed a major intellectual chunk of work, because they just fixed a difficult bug, or because their laptop battery is almost dead and they don't want to lose their work. Furthermore, different students save at different intervals—some students may achieve a perfect score on a project with 600 snapshots while other students may also achieve a perfect score with only 15 snapshots. In addition, it is also somewhat problematic to cluster students' snapshots into "work sessions" based on snapshots because we don't know where to break the clusters (i.e. Should two snapshots 15 minutes apart be in the same cluster? How about 25 minutes?), nor how long students were working before the first save operation of a fresh session. All of these factors combine to make it difficult to use snapshots to

semester	# students	# projects	# submissions	# snapshots	# test outcomes
Fall 2004	84	8	3,919	44,771	627,320
Spring 2005	103	8	3,932	45,558	451,727
Fall 2005	69	7	1,505	14,214	152,359
Spring 2006	101	6	2,680	43,052	940,406
Total		29	12,036	147,595	2,171,812

Table 5.1: Overall data collected by Marmoset for 4 consecutive semesters of CS-2 at the University of Maryland

measure programming effort or time spent programming[1].

Despite these complications, capturing data at the granularity of each snapshot has a number of advantages over other approaches, such as capturing snapshots at certain time intervals, for example by forcing a save operation every 5 minutes. Such an approach may improve the quality of the data harvested, but would suffer from other complications that could reduce the quality of data; for example, snapshots grabbed at regular time intervals might be less likely to compile, would force the student to wait for the synthetic save operations to complete, and may alter the "undo" history of the students' edit actions in unpredictable or surprising ways. Marmoset at its core is a pedagogical tool that should not make learning to program more difficult for students in any way. Our approach is simple and unambiguous, and produces high-quality datasets for study.

# lines added/ changed	F04	% so far	S05	% so far	F05	% so far	S06	% so far	total	% so far
1	15,106	40	14,429	38	4,472	38	14,904	40	48,911	39
2	6,876	58	6,398	55	2,73	56	5,944	56	21,291	56
3-4	5,990	74	5,825	70	1,730	72	5,140	70	18,685	71
5-8	4,380	85	4,524	82	1,419	84	4,387	82	14,710	83
9-16	2,863	93	3,86	90	815	91	3,329	92	10,93	91
17-32	1,400	97	1,866	95	490	95	1,630	96	5,386	96
33-64	699	99	1,33	98	244	97	678	98	2,654	98
65+	359	100	662	100	245	100	593	100	1,859	100

Table 5.2: Number of lines changed between snapshots collected by the Course Project Manager plugin over four semesters of CS-2 at the University of Maryland.

5.3 CS-2 Data Collected with Marmoset

Figure 5.1 shows a breakdown of the number of snapshots collected over four semesters of CS-2 at the University of Maryland. The Fall 2005 data had fewer snapshots, submissions and test outcomes than the other semesters, even though the average number of unit tests (public, release and secret) per project was on par with other semesters. The only major difference was that there were more public tests (and therefore fewer release and secret tests), although it is unclear what effect, if any, this has on the frequency of student saves and the proportion of snapshots that are compilable.

In general, the data collected by the CPM is extremely fine-grained, as is shown in Table 5.2: Just under 40% of the compilable snapshots add or change only a single line, while over 70% of compilable snapshots add or change 4 lines or fewer. This fine-grained dataset contains a wealth of information that can be used for a

[1]In the future, we hope to collect additional data that will better help us measure programmer effort.

Exception	# student projects	# snapshots
NullPointer	1,172	31,454
ClassCast	512	8,122
IndexOOB	400	5,453
ArrayIndexOOB	359	4,996
StackOverflow	281	3,754
NoSuchElement	238	3,289
IllegalState	228	3,956
StringIndexOOB	195	3,960
IllegalArgument	195	3,698
OutOfMemory	158	1,311
Total	2,371	147,597

Table 5.3: Most common exceptions over four semesters of CS-2 at the University of Maryland. Course taught in Java, 29 total projects represented in this data, data sorted by number of projects in which the exception occurred.

variety of purposes, such as mining new bug patterns or studying the development process for novices in more detail.

5.4 Examining Runtime Exceptions in CS-2

Figure 5.3 shows the top 10 run-time exceptions plaguing students over 4 consecutive semesters of CS-2 at the University of Maryland. That almost half of all student projects and over one in five snapshots had at least one `NullPointerException` did not surprise us. It was similarly expected that the various bounds-check exceptions, such as `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException`, would be in the top ten, which they were. However, we were mildly surprised that `ClassCastException` was the second most common run-time exception (about 22%) and `StackOverflowError` the sixth most common run-time exception (about 12%) based on the number of projects in which they occurred.

```
/**
 * Create a new Web Spider
 *
 */
public Spider(boolean isDFS, int limit, String root) {
    Spider spider= new Spider(isDFS,limit,root);
}
```

Figure 5.1: Example of CS-2 infinite recursive loop bug pattern.

```
public WebPage(URL u) {
     this.webpage = (WebPage)((Object)u);
}
```

Figure 5.2: Example of CS-2 bad cast bug pattern.

We first noticed the large number of `StackOverflowError`s in Fall 2004 during the inaugural semester of Marmoset. To figure out why, we examined submissions with test cases failures due to `StackOverflowError`; a pattern quickly emerged.

Students were causing `StackOverflowError`s by writing infinite recursive loops in a very specific way, as show in Figure 5.1. Students had recently learned constructors in lecture; the Javadoc explained that the method needed to create a new `WebSpider` object, so the student called the constructor. Of course the method described by the Javadoc *was* the constructor, which lead to an infinite recursive loop.

Similarly, the pattern shown in Figure 5.2 shows a sample bug students made that caused `ClassCastException`s in their code. Students had learned typecasts in lecture that week, and were trying to apply them in inappropriate places. This error pattern is an example of students' "fragile knowledge" of a concept as described by Lister et al [33]. Fragile knowledge can lead students to mis-apply concepts, or apply

them in the wrong situations, as the student has clearly done in this example.

We implemented new bug detectors for these patterns in the open-source static bug finder FindBugs [16]. The infinite loop detector (IL) simply looked for methods that unconditionally call themselves, while the bad-cast detector (BC) looked for casts that were statically doomed to fail. Running these checker over some student code revealed a number of both types of bugs, alerting us to the fact that many students were making a similar mistakes and suggesting that typecasting and constructors were proving more difficult to the students than the instructors had expected.

After writing a new bug detector for FindBugs, we typically run the new detector over a large production codebase, such as the core Java runtime libraries, Eclipse, or JBoss [27], manually evaluate the false positive rate of a small sampling of the warnings, and if possible tune the detector to eliminate some of the false positives.

We followed this procedure after writing these new detectors; interestingly, the detectors uncovered infinite loops and statically-doomed casts in production code as well as student code. This is not to say that professional programmers writing production quality software make the same mistakes as novices programmers; instead, it suggests that bugs in production code can be taught using techniques similar to those that effectively find bugs in novice programs. For a detailed discussion of these bug patterns in production code, see David Hovemeyer's 2005 PhD thesis [23].

5.5 Evaluating and Tuning FindBugs

We evaluated the precision and recall of three FindBugs detectors: the infinite-loop and bad-cast detectors discussed in the previous section and the suite of null-pointer dereference detectors, which can effectively be grouped into one category because they all warn about null-pointer dereferences that will lead to a `NullPointerException`. We did this study by matching up bug warnings and their corresponding runtime exceptions in the Marmoset dataset: infinite-loop warnings are paired with `StackOverflowError`, bad-cast warnings with `ClassCastException`, and null-pointer dereference warnings with `NullPointerException`. This is possible for these particular warning categories because any bugs other than the ones our detectors check for are very unlikely to cause these runtime exceptions.

In this context, precision, also known as the *false positive rate*, means determining the percentage of warnings that correspond to actual bugs. For example, if snapshot N contains a warning of type W; does snapshot N also contain a runtime exception E that corresponds to warning W?

Recall, also known as the *false negative rate*, means determining the percentage of runtime exceptions for which FindBugs issues a warning. For example, if a snapshot contains a runtime exception of type E, does it also contain the corresponding warning W?

Measuring the precision for a static checker can always be done by brute force; if a checker issues 500 warnings, someone can always manually examine each warning and the source code to determine whether or not the warning indicates

a bug. Recall, however, is more difficult to measure because it requires a priori knowledge of the location of bugs in the code, and for most software, we don't know where the bugs are (and if we did, we would have already solved the problem of finding the bugs and would not need to write and tune a bug-finder). Because the Marmoset dataset contains a large number of test case failures—essentially known bugs—we can measure the recall of bug checkers, something that has proved difficult for the static analysis community.

When counting code features such as warnings and exceptions in the Marmoset dataset, the question naturally arises: What are we counting? We cannot simply count the warnings or exceptions in every snapshot: Warnings may persist across many snapshots, leading to over-counting; and exceptions may mask each other, leading to under-counting. For example, students have no incentive to fix a bug warning that is a false positive (since it does not lead to an error); thus that warning is likely to persist across multiple snapshots, making the warning's precision appear worse. Similarly, if several test cases may fail with the same exception, *at the same line number*, for a given snapshot, then these exceptions are likely the result of the same error and should be matched together with a single warning. On the other hand, two test cases may fail with the same runtime exception at different lines of the same method; should we also group these exceptions together? What about if two test cases fail at different lines but have the same calling context earlier in their stack traces? Also, because each test case can only throw a single runtime exception, one exception may "mask" another, making it more difficult to match exceptions with warnings. Finally, it is difficult to evaluate the accuracy of warnings that are

Detector	Warnings	With Exception	Observed Precision
InfinteLoop (IL)	83	73	88%
BadCast (BC)	7	7	100%
NullPointer (NP)	1343	579	43%

Table 5.4: Observed false positive rates for selected FindBugs detectors.

Detector	With Exception	With Warning	Observed Recall
InfinteLoop (IL)	386	73	19%
BadCast (BC)	724	6	$< 1\%$
NullPointer (NP)	2269	475	21%

Table 5.5: Observed false negative rates for selected FindBugs detectors.

issued by a detector but are never executed by any test case.

We used a series of simple heuristics to prune the number of warnings and exceptions under consideration to better reflect the true precision and recall of our detectors. First, we computed the code coverage for each test case in order to eliminate uncovered warnings from consideration. Next, when computing the false positive rate, we only consider warnings that are either present in the final snapshot or present in one snapshot then removed in the subsequent snapshot. We compute warning-removal information using FindBugs functionality that tracks bug warnings across versions of software. We describe this work in more detail in [47]. Finally, when computing false negatives, we group exceptions together if they happen at the same line number or in the same method.

Table 5.4 shows the false positive rates for selected FindBugs detectors, while Table 5.5 shows the false negative rates for those same detectors. The detectors are quite precise when issuing warnings; however, the recall we have observed is

very poor, especially for the bad-cast detector. While there were too many false negatives to examine each one individually, a spot-check of a sample of the false negatives reveals that many of the `ClassCastException`s for which FindBugs is not issuing a warning are improper casts of objects coming out of a collection, and would be eliminated with the use of generics.

Chapter 6

Helping Students Appreciate Test-Driven Development (TDD)

In this chapter, we report on our initial experiences teaching and motivating students to write test cases and then evaluating student-written test suites, with an emphasis on our observation that, without proper incentive to write test cases early, many students will complete the programming assignment first and then add the bulk of their test cases afterward. Based on these experiences, we propose new mechanisms to provide better incentives for students to write their test cases early.

6.1 Motivation for Test-Driven Development

We agree with other educators [32, 30, 21, 12, 9] that learning to understand, appreciate and construct tests is an important part of learning to develop software, and a topic that needs to be better addressed in our undergraduate Computer Science curriculum. The question is, how do you teach testing? We can lecture on topics such as unit tests, integration tests, testing frameworks (such as JUnit) and code coverage. But testing can't really be adequately covered or evaluated in lecture. Instead, we need to find ways to encourage and/or require students to practice it in programming projects, and to assess their mastery of the topic.

We believe that simply mandating testing as part of programming assignments (e.g., "you will write test cases, and 30% of your grade will be based on your test

cases") can be counter-productive. We want to help students understand and appreciate the value of devising their own test cases *without* making the requirement that students write their own test suites seem like an artificial hurdle.

We also agree with previous work [30, 12] that to demonstrate its importance, testing needs to be part of the curriculum throughout the major, rather than taught in an upper-level testing course. We have used a number of different techniques across several courses to help students learn to appreciate, understand and perform testing. Several of these techniques are stand alone techniques that can be easily incorporated into any curriculum, while other are implemented within Marmoset. We were also concerned as to whether some of the features of Marmoset—namely, release testing—might have the perverse effect of reducing the incentive for students to write their own tests cases, and discuss steps we have taken to mitigate that possibility.

6.1.1 Marmoset's support for TDD

Marmoset supports student-written test suites and computes code coverage metrics using the Clover [7] code coverage tool. Instructors can evaluate how many test cases students are writing, and how well their test suites cover all of the statements, branches or methods in a programming assignment.

6.2 Getting students to write tests

We need to provide students with incentives to write test cases. In CS1/CS2 courses, students have not yet learned to appreciate the value of TDD, and we believe that even if students have been told that they will help themselves by writing their own test cases, all too often students work on only the things that they are graded on.

Although we believe the Marmoset system has a number of advantages, we are also concerned that it may reduce the incentive for students to write their own test cases. Since students are provided (limited) opportunities to test their implementations against the instructor test cases, they may feel that they do not need to develop their own test cases.

In some projects, we have blunted some of the feedback provided by Marmoset to further encourage students to write their own test cases, rather than depending upon release testing. One CS2 project is a binary search tree project where rather than the typical descriptive names used for release tests, we simply number the tests (test1, test2, ...). We then inform the students that the release tests will provide them very little information about why their program fails, and that their best hope of figuring out why is to write their own comprehensive test suite.

We can also use a blunter instrument of encouragement: make writing tests part of their project grade. The easiest way to do this is to measure code coverage. On several of the projects we assign, we tell students that part of their grade will be based on the code coverage they achieve. Since we provide students with some test cases (the public tests), we measure the code coverage from the combination of

public and student-written test cases and base part of their project grade on this. We can scale this as

$$\frac{(coverage\ from\ public\ and\ student\ tests) - (coverage\ from\ public\ tests)}{(coverage\ from\ public\ and\ instructor\ tests)}$$

We have used a combination of these techniques to encourage students to write test cases, and most students responded to this incentive by submitting test cases as part of their final submission. However, upon further exploration, we found that a significant number of students did substantial work to improve their test cases after they had submitted and release tested a submission that passed all of the release tests. Obviously, these students aren't learning that writing their own test cases can help them develop reliable and correct software; instead, they are simply responding to the carrot/stick of being graded based on their code coverage. We worry that as a result, these students likely to view testing as a hurdle to jump rather than an important part of the development process.

To report on this phenomena, we studied the submissions by each student for two CS2 projects in the Spring 2006 semester: a binary search tree project and a MediaPlayer project (in which students stored a database of songs, playlists, podcasts and podcast entries). In both projects, students were told that part of their grade would be based on code coverage of their own test cases. The MediaPlayer project provided descriptive release test names as is normal for Marmoset; for the binary search tree project, we used meaningless names for release tests to reduce the ability of students to depend upon release testing for all of their testing needs.

We looked at all the students who completed the project (turned in a solution that passed all of the instructor test cases). For each such student, we looked at

whether they continued to work on their test cases after having performed a release test and found that they passed all of the instructor test cases.

On the binary search tree project, 16 of 34 students extended their test cases after they had completed an implementation that passed all instructor tests. Of those 16, 11 students made significant improvements to their tests (significant is defined as a change that increased the number of covered statements and methods by more than 1/3 the total number of statements and methods, or a reduction of more than 25 % in the number of uncovered methods and statements). In the MediaPlayer project, 24 of 40 students extended their test cases after having completed an implementation that passed all instructor tests. Of those 22, 17 made significant improvements to their tests. Figure 6.1 shows the changes in code coverage after reaching functional correctness (for the MediaPlayer project, the number given is the amount of code coverage over baseline of the coverage provided by the public test cases that were provided to students).

Ideally, we would like to see most students clustered in the top right portion of the graph, i.e. when they achieve full functional correctness, they also have good test coverage from the test suites they've developed.

Our hypothesis was that students would write more test cases before achieving functional correctness for the binary search tree project than for the media player project because the names of the release tests used in the binary search tree project were generic (test1, test2, etc) and therefore students could not rely as heavily on release testing to find their errors without writing their own test cases.

Table 6.2 shows the breakdown of when, if at all, students who achieved perfect

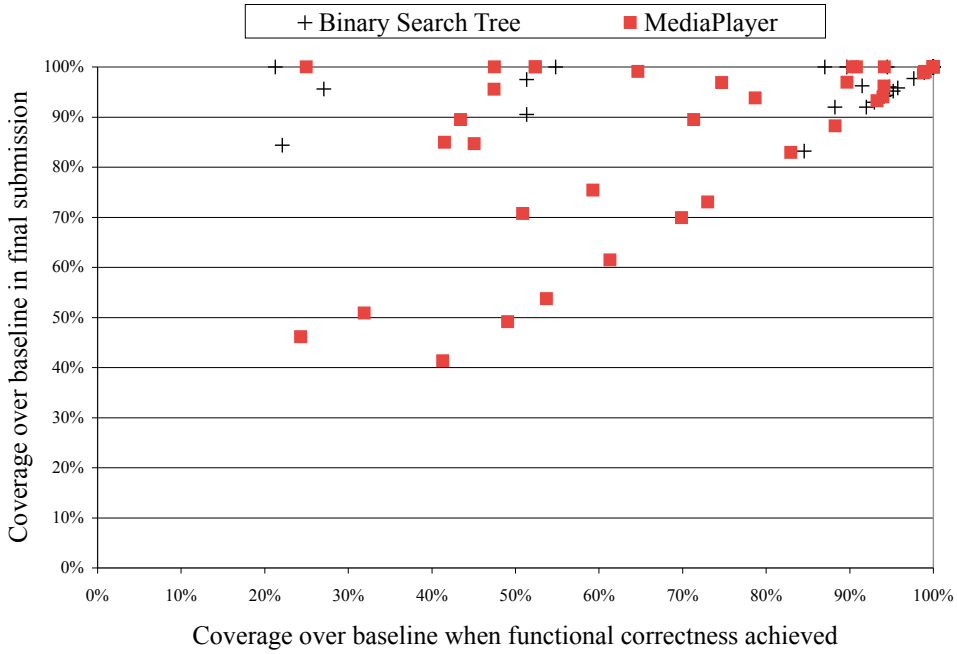


Figure 6.1: Code coverage added after reaching functional correctness. Values along the diagonal represent students who did not improve their test suites after achieving functional correctness; values along the top represent students who eventually achieved 100% code coverage.

project	never achieved 80% coverage	achieved 80% coverage after functional correctness	achieved 80% coverage before functional correctness
Media Player	10	12	16
Binary Search Tree	0	6	28

Figure 6.2: Breakdown of when students achieved at least 80% coverage.

functional correctness (i.e. passed all public and release tests) achieved at least 80% coverage of their code. Our results indicate that there was a statistically significant difference between the number of students achieving at least 80% coverage *before* achieving functional correctness on the binary search tree project as there was for the media player. While we believe that the difference can be accounted for by the lack of descriptive release test names, we cannot draw such a strong conclusion from the data. The binary search tree project was assigned later in the semester, so students may have simply learned to write better test cases.

Finally, some of the results of this study are not comforting. A substantial number of students are writing many of their test cases after having completed the non-test code, and as a result are less likely to appreciate or understand the value of test driven development.

6.3 When coverage is not enough

One interesting issue related to teaching testing is that there are many scenarios where code coverage is not sufficient to expose a bug. For example, we expect that

the majority of failing test cases in student assignments will *uniquely cover* lines of code that are not covered by any passing tests. However, there will also be failing tests that are *redundant* in that they only cover lines of code that are also covered by a passing test case, and if we were to remove these *redundant tests* from the coverage set, the percentage of covered statements, methods and branches would remain unchanged.

A failing test that does not cover any "new" code is interesting because it may expose a "fault of omission" [20] or a bug that is control-dependent on a particular set of conditions.

One question is, what is the proportion of failed tests that uniquely cover code versus the proportion of failed tests that are redundant? We hypothesize that the majority of failing tests will execute code that is uncovered by any passing test, and that the bug will be located in the code that is uniquely covered by the failing test. We also expect that redundant failing tests (i.e. failing tests that cover only code also covered by passing tests) will represent the more difficult test cases, because these tests exercise more difficult or subtle interactions in the code.

Figure 6.3 shows the breakdown of test case outcomes for all submissions for the first two projects assigned in CS2 in the Spring 2006 semester. In this chart, "passed" is the portion of test cases that passed, "redundant" represents the portion of test cases that failed but only covered code that was also covered by a passing test, "statement" describes test cases that failed but covered at least one statement that was not covered by any passing test, and finally "method" represents the portion of failed test cases that covered at least one method that was not covered by any

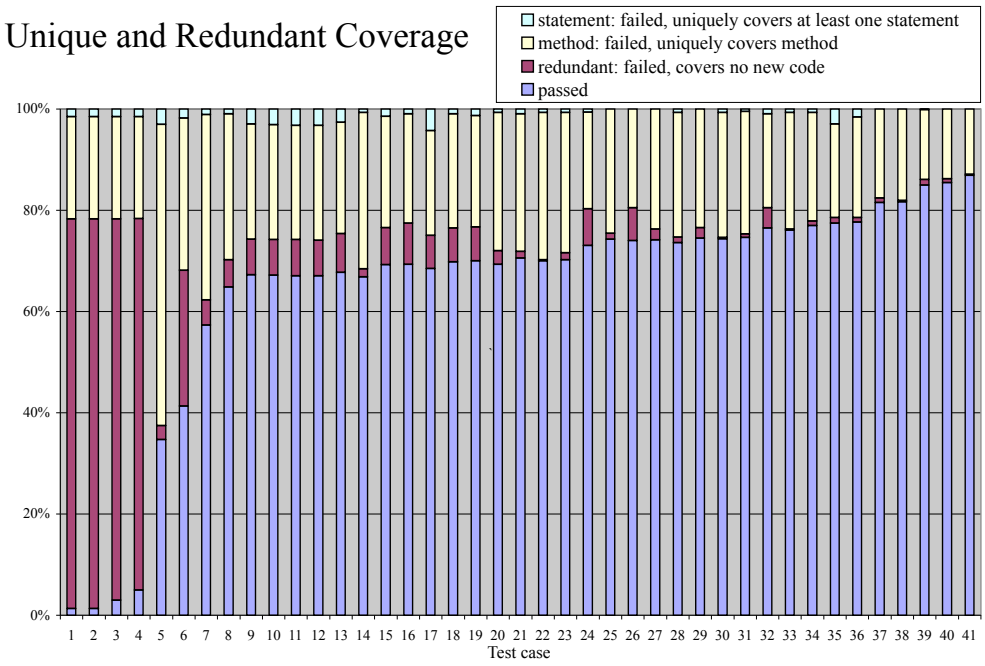


Figure 6.3: Unique and redundant coverage by failing test cases

passing test.

If our hypothesis is true, we would expect the most difficult test cases to have both low rates of passing tests and also high rates of redundant failures, or failures that don't cover any new code.

The chart is sorted along the x-axis in ascending order by the passing rate. The first 4 test cases have the lowest passing rates and the highest rates of redundant coverage. This is not surprising since these four test cases were challenge problems assigned to the honors section but not required of the other sections of the course.

Test case #5 tested a difficult method that was not exercised by any other test. Test case #6 checks whether a student's code performs a deep copy of an important data structure that is specified in the project description. This is a prime example of code that will pass many test cases even when not implemented according to the specification because a shallow copy will work in some—but not all—places where a deep copy is expected.

Our data suggests that difficult test cases do indeed exercise code that is also covered by passing test cases elsewhere. However, our dataset is far too small to draw any sort of wider conclusions.

The lesson to learn from redundant failing tests is that, as educators, we need to understand the limits and weaknesses of code coverage (namely, that high coverage does not necessarily imply adequate testing), and to reiterate to students that code coverage is a useful tool but that, like any other tool, it needs to be understood and utilized properly to be effective.

6.4 Future Work: Enchancing Marmoset

Given our observations reported in section 6.2, namely that students will write test cases when required to do so, but often only *after* completing the project, we want to create incentives that reward students for writing their test cases early in the development cycle.

To achieve this goal, we have provided "knobs" so that the feedback provided by Marmoset can be adjusted depending on the quality of test cases that students have written. For example, if students have written few or no test cases, Marmoset can be configured to provide less information when students perform a release test. Thus, students will be motivated to write test cases early in their development, so that their early release tests will return as much information as possible. If students write tests earlier, we hope that they will gain more value from the tests and learn to appreciate them more than if they wait until finishing their implementation before writing test cases.

6.4.1 Code coverage information

Although UMD has obtained a license to use the Clover code coverage tool in courses, students sometimes find the tool difficult to install and use. Thus, we also provide code coverage results to students through the SubmitServer's web interface. We also provide a summary view, showing the coverage from the combination of all public and student tests. For each of these views, we give a list of all the source files, and for each source file report the number of covered and uncovered methods, statements

and branches. For each source file, students can see a view of the source of that file which each line labeled by the coverage.

While this detailed summary information is useful, it can also be a little overwhelming and confusing, particularly to students just starting an intro programming sequence. So in addition, we provide a high level summary that just lists methods that are not covered by any public or student test case.

Instructional staff is provided with additional coverage views, such as a view of all the methods, statements and branches covered by a release or secret test but not by any public or student test case.

6.4.2 Tests that cover uncovered methods

For each release test, we also record the coarsest granularity covered by the release test but not by any public or student test case. This information is provided as a high level summary/feedback to both instructors and students.

A project can also be configured to provide an additional incentive for writing test cases. The instructor can specify that for a particular project, if a failing release test covers too much code that is not covered by a public or student test, then the student will not be told the name of the release test even if the rules for release tests would otherwise allow the student to be told the name of the release test. For example, the instructor might specify that students are never told the name of failing release tests that covers a method not covered by some student or public test case.

In this way, we provide students with an incentive to test their own methods

and prevent them from depending too heavily on release testing as their only testing framework.

6.4.3 Covering the source of exceptions

When a release test fails, we reveal the kind of failure: a test failure (when a condition asserted by the test case does not hold), an error (when execution of the test results in an uncaught exception being thrown), or a timeout (when execution does not terminate in a timely matter).

Normally, that is all the information we provide; we don't tell students whether an error arose from an null pointer exception or an index out of bounds exception.

Instructors can now enable a feature that works as follows: if a release test terminates with an error occurring on a line that is covered by a public or student test, we tell students the kind of exception and the line number where the exception occurred.

In addition to providing students with yet another incentive to write their own test case, this will be helpful in the situation where some core component consistently fails across multiple test cases. If students see that several of their release tests generate the same error at the same line, this suggests that a common error might be at fault and students will look for one common fault, rather than for a series of disconnected faults.

We have also considered, although not yet implemented, providing feedback whenever several release tests all terminate with the same exception at the same line number. Here, we would not require that the place where the exception occurs

be covered by student code. Instead, we should simply identify that several release tests all failed with the same exception at the same line number. Whether the name of the release test, the kind of exception and the line number where the exception occurred are revealed is determined by the other rules of the Marmoset system.

These new features were first introduced during the Spring 2006 semester; we do not have any data about their effectiveness at this time.

Chapter 7

Survey

In July 2006, we administered a multi-institution survey of educators about grading practices for programming assignments in computer science courses. A copy of the text of the survey is available in Appendix A. We sent the survey to the Special Interest Group on Computer Science Education (SIGCSE) mailing list, as well as educators at larger institutions that we know personally. This last step was necessary because SIGCSE members tend to come from smaller colleges and other teaching-oriented institutions, and their practices do not necessarily reflect the practices at larger state schools or research-focused institutions ("R1" schools).

7.1 Survey Goals

Our main goal in administering the survey was to determine what features of programs instructors evaluate, how long faculty spend grading student programs, how much time spent grading could be saved through automation, and what the perceived impediments to effective grading are. In addition, we measured other general trends related to teaching programming courses, such as the programming language used, as well as the use of online course management systems, Integrated Development Environments (IDEs), and automated testing frameworks.

Of particular interest to us was the weighting of style vs. functional correct-

School size	number	average size
less than 3,000	17	1,700
3,000 to 10,000	20	4,900
over 10,000	17	20,900
Total	54	8930

Table 7.1: Breakdown of the sizes of schools where survey respondents taught.

ness. Ultimately, we would like to know to what extent faculty "grade what they can see". In other words, do faculty weight style more heavily because they lack an efficient way of running student code against test cases and feel that it's easier to grade style than correctness simply by looking at code? Furthermore, would faculty weight functional correctness more heavily, or even assign different types of projects, if they could measure functional correctness cheaply and easily?

7.2 Overview of Survey Results

A total of 56 people responded to the survey, including one high school teacher and one community college professor. While several respondents were from larger institutions, most were from smaller schools or teaching-oriented larger institutions. The relatively small sample size and the bias towards the SIGCSE members mailing list and personal contacts makes it difficult to draw overly-broad conclusions based on the survey data. However, the respondents provided a wealth of interesting and instructive quantitative and qualitative responses from which we can learn quite a bit.

Table 7.1 outlines the breakdown of the sizes of schools where the survey respondents taught. The cutoff points are somewhat arbitrary, but give a rough

Course (56 responses)	#	%
CS-1	31	55%
CS-2	12	31%
Software Engineering	3	0.5%
Other	3	0.5%
Computer Graphics	2	<1%
Operating Systems	2	<1%
Compilers	1	<1%
Databases	1	<1%
CS-0	1	<1%

Table 7.2: Breakdown of the courses taught by survey respondents.

estimate as to the types of schools responding to the survey. The data includes one high school and one community college, as well as several schools outside of the United States.

Table 7.2 shows the breakdown of the courses taught by survey respondents; 45 of 56 respondents taught CS-0, CS-1 or CS-2, with over 87% of CS-1 courses taught in Java. Drawing any conclusions about grading practices for upper-level courses based on our survey results is difficult because there are so few data points.

Full results of the survey will be made available online after this work is accepted for publication.

7.3 Evaluating Student Programs

One major question is to what extent faculty are evaluating things that they can measure easily, such as programming style, because they lack the infrastructure required to efficiently measure functional correctness, versus to what extent faculty have made a conscious choice to emphasize programming style, regardless of their

		Minutes spent grading correctness				
		< 5	5-10	10-15	15-30	> 30
Min.	< 5	7	12	5	4	0
spent	5-10	4	3	6	4	1
grading	10-15	2	0	1	2	0
style	15-30	0	0	1	0	0
	> 30	0	0	1	0	0

Table 7.3: Grid showing number of minutes spent on each submision evaluating style concerns and functional correctness.

ability to evaluate functional correctness. In other words, if instructors who currently weight style more than functional correctness had access to an automated testing technology that measured functional correctness cheaply, would they begin to weight functional correctness more highly, or even assign different types of projects? The results of this survey certainly do not resolve the issue, but rather take the necessary first step of measuring the current practice.

7.3.1 Time Spent Grading per Submission

Table 7.3 shows the amount of time spent grading submissions for style as well as functional correctness. These results show that on average respondents spent 10 minutes grading submissions for correctness and 5 minutes grading submissions for style.

Spending 10 minutes per submission grading the correctness of a program does not sound daunting, especially if the enrollment in the course is small. However, the 20 respondents who used some kind of framework to automatically run student code against intstructor-supplied test cases showed a statisticlly significant difference in the amount of time they spent grading student-written programs for correctness,

with the amount of time being on average 5 minutes less than their counterparts who did not use any automation. These results suggest that recent work on large-scale systems capable of automated testing, such as Marmoset [48] and Web-CAT [14], could save instructors a substantial amount of time when grading programs for correctness.

A mere 5 minutes per submission may seem like a meager amount of time saved for investing effort into adopting an automated testing framework; however, those 5 minutes add up quickly if we consider the larger context. For example, according to the annual Taulbee Survey [50], in 2005 about 15,000 students were awarded bachelors' degrees in Computer Science at PhD-granting institutions in the United States. Assuming that total nationwide enrollment in Computer Science is about four times higher, or 60,000 students, and that each student does 15 programming assignments per year... that's over 35 years worth of labor (split into 40 hour weeks, 50 week-years) that goes into grading all those submissions *every year*. The exta time sunk into grading work that could be partially or completely automated takes time away from the more creative aspects of being faculty: developing innovative teaching methods or curricula, performing cutting-edge research, or otherwise growing the knowledge base of computer science—in short, tasks that could help reverse the recent trend of declining enrollments in undergraduate Computer Science programs throughout the United States.

Code features evaluated	#	%
Functional Correctness	55	100%
Programming Style	53	96%
Comments	49	89%
Student-Written Test Cases	25	45%
Other	20	36%

Table 7.4: Factors that contribute to the final score of a programming assignment (55 responses). Note that the totals will sum to more than the number of responses because respondents could select more than one answer.

7.3.2 What Contributes to the Final Score of an Assignment?

Table 7.4 shows a breakdown of code features that contribute to the final score of a programming assignment. Note that respondents could select more than one response to this survey item.

We were not surprised that every respondent evaluates functional correctness, or that all but two respondents evaluate programming style. We were surprised that so many respondents evaluate comments and documentation in addition to style. Finally, as advocates of Test-Driven Development, we were pleasantly surprised that student-written test cases are evaluated by 45% of respondents.

Some of the other factors evaluated that respondents listed included non-functional issues such as "elegance" and "readability", which arguably could be lumped in with style, as well as other non-functional issues such as "using a for loop instead of a while loop", which aren't necessarily stylistic but would not affect the functional correctness of the program. No single feature garnered more than a couple of write-in votes ("design" was mentioned three times), suggesting that the four major categories in the survey encompass the vast majority of what is graded

Correctness vs Style Points per assignment (50 responses)			
Correctness Points	# that execute code against test cases	# where code not executed against test cases	Style Points
30	1	1	70
40	1	0	60
50	7	1	50
60	4	3	40
70	14	2	30
80	8	3	20
90	4	1	10
Totals	39	11	

Table 7.5: Weighting of style and functional correctness in grades.

in a programming assignment.

7.3.3 Style vs. Functional Correctness

Table 7.5 shows a breakdown of the style and functional correctness weights for the 50 survey respondents who supplied this information. The average for all responses was 67 points for functional correctness and 33 points for style. We further break down each category by whether the respondents compile and execute the student code against instructor-written tests, or not. For example, the first line of this table means that of the 2 survey respondents who assigned 30% of the points to functional correctness and 70% of points to style, one compiles and executes the student code, while the other does not. However, because this question was framed as "functional correctness" vs. "style", it is not clear to which category respondents have assigned other non-functional factors mentioned in Table 7.4, such as student-written test

inputs.

Our hypothesis was that instructors who did not compile and execute the code would give more weight to style than to functional correctness. The data does not appear to support this hypothesis, however, as the distribution of respondents who do not execute code is not skewed towards more style points. The sample size in this case is only 11, which is too small to draw any broad conclusions.

One threat to validity for this particular question is that of the 11 respondents who did not execute student code, 9 taught CS-1, where programs are likely small enough to evaluate without executing the code, and where the primary focus of each assignment may be algorithmic or stylistic rather than functional.

Interestingly, 6 of the 11 courses where student code was not executed had enrollments of at least 50 students, with an average enrollment of 132 students and an average of 10 minutes required to evaluate the functional correctness of each submission. That's over 22 hours of grading *per assignment* that could be reduced through automation, freeing the instructor or TAs to focus more time and energy on higher-level cognitive functions, such as innovative project or curriculum design or additional office hours.

The opposite of our hypothesis does appear to be true, i.e. that instructors who *do* execute code against their test cases tend to weight functional correctness more highly than style. This suggests that at least to some extent instructors give more weight to factors that they can measure more easily.

A final threat to the validity of these conclusions is that the names of the categories used—functional correctness and style—are too coarse. While the term

Submission Mechanism (55 responses)	#	%
Electronic submission system only	29	53%
Email only	9	16%
Printout and electronic system	9	16%
Email and printout	4	7%
Source code printout only	2	2%
Email and electronic system	1	<1%
Email, printout and electronic system	1	<1%

Table 7.6: Submission mechanisms used by survey respondents.

Course Management System (44 responses)	#	%
In-house system	16	36%
Blackboard	11	25%
WebCT	7	16%
Moodle	6	14%
Other commercial product	4	1%

Table 7.7: Electronic submission systems other than emailed used by survey respondents.

"functional correctness" unambiguously identifies whether a program obeys its specification, "style" does not adequately represent all types of non-functional correctness. For example, in a course that stresses object oriented design, non-functional code features such as modularity are far more important than the style features typically evaluated. Similarly, in a senior-level software engineering course, quality of documentation and design specifications are non-functional requirements that are substantially more important than either the style or correctness or the actual code.

7.4 Course Management Systems

Table 7.6 shows a breakdown of the submission mechanisms used by survey respondents to collect students' work. Over half of all respondents use some kind of

electronic submission system *other* than email, while about 16% rely on email alone. Only two respondents rely on paper printouts alone as the sole means of collecting students' work, which is encouraging as this method is not easily maintainable or scalable.

Table 7.7 shows the systems used by the 44 respondents who mentioned an electronic submission mechanism other than email. Blackboard proved to be the most popular product, although one respondent switched from blackboard to a Unix-based command line program because "Blackboard proved to be inadequate". The 16 in-house systems varied widely, ranging in complexity from electronic drop-boxes and Unix-based command-line submit scripts to a "Custom web-based submission application". That more than a third of faculty (or the support staff at their institutions) use an electronic submission system built and maintained in-house represents a lot of effort going into multiple systems that serve similar purposes. However, it is still encouraging that over 80% (44 out of 56) of respondents use some kind of electronic submission system rather than email and paper printouts, which are generally messy, time-consuming systems of course management.

7.5 Automated Style Checkers

Of the 56 respondents, 52 differentiate between style elements (indentation, variable naming scheme, comments, and so on) and functional correctness of code when grading student programs. Of the 52 respondents who differentiate between style and correctness, 50 answered that some of the features contributing to the style grade

include coding conventions like "following an indentation scheme, naming conventions, etc". Many of these simple coding conventions can be checked quickly, easily, consistently, and accurately for many programming language using an automated style checker, such as cxxchecker for C++ [10], FxCop for C# [18], and PMD [43] or Checkstyle [8] for Java.

Because the quality, maturity, licensing and availability of style checkers varies greatly between languages (i.e. C++ in general is difficult to parse and therefore can be difficult to evaluate stylistically), a suitable automated style checker may not be available for the language taught by each survey respondent. However, Java was taught by 35 of 50 the respondents who graded coding conventions, while only four of these 35 instructors used one of the freely available open-source products such as PMD or Checkstyle. Why are automated tools not being used to evaluate these aspects of style?

Course enrollment is likely one factor, as 12 of the 35 Java teachers who evaluate style have fewer than 20 students to grade, and while evaluating the stylistic conventions of twenty submissions is tedious, it is feasible in a reasonable amount of time. Of the remaining 23 survey respondents, 7 had course enrollments of more than 50 students—a substantial time commitment at 5 minutes per submission.

This type of work can always be parceled out to be done by TAs; however, having multiple TAs evaluate style is tricky, as TAs often have different interpretations of style, and maintaining and using an accurate, standardized rubric of style conventions can prove challenging. This is one area where a style checker can consistently and accurate handle the mundane details of style evaluation and free the

IDE (56 responses)	#	%
Eclipse	16	29%
BlueJ	9	16%
jGrasp	4	7%
Visual Studio Team System	3	5%
Dr Scheme	2	3%
Dr Java	1	2%
Netbeans	1	2%
Bordland	1	2%
None	19	34%

Table 7.8: IDEs used by survey respondents.

instructor or TA to focus on more complex stylistic issues, such as program design, choice of algorithms, and so on.

We are not advocating style checkers as a silver bullet that will completely eliminate humans from the evaluation of style. Some code features, such as the use of a for-loop rather than a while loop, algorithmic efficiency, or the modularity of an object-oriented design, still require a human to look at the code. However, we feel that a human can more effectively evaluate these higher-level stylistic concerns once freed from the drudgery of checking things like variable names and indentation.

7.6 Integrated Development Environments (IDEs)

Table 7.8 shows the breakdown of IDEs survey respondents used in their course. Eclipse [11] tops the list at 29% of respondents, with BlueJ coming in second at 16%. Of the 19 respondents who do not use an IDE in their classes, almost half of them (9 out of 19) teach in Java, for which several well-supported educational IDEs are available [1, 4], as well as powerful and freely available commercial IDEs [11, 42].

The other 10 respondents who don't use an IDE taught courses in languages for which an IDE is not as readily available (Tcl, C/C++, OCaml, Python) as for Java.

The most interesting result of this survey item is that about two-thirds of respondents (37 out of 56) use some kind of IDE, implying that for many the world has changed from the days of Unix-based text editors like vi and emacs and command-line compilation.

7.7 Perceived Impediments to Effective Grading

Table 7.9 presents the results of a section of our survey that asked respondents to evaluate to what extent something is "a significant impediment to providing effective feedback on student programming assignments". This section of the survey uses a four-point Likert scale of "Not at All", "Very Little", "Somewhat", or "To a Great Extent". These responses were assigned values from 1 to 4, respectively, when computing statistical rankings for each impediment. The text of these survey questions was taken directly from a similar survey conducated in 2004 by Hussein Vastani at Virginia Tech as part of his thesis [51]

The last two columns of Table 7.9 contain the mean of the Likert ranks for each impediment on this survey and for Vastani's survey in 2004. We applied the Mann-Whitney-Wilcoxon test to the observed responses for each potential impediment for both surveys; those marked with a * (star) on the table showed a statistically significant ($p < 0.05$) difference in the distribution of their responses.

It is difficult to evaluate why the results differ between Vastani's survey in

Impediment	Not at All	Very Little	Some-what	To a Great Extent	Avg	Vastani Avg
Too many assignments to assess	16%	29%	39%	14%	2.53*	3.10*
Not enough time or resources to do a thorough job	11%	20%	41%	29%	2.88	3.19
Technical knowledge, capabilities, or experience of the grader(s)	52%	27%	20%	2%	1.71	1.94
Lack of a consistent rubric (grading criteria) for grader(s) to follow	46%	30%	21%	2%	1.79	1.90
Poor code readability of student code	18%	39%	38%	4%	2.27*	2.76*
Poor layout and indentation of student code	23%	46%	25%	5%	2.12*	2.45*
Little or no commenting within the student code	16%	30%	50%	2%	2.38	2.56
The density of defects (bugs) in the student code	11%	54%	30%	5%	2.30*	2.86*
Poor testing of work by the student before submission	12%	27%	43%	18%	2.66*	3.15*
The logistics of executing code against instructor-provided tests to see if it works	25%	34%	21%	18%	2.33	2.85
Managing the submission of assignments and the return of results	30%	39%	21%	9%	2.09	2.26

Table 7.9: Survey results of respondents' evaluation of perceived impediments to effective grading, from the survey administered for this thesis. Impediments marked with a * showed statistically different (with $p < 0.05$) distributions of responses between Vastani's 2004 survey and this survey.

April 2004 and our survey in July 2006. One hypothesis as to why respondents of this survey rated "poor layout of student code" and "poor readability of student code" as less of an impediment than respondnets to Vastani's survey is that the use of IDEs (which generally handle layout and indentation automatically) has increased in the last two years. There is no way to test this hypothesis across the two surveys because Vastani's survey did not collect information about the use of IDEs in the classroom.

In our own survey, to our surprise we did *not* find a statistically significant difference between respondents who used an IDE and those who did not regarding either poor layout or poor readability of student code as a perceived impediment to effective grading. Similarly, we did *not* find a statistically significant difference between respondents who used course management software and those who did not regarding the management of student submissions as a perceived impediment.

Our results suggest that, while technologies such as IDEs and course management software are being adopted, they have no measurable effect on the perceived impediments to grading. One possible reason for this is that a four-point Likert scale is not fine-grained enough; many psychometricians recommend a seven or nine-point scale.

Finally, instructors rated "Poor testing of work by the student before submission" as one of the strongest impediments to grading, in both this survey and in Vastani' 2004 survey. This suggests that we should take a careful look at recent work geared towards introducing Test-Driven Development (TDD) to the curriculum [32, 29, 14].

7.8 Conclusions

We found statistically significant evidence that using an automated testing system takes about a third less time when evaluating correctness of student programs; this result is not unexpected, but is nonetheless an important data point supporting the use of automation in the grading process when possible.

We found that static style checkers have not penetrated very deeply into the grading process, that electronic course management systems and IDEs have achieved fairly deep penetration into academia, and that Java is taught in 87% of CS-1 courses taught by our survey respondents.

We also found evidence that instructors find the lack of testing by students to be a significant impediment to effective grading, lending additional support for testing in the undergraduate curriculum.

Chapter 8

Related Work

In this chapter, we discuss related work.

8.1 Automated Grading Systems

Hollingsworth [22] describes a very early automated grading system used at Rensselaer Polytechnic Institute in the early 1960s. This system is primarily of interest for historical purposes since it is the first published example of an automated assessment system we have found. Some of the major issues the system deals with, such as students who use the system not being "as skilled in machine operation", the fact that "[s]tudent programs can modify the grader itself", and the limitation to programs written in machine language, are no longer obstacles.

Isaacson and Scott [24] describe a Unix-based command-line system for automating the execution of student programs against test input. Their system reads test data from standard-in and writes outputs to standard-out; without a precise specification of the format for outputs, determining if an output is correct is difficult to automate.

Jackson et al. [25] describe the ASSYST, a semi-automated assessment system for evaluating programming assignments. Their system evaluates correctness, efficiency, style, complexity and test data adequacy. Their pre-dates the explosion

of unit testing by several years, and therefore only works with programs that read test inputs from standard-in and write output to standard-out. Such an approach is limited in that students must have all of the code for reading inputs and writing outputs working precisely according to specification before they can begin writing their own test cases. Furthermore, ASSYST does not assume that instructors will provide (nor that students will implement) a precise format for the output files, but rather employs a heuristic pattern-matching algorithm to determine if an output is correct. The extra level of complexity required to determine the correctness of an output could be reduced or eliminated using unit testing; however, as mentioned previously, their system pre-dates unit testing by several years. The interest that the ASSYST system designers showed in measuring the adequacy of students' test cases is an example of the recent swelling of interest in test-driven development.

Zeller describes Praktomat [54], an automated testing and code review system for introductory students. Praktomat allows students to submit multiple versions of their code which will be automatically tested. The automated testing system used *public* and *secret* test cases so the information revealed to the students upon submission was only part of a correct solution. Upon submission, students can see other students' solutions to the project, and can comment on the other student's code while also receiving comments for their own code. To prevent plagiarism, the requirements for each project were different. The comments from the students about the utility of sharing commentary about code were positive. Their results imply that students who sent more code commentaries tended to write clearer code than students who did not, though this conclusion is not strongly backed up in the

paper.

Ellsworth et al describe Quiver [15], an automated QUIz VERification tool. Quiver provides a closed-lab environment where students have a limited amount of time to complete a small programming assignment, such as sorting an array or building a binary adder with GUI tools. Quiver works with C++ and Java and allows quizzes to be built automatically from descriptions of the required test cases that should be passed. Students are required to write the code in a specific editor running on the client machine. They discovered anecdotal evidence that students who were able to pass the course with a C by perform well enough on exams and pouring many hours into out-of-class programming assignments struggled to perform adequately on quizzes given through the Quiver system.

Stephen Edwards describes Web-CAT [12], an automated testing system developed at Virginia Tech. Web-CAT automatically tests students' submissions against instructor-written test-suites and evaluates the adequacy of student-written test suites by evaluating their code-coverage. Web-CAT is stable and mature enough to be used at other institutions. Web-CAT does not address the issue of giving limited feedback to students before the submission deadline, nor does it collect the fine-grained research data collected by Marmoset.

8.2 Data Collection with BlueJ

Jadud's work [26] attempts to discover particular patterns of syntax errors by novice users. He instruments BlueJ to capture a variety of information about when students

compile and what errors are made in order to characterize the typical errors that novices make. All capturing is done in a laboratory setting during weekly lab sections and there is no comparison of novice and expert behavior. This study reveals that a small number of frequent errors account for the majority of the total errors and that students tend to make small changes quickly when their code does not compile and tend to make larger changes after a successful compilation. This is not at all surprising, and suggests that students program in a different way when they are fixing syntax errors than when they are adding new functionality.

8.3 Hackystat

Hackystat [28] is a data-collection framework developed by Philip Johnson's research group at the University of Hawaii. Hackystat is the collective name for a suite of tools that plug in to various software development components, such as `emacs, vi, Eclipse`, or `make`, tracks detailed information about when and how developers use these tools. Hackystat is a pioneering technology for studying the software development "microprocess", where instead of studying hundreds of developers working on millions of lines of code for several years, we instead study the interactions between a single developer and her development tools over the course of days or even hours.

The Eclipse plugin for Hackystat is similar to the Course Project Manager plugin used by Marmoset, but with the notable difference that Hackystat does not capture snapshots of the full state of students' files. The data collected by Hackystat is orthogonal to the data collected by Marmoset, but would be extremely useful for

future studies of the Marmoset data.

There were two reasons why we did not use Hackystat to collect additional data: First, installing and configuring Hackystat requires more effort on the part of the user than installing Marmoset's Course Project Manager plugin, and we did not want to burden novices in CS-1 and CS-2 with the extra overhead; and second, the data collected by the Course Project Manager plugin is of direct benefit to the students (since it essentially records automatic backups), while Hackystat collects a lot of data that is not directly beneficial to students and would likely require a separate Institutional Review Board (IRB) authorization.

8.4 Software Repository Mining

Much of the interest in CVS repository mining was spurred by Ball et al's seminal paper [3] that encouraged analysis of source code repositories. This initial paper, while sparse, lays the groundwork for further exploration of source code repository mining.

Liu and Stroulia [34] talk about JReflex, the name for their system consisting of plugins built into Eclipse and a set of web-accessible wiki services. JReflex is designed to help students plan collaborative software development projects online. They use the same data in later work [35] for a case study using JReflex. Their work is still preliminary and their sample size thus far is only 5 teams of student-programmers. One interesting conclusion of their work is that many students use CVS in ways other than what was intended, for example primarily as a place to

store data. Their work primarily focuses on studying and improving the interactions between many students working on a group project. Our work differs in that we focus on individual students all working on the same large project, and we capture CVS commit information transparently, without the students' needing to do anything.

Zimmerman and Weisgerber [55] outline some of the issues involved in pre-processing CVS repositories for fine-grained analysis, making the observation that the quality of the data is directly proportional to the quality of the pre-processing. They are concerned with processing the repositories for large projects with multiple developers, and are interested in four major issues: data extraction to a database, recovery of transactions (i.e. commits of multiple files at the same time), mapping changes to fine-grained program entities such as functions rather than simple source lines, and special handling of certain transactions such as merge changes or large changes resulting from major infrastructure modifications. They present techniques for each of these issues.

Mierle et al [39] examine CVS logs for a second-year computer science course. They do not mention anything out-of-the ordinary regarding the collection of this data, so we assume that the CVS data was collected in the usual fashion (i.e. the students manually perform CVS operations). They focus on trying to find source-code artifacts in the CVS repositories that are co-related with whether a student finishes in the top or bottom third of the class. They examine many artifacts and apply data-mining techniques, and found that the only two factors that showed a weak correlation with a student's placement were the number of lines of code written by the student and the number of commas with spaces after them. It is unclear form

the paper if they controlled for the fact that the number of commas with spaces after them is related to the size of the file. Interestingly, they found no correlation between the amount or type of CVS repository activity and the student's performance in the course. Based on their inquiries, they have concluded that mining information from a source code repository is perhaps not as easy as expected, or else they feel they would have found better predictors of a student's success in the course.

Purushothaman and Perry [44] have conducted an interesting study of the types of changes made to a source code repository and their effects on the codebase. Specifically, they study the effects of one-line changes to source code over time. For the codebase they studied, one-line changes made up over 10% of all changes during maintenance and one-line changes had a 4% chance of introducing a bug. They also found that all one-line changes were not equal, in that additions, deletions and modifications had different properties. Based on their findings they conclude that one-line changes can cause bugs but did not find any so-called disastrous one-line changes. Future work for this project is to study similar properties of one-line changes in other software projects.

German [19] describes mining source code repositories with the softChange tool. SoftChange [45] is a tool designed to automatically extract software change "trails" from open-source projects by looking principally at mailing list archives, bugzilla databases, and the CVS repository. This work describes preliminary experiences using softChange on the Evolution codebase. German concludes that the amount of data is overwhelming and that more work is needed to find useful information in the source code repository.

Williams and Hollingsworth [53] analyze bug database and source code repositories to develop bug finders useful for that software project. They also have a novel false-positive filtering mechanism based on identifying patterns that triggered previous bug fixes.

8.5 Test-Driven Development in the Curriculum

Recent literature on introducing testing into the computer science curriculum focuses on several key issues:

- Testing cannot only be taught in an upper-division course. In order to truly learn the value of testing, students need to be exposed to software testing throughout the curriculum.

- Instructors need to design testable programming assignments. This requires additional overhead for instructors who have already developed project descriptions that are not easily amenable to any type of automated testing framework.

- Students need to directly experience benefits from writing test suites. Requiring students to write test cases simply because test suite quality will be graded does not help students learn the value of testing.

- Teaching testing well requires additional infrastructure over-and-above what would be required in a more traditional programming course. For example, students must be taught to use a testing framework (such as JUnit), and

instructors need a way to evaluate the quality of student-written test suites (such as a code coverage tool). This requires more automation than is typically required of grading scripts, and can be a barrier to entry for some instructors.

- Software testing is large subject area that could include many activities, such as using a unit test framework, reading code, using a debugger, learning to identify faults based on error logs, and writing and testing specifications. Thus introducing testing into the curriculum can mean introducing any or all of these activities. However, most of the literature focuses on the much simpler task of introducing test-driven development, primarily through unit testing, into the curriculum.

These issues imply one other issues not explicitly stated in the literature:

- Test-Driven Development relies on rapid feedback. For programming assignments that perform all the grading after the deadline, students are *not* receiving feedback quickly enough to learn from their mistakes. Students need feedback about the quality of their code as well as about the quality of their test suites *while they are working* in order to fully appreciate the value of test-first coding practices.

A more detailed survey of the literature follows.

Christensen [9] argues that software testing cannot be an isolated topic taught in an upper-division course; it needs to permeate the curriculum so that students learn the value of testing, and have that knowledge constantly reinforced throughout their education.

Christensen observes that "it is vital that teachers ensure that the students benefit from their tests." In other words, instructors need to create assignments that allow students to directly experience the value of testing.

Christensen teaches testing by having each project consist of a progression of steps such that step N requires step N-1. In this way students need to use code from previous steps and cannot implement each project from scratch. This approach means that students are constantly modifying code they've already written and will directly experience the benefit their test suites provide through regression testing.

The Web-CAT system [12], built by Stephen Edwards at Virginia Tech, has a very rich set of features designed to support TDD and is stable and mature enough to be used at other universities. Web-CAT allows students to submit their own test suites, which are evaluated for code coverage. In addition, Edwards takes the novel approach of basing a students grade on the product of the percent of their test own cases passed, the percent of instructor's tests passed, and the percent of code coverage achieved. Web-CAT also provides style feedback from style checkers such as PMD and CheckStyle, as well as the ability for TAs or instructors to enter style comments of their own. Students can then access a web page that incorporates their code coverage, style warnings, and instructor or TA style comments into a single marked-up view of their source code.

Web-CAT solves several practical problems that Marmoset also solves: It automates the execution of student code against the instructor's test cases to be used for grading and collects detailed code coverage information on student-written test suites. However, Web-CAT does not address some key issues, such as the situation

where a test case fails but covers only code also covered by passing test cases, how to create additional incentives for students to write their own test cases, or how to provide feedback to students about their progress on a programming assignment without giving away all of the instructor's test cases (and inadvertently encouraging students to code to the test cases).

Jones[30] calls for the integration of a variety software testing and debugging experiences into the curriculum by exposing students to as much of the SPRAE (Specification, Premeditation, Repeatability, Accountability, Economy) framework as possible. This paper is somewhat broader in scope than much of the literature: Jones recommends that students learn not only to debug a program given test cases and a test log, but also to develop test logs given test results and test cases (presumably with sufficient detail so that someone else could perform the debugging) and to write test cases based only on a specification. This is in contrast with much of the literature, which is more narrowly focused on getting students to write unit tests as they develop their own code.

The framework built by Jones includes an automated program grading system (APGS) which can compile and execute both student and instructor-written test cases automatically. The APGS rigidly specifies the format for input and output and performs deductions based on incorrect lines of output. Based on the paper, the APGS seems to require more overhead for instructors than a lightweight unit testing framework, such as JUnit of cxxUnit.

This work touches on two important issues: First, to teach students how to write their own test cases, instructors need to design *testable* assignments—this may

seem an obvious point, but it is nonetheless important; and second, the results of test cases need to be *repeatable*. This notion of repeatability of test outcomes does not show up in the literature; it is somewhat disconcerting to imagine the number of buggy grading scripts that have assigned students a low grade one or two weeks after the project deadline, at which point the student was so busy working on the next project that they never noticed the error.

Goldwasser[21] proposes a novel, innovative system where students submit their test suites in addition to their implementations and receive points for exposing bugs in other students' programs (or even in the instructor's reference implementation) with their test suites. The projects used rely on reading and writing from the standard input and therefore typically require the instructor to write a reliable front-end parser. Furthermore, because these projects rely on standard-in, they are not as easily amenable to unit or API testing. Furthermore, the quadratic growth of running all students' test suites against all other students' submissions requires automation, which they provide in the form of a perl script. Finally, since all submissions are needed to start the all-to-all testing process, this means students do not receive feedback until after the deadline.

This work suggests two things: First, providing students with appropriate incentive to write test cases is an important issue (presumably because students don't write test cases otherwise), and second, innovative ways to motivate testing require additional infrastructure over what would be requires to teach a traditional programming course.

Leska incorporated testing into a CS1 course taught in Java with an "objects first" flavor [32]. He broke testing down into three related "quality assurance" activities: system-level (black-box) testing, unit or API testing (though they didn't use JUnit), and code reading exercises. This approach was influenced by recommendations by Jones [30] that students be exposed to a variety of software quality assurance methods throughout the curriculum.

The paper makes no mention of how the students' test suites were evaluated, nor whether student programs were graded using the instructor's test suite. The author mentions that he wanted to introduce JUnit but didn't want to overburden himself or the students. This suggests that infrastructure required to focus on testing in introductory courses is greater than what is required to teach the same course without the emphasis on testing.

Jones performed a survey of eight different efforts towards introducing Test-Driven Development (TDD) into the curriculum [29]. Three of the papers, by Edwards [13], Muller [40], and Kaufmann [31], that were studied contained controlled studies of the effectiveness of teaching test-first coding.

Edwards [13] provided some students instruction on how to use the Test-Driven Development features of Web-CAT [12], a rich framework for executing and evaluation student-written test cases. These students were then compared with students who had taken the course prior to the existence of Web-CAT, and found that the students who used TDD wrote code that was statistically better along several axes, including defects per thousand lines of code and adherence of the code to the specification. In addition, around two thirds of students who used TDD

reported in an anonymous survey that TDD increased their confidence in both the correctness of their code as well as in the ability to make changes to their code without introducing new bugs.

Muller [40] conducted a controlled study with graduate students who had taken a course on Extreme Programming methodology. The students were split into a TDD group and a control group. They found that there was not a statistically significant difference in productivity or quality between the TDD group and the control group; however, the TDD group did a better job of reusing methods at a statistically significant level.

Kaufman and Janzen [31] performed a controlled study of test-first versus test-last coding in an upper-level "Software Studio" course. They found that students who employed test-first coding practices produced more total code, but that their code exhibited a weaker coupling between classes.

Jones points out in his survey that while the results of these three controlled studies are mixed and the sample sizes were not large enough, nor the methodologies rigorous enough, to draw any major conclusions, TDD nonetheless has shown promise as a valuable part of the computer science curriculum.

Jones also wisely cautions that TDD is *not free*, in that it requires more infrastructure, expertise, and effort by instructors than is required to teach a CS course that does not focus on TDD.

Marrero and Settle [37] introduced testing to two introductory Java programming courses by requiring students to write test cases for a binary implementation *before* submitting their own implementation. To further motivate testing, they staged

a competition similar to that recommended by Goldwasser [21] in one of courses that had few students. This work mentions that the "goal is to have students place a greater emphasis on testing with minimal added work for the instructor", suggesting that they lacked appropriate infrastructural support to fully embrace the approach.

This was a very small study, and the results of this work from a quantitative perspective were inconclusive. However, the anecdotal evidence presented—that students who were required to test code began asking more detailed questions about project specifications in order to clarify the type of test cases they should write—was very promising for this approach.

Marick [36] details how an organization can easily misuse or misinterpret code coverage results. Although the paper is geared towards industry, the main lessons are directly applicable to the classroom (where code coverage will most likely be used to evaluate student-written test suites).

The primary lesson is that a high code coverage number does not mean that a program is adequately tested. Marick illustrates this point by highlighting the subtle difference between a programmer who "expects" a high level of coverage when they write a test suite and a manager who "requires" a high level of coverage before a product can ship. The manager in this example is analogous to an instructor who requires students to achieve a certain level of code coverage—say 80% of statements and 90% of branches— in order to achieve a certain grade. The problem is that achieving high code coverage does not necessarily imply adequate testing because it is often easy to write trivial test cases simply to improve the reported coverage numbers. Poor code coverage should be interpreted as a hint that part of the test

suite is weak and requires some additional thought to strengthen. Marick observes: "If a part of your test suite is weak in a way coverage can detect, it's likely also weak in a way coverage can't detect".

Chapter 9

Conclusions and Future Work

In this chapter we conclude and discuss directions for future work.

9.1 Conclusions

In this work, we have successfully built and deployed Marmoset, an automated testing system that provides advanced feedback to both students and instructors, and collects fine-grained data for researchers to study the novice programming process.

We have demonstrated the richness and usefulness of the Marmoset dataset by mining new bug patterns from student mistakes, and by evaluating the precision and recall of the open-source static checker FindBugs. The Marmoset dataset allows us to evaluate not only precision (false positive rates) but also recall (false negative rates), something that had proved extremely difficult to the static error checking community.

We have also conducted a multi-institution study of grading practices for programming assignments; this survey shed light on the current grading practices for programming assignments as well as the relative weights of style vs. functional correctness as proportions of the final grade. Our survey revealed a number of interesting trends, such as the penetration of industrial-strength IDEs such as Eclipse and educational IDEs such as BlueJ, into the classroom, and the lack of penetration

of automated style checkers such as PMD and Checkstyle into the classroom.

We also examined projects designed at encouraging students to adopt a Test-Driven Development methodology; these projects caused students to write more test cases, but did not always encourage students to write test cases while development their software, as many test cases were written *after* students had already passed all the test cases. This suggests that without proper motivation, many students fail to see the importance of writing good cases, but rather view testing as an artificial hurdle. This line of research has inspired us to make changes to Marmoset that reward students for writing better test suites; these changes will be evaluated in future work.

9.2 Future Work

This research has been wildly successful at opening up avenues of future work and setting up future collaborations.

The Marmoset dataset was one of the datasets used by Chadd Williams in his thesis [52]. He used the submission history of the CS-3 data to detect source code properties and then find violations of those properties.

The dataset has also been used in the context of educational data mining [49] to evaluate the accuracy of clustering algorithms for figuring out related unit tests based on the unit test outcomes for a large number of students.

In the Fall 2006 semester, Marmoset will be used at other institutions.

All of the studies we've performed using Marmoset data, such as mining new

bug patterns and evaluating the precision and recall of FindBugs, were done using the four semesters of CS-2 data that are available. We have not had a chance to look at the CS-1 data, or at the data available for other classes that have adopted Marmoset; one future avenue of research is to examine the CS-1 data.

Our dataset swells with every semester that passes. At Maryland, CS-1, CS-2 and CS-3 reliably use Marmoset every semester. Furthermore, each semester new courses at all levels adopt their programming project sequence to use Marmoset. We have two semesters of projects for a senior-level course on advanced Java technologies that we haven't analyzed; some of these projects use threads, XML, and RMI, and may contain interesting bug patterns to be analyzed.

We also have not performed a controlled study of release testing. This will probably happen in the next year because two courses at Maryland adopted Marmoset purely to automate grading by making all the test cases secret. In future semesters, these two courses will use the same project sequence and will make some of these test cases public and release tests, allowing us to study release testing as an independent variable between semesters.

Appendix A

Text of the Survey

This appendix contains the text of each of the survey questions.

Survey about grading programming assignments for computer science courses

This survey should take you only a few minutes to fill out. Please answer from the point of view of the most recent course you taught that involved programming assignments.

In the following questions the term "grader" refers to any instructional staff responsible for marking up or making comments on students programming assignments for the purpose of grading or reviewing.

If you are the grader, please answer the questions for yourself. If you have appointed someone to do part or all of the grading (e.g., a TA), then answer the questions with respect to the expectations you have with your grader.

Please use the "Additional Details" box to the right of each question to provide any additional information you feel would be useful, for example if the question does not pertain to your situation or if the answer you would like to provide is not listed.

Note that the text of some of these questions is taken directly from a survey administered in 2004 by Hussein Vastani and Stephen Edwards from Virginia Tech University.

Background on you and your institution:		**Additional Details**
What is the name of your college or university?		
About how many total undergraduates at your institution (in all fields, not just Computer Science)?		
Does your institution have graduate students and Teaching Assistants?	○ No ○ Yes	
Are you a Professor, Instructor or Teaching Assistant?	○ Professor ○ Instructor ○ Teaching Assistant	
About how many Computer Science majors graduate per year at your institution?		
Background for the course on which you will base the rest of your answers on this survey:		**Additional Details:**
What level best describes this course?	○ Freshman ○ Sophomore ○ Junior ○ Senior ○ Graduate	
What best describes the title of the course?	○ CS-0 ○ CS-1 ○ CS-2 ○ Programming Languages ○ Compilers ○ Networks ○ Operating Systems	

	○ Artificial Intelligence ○ Computer Organization / Architecture ○ Databases ○ Software Engineering ○ Computer Graphics ○ Other:	
About how many students were enrolled in the sections of the course that you taught?		
What was the primary programming language that students used to implement programming assignments?	○ Java ○ C ○ C++ (choose C++ if students can use either C or C++) ○ Scheme ○ Lisp ○ C# ○ Haskell ○ Prolog ○ Python ○ Ruby ○ Perl ○ Objective Caml ○ Pascal ○ Fortran ○ Other:	
Did this course use an Integrated Development Environment (IDE)?	○ Eclipse ○ BlueJ ○ Dr Java ○ Net Beans ○ Visual Studio ○ Other: ○ No, this course did not use an IDE.	

Please rate the degree to which you believe that each of the following is a significant impediment to providing effective feedback on student programming assignments:

Issue	To a Great Extent	Somewhat	Very Little	Not At All	Additional Details:
Too many assignments to assess	○	○	○	○	
Not enough time or resources to do a thorough job	○	○	○	○	
Technical knowledge, capabilities, or experience of the grader(s)	○	○	○	○	

Lack of a consistent rubric (grading criteria) for grader(s) to follow					
Poor code readability of student code					
Poor layout and indentation of student code					
Little or no commenting within the student code					
The density of defects (bugs) in the student code					
Poor testing of work by the student before submission					
The logistics of executing code against instructor-provided tests to see if it works					
Managing the submission of assignments and the return of results					

General background on grading criteria and submission mechanisms											**Additional Details:**
When grading, do you evaluate style (indentation scheme, variable naming conventions, comments, etc) separately from functional correctness (does the code do what the specification says that it is supposed to do)? **Note:** If you do not distinguish between style and functional correctness when grading, answer the rest of the survey as best you can.	Yes No										
Average # of points (out of 100) for functional correctness. Relative weights of functional	0 100	10 90	20 80	30 70	40 60	50 50	60 40	70 30	80 20	90 10	100 0	

correctness vs. style when grading a programming assignment. Average # of points (out of 100) for other considerations (style, comments, documentation, etc).		
How do students submit their programming assignments for assessment? Check all that apply.	☐ They turn in a source code printout ☐ They email their submission ☐ They submit their code electronically using a mechanism *other* than email. (Please describe in the "additional details" box). ☐ Other	
When grading programming assignments, what contributes to the final score for an assignment? Check all that apply.	☐ Programing Style (Are students following an indentation scheme, naming conventions, etc?) ☐ Comments (Is the code adequately commented, is there documentation, etc?) ☐ Student-Written Test Cases (Have students written test cases for their code?) ☐ Functional Correctness (Does the program do what it is supposed to do?) ☐ Other:	
Evaluating the *style* of student submissions.		**Additional Details:**
On average, how many minutes does the grader spend on each student's assignment for style?	○ < 5 ○ 5-10 ○ 10-15 ○ 15-30 ○ > 30	
How does the grader read student submissions while evaluating *style*? Check all that apply.	☐ Using a paper printout ☐ Directly accessing the source code files at a computer ☐ Using an interface provided by an electronic submission and/or grading system ☐ Other	
Do you use any automated tools to evaluate the style of student source code?	○ FindBugs ○ PMD ☐ Checkstyle ○ FxCop ○ Other:	

	☐ No, we do not use any automated tools to evaluate style.	
Evaluating the *functional correctness* of student submissions		**Additional Details:**
On average, how many minutes does the grader spend on each student's assignment evaluating functional correctness?	☐ < 5 ○ 5-10 ○ 10-15 ☐ 15-30 ○ > 30	
How is the student work assessed for functional correctness? Check all that apply.	☐ By reading the source code ☐ The student provides a printout of test results ☐ The student provides a live demonstration for the grader ☐ The grader hand-executes the student program against instructor-provided data ☐ An automated tool compiles and executes the student code against instructor-provided data ☐ Other	
Are student submissions executed against instructor-written test cases or test-data?	○ No. ○ Yes.	
If student submissions are executed against instructor-written test cases, what testing framework (if any) do you use?	○ JUnit ○ CppUnit ☐ CxxTest ○ MbUnit ○ NUnit ☐ Visual Studio Team System ○ Other: ☐ We do not execute student submissions against instructor-written tests	
If student submissions are executed against instructor-written test data, please describe the infrastructure used in more detail. -OR- If student submissions are *not* executed against instructor-written test data, what do		

you see as the biggest impediment towards doing so?		
If you run students' programs against test inputs or test cases to determine functional correctness, do you give students any of the test cases ahead of time?	○ We give students *all* the test cases that will be used for grading *before* the deadline ○ We give students *some* test cases before the deadline, but keep some test cases private until *after* the deadline. ○ We do not give students *any* of the test cases before the deadline. ○ Other:	
Is there any incentive for students to submit *before* the deadline? For example, if students submit early, do they receive any additional feedback, such as the results of running their submission against some or all of the instructor's private test inputs?	○ No. ○ Yes. Please describe:	
Please describe any other automated tools or technologies that were used in this course (i.e. memory checkers such as Purify or Valgrind, code coverage tools such as Clover or Emma, etc)		
Archiving submissions, course management software		**Additional Details:**
Do you maintain an archive of electronic copies of student program submissions for	○ No ○ Yes, for all assignments in this course this semester/term ○ Yes, for all assignments in this course over multiple semesters/terms ○ Yes, for all assignments in multiple courses over multiple semesters/terms	

your course?		
If you maintain an archive of electronic copies of student program submissions, what are the archived programs used for? Check all that apply.	☐ Resolving grade disputes ☐ Providing a backup of student work ☐ Detecting plagiarism/cheating ☐ Learning outcomes assessment for my course ☐ Longitudinal curricular assessment over multiple courses ☐ Computer Science Education (CSEd) research ☐ Other:	
Do you use any course management software?	○ No, we don't use course management software ○ Blackboard ○ WebCT ○ Moodle ○ Other	
Free response section:		
Given a choice, what part of the grading process would you automate and why?		
What is the most time consuming or difficult part of the grading process and why?		
What is the biggest disadvantage of the grading method used by you or your grader and why?		
Submit!		

BIBLIOGRAPHY

[1] E. Allen, R. Cartwright, and B. Stoler. Drjava: A lightweight pedagogic environment for java sigcse, 2002.

[2] Apache Tomcat. `http://tomcat.apache.org/`, 2006.

[3] Ball, T. and Kim, J. and Porter, A. and Siy, H. If your version control system could talk. In *Proceedings of the International Workshop on Principles of Sofware Evolution (IWPSE)*, May 1997.

[4] David J. Barnes and Michael Klling. *Objects First with Java - A Practical Introduction using BlueJ.* Prentice-Hall, September 2002.

[5] Kim B. Bruce. Controversy on how to teach cs 1: a discussion on the sigcse-members mailing list. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 29–34, New York, NY, USA, 2004. ACM Press.

[6] Caucho Resin : Fast, Open-Source Application Server. `http://www.caucho.com/`, 2006.

[7] Cenqua Clover Code Coverage for Java. `http://www.cenqua.com/clover/`, 2006.

[8] Checkstyle. `http://checkstyle.sourceforge.net/`, 2006.

[9] Henrik Bærbak Christensen. Systematic testing should not be a topic in the computer science curriculum! In *ITiCSE '03: Proceedings of the 8th annual*

conference on Innovation and technology in computer science education, pages 7–10, New York, NY, USA, 2003. ACM Press.

[10] C++ source-code style check. url=https://gna.org/projects/cxxchecker, 2006.

[11] Eclipse.org main page. http://www.eclipse.org, 2004.

[12] Stephen H. Edwards. Rethinking computer science education from a test-first perspective. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 148–155, New York, NY, USA, 2003. ACM Press.

[13] Stephen H. Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on perfomance. 2003.

[14] Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 26–30, New York, NY, USA, 2004. ACM Press.

[15] Christopher C. Ellsworth, Jr. James B. Fenwick, and Barry L. Kurtz. The quiver system. *SIGCSE Bull.*, 36(1):205–209, 2004.

[16] FindBugs—Find Bugs in Java Programs. `http://findbugs.sourceforge.net`, 2006.

[17] Fortify Software. `http://www.fortifysoftware.com`, 2006.

[18] Fxcop team page. url=http://www.gotdotnet.com/team/fxcop/, 2006.

[19] Daniel German. Mining CVS repositories, the softChange experience. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[20] Robert L. Glass. Persistent software errors. *IEEE Trans. Software Eng.*, 7(2):162–168, 1981.

[21] Michael H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 271–275, New York, NY, USA, 2002. ACM Press.

[22] Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, 1960.

[23] David Hovemeyer. *Simple and Effective Static Analysis to Find Bugs*. PhD in Computer Science, University of Maryland, College Park, College Park, MD, 2005.

[24] Peter C. Isaacson and Terry A. Scott. Automating the execution of student programs. *SIGCSE Bull.*, 21(2):15–22, 1989.

[25] David Jackson and Michelle Usher. Grading student programs using assyst. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 335–339. ACM Press, 1997.

[26] Matthew C Jadud. A first look at novice compilation behavior using bluej. In *Proceedings of the 16th Workshop on Psychology of Programming Interest Group*, Carlow, Ireland, April 2004.

[27] JBoss. `http://www.jboss.org`, 2005.

[28] Philip M. Johnson, Hongbing Kou, Joy 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 *ISESE*, pages 136–144. IEEE Computer Society, 2004.

[29] Christopher G. Jones. Test-driven development goes to school. *J. Comput. Small Coll.*, 20(1):220–231, 2004.

[30] Edward L. Jones. Software testing in the computer science curriculum – a holistic approach. In *ACSE '00: Proceedings of the Australasian conference on Computing education*, pages 153–157, New York, NY, USA, 2000. ACM Press.

[31] Reid Kaufmann and David Janzen. Implications of test-driven development: a pilot study. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299. ACM Press, 2003.

[32] Chuck Leska. Testing across the curriculum: square one! *J. Comput. Small Coll.*, 19(5):163–169, 2004.

[33] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate

Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 119–150, New York, NY, USA, 2004. ACM Press.

[34] Ying Liu and Eleni Stroulia. Reverse engineering the process of small novice software teams. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 102. IEEE Computer Society, 2003.

[35] Ying Liu, Eleni Stroulia, Ken Wong, and Daniel German. Using CVS historical information to understand how students develop software. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[36] Brian Marick. How to misuse code coverage. In *International Conference and Exposition on Testing Computer Software*, June 1999.

[37] Will Marrero and Amber Settle. Testing first: emphasizing testing in early programming courses. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 4–8, New York, NY, USA, 2005. ACM Press.

[38] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of

programming skills of first-year cs students. In *ITiCSE-WGR '01: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180, New York, NY, USA, 2001. ACM Press.

[39] Keir B. Mierle, Kevin Laven, Sam T. Roweis, and Greg V. Wilson. CVS Data Extraction and Analysis: A Case Study. Technical Report 2004-002, University of Toronto, September 2004.

[40] M. Muller and O. Hagner. Experiment about test-first programming. *Software, IEE Proceedings- [see also Software Engineering, IEE Proceedings]*, 149(5):131–136, 2002.

[41] MySQL AB :: The world's most popular open source database. `http://www.mysql.com/`, 2006.

[42] Netbeans. `http://www.netbeans.org/`, 2006.

[43] PMD. `http://pmd.sourceforge.net`, 2005.

[44] Ranjith Purushothaman and Dewayne Perry. Towards Understanding the Rhetoric of Small Changes. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[45] softChange, software change analysis tool. http://sourceforge.net/projects/sourcechange, 2004.

[46] Jaime Spacco, David Hovemeyer, and Bill Pugh. An eclipse-based course project snapshot and submission system. In *3rd Eclipse Technology Exchange Workshop (eTX)*, Vancouver, BC, October 24, 2004.

[47] Jaime Spacco, David Hovemeyer, and Bill Pugh. Tracking defect warnings across versions. In *MSR 2006: Proceedings of the 3rd annual workshop on Mining Software Repositories*, 2006.

[48] Jaime Spacco, David Hovemeyer, William Pugh, Jeff Hollingsworth, Nelson Padua-Perez, and Fawzi Emad. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. In *ITiCSE '06: Proceedings of the 11th annual conference on Innovation and technology in computer science education*. ACM Press, 2006.

[49] Jaime Spacco, Titus Winters, and Tom Payne. Inferring use cases from unit testing. In *AAAI Workshop on Educational Data Mining*, New York, NY, USA, July 2006. ACM Press.

[50] 2004-2005 taublee survey. http://www.cra.org/CRN/articles/may06/taulbee.html, 2006.

[51] Hussein K. Vastani. Supporting Direct Markup and Evaluation of Students Projects On-line. Master's thesis, Virginia Tech University, Blacksburg, VA, June 2004.

[52] Chadd Williams. *Using Historical Data From Source Code Revision Histories to Detect Source Code Properties*. PhD in Computer Science, University of Maryland, College Park, College Park, MD, 2006.

[53] Chadd Williams and Jeff Hollingsworth. Bug-Driven Bug Finders. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[54] Andreas Zeller. Making students read and review code. In *ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education*, pages 89–92, New York, NY, USA, 2000. ACM Press.

[55] Thomas Zimmerman and Peter Weisgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.