Automatic Continuous Testing to Speed Software Development

by

David Saff

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2004

Author .....
Department of Electrical Engineering and Computer Science
February 3, 2004

Certified by .....
Michael D. Ernst
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Automatic Continuous Testing to Speed Software Development

by

David Saff

Submitted to the Department of Electrical Engineering and Computer Science
on February 3, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Continuous testing is a new feature for software development environments that uses excess cycles on a developer's workstation to continuously run regression tests in the background, providing rapid feedback about test failures as source code is edited. It is intended to reduce the time and energy required to keep code well-tested, and to prevent regression errors from persisting uncaught for long periods of time. The longer that regression errors are allowed to linger during development, the more time is wasted debugging and fixing them once they are discovered.

By monitoring and measuring software projects, we estimate that the *wasted time*, consisting of this preventable extra fixing cost added to the time spent running tests and waiting for them to complete, accounts for 10–15% of total development time. We present a model of developer behavior that uses data from past projects to infer developer beliefs and predict behavior in new environments — in particular, when changing testing methodologies or tools to reduce wasted time. This model predicts that continuous testing would reduce wasted time by 92–98%, a substantial improvement over other approaches we evaluated, such as automatic test prioritization and changing manual test frequencies.

A controlled human experiment indicates that student developers using continuous testing were three times more likely to complete a task before the deadline than those without, with no significant effect on time worked. Most participants found continuous testing to be useful and believed that it helped them write better code faster. 90% would recommend the tool to others. We show the first empirical evidence of a benefit from continuous compilation, a popular related feature.

Continuous testing has been integrated into Emacs and Eclipse. We detail the functional and technical design of the Eclipse plug-in, which is publicly beta-released.

Thesis Supervisor: Michael D. Ernst
Title: Assistant Professor

Acknowledgments

Many thanks to my wife Rebecca and son Ethan for their patience and support beyond the call of duty, and to my advisor for his wisdom, flexibility, and hard work on my behalf.

Special thanks to the students of MIT's 6.170 course who participated in the user study, and the course staff who helped us to run it. This research was supported in part by NSF grants CCR-0133580 and CCR-0234651 and by a gift from IBM. Thank you to Sebastian Elbaum, Derek Rayside, the Program Analysis Group at MIT, and anonymous ISSRE and ICSE reviewers for their comments on previous versions of some of this material.

Contents

List of Figures

Chapter 1

Introduction

We introduce here the idea of *continuous testing*, which uses real-time integration with the development environment to asynchronously run tests against the current version of the code, and notify the developer of failures. The developer never needs to explicitly run the test suite. The process can be tuned, by prioritizing tests and parts of tests, to approach the illusion that the entire test suite runs instantaneously after every small code change, immediately notifying the developer of regression errors.

The purpose of continuous testing is to reduce two varieties of wasted time related to testing. The first source of wasted time is time spent running tests: remembering to run them, waiting for them to complete, and returning to the task at hand after being interrupted to run tests. The second source of wasted time is time spent performing development while errors exist in the system. Performing development in the presence of an error lengthens the time to correct the error: more code changes must be considered to find the changes that directly pertain to the error, the code changes are no longer fresh in the developer's mind, and new code written in the meanwhile may also need to be changed as part of the bug fix. The longer the developer is unaware of the error, the worse these effects are likely to be.

Developers without continuous testing can trade these two sources of wasted time off against one another by testing more or less frequently. Test case selection [21, 31] and prioritization [39, 32] can reduce the time spent waiting for tests. Editing while tests run can also help, but further complicates reproducing and tracking down errors.

However, there is intuitive appeal to the idea that automating the testing process, and freeing developers from having to consciously decide how to budget their time to keep code well-tested during development, would yield even better results:

- It has been shown [2, 5] that a long interval between the introduction and discovery of a defect dramatically increases the time required to fix it, when these intervals are measured in days or weeks. Continuous testing should be useful if this effect also holds on scales of hours, minutes, or even seconds.

- Test-driven development [4], an aspect of agile development methodologies such as Extreme Programming [3], emphasizes test suites that are maintained during all development activity, and run very frequently to increase developer confidence. Continuous testing automatically enforces frequent test running, which should yield greater benefit with less distraction.

- Continuous compilation (see Chapter 2), which automatically keeps code compiled and notifies users of compilation errors, has widespread use among industrial Java developers. Continuous testing extends this idea to keeping code well-tested, and should also receive wide industry acceptance.

After reviewing related work (Chapter 2), this thesis presents three initial steps toward verifying and quantifying the benefits of continuous testing.

First (Chapter 3), we captured the behavior of developers not using continuous testing, used the data to develop a model of behavior, and used the model to predict the effect that changes in testing behavior or technology might have had on their productivity. For the monitored projects, continuous testing could have reduced development time by 8–15%, a substantial improvement over other evaluated approaches such as automatic test prioritization and changing manual test frequencies.

Second (Chapter 4), we ran a controlled experiment comparing student developers provided with continuous testing with a control group. We found a statistically significant effect on developer success in completing a programming task, without affecting time worked. Student developers using continuous testing were three times

more likely to complete a task before the deadline than those without. Students with only continuous compilation were twice as likely to complete the task as those without, providing empirical evidence of continuous compilation's effectiveness. Most participants found continuous testing to be useful, did not consider it distracting, and believed that it helped them write better code faster. Furthermore, 90% would recommend the tool to others.

Third (Chapter 5), we built a beta version of a continuous testing plug-in for the Eclipse development environment. In this thesis, we detail its functional and technical design. Creating an implementation of the feature that seamlessly integrates with a development environment with widespread industrial use should bring the benefits of continuous testing to a wide range of developers. It will also help us to discover more about how developers put it to use, and what additional features would increase its benefits.

Finally, Chapter 6 examines future directions to be explored, and reviews contributions. The first four appendices cover materials given to participants in the user study of Chapter 4. The final appendix contains a tutorial from the on-line help of the Eclipse plug-in (Chapter 5).

This thesis contains work appearing in [33].

Chapter 2

Related work

Continuous testing can be viewed as a natural extension of continuous compilation. Modern IDE's (integrated development environments) with continuous compilation supply the developer rapid feedback by performing continuous parsing and compilation, indicating (some) syntactic and semantic errors immediately rather than delaying notification until the user explicitly compiles the code. The Magpie [35] and Montana [18] systems pioneered practical incremental compilation to enable continuous compilation of large programs by only recompiling the part of a program that has changed, and this capability is standard in IDE's such as Eclipse and IDEA. This thesis appears to contain the first empirical evaluation of the productivity improvement provided by continuous compilation, but our programs were small enough that incremental compilation was unnecessary.

Continuous testing can also be viewed as a natural extension of Extreme Programming [3], which emphasizes the importance of unit test suites that are run very frequently to ensure that code can be augmented or refactored rapidly without regression errors. Gamma and Beck [12] appear to have independently arrived at this extension.

Henderson and Weiser [15] propose *continuous execution.* By analogy with a spreadsheet such as VisiCalc, their proposed VisiProg system (which they hypothesized to require more computational power than was available to them) displays a program, a sample input, and the resulting output in three separate windows. VisiProg

treats a program as a data-flow-like network and automatically updates the output whenever the program or the input changes. Rather than continuously maintaining a complete output, which would be likely to overwhelm a developer, the test suite abstracts the output to a simple indication of whether the output for each individual test case is correct.

Programming by Example [9, 20] and Editing by Example [26, 23] can be viewed as varieties of continuous execution: the user creates a program or macro (possibly by providing input–output pairs), immediately sees its results on additional inputs, and can undo or change the program or macro in response. Our work differs in both its domain and in abstracting the entire program output to the test output, lessening the user's checking burden. (The user is required to write tests, which behave as a partial specification.)

Johnson, et al. [17] evaluate the HackyStat client/server framework, which monitors individual developer activity and provides feedback on how development time is being spent. This information is intended to allow the individual to make changes to their own development process. Our monitoring framework is similar to theirs, but provides more data and allows deeper analysis, because it enables recreating the state of the source code at any point in time. Also, we were not evaluating the monitoring framework itself nor providing feedback on collected data, but monitoring the impact of continuous testing.

While it is desirable to run complete test suites, that may be too expensive or time-consuming. Regression test selection [21, 14, 31] and prioritization [39, 32, 38] aim to reduce the cost or to produce useful answers more quickly. Test prioritization is a key enabling technology for realistic continuous testing, and Section 3.4.4 compares several simple strategies. Our domain allows using data from previous runs of the test suite to aid prioritization, which we focus on here rather than data collected from coverage analysis of the program and test suite.

Kim, Porter, and Rothermel [19] examine the impact of testing frequency on the costs and benefits of regression test selection techniques, by artificially creating development histories that add defects to a working code base. Our work differs by

using real development histories, and focusing on the impact on development time of changing test frequency and other techniques.

Boehm [5] and Baziuk [2] have shown that in projects using a traditional waterfall methodology, the number of project phases between the introduction and discovery of a defect has a dramatic effect on the time required to fix it. We believe that similar results hold on the order of seconds rather than days or months.

Several other authors use terms similar to our uses of continuous compilation, continuous execution, and continuous testing. Plezbert [30] uses the term "continuous compilation" to denote an unrelated concept in the context of just-in-time compilation. His continuous compilation occurs while the program is running to amortize or reduce compilation costs and speed execution, not while the program is being edited in order to assist development. Childers et al. [8] use "continuous compilation" in a similar context. Siegel advocates "continuous testing", by which he means frequent synchronous testing during the development process by pairs of developers [36]. Perpetual testing or residual testing [29] (also known as "continuous testing" [37]) monitors software forever in the field rather than being tested only by the developer; in the field, only aspects of the software that were never exercised by developer testing need be monitored. Software tomography [28] partitions a monitoring task (such as testing [27]) into many small subpieces that are distributed to multiple sites; for instance, testing might be performed at client sites. An enabling technology for software tomography is continuous evolution of software after deployment, which permits addition and removal of probes, instrumentation, or other code while software is running remotely.

More feedback is not always better: an interface that provides too much information (particularly low-quality information) might interrupt, distract, and overload the developer, perhaps even to the point of retarding productivity. In a survey of 29 experienced COBOL programmers, Hanson and Rosinski [13] found that they fell into two groups (of approximately equal size) regarding their tool preferences. One group preferred a larger, more integrated set of interacting tools; the other preferred to have fewer, more distinct, and less interrelated tools. Our work shows that for at

least one tool providing additional feedback, students were largely positive about the experience.

Chapter 3

Reducing wasted time

3.1 Introduction

Wasted time during software development costs money and reduces morale. One source of wasted development time is regression errors: parts of the software worked in the past, but are broken during maintenance or refactoring. Intuition suggests that a regression error that persists uncaught for a long time wastes more time to track down and fix than one that is caught quickly, for three reasons. First, more code changes must be considered to find the changes that directly pertain to the error. Second, the developer is more likely to have forgotten the context and reason for these changes, making the error harder to understand and correct. Third, the developer may have spent more time building new code on the faulty code, which must now also be changed. If the error is not caught until overnight or later, these problems are exacerbated by the long period away from the code.

To catch regression errors, a developer can run a test suite frequently during development. After making a sequence of code changes, the developer runs the test suite, and waits for it to complete successfully before continuing. This synchronous use of the test suite leads to a dual inefficiency. Either the developer is wasting potential development time waiting on the CPU to finish running tests, or the CPU is idle waiting for the developer to finish a sequence of code changes, while regression errors potentially continue to fester, leading to wasted time fixing them down the

road.

How can these two sources of wasted time be reduced? The testing framework with which the test suite is built could perhaps be enhanced to produce useful results in less time, by changing the way that errors are reported or the order in which they are run. The developer could test more frequently (catching regression errors more quickly at the expense of more time waiting for tests), or less frequently (reversing the trade-off), in hopes of striking a better balance between the two sources of wasted time.

Or the developer might use the test suite asynchronously, using the processor cycles unused during development to run tests. Without development environment support, asynchronous testing means that the developer starts the test suite, and continues to edit the code while the tests run on an old version in the background. This is unsafe, however. If the test suite exposes an error in the code, it may be an old error that no longer exists in the current version of the code, and recently introduced errors may not be caught until the suite is re-run.

3.1.1 Continuous testing

Continuous testing should combine the efficiency of asynchronous testing with the safety of synchronous testing. We conducted a pilot experiment to experimentally verify our intuitions about continuous testing before proceeding with a controlled user study. Are regression errors caught earlier easier to fix? Does continuous testing really promise a sizable reduction in wasted time, compared to simpler strategies like test reordering or changing test frequency?

To answer these questions, we needed both real-world data on developer behavior, and a model of that behavior that allowed us to make predictions about the impact of changes to the development environment. The data came from monitoring two single-developer software projects using custom-built monitoring tools to capture a record of how the code was changed over time, when tests were run, and when regression errors were introduced and later revealed by tests.

Our model, which is central to our analysis, is a finite automaton where states

represent both whether the code under development actually passes its tests, and whether the developer believes that it does. Transitions among states are triggered by events such as editing code (to introduce or fix errors), running tests, and being notified of test results.

Used observationally, following the transitions triggered by observed events in the recorded data, the model can be used to infer developer beliefs about whether tests succeed or fail, distinguishing true accidental regression errors from intentional, temporary, changes to program behavior. Our analysis indicates a correlation between ignorance time (time between the accidental introduction of an error and its eventual discovery) and fix time (time between the discovery and correction of the error). This confirms that regression errors caught earlier are easier to fix, and permits us to predict an average expected fix time, given an ignorance time. We were then able to calculate that wasted time from waiting for tests and fixing long-festering regression errors accounted for 10% and 15% of total development time for the two monitored projects.

Used predictively, the model, together with the correlation observed above, can be used to evaluate the effect a change to the testing strategy or development environment has on wasted time. In this case, the observed events are altered to reflect a hypothetical history in which the new technique is used, and the model is re-run on the new event stream.

We evaluated three techniques for reducing wasted time. The first technique is to (manually) run tests more or less frequently. We discovered that the optimal frequency, for the projects we studied, would have been two to five times higher than the developer's actual frequency. The second technique is test prioritization, which reduces test-wait time by more quickly notifying developers of errors. In our experiments, the impact of test prioritization was non-negligible, but less than that of test frequency. The third technique is continuous testing, introduced above, which dominated the other two techniques and eliminated almost all wasted time.

The remainder of this section is organized as follows. Section 3.2 presents our model of developer behavior and beliefs. Section 3.3 presents the quantities measured,

the methodology used to gather data, and the specific development projects to which these methods were applied. Section 3.4 gives experimental results.

3.2 Model of developer behavior

Regression testing notifies a developer that an error has been introduced. If the developer was not already aware of the error, then they have the opportunity to correct it immediately or to make a note of it, rather than being surprised to discover it at a later time when the code is no longer fresh in their mind. (Notification is useful not only in cases where the developer wrongly believes that there is no error, but in cases where the developer believes there may be an error, but doesn't know which test fails, or why it fails.) If the developer was already aware of the error, then the notification confirms the developer's beliefs but does not affect their behavior. The notification is more useful in the former case, where the error was introduced unknowingly, than in the latter case, where the error was introduced knowingly.

Since continuous testing operates by increasing failure notifications, assessing its usefulness for real development projects requires distinguishing when the developer is aware or unaware of an error. Querying for the developer's beliefs regarding how many and which tests may be failing is distracting and tedious, affecting developer behavior and degrading quality of the answers.

Our approach is to unobtrusively observe developer behavior, then to infer, from developer actions, whether the developer believed a regression error to be present. This approach does not affect developer behavior, nor does it suffer from developer mis-reporting. The inferred beliefs are not guaranteed to be an accurate characterization of the developer's mental state; however, they do match our intuition about, and experience with, software development. In Section 3.2.2, we detail the way that developer knowledge of errors is inferred.

This section describes two models of developer behavior. The *synchronous model* describes the behavior of a developer who only gets test feedback by running the suite and waiting for it to complete before continuing development. The *safe asynchronous*

model extends the synchronous model to describe developer behavior when test feedback on the current version of the code may be provided during development without the developer's explicit invocation (such as when the developer is using continuous testing). (We do not here consider an unsafe asynchronous model, in which a developer continues developing while tests run in the background on an old version of the code.)

Each model is a nondeterministic finite state machine. States indicate developer goals and beliefs about the absence or presence of regression errors; events (edges between states) are actions by the developer or the development environment. The synchronous model is a special case of the asynchronous model in which no notifications of test failures are ever received without explicit test invocation.

These models have resemblances to the TOTE (Test-Operate-Test-Exit) model of cognitive behavior [22]. In the TOTE model, plans consist of an Image of the desired result, a Test that indicates whether the result has been reached, and Operations intended to bring reality closer to the Image. Here, running the test suite is the Test, changing the code to make it work is the Operation, and a successful test run allows Exit. The test suite is an externally communicated Image of the developer's desired behavior for the program.[1]

A model of developer behavior can be used in two different ways: observationally and predictively. When used observationally, events (developer behavior and facts about the state of the code) drive the model into states that indicate the developer's beliefs. When used predictively, both states and actions are replayed into a model in order to infer developer behavior under different conditions. This paper uses the synchronous model both observationally and predictively, and the asynchronous model predictively.

Section 3.2.1 informally describes the synchronous model and the environment that it assumes. Section 3.2.2 presents the synchronous model for a test suite con-

[1]The TOTE model is more general than the better-known GOMS model [7], but is not designed for estimation of the time required to complete a task. Here, we predict the impact of changes by estimating the difference in effort for the problem-solving activity of recovering from programming errors, rather than by counting individual actions, as would a GOMS task analysis.

taining a single test. Section 3.2.3 extends the model to a test suite containing more than one test. Section 3.2.4 describes the safe asynchronous model.

3.2.1 Assumptions and terminology

The synchronous model applies to developers using a "guess-and-check" testing strategy. The developer changes the source code until he or she believes that the software is passing (the *guess* stage), then runs the test suite and waits for completion (the *check* stage). If the test suite fails, the developer has inadvertently introduced a regression error. The developer iteratively tries to fix the error and runs the tests, until no errors remain. Then the developer resumes changing the source code. We believe that this methodology is followed by many developers in practice. More relevantly, it was followed in our case study. Our models account for the possibility that developers are following a test-first methodology such as Extreme Programming [3], in which a developer knowingly augments a working test suite with a failing test case, and then fixes the code to make the test pass.

The model assumes that the developer maintains an automated test suite that tests (part of) the software system's behavior. The developer may modify the source code or the test suite at any point in time. It is possible that the tests may be incomplete or incorrect with respect to some external specification of the program behavior, but these kinds of errors are not modeled. Our model ignores all points at which a test is unrunnable — for instance, due to a compilation error. A runnable test is either *failing* or *passing*, depending on what would happen if the test were run at that instant using the developer's current view of the code, including modifications made in unsaved editor buffers. A test that throws an unexpected runtime error is counted as failing.

3.2.2 Synchronous model for a single test

The nondeterministic finite-state automaton (NFA) shown in Figure 3-1 models the synchronous development process with respect to a single test case. The NFA has

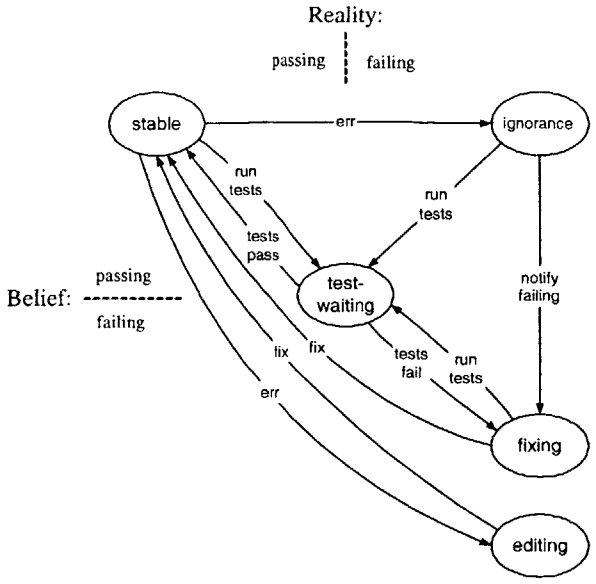

Figure 3-1: Nondeterministic finite-state automaton (NFA) for the synchronous model of developer behavior with a test suite consisting of a single test (Section 3.2.2).

five states:

stable: The test is passing, and the developer knows it. The developer is refactoring, adding new functionality, working on an unrelated part of the code, or not developing.

editing: The developer has temporarily caused the test to be failing, and knows it. The developer is in the middle of an edit that is intended to make the test work again.

ignorance: A regression error has been unknowingly introduced. The test is failing, but the developer does not know it. The developer continues to act as if the test is passing.

test-waiting: The developer has started the test suite, and is waiting for it to finish.

fixing: The developer knows (from a test failure) that a regression error has been introduced, and is working to fix it.

In the absence of continuous testing, these observable events map to transitions between the states in the model:

add test: The test is first added to the test suite. This puts the model in the *stable* or *fixing* state, depending on whether the test is passing or failing. (For simplicity, Figures 3-1 and 3-2 omit these events.)

run tests: The developer starts running the test suite.

test fail: The test suite reports that the test has failed, together with details of the failure.

test pass: The test suite reports that the test has passed.

err: The developer (intentionally or inadvertently) introduces an error, making an edit that causes the previously passing test to fail.

fix: The developer (intentionally) fixes an error, making an edit that causes the previously failing test to pass.

The synchronous model forbids some plausible behaviors. For example, it lacks a state corresponding to the situation in which the code is passing, but the developer believes it is failing. Although such a state would bring some symmetry to the model, we observed it to be very rare, and it does not capture any behavior that we wanted to consider in this research. As another example, the *fix* event always occurs with the developer's full knowledge. In reality, a regression error might be accidentally fixed, but we believe that such an event is uncommon and any such error is unlikely to be a serious one. bothers to run the test suite when unsure about whether (some) tests fail, or when the developer believes that the tests pass and wishes to double-check that belief. The developer does not intentionally make a code change that introduces a specific regression error and then run the test suite, but without even trying to correct the error. (Note that this situation of causing an existing test to fail is quite different than augmenting the test suite with a new test that initially fails: we specially handle adding new failing tests, which is a common practice in test-first methodologies such as Extreme Programming.)

The reason for these restrictions to the model is two-fold. First, they make the model more closely reflect actual practice. Second, they enable resolution of nondeterminism. In Figure 3-1, the *err* event may happen with or without the developer's knowledge, transitioning from the *stable* state into either *editing* or *ignorance*. The

nondeterminism is resolved by whether the developer fixes the error before the next time the tests are run: a *fix* event is interpreted as meaning that the developer knew that an error had been introduced. A *run tests* event is interpreted as meaning that the developer thought there was no error the whole time.

3.2.3 Model for multiple tests

The model for a test suite containing multiple tests is built by combining the model for each individual test in the suite. Its state is determined as follows:

- If any test is in *ignorance*, so is the combined model.
- Otherwise, if any test is in *fixing*, so is the combined model.
- Otherwise, if any test is in *test-waiting*, so is the combined model.
- Otherwise, the combined model is in *stable*.

The multiple-test model has no *editing* state: nondeterminism is introduced and resolved at the level of individual tests.

The exact point when the model leaves the *test-waiting* state depends on the operation of the test harness that runs the test suite. If the harness waits until the end of all testing to give details about the success or failure of individual tests, then the model stays in the *test-waiting* state until the end of the entire suite run. If, however, the harness reports information on failures as soon as they occur, the model could immediately transition from *test-waiting* to *fixing*, on the assumption that the developer could immediately begin working on the new error before waiting for the tests to complete. For a passing run of the suite, we assume that the developer waits to see whether all tests have passed before continuing work. This indicates that even without introducing continuous testing, changes to a test suite or test harness may impact the speed of development. This notion is familiar from test selection and prioritization research; we explore this idea further in Section 3.4.3.

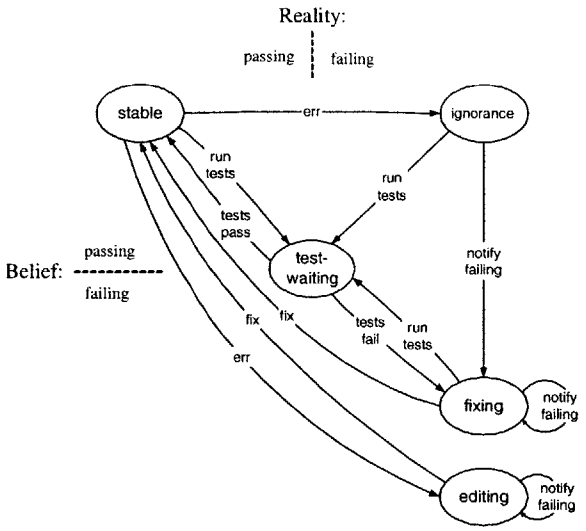

Figure 3-2: Nondeterministic finite-state automaton (NFA) for the safe asynchronous model of developer behavior with a test suite consisting of a single test (Section 3.2.4).

3.2.4 Safe asynchronous model

This section presents a model of developer behavior in an environment including continuous testing, which immediately displays feedback to the developer when a test fails. We use this model to predict the effectiveness of continuous testing.

The model of Figure 3-2 differs from that of Figure 3-1 only in adding a *notify failing* transition. This transition happens when the development environment notifies the developer that a particular test is failing, thereby ending the developer's ignorance about that error. If the notification happens in the *editing* state, it is confirmatory rather than providing new information — like a word processor's grammar checker flagging a subject-verb agreement error after the user changes the subject, when the user intended to change the verb next as part of a single logical edit.

Just as different test harnesses can cause different delays before the *tests fail* event, different continuous testing implementations can cause different delays before the *notify failing* event.

Again, the multi-test model is built from the single-test model as described in Section 3.2.3.

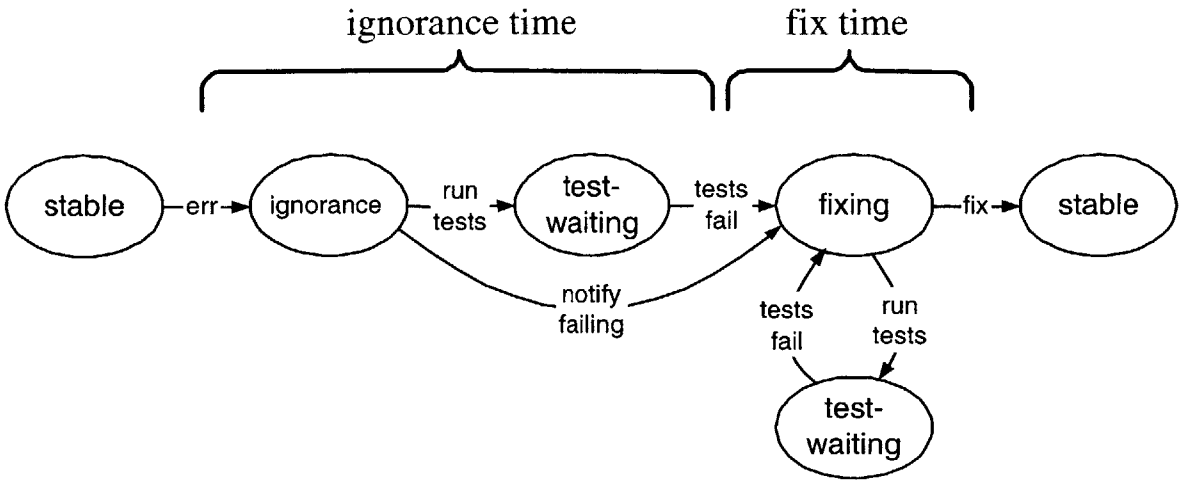

Figure 3-3: Definitions of ignorance time and of fix time. The diagram shows the transitions that occur when an error is inadvertently introduced, discovered, and then fixed. This diagram is based on the multiple-tests model of Section 3.2.3, which has no *editing* state.

3.3 Experimental methodology

Using the model of developer behavior introduced in Section 3.2, we were able to define precisely the kinds of wasted time we wished to reduce (Section 3.3.1), develop an infrastructure for measuring them (Section 3.3.2), and apply this infrastructure to two development projects (Section 3.3.3).

3.3.1 Measured quantities

The key quantity that we measure is wasted time. We posit two varieties of wasted time: test-wait time and regret time. *Test-wait time* is the entire time spent in the test-wait state — reducing this time, all other things being equal, should lead to faster software development. *Regret time* is extra time that is wasted tracking down and fixing errors that could have been prevented on the spot with instant feedback, and fixing code based on a faulty assumption. Therefore, regret time manifests itself as increased *fix time*, the amount of working time between learning of an error and correcting it (see Figure 3-3). Some fix time is unavoidable: regret time must be inferred from *ignorance time*, the amount of working time spent between introducing

an error and becoming aware of it.[2] The relationship between ignorance time and regret time may be computed for a particular development project (as we do in Section 3.4.1) by averaging over many observed instances to assign constants in a sub-linear polynomial function. We calculate wasted time using a baseline of the predicted fix time for the minimum ignorance time observed, which is the lowest ignorance time for which it seems valid to make a prediction. This is a conservative estimate, and likely underestimates the possible benefits of continuous testing. To review, for a single error:

1. ignorance time = observed time in the ignorance state
2. predicted fix time = $k_1 *$ ignorance time ^{k_2}
3. regret time = predicted fix time[ignorance time] − predicted fix time[minimum observed ignorance time]
4. wasted time = regret time + test-wait time

To normalize test-wait time and regret time across projects, we express them as a percentage of the total time worked on the project. We estimated the total time worked by assuming that every five-minute interval in which the source code was changed or tests run was five minutes of work, and no work was done during any other times. This also applies to measuring ignorance time and fix times.

3.3.2 Implementation

We collected and analyzed data using two custom software packages that we implemented: `delta-capture` and `delta-analyze`.

`delta-capture` monitors the source code and test suite, recording changes and actions to a log. `delta-capture` includes editor plug-ins that capture the state of in-memory buffers (in order to capture the developer's view of the code), and a background daemon that notes changes that occur outside the editor. The developer's

[2] As a simplification, we assume that the contribution of ignorance time to wasted time applies only to the immediately following fix time interval.

test harness is modified to notify `delta-capture` each time the tests are run. A *capture point* is a point in time at which a change has been captured. A capture point occurs on a regular basis during editing, and immediately after each save and before any run of the test suite.

`delta-analyze` processes data collected by `delta-capture` in order to compute wasted time and determine how the programmer would have fared with a different testing strategy or development environment. First, `delta-analyze` performs *replay*: it recreates the developer's view of the source code and tests at each capture point, and runs each test on the recreated state. Second, `delta-analyze` uses the replay output, along with model of Section 3.2, to determine the state of the suite and to predict developer behavior, such as fix times. `delta-analyze`'s parameters include:

- A test-frequency strategy indicating how often the developer runs the test suite (see Section 3.4.2).
- A test-prioritization strategy that determines the order of synchronous tests run in the test harness and how the results are communicated to the developer (see Section 3.4.3).
- A continuous testing strategy (possibly none) that determines the order of asynchronous tests run during development (see Section 3.4.4).

3.3.3 Target programs and environments

We used an early version of `delta-capture` to monitor the remainder of the development of the tools of Section 3.3.2. The monitored development included a command-line installation interface, robustness enhancements, and code re-structuring for `delta-capture`, and the creation of `delta-analyze` from scratch. Some of the code and test suite changes were in response to user comments from the 6 developers being monitored; this paper reports only data from the one developer who was working on the delta tools themselves, however. (Less data or lesser-quality data is available from the other half-dozen developers.) The developer used a test-first methodology. The test cases were organized by functional area, and not with any intention of catching

Attribute	Perl	Java
lines of code	5714	9114
total time worked (hours)	22	22
total calendar time (weeks)	9	3
total test runs	266	116
total capture points	6101	1634
total number of errors	33	12
mean time between tests (minutes)	5	11
mean test run time (secs)	16	3
min test run time (secs)	2	0
max test run time (secs)	61	15
mean ignorance time (secs)	218	1014
min ignorance time (secs)	16	20
median ignorance time (secs)	49	157
max ignorance time (secs)	1941	5922
mean fix time (secs)	549	1552
min fix time (secs)	12	2
median fix time (secs)	198	267
max fix time (secs)	3986	7086
max time between captures (secs)	15	60

Figure 3-4: Statistics about the Perl and Java datasets.

errors early.

Because of different development environments, we separate the data into two groups and analyze them separately. The *Perl dataset* consisted of the development in Perl 5.8 of `delta-capture` and an early version of `delta-analyze`. The Perl tools were based on a shared library, and shared a single test suite. This project used the `Test::Unit` test harness and the Emacs development environment. The *Java dataset* consisted of a rewrite of most of `delta-analyze` in Java 1.4, using the JUnit test harness and the Eclipse development environment.

Table 3-4 shows statistics for the two monitored projects. Recall that time worked is based on five-minute intervals in which code changes were made. On average, the developer ran the Perl test suite every 5 minutes and the (faster) Java test suite every 11 minutes. The more frequent Perl test suite runs result from the developer's use of the Perl test suite to check the code's syntax as well as its functionality. The Eclipse environment provides real-time feedback about compilation errors, so

the developer did not need to run the test suite to learn of syntax errors, type and interface mismatches, and similar problems.

3.4 Experiments

The data collected in the case studies of Section 3.3 allowed us to determine that a correlation exists between ignorance time and fix time (Section 3.4.1). This allowed us to compute the total wasted time for each project, which was 10% of total development time for the Perl project, and 15% for the Java project. We then used our developer behavior model predictively to evaluate three techniques for reducing this wasted time: more or less frequent testing by the developer (Section 3.4.2), test suite ordering and other test harness changes (Section 3.4.3), and continuous testing (Section 3.4.4).

3.4.1 Effect of ignorance time on fix time

Figure 3-5 plots the relationship between ignorance time and subsequent fix time. The plot uses the the multiple-test model of Section 3.2.3: if a change breaks multiple tests, the error introduced by that change is represented only once.

The figures also show the best-fit line relating ignorance time and fix time. A line on a log-log plot represents a polynomial relationship; the resulting polynomials are sub-linear (degree < 1; concave down). This relationship fits both the data and our intuition: the difference between 5 and 10 minutes of ignorance is more significant than the difference between 55 and 60 minutes of ignorance.

The relationship between ignorance time and fix time permits prediction of the impact on development time of changes to the development process. In particular, a tool that reduces ignorance time is expected to save the difference between the average fix time for the original ignorance time and the average fix time for the new ignorance time. Thus, we can fill in the constants in the formula for predicted fix time given in Section 3.3.1, as shown in Figure 3-5. In Section 6.1, we discuss possible techniques to improve the accuracy of these predictions–however, the trend is clear enough to

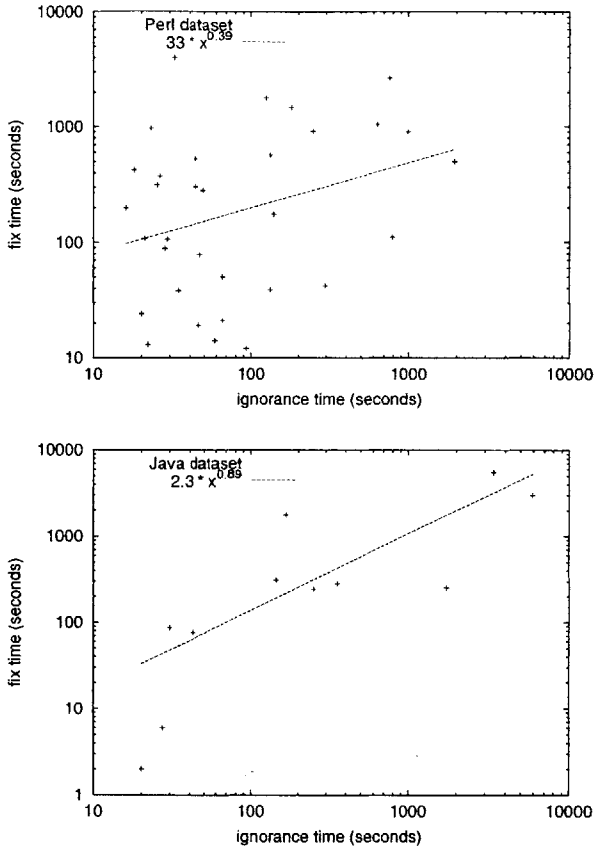

Figure 3-5: Scatter plot and best-fit line for fix time vs. ignorance time. Axes are log-scale, and the best-fit line is plotted. For perl, $R^2 = 0.14$. For java, $R^2 = 0.38$

allow us to estimate time saved and compare between different testing techniques

We treat the Perl dataset and the Java dataset separately because the different development environments, programming languages, and problem domains make errors easier or harder to introduce and to fix, yielding different relationships between ignorance time and fix time. Our subsequent experiments use the appropriate relationship to predict the effect of changes to the development process. As an example, for the Perl dataset, the relationship predicts that errors with an ignorance time of one minute take 3 minutes to correct, whereas errors with an ignorance time of one hour take 13 minutes to correct. For the Java dataset, the corresponding fix times are 2 minutes and 49 minutes.

Using the observed test-wait times and the formula for regret time given in Section 3.3.1, the total wasted time for the two projects can be broken down as seen in Figure 3-6

	Perl	Java
Test-wait time	0.055	0.004
Regret time	0.044	0.145
Total wasted time	0.099	0.149

Figure 3-6: Total wasted time for the Perl and Java datasets, as a fraction of total development time

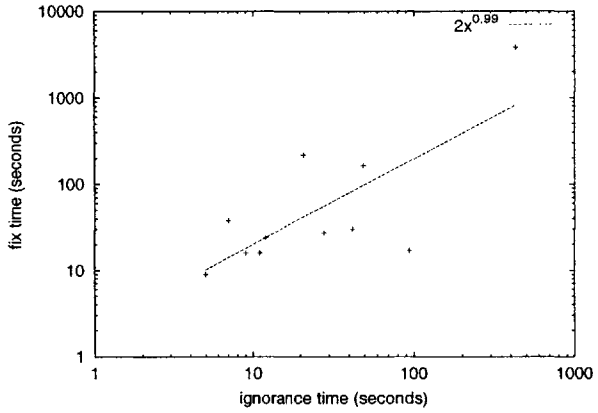

Figure 3-7: Scatter plot and best-fit line for fix time vs. ignorance time for a single undergraduate student. Axes are log-scale, and the best-fit line is plotted.

Undergraduate data

For additional indication of the relationship between ignorance time and fix time, we monitored the progress of undergraduate students working on problem sets with provided test suites (see Section 4). We found that the same trend holds, even more strongly for individual students, but not so much for students taken as a whole. This makes sense, because patterns of development and debugging skill are variable from developer to developer, but likely play a large role in the time required to fix an error.

Figure 3-7 plots ignorance time and fix time for a single student (the one who was observed to have introduced and fixed the most regression errors). For this single student, regression analysis indicated an R-squared value of 0.54. Taking all students together as a group, individual variations in programming style and other factors influencing fix time introduce more noise, but the trend is still clear.

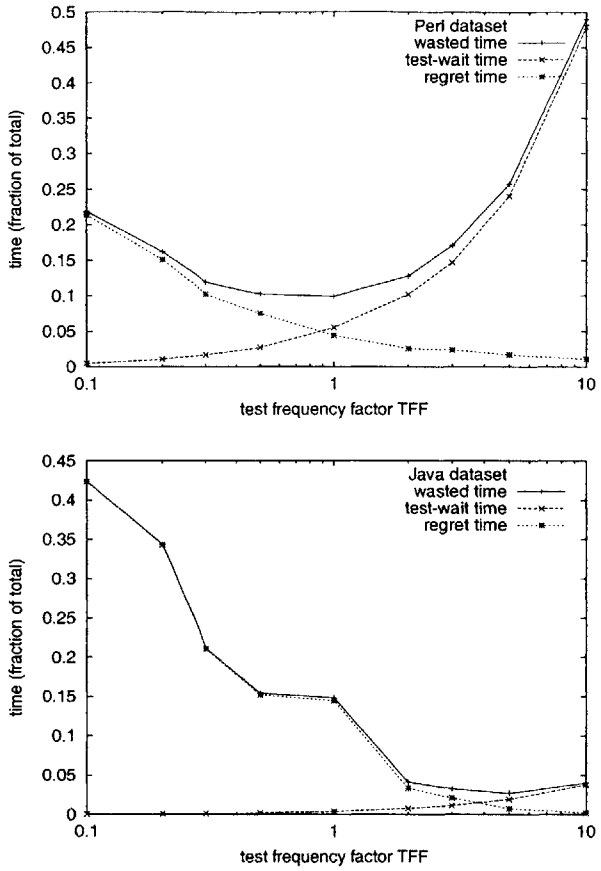

Figure 3-8: Wasted time as a function of testing frequency. Wasted time is the sum of test-wait time and regret time. The graphs show how increasing or decreasing the testing frequency by a multiplicative test frequency factor *TFF* affects wasted time.

3.4.2 Frequency of testing

Section 3.4.1 showed that ignorance time is correlated with fix time: reducing ignorance time should reduce fix time. Developers do not need a tool to obtain this benefit: they can simply run their tests more often to catch more errors more quickly, at the expense of more test-wait time. Running tests less often would involve the opposite trade-off. We investigated whether developers would be better off overall (and how much better off they would be) by increasing or decreasing the frequency at which they run tests. We simulated these behavior changes by adding and removing *test points*, which are capture points at which a run of the test suite was recorded.

Figure 3-8 shows the effect on wasted time as a developer tests more or less frequently. The test frequency factor *TFF* is the ratio of the number of simulated

test suite runs to the number of actual test case runs. $TFF = 1$ corresponds to the observed developer behavior. For $TFF < 1$, we removed random test points; according to our model, test cases in the ignorance state at the time of a removed test point remain in that state until the next intact test point. For $TFF > 1$, we added random capture points to the actual test points; according to our model, when one of these new test points is encountered, any errors then in ignorance are transitioned into the fixing state. As always, regret time is computed from ignorance time using the formula given in Section 3.3.1, with constants derived from the data in Section 3.4.1.

As expected, increasing test frequency increases test-wait time, and decreases regret time. In the Perl dataset, the developer chose a good frequency for testing. In the Java dataset, the developer could have enjoyed 82% less wasted time by running the tests 5 times as frequently — every 2 minutes instead of every 11 minutes. Here, a faster test suite and different development style mean that the increased test-wait time that results from more frequent testing is insignificant in comparison to the regret time, at least over the frequencies plotted.

3.4.3 Test prioritization and reporting

Changes to the test harness may save overall development time by reducing test-wait time. There are two related factors to consider. First is how long the test harness takes to discover an error, and second is how long the harness takes to report the error to the developer. Time for the harness to discover the error depends on the order in which tests are run. Figure 3-9 lists the test prioritization strategies that we experimentally evaluated. Time for the harness to report the error depends on its reporting mechanism. We considered three scenarios.

Full suite: The text harness runs to completion before reporting any results. This is the default behavior for the Perl `Test::Unit` harness. With this reporting mechanism, test prioritization is irrelevant.

Real-time: The test harness reports failures as they occur. This is the default behavior for the Java JUnit harness as integrated into Eclipse. Whenever a

Suite order: Tests are run in the order they appear in the test suite, which is typically ordered for human comprehensibility, such as collecting related tests together, or is ordered arbitrarily.

Round-robin: Like the suite order, but after every detected change, restart testing at the test after the most recently completed test. This is relevant only to continuous testing, not synchronous testing.

Random: Tests are run in random order, but without repetition.

Recent errors: Tests that have failed most recently are ordered first.

Frequent errors: Tests that have failed most often (have failed during the greatest number of previous runs) are ordered first.

Quickest test: Tests are ordered in increasing runtime; tests that complete the fastest are ordered first.

Failing test: The quickest test that fails, if any, is ordered first. This (unrealistic, omniscient) ordering represents the best that a perfect test prioritization strategy could possibly achieve. (Even better results might be achieved by introducing new, faster tests and prioritizing them well; see Section 6.1.)

Figure 3-9: Test case prioritization strategies used in the experiments of Sections 3.4.3 and 3.4.4.

prioritization strategy listed in Figure 3-9 is mentioned, it is implicit that the harness reports failures in real time.

Instantaneous: All tests are run, and all failures are reported, instantaneously. With this reporting mechanism, test prioritization is irrelevant. This scenario represents the unachievable ideal.

Figure 3-10 shows, for each test harness and prioritization listed above, the effect of using it in terms of test-wait time, ignorance time, regret time, total wasted time (the sum of test-wait time and regret time), and improvement. Improvement is reported as reduction of wasted time and as a percentage reduction from the baseline measurement of wasted time.

For the Perl dataset, the recent errors test prioritization strategy dominated other achievable test prioritizations. In other words, the developer tended to break the same tests over and over. To our surprise, the frequent errors test prioritization strategy performed poorly. We speculate that this is because often a particularly difficult error persists for many test executions, thus dominating this metric due to test runs while in the *fixing* state, but may not be representative of regression errors discovered in

Perl dataset:

Test Order	Test-wait	Regret	Wasted	Improvement	
Full Suite	0.055	0.044	0.100	0.000	0.0%
Random	0.038	0.042	0.080	0.020	19.6%
Frequent Errors	0.037	0.042	0.078	0.021	21.2%
Round-robin	0.029	0.041	0.071	0.029	28.9%
Suite Order	0.029	0.041	0.071	0.029	28.9%
Quickest Test	0.028	0.040	0.068	0.032	32.1%
Recent Errors	0.018	0.041	0.059	0.041	41.0%
Failing Test	0.018	0.041	0.059	0.041	41.1%
Instantaneous	0.000	0.039	0.039	0.060	60.5%

Java dataset:

Test Order	Test-wait	Regret	Wasted	Improvement	
Full Suite	0.005	0.145	0.150	0.000	0.0%
Frequent Errors	0.004	0.145	0.149	0.002	1.0%
Round-robin	0.004	0.145	0.149	0.001	1.1%
Suite Order	0.004	0.145	0.149	0.001	1.1%
Recent Errors	0.004	0.145	0.148	0.002	1.4%
Quickest Test	0.003	0.145	0.145	0.003	1.9%
Random	0.004	0.142	0.147	0.004	2.5%
Failing Test	0.003	0.142	0.145	0.005	3.6%
Instantaneous	0.000	0.142	0.142	0.009	5.7%

Figure 3-10: Effect of test harness and prioritization on the synchronous testing methodology, for the Perl and Java dataset. Each column is a fraction of total time (except percent improvement, which is based on the wasted time). Wasted time is the sum of test-wait time and regret time.

the *ignorance* state.

For the Java dataset, none of the test prioritization strategies helped very much — they improved overall time by less than 1% and reduced wasted time by less than 4%. The reason is that the tests already ran very fast; little benefit could be had by reordering them.

3.4.4 Continuous testing

Using the same methodology as in Section 3.4.3, we evaluated the effect of various continuous testing techniques. Whenever `delta-capture` detects a change to the program or its tests (in other words, at each capture point that is not a test point), we simulate running tests until the next capture point. The full test suite may not

Perl dataset:

Testing Strategy	Test-wait	Regret	Wasted	Improvement	
None	0.055	0.044	0.100	0.000	0.0%
Random	0.038	0.002	0.040	0.060	60.2%
Frequent Errors	0.037	0.003	0.039	0.060	60.5%
Suite Order	0.029	0.003	0.032	0.070	67.8%
Round-robin	0.029	0.000	0.030	0.070	70.0%
Quickest Test	0.028	0.001	0.030	0.070	70.3%
Recent Errors	0.018	0.000	0.018	0.082	82.0%
Failing Test	0.017	0.000	0.018	0.082	82.4%
Instantaneous	0.000	0.000	0.000	0.100	100.0%

Java dataset:

Testing Strategy	Test-wait	Regret	Wasted	Improvement	
None	0.005	0.145	0.150	0.000	0.0%
Random	0.004	0.000	0.004	0.146	97.2%
Round-robin	0.004	0.000	0.004	0.146	97.3%
Suite Order	0.004	0.000	0.004	0.146	97.3%
Frequent Errors	0.004	0.000	0.004	0.146	97.3%
Recent Errors	0.004	0.000	0.004	0.147	97.7%
Quickest Test	0.003	0.000	0.003	0.147	98.0%
Failing Test	0.003	0.000	0.003	0.148	98.1%
Instantaneous	0.000	0.000	0.000	0.150	100.0%

Figure 3-11: Effect of continuous testing on wasted time. Each column is a fraction of total time (except percent improvement, which is based on the wasted time). Synchronous tests are executed at the same frequency used by the developer, using the same test harness strategy as the continuous testing strategy.

be able to complete in this period, and we ignore any partially completed tests. This simulates a continuous testing environment that, every time the source is changed, restarts the test suite on the new version of the system, if the source is compilable. The same prioritization strategies listed in Section 3.4.3 are used to indicate results in Figure 3-11. "None" indicates that continuous testing is not used; no *notify-failing* events occur.

Continuous testing using the recent-errors test prioritization reduces wasted time in the Perl project by 80%, and overall development time by more than 8%, as shown in the last two columns of Figure 3-11. The recent-errors test prioritization is nearly as good as the omniscient "failing test" strategy that runs only the fastest test that fails. Even using a random ordering saves a respectable 6% of development time. The

frequent-errors prioritization performs nearly as badly as random ordering.

The improvement for the Java dataset is even more dramatic: Continuous testing improves development time by 14–15%, reducing wasted time by over 97%, regardless of the prioritization strategy, taking us near our conservative predicted limit for development speed-up from improved testing techniques. The prioritization strategies are essentially indistinguishable, because the test suite is quite small. We speculate that the greater gain for the Java dataset is due to the fact that the developer ran tests relatively infrequently. Section 3.4.2 showed that the developer could have reduced wasted time by 81% simply by running the tests more frequently.

Together, these results indicate that continuous testing may have a substantial positive effect on developers with inefficient testing habits, and a noticeable effect even on developers with efficient testing habits.

Chapter 4

Ensuring success

The previous section suggested, based on two development projects by a single developer, that development time could be reduced by 10–15%, which is substantially more than could be achieved by changing test frequency or reordering tests.

This section reports a study that evaluated whether the extra feedback provided by continuous testing improved the productivity of student developers, without producing detrimental side effects like distraction and annoyance. In a controlled human experiment, 22 students in a software engineering course completed two unrelated programming assignments as part of their normal coursework. The assignments supplied partial program implementations and tests for the missing functionality, simulating a test-first development methodology. All participants were given a standard Emacs development environment, and were randomly assigned to groups that were provided with continuous testing, just continuous compilation, or no additional tools. Data was gathered from remote monitoring of development progress, questionnaires distributed after the study, and course records. In all, 444 person-hours of active programming were observed.

The experimental results indicate that students using continuous testing were statistically significantly more likely to complete the assignment by the deadline, compared to students with the standard development environment. Continuous compilation also statistically significantly increased success rate, though by a smaller margin; we believe this is the first empirical validation of continuous compilation.

Our analysis did not show a higher success rate for students who tested frequently manually. Furthermore, the tools' feedback did not prove distracting or detrimental: tool users suffered no significant penalty to time worked, and a large majority of study participants had positive impressions of the tool and were interested in using it after the study concluded.

This section is organized as follows. We first detail the tools provided to students (Section 4.1) and the design of the experiment (Section 4.2). We then present quantitative (Section 4.3) and qualitative (Section 4.4) results and threats to validity (Section 4.5).

4.1 Tools

We have implemented a continuous testing infrastructure for the Java JUnit testing framework and for the Eclipse and Emacs development environments.

Our JUnit extension (used by both plug-ins) persistently records whether a test has ever succeeded in the past, the history of successes and failures of each test, and the most recent running time of each test. It then references a pluggable prioritization strategy to change both the order in which tests are run and the order in which results are printed. For instance, regression errors, which are typically more serious, can be prioritized over unimplemented tests.

We have built continuous testing plug-ins for both Eclipse and Emacs. We focus here on the Emacs plug-in, which was used in our experiment. We describe how the user is notified of problems in his or her code and how the plug-in decides when to run tests. We conclude this section with a comparison to pre-existing features in Eclipse and Emacs.

Because Emacs does not have a standard notification mechanism, we indicated compilation and test errors in the mode line. The mode line is the last line of each Emacs text window; it typically indicates the name of the underlying buffer, whether the buffer has unsaved modifications, and what Emacs modes are in use. The Emacs plug-in (a "minor mode" in Emacs parlance) uses some of the empty space in the mode

line. When there are no errors to report, that space remains blank, but when there are errors, then the mode line contains text such as "Compile-errors" or "Regressions:3". This text indicates the following states: the code cannot be compiled; regression errors have been introduced (tests that used to give correct answers no longer do); some tests are unimplemented (the tests have never completed correctly). Because space in the mode line is at a premium, no further details (beyond the number of failing tests) are provided, but the user can click on the mode line notification in order to see details about each error. Clicking shows the errors in a separate window and places the cursor on the line corresponding to the failed assertion or thrown exception. Additional clicks navigate to lines corresponding to additional errors and/or to different stack frames within a backtrace.

The Emacs plug-in performs testing whenever there is a sufficiently long pause; it does not require the user to save the code, nor does it save the code for the user. The Emacs plug-in indicates errors that would occur if the developer were to save all modified Emacs buffers, compile, and test the on-disk version of the code. In other words, the Emacs plug-in indicates problems with the developer's current view of the code.

The Emacs plug-in implements testing of modified buffers by transparently saving them to a separate shadow directory that contains a copy of the software under development, then performing compilation and testing in the shadow directory. Users never view the shadow directory, only the test results. This approach has the advantage of providing earlier notification of problems. Otherwise, the developer would have no possibility of learning of problems until the next save, which might not occur for a long period. Notification of inconsistent intermediate states can be positive if it reinforces that the developer has made an intended change, or negative if it distracts the developer; see Section 4.4.

The continuous testing tool represents an incremental advance over existing technology in Emacs and Eclipse. By default, Emacs indirectly indicates syntactic problems in code via its automatic indentation, fontification (coloring of different syntactic entities in different colors), indication of matching parentheses, and similar mecha-

nisms. Eclipse provides more feedback during editing (though less than a full compiler can), automatically compiles when the user saves a buffer, indicates compilation problems both in the text editor window and in the task list, and provides an integrated interface for running a JUnit test suite. Our Emacs plug-in provides complete compilation feedback in real time and JUnit integration, and both of our plug-ins provide asynchronous notification of test errors.

4.2 Experimental design

This section describes the experimental questions, subjects, tasks, and treatments in our evaluation of continuous testing.

4.2.1 Experimental questions

Continuous testing exploits a regression test suite to provide more feedback to developers than they would get by testing manually, while also reducing the overhead of manual testing. Continuous testing has intuitive appeal to many developers, but others are skeptical about its benefits. An experimental evaluation is required to begin to settle such disagreements.

We designed an experiment to address these issues, in hopes of gaining insight into three main questions.

1. Does continuous testing improve developer productivity? Increased developer productivity could be reflected either by accomplishing a task more quickly, or by accomplishing more in a given timeframe. We could control neither time worked nor whether students finished their assignments, but we measured both quantities via monitoring logs and class records.

2. Does the asynchronous feedback provided by continuous testing distract and annoy users? Intuition is strongly divided: some people to whom we have explained the idea of continuous testing have confidently asserted that continuous testing would distract developers so much that it would actually make them

slower. To address this question, we used both qualitative results from a participant questionnaire (reproduced in Appendix D.1 and quantitative productivity data as noted above.

3. If the continuous testing group was more productive, why? Continuous testing subsumes continuous compilation, and it enforces frequent testing; perhaps one of those incidental features of the continuous testing environment, or some other factor, was the true cause of any improvement.

   (a) Continuous compilation. We compared the performance of three treatment groups: one with continuous testing, one with only continuous compilation, and one with no additional tool.

   (b) Frequent testing. Although all students were encouraged to test regularly, not all of them did so. We compared the performance of students who tested more frequently with those who tested less frequently and with those who had a continuous testing tool. Any effect of easy or fast testing was unimportant, since all students could run the tests in five seconds with a single keypress.

   (c) Demographics. We chose a control group randomly from the volunteers, and we measured and statistically tested various demographic factors such as programming experience.

4.2.2 Participants

Our experimental subjects were students, primarily college sophomores, in MIT's 6.170 Laboratory in Software Engineering course (`http://www.mit.edu/~6.170`). This is the second programming course at MIT, and the first one that uses Java (the first programming course uses Scheme). Of the 100 students taking the class during the Fall 2003 semester, 34 volunteered to participate in the experiment. In order to avoid biasing our sample, participants were not rewarded in any way. In order to maintain consistency between the experimental and control groups, we excluded all volunteers

	Mean	Dev.	Min.	Max.
Years programming	2.8	2.9	0.5	14.0
Years Java programming	0.4	0.5	0.0	2.0
Years using Emacs	1.3	1.2	0.0	5.0
Years using a Java IDE	0.2	0.3	0.0	1.0

	Frequencies
Usual environment	Unix 29%; Win 38%; both 33%
Regression testing	familiar 33%; not familiar 67%
Used Emacs to compile	at least once 62%; never 38%
Used Emacs for Java	at least once 17%; never 83%

Figure 4-1: Study participant demographics (N=22). "Dev" is standard deviation.

Don't use Emacs	45%
Don't use Athena	29%
Didn't want the hassle	60%
Feared work would be hindered	44%
Privacy concerns	7%

Figure 4-2: Reasons for non-participation in the study (N=31). Students could give as many reasons as they liked.

who did not use the provided development environment for all of their development. The excluded students used a home PC or a different development environment for some of their work. This left us with 22 participants for the first data set, and 17 for the second data set.

On average, the participants had 3 years of programming experience, and one third of them were already familiar with the notions of test cases and regression errors. Figure 4-1 gives demographic details regarding the study participants.

Differences between participants and non-participants do not affect our internal validity, because we chose a control group (that was supplied with no experimental tool) from among the participants who had volunteered for the study. Our analysis indicates that it would have been wrong to use non-participants as the control group, because there were statistically significant differences between participants and non-participants with respect to programming experience and programming environment preference. This is a threat to the external validity of the study (see Section 4.5).

Only two factors that we measured predicted participation to a statistically sig-

nificant degree. First, students who had more Java experience were less likely to participate: participants had an average of .4 years of Java experience, whereas non-participants had an average of almost .8 years of Java experience. Many of the more experienced students said this was because they already had work habits and tool preferences regarding Java coding. Overall programming experience was not a predictor of participation. Second, students who had experience compiling programs using Emacs were more likely to participate; this variety of Emacs experience did not predict any of the factors that we measured, however.

Figure 4-2 summarizes the reasons given by students who chose not to participate; 31 of the non-participants answered a questionnaire regarding their decision. Very few students indicated that privacy concerns were a factor in their decision not to participate, which was encouraging considering the degree of monitoring performed on the students (see Section 4.2.5). The 6.170 course staff only supports use of Athena, MIT's campus-wide computing environment, and the Emacs editor. In the experiment, we provided an Emacs plug-in that worked on Athena, so students who used a different development environment could not participate. The four non-Emacs development environments cited by students were (in order of popularity): Eclipse, text editors (grouping together vi, pico, and EditPlus2), Sun ONE Studio, and JBuilder. Students who did not complete their assignments on Athena typically used their home computers. Neither student experience with, nor use of, Emacs or of Athena was a statistically significant predictor of any measure of success (see Section 4.3.2).

4.2.3 Tasks

During the experiment, the participants completed the first two assignments (problem sets) for the course. Participants were treated no differently than other students. All students were encouraged by staff and in the assignment hand-out to run tests often throughout development. The problem sets were not changed in any way to accommodate the experiment, nor did we ask participants to change their behavior when solving the problem sets. All students were given a 20-minute tutorial on the experimental tools and had access to webpages explaining their use. A few students

	PS1	PS2
participants	22	17
skeleton lines of code	732	669
written lines of code	150	135
written classes	4	2
written methods	18	31
time worked (hours)	9.4	13.2

Figure 4-3: Properties of student solutions to problem sets. All data, except number of participants, are means. Students received skeleton files with Javadoc and method signatures for all classes to be implemented. Students then added about 150 lines of new code to complete the programs. Files that students were provided but did not need to modify are omitted from the table.

in the treatment groups chose to ignore the tools and thus gained no benefit from them.

Each problem set provided students with a partial implementation of a simple program. Students were also provided with a complete test suite (see Section 4.2.3). The partial implementation included full implementations of several classes and skeleton implementations of the classes remaining to be implemented. The skeletons included all Javadoc comments and method signatures, with the body of each method containing only a RuntimeException. No documentation tasks were required of students for these assignments. The code compiled and the tests ran from the time the students received the problem sets. Initially, most of the tests (all those that exercised any code that students were intended to write) failed with a RuntimeException; however, some tests initially passed, and only failed if the student introduced a regression into the provided code.

The first problem set (see Appendix B for the student handout) required implementing four Java classes to complete a poker game. The second problem set (see Appendix C for the student handout and some of the provided code) required implementing two Java classes to complete a graphing polynomial calculator. Both problem sets also involved written questions, but we ignore those questions for the purposes of our experiment. Figure 4-3 gives statistics regarding the participant solutions to the problem sets.

	PS1	PS2
tests	49	82
initial failing tests	45	46
lines of code	3299	1444
running time (secs)	3	2
compilation time (secs)	1.4	1.4

Figure 4-4: Properties of provided test suites. "Initial failing tests" indicates how many of the tests are not passed by the staff-provided skeleton code. Times were measured on a typical X-Windows-enabled dialup Athena server under a typical load 36 hours before problem set deadline.

Test suites Students were provided with JUnit test suites prepared by the course staff (see Figure 4-4). Passing these test suites correctly accounted for 75% of the grade for the programming problems in the problem set. Each test case in the suites consists of one or more method calls into the code base, with results checked against expected values. The provided continuous testing tools were hard-coded to use these suites.

The suites are optimized for grading, not performance, coverage, or usability. (That is, the test cases were developed and organized according to the pedagogical goals of the class.) However, experience from teaching assistants and students suggests that the tests are quite effective at covering the specification students were required to meet. Compiling and testing required less than five seconds even on a loaded dialup server, since the suites were relatively small (see Figure 4-4). Thus, there was no point in using test prioritization or selection when running these particular test suites.

It is rare in professional development for a developer to develop or receive a complete test suite for the desired functionality before they begin writing code on a new development project. However, since the students were directed to concentrate on one failing test at a time, the effect of the development scenario was similar to the increasingly common practice of test-driven development [4]. The task also had similarities to maintenance, where a programmer must ensure that all tests in a regression test suite continue to succeed. Finally, when developers are striving for compatibility or interoperability with an existing system, a de facto test suite is available, since the

	volunteers	non-volunteers
waited until end to test	31%	51%
tested regularly throughout	69%	49%

test frequency (minutes)		
mean	20	18
min	7	3
max	40	60

Figure 4-5: Student use of test suites, self-reported. "Volunteers" omits those who used continuous testing. Only students who tested regularly throughout development reported test frequencies.

two systems' behavior can be compared.

Several deficiencies of the provided test suites and code impacted their usefulness as teaching tools and students' development effectiveness. The PS1 test suite made extensive use of test fixtures (essentially, global variables that are initialized in a special manner), which had not been covered in lecture, and were confusing to follow even for some experienced students. In PS2, the provided implementation of polynomial division depended on the students' implementation of polynomial addition to maintain several representation invariants. Failure to do so resulted in a failure of the division test, but not the addition test. Despite these problems, students reported enjoying the use of the test suites, and found examining them helpful in developing their solutions. Figure 4-5 gives more detail about student use of the provided test suites, ignoring for now participants who used continuous testing.

4.2.4 Experimental treatments

The experiment used three experimental treatments: a control group, a continuous compilation group, and a continuous testing group. The control group was provided with an Emacs environment in which Java programs could be compiled with a single keystroke and in which the (staff-provided) tests could be run with a single keystroke. The continuous compilation group was additionally provided with asynchronous notification of compilation errors in their code. The continuous testing group was further provided with asynchronous notification of test errors. The tools are described in

Section 4.1.

For each problem set, participants were randomly assigned to one of the experimental treatments: 25% to the control group, 25% to the continuous compilation group, and 50% to the continuous testing group. Thus, most participants were assigned to different treatments for the two problem sets; this avoids conflating subjects with treatments and also permits users to compare multiple treatments.

4.2.5 Monitoring

Participants agreed to have an additional Emacs plug-in installed on their system that monitored their behavior and securely transmitted logs to a central remote server. The log events included downloading the problem set, remotely turning in the problem set, changes made to buffers in Emacs containing problem set source (even if changes were not saved), changes to the source in the file system outside Emacs, and clicking the mode line to see errors. A total of 444 person-hours, or almost 3 person-months, of active time worked were monitored.

This allowed us to test the effect of the predictors on 36 additional variables indicating the number of times users entered the following 6 states, the period between being in the states, and the absolute and relative average and total time in the states. The states are:

- ignorance: between unknowingly introducing an error and becoming aware of it via a test failure
- fixing: between becoming aware of an error and correcting it
- editing: between knowingly introducing an error and correcting it.
- regression: between introducing, knowingly or unknowingly, and eliminating a regression error
- compile error: between introducing and eliminating a compilation error
- testing: between starting and stopping tests; elapsed times are always very short in our experiments, but the number and period are still interesting quantities

Not many statistically significant results were found on these variables; however,

collection allowed us to confirm the trends seen in Section 3.4.1. (See Section 3.4.1 for details.)

4.3 Quantitative results

This section reports on quantitative results of the experiment; Section 4.4 gives qualitative results.

4.3.1 Statistical tests

This paper reports all, and only, effects that are statistically significant at the $p = .05$ level. All of our statistical tests properly account for mismatched group and sample sizes.

When comparing nominal (also known as classification or categorical) variables, we used the Chi-Square test, except that we used Fisher's exact test (a more conservative test) when 20% or more of the the cells of the classification table had expected counts less than 5, because Chi-Square is not valid in such circumstances. When using nominal variables to predict numeric variables, we used factorial analysis of variance (ANOVA). Where appropriate, we determined how many of the effects differed using a Bonferroni correction.

When using numeric variables as predictors, we first dummy-coded or effect coded the numeric variables to make them nominal, then used the appropriate test listed above. We did so because we were less interested in whether there was a correlation (which we could have obtained from standard, multiple, or logistic regression) than whether the effect of the predictor on the criterion variable was statistically significant in our experiment.

4.3.2 Variables compared

We used 20 variables as predictors: experimental treatment, problem set, and all quantities of Figures 4-1 and 4-7.

Treatment	N	Correct
No tool	11	27%
Continuous compilation	10	50%
Continuous testing	18	78%

Figure 4-6: Treatment predicts correctness. "N" is the number of participants in each group. "Correct" means that the participant's completed program passed the provided test suite.

The key criterion (effect) variables for success are:

- time worked. Because there is more work in development than typing code, we divided wall clock time into 5-minute intervals, and counted 5 minutes for each interval in which the student made any edits to the `.java` files comprising his or her solution.
- errors. Number of tests that the student submission failed.
- correct. True if the student solution passed all tests.
- grade, as assigned by TAs. We count only points assigned for code; 75% of these points were assigned automatically based on the number of passed test cases.

4.3.3 Statistical results

Overall, we found few statistically significant effects. We expected this result, because most effects are not statistically significant or are overwhelmed by other effects—either ones that we measured or other factors such as the well-known large variation among individual programmers. (Another reason might be the relatively small sample size, but the fact that some effects were significant suggests that the sample was large enough to expose the most important effects.) This section lists all the statistically significant effects.

1. Treatment predicts correctness (see Figure 4-6). This is the central finding of our experiment, and is supported at the $p < .03$ level. Students who were provided with a continuous testing tool were 3 times as likely to complete the assignment correctly as those who were provided with no tool. Furthermore, provision of

continuous compilation doubled the success rate. The latter finding highlights the benefit that developers may already get from the continuous compilation capabilities included in modern IDE's such as Eclipse [11] and IDEA [16].

2. Continuous testing vs. regular manual testing predicts correctness. Of participants who were not given continuous testing, but reported testing regularly, only 33% successfully completed the assignment, significantly less than the 78% success rate for continuous testing. There was no statistically significant difference in test frequency between complete and incomplete assignments within this group. However, the mean frequency for manual testing (see Figure 4-5) among those who tested regularly was once every 20 minutes, which is longer than the mean time to pass an additional test during development (15 minutes), possibly indicating that students were often writing code to pass several tests at a time before running the tests to confirm. Even if we were to hypothesize that students could have done better if all had tested more regularly and more often, it is clear that continuous testing accomplished what a fair amount of "managerial encouragement" could not.
3. Problem set predicts time worked (PS1 took 9.4 hours of programming time on average, compared to 13.2 hours for PS2). Therefore, we re-ran all analyses considering the problem sets separately. We also re-ran all analyses considering only successful users (those who submitted correct programs).
4. For PS1 only, years of Java experience predicted correctness and grade. For the first problem set, participants with previous Java experience had an advantage: 80% success rate and average grade 56/75, versus 35% success rate and average grade 66/75 for those who had never used Java before. By one week later, the others had caught up (or at least were no longer statistically significantly behind).
5. For PS2 only, treatment predicts the the average duration of compilation errors (periods when the code is uncompilable) and the average time between such periods; it does not predict the number of such periods, however. This may indicate some effect from continuous compilation, but this is unclear, because

the effect was not seen on PS1; it may have been a coincidence.

6. For PS1 participants with correct programs, years of Java IDE experience predicts time worked: those who had previously used a Java IDE spent 7 hours, compared to 13 hours for the two who reported never previously using a Java IDE. No similar effect was observed for any other group, including PS1 participants with incorrect programs or any group of PS2 participants.

It is worth emphasizing that we found no other statistically significant effects. In particular, of the predictors of Section 4.3.2 (including user perceptions of the experimental tools), none predicted number of errors, time worked, or grade, except for the effects from experience seen in PS1. The only effects on student performance throughout the study period were the effects that continuous testing and continuous compilation had in helping significantly more students complete the assignment.

4.4 Qualitative results

We gathered qualitative feedback about the tools from three main sources. All students were asked to complete an online questionnaire containing multiple-choice and free-form questions.

We interviewed staff members about their experiences using the tools, helping students with them, and teaching Java while the tools were running. Finally, some students provided direct feedback via e-mail.

Section 4.4.1 discusses the results of the multiple choice questions. The remainder of this section summarizes feedback about changes in work habits, positive and negative impressions, and suggestions for improvement.

4.4.1 Multiple choice results

Figure 4-7 summarizes the multiple-choice questions about experiences with the tools. Participants appear to have felt that continuous compilation provided somewhat more

	Continuous compilation (N=20)	Continuous testing (N=13)
The reported errors often surprised me	1.0	0.7
I discovered problems more quickly	2.0	0.9
I completed the assignment faster	1.5	0.6
I wrote better code	0.9	0.7
I was distracted by the tool	−0.5	−0.6
I enjoyed using the tool	1.5	0.6
The tool changed the way I worked	1.7	
I would use the tool in 6.170	yes 94%; no 6%	
... in my own programming	yes 80%; no 10%	
I would recommend the tool to others	yes 90%; no 10%	

Figure 4-7: Questionnaire answers regarding user perceptions of the continuous testing tool. The first 6 questions were answered on a 7-point scale ranging from "strongly agree" (here reported as 3) through "neutral" (reported as 0) to "strongly disagree" (reported as -3). The behavior change question is on a scale of 0 ("no") to 3 ("definitely").

incremental benefit than continuous testing (though the statistical evidence of Section 4.3.3 indicates the opposite). Impressions about both tools were positive overall.

The negative response on "I was distracted by the tool" is a positive indicator for the tools. In fact, 70% of continuous testing and continuous compilation participants reported leaving the continuous testing window open as they edited and tests were run. This confirms that these participants did not find it distracting, because they could easily have reduced distraction and reclaimed screen space by closing it (and re-opening it on demand when errors were indicated).

Especially important to us was that 94% of students wanted to continue using the tool in the class after the study, and 80% wanted to apply it to programming tasks outside the class. 90% would recommend the tool to others. This showed that developers enjoyed continuous testing, and most did not have negative impressions of distraction or annoyance.

4.4.2 Changes in work habits

Participants reported that their working habits changed when using the tool. Several participants reported similar habits to one who "got a small part of my code working

before moving on to the next section, rather than trying to debug everything at the end." Another said, "It was easier to see my errors when they were only with one method at a time." The course staff had recommended that all students use the single-keystroke testing macro, which should have provided the same benefits. However, some participants felt they only got these benefits when even this small step was automated.

This blessing could also be a curse, however, exacerbating faults in the test suites (see Section 4.2.3): "The constant testing made me look for a quick fix rather than examine the code to see what was at the heart of the problem. Seeing the 'success' message made me think too often that I'd finished a section of code, when in fact, there may be additional errors the test cases don't catch."

4.4.3 Positive feedback

Participants who enjoyed using the tools noted the tools' ease of use and the quickness with which they felt they could code. One enjoyed watching unimplemented tests disappear as each was correctly addressed. Several mentioned enjoying being freed from the mechanical tedium of frequent manual testing: "Once I finally figured out how it worked, I got even lazier and never manually ran the test cases myself anymore." One said that it is "especially useful for someone extremely prone to stupid typo-style errors, the kind that are obvious and easily fixable when you see the error line but which don't jump out at a glance."

Staff feedback was predominantly positive. The head TA reported, "the continuous testing worked well for students. Students used the output constantly, and they also seemed to have a great handle on the overall environment." Staff reported that participants who were provided the tools for the first problem set and not for the second problem set missed the additional functionality.

Several participants pointed out that the first two problem sets were a special case that made continuous testing especially useful. Full test suites were provided by the course staff before the students began coding, and passing the test suite was a major component of students' grades on the assignments. Several participants mentioned

that they were unsure they would use continuous testing without a provided test suite, because they were uncomfortable writing their own testing code, or believed that they were incapable of doing so. One said that "In my own programming, there are seldom easily tested individual parts of the code." It appears that the study made participants think much more about testing and modular design, which are both important parts of the goals of the class, and are often ignored by novice programmers. The tools are likely to become even more useful to students as they learn these concepts.

4.4.4 Negative feedback

Participants who didn't enjoy using the tools often said that it interfered with their established working habits. One said "Since I had already been writing extensive Java code for a year using emacs and an xterm, it simply got in the way of my work instead of helping me. I suppose that, if I did not already have a set way of doing my coding, continuous testing could have been more useful." Many felt that the reporting of compilation errors (as implemented in the tool) was not helpful, because far too often they knew about the errors that were reported. Others appear to have not understood the documentation. Several didn't understand how to get more information about the compile errors reported in the modeline.

Some participants believed that when the tool reported a compilation or test error, that the tool had saved and compiled their code. In fact, the tool was reporting what would happen were the user to save, compile, and test the code. Some users were surprised when running the tests (without saving and compiling their work) gave different results than the hypothetical ones provided by the tool.

4.4.5 Suggestions for improvement

Participants had many suggestions for improvement. One recommended more flexibility in its configuration. (As provided, the tools were hardcoded to report feedback based on the staff-provided test suite. After the study completed, students were given

instructions on using the tools with their own test suite.) Another wanted even more sophisticated feedback, including a "guess" of why the error is occurring. Section 6 proposes integrating continuous testing with Delta Debugging [40], to provide some of these hints.

Implementation issues

A problem that confused some students is that the continuous testing tool filtered out some information from the JUnit output before displaying it. In particular, it removed Java stack frames related to the JUnit infrastructure. These were never relevant to the code errors, but some users were alarmed by the differences between the continuous testing output and the output of the tests when run directly with JUnit.

Another problem was that when a test caused an infinite loop in the code under test, no output appeared. This is identical to the behavior of standard JUnit, but since students had not manually run the tests, they thought that the tool had failed.

Some participants reported an irreproducible error in which the results appeared not to change to reflect the state of the code under particular circumstances. One participant reported that this happened 2 or 3 times during the two weeks of the study. These participants still reported that they would continue using the tools in the future, so we assume it was not a huge impediment to their work.

The most common complaint and improvement recommendation was that on compute-bound workstations (such as a 333-MHz Pentium II running a modern operating system and development environment, or a dialup workstation shared with up to 100 other users all running X applications), the background compilation and testing processes could monopolize the processor, sometimes so much that "cursor movement and typing were erratic and uncontrollable." One said that "it needs a faster computer to be worthwhile." However, most students found the performance acceptable. We conclude that potential users should be warned to use a system with acceptable performance, and that additional performance optimizations are worthwhile.

4.5 Threats to validity

Our experiment has produced statistically significant results showing that for student developers using a test-first methodology, a continuous compilation tool doubles the success rate in creating a correct program, and a continuous testing tool triples the success rate. However, the circumstances of the experiment must be carefully considered before applying the results to a new situation.

One potential problem with the experiment is the relative inexperience of the participants. They had on average 2.8 years of programming experience, but only 0.4 years of experience with Java. Two thirds of them were not initially familiar with the notion of regression testing. More experienced programmers might not need the tools as much — or they might be less confused by them and make more effective use of them. Student feedback suggests that the benefits are most available to users who are open to new development tools and methodologies, not those who are set in their ways (see Sections 4.2.2 and 4.4.4).

The participants were all students; this is both a strength and a weakness. On the plus side, use of students has several benefits. The subject demographics are relatively homogeneous, each subject performed the same task, the subjects were externally motivated to perform the task (it was part of their regular coursework, not an invention of the experimenters), and most subjects were exposed to two experimental treatments. It would be very difficult to reproduce these conditions with professional developers. [1] For this reason, controlled experiments in software engineering commonly use students.[1] On the minus side, use of students limits our ability to generalize these results to other software developers. However, the results strongly suggest that continuous compilation and continuous testing are valuable at least for beginning programmers. The enthusiastic adoption by professionals of programming environments offering continuous compilation suggests that its benefits are not limited to students, and the qualitative feedback from students, from TAs, and from

[1]The authors reviewed the last two proceedings of each of ICSE, FSE, and ISSTA. Of 5 papers that reported conducting controlled experiments on human subjects ([25, 43, 10, 6, 24]), all used students.

other experienced programmers leads us to believe that the benefits of continuous testing, too, will apply more broadly. We emphasize, however, that there is not yet quantitative evidence of general benefit. Were such evidence desired, it would require a series of experiments with a broad range of programmers and tasks. Such experiments would complement the one we have performed, in that they would be likely to have lesser external threats to validity but greater internal threats to validity.

The experiment involved relatively few participants. We addressed the risk of type I error (a false alarm) by only reporting statistically significant effects; the statistical tests account for the number of data points. There is also a risk of type II error (a failed alarm): only relatively large differences in effects are statistically significant. There may be other effects that are less important (smaller in magnitude) yet would become statistically significant, given a larger data set. The statistics that are available to us at least indicate the most important effects.

For example, neither programming experience nor Java experience predicted success at the $p = .05$ level, but both predicted success at the $p = .10$ level. Perhaps with additional participants, the p value would have been lower and we would have concluded that these effects were significant. (Or perhaps the course instructor did a good enough job that prior experience was only a weak predictor of experience, and certainly weaker than which tool the participants used — we are forced to this conclusion by the statistics that are available to us.)

Five participants dropped out of the study in problem set 2. They did so simply by doing some of their work outside Emacs or off the Athena computer system. We eliminated such participants — even those in the control group — because we were unable to measure their time worked and similar quantities, and could not account for effects that the other development environments (and switching between environments) may have had on their success.

We identified several problems with the tools (Section 4.4.5). Many of these have since been corrected, which would likely improve the results for the participants who were provided the tool. Furthermore, some students ignored the tools and thus gained no benefit from them. The results would probably be even better had those students

used the tools.

We proposed continuous testing as an aid to a developer performing maintenance tasks such as adding functionality or refactoring in the presence of a regression test suite (though it may also be useful in other circumstances, such as test-first development). Continuous testing can be thought of as a way of making a test suite more valuable by using it more effectively: failing tests are discovered more quickly than they otherwise would be. Continuous testing is most useful when it relieves developers of difficult, lengthy, or easily forgotten tasks, and when developers are performing maintenance or other tasks that are likely to introduce regression errors. Because these circumstances were not present in our experiment, the experiment provided a much less than ideal scenario for a continuous testing tool. Testing was easy (it required only a single keystroke) and fast (a few seconds, see Figure 4-4); students were repeatedly reminded by the course staff to test often; and initial development tends to introduce fewer regression errors than does maintenance. These factors suggest that use of continuous testing in software maintenance (and in real project settings rather than the classroom) may yield even better results than measured by this experiment. As noted above, future experiments should build upon this one to evaluate continuous testing in other environments.

In our experiment, the developers were given a test suite ahead of time. Thus, our experimental results yield insight into the use of continuous testing in the increasingly popular practice of test-first development (and in certain other circumstances, such as when striving for compatibility or interoperability with an existing system). The experimental results are somewhat less applicable to maintenance, as noted above. Finally, they are not applicable to initial development in the absence of a test suite. Naturally, no test execution methodology can help in the absence of a test suite, or with errors that are not exposed by the test suite.

Chapter 5

Design of a continuous testing plug-in

Section 4.1 described the implementation of a continuous testing feature for Emacs. Encouraged by the results of our user study, we built a plug-in for the Eclipse integrated development environment [11]. This allowed us to compare the challenges of adapting the idea of continuous testing to two very different development environments, around which have grown two user populations with different expectations of their tools and metaphors for their software activities.

Eclipse already contains support for integrated unit testing, automatic asynchronous compilation, and automatic notification of compile errors. This meant that a lot of the infrastructure for continuous testing already existed, both in the technical implementation, and in user expectations and metaphors. We hoped that this would allow us to provide more functionality more quickly than we could in Emacs and to develop a user base more quickly.

In the following sections, we detail the architectural principles behind the plug-ins design (5.1), the design features of Eclipse that influenced its design (5.2), the design of the plug-in itself (5.3), and some remaining engineering tasks for future releases (5.4). Prior to reading this chapter, it may be useful to read Appendix E, which gives a tutorial for users and an overview of the plug-in's functionality.

5.1 Architectural principles and functional design

In building this plug-in, we followed these architectural principles to the greatest extent possible:

- Whenever possible, reuse what's in Eclipse. Eclipse was designed from the ground up for extensibility. The base platform is small, and most of the actual software development functionality is provided by add-on plug-ins that declaratively contribute to extension points in the base platform, and declare their own extension points to which other plug-ins can contribute. This allowed us to, in many places, simply contribute the functionality we wanted to provide into the right place without any duplication.

- When reuse is impossible, design for future reuse. In some cases, the plug-in we wished to extend did not provide an extension point where we needed it, so it was necessary to cut and paste code from the existing plug-in into our plug-in. In these cases, we kept the copied code as close to the original as possible, so that if extension points are provided in the future, it will be easy to make use of them.

- At all times, maintain a consistent front end to the user. Consistency meant not changing behavior in old circumstances (for example, leaving the existing JUnit result view to only show the result of manually launched tests), and allowing users to bring old metaphors to bear on new functionality (test failures look similar to, but distinguishable from, compile errors).

- Without breaking the above principles, make code as testable as possible. Writing automated tests for new UI features in Eclipse is challenging. The tasks of interest to continuous testing, such as compiling, running tests, adding error notifications to documents, and updating the GUI, are often performed by several asynchronous components communicating across extension points defined in XML files, making it difficult to know the right way to initiate a process and determine when it had finished. However, the testing paid off many times

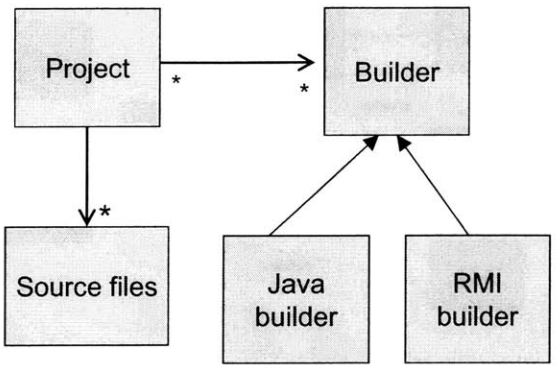

Figure 5-1: Static structure of auto-build framework in Eclipse

by preventing backslides in functionality even when sizable design changes were considered. Testability influenced our design in several ways, especially by leading us to build in API's to the test running process that made it easier to monitor progress.

For the user-visible functional design of the plug-in, please refer to Appendix E, which is a tutorial that ships with the plug-in.

5.2 Eclipse design overview

In this section, we put forth some of the salient design points of Eclipse that influenced the technical design of the continuous testing plug-in. We focus on the static structure and dynamic behavior of both the auto-building framework and the JUnit launch framework. This is a high-level overview of the design, omitting lower-level details where possible.

5.2.1 Auto-building

Figure 5-1 shows the static structure of projects, source files, and builders in Eclipse. Source files are grouped into projects for purposes of auto-building, running, and version control. Each project references an ordered list of zero or more builders, which are contributed through an extension point. Each builder is responsible for

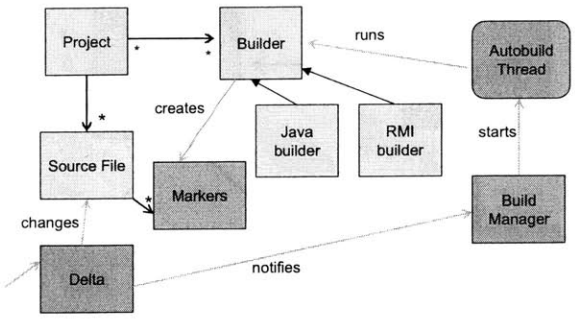

Figure 5-2: Dynamic behavior of auto-build framework in Eclipse when the user changes a source file.

responding to incremental changes to keep project state consistent; for example, the Java builder uses a compiler to keep built class files up to date with the Java source files in the project, whereas the RMI builder keeps generated skeleton and stub files up to date with any RMI interfaces in the project.

Figure 5-2 shows the behavior of the auto-building framework when the user changes one of the project source files. A delta is created and applied to the source file. A global build manager is notified of the delta, and decides which projects are affected. The build manager then starts a new thread to actually carry out the autobuild. This new thread passes control to each builder for the project in turn. If a builder encounters an error while responding to the delta (such as a compile error or classpath problem), it can create one or more markers that attach to source files in the project at particular places. The GUI is updated to indicate these markers in several ways, with line items in the "Problems" table, icons in the margin of the source, and, possibly, a wavy underline under the relevant characters in the source.

5.2.2 JUnit launching

Eclipse provides a launching framework for users to start new Java processes to execute code within a project (the code is kept constantly compiled by the auto-build framework detailed above). Figure 5-3 shows the persistent static structure that enables this functionality.

Eclipse keeps references to a set of user-defined or automatically generated launch

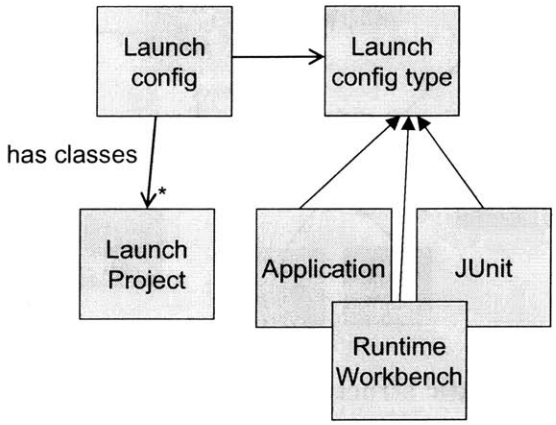

Figure 5-3: Static structure of launching framework in Eclipse

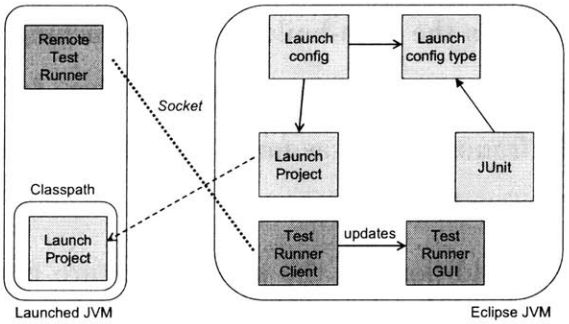

Figure 5-4: Dynamic behavior of launching framework for JUnit in Eclipse

configurations. Each configuration references one or more projects that (by themselves or in referenced dependent projects and libraries) contain all of the classes needed for the process to run. Each configuration also is of a particular launch type. Three examples shown in the figure include a launch type for running Java applications, one for running JUnit tests using a graphical test runner, and one for starting a new instance of Eclipse that includes a plug-in under development.

Figure 5-4 shows how this is put to use when specifically a JUnit launch is initiated. A new JVM instance is created, with all of the needed projects and libraries placed in its classpath. The main class in this new JVM is a remote test runner. A client in the Eclipse JVM opens a socket to the remote test runner, which the runner uses to communicate its progress and the outcome of each test. The client uses this feedback to update the graphical test result view.

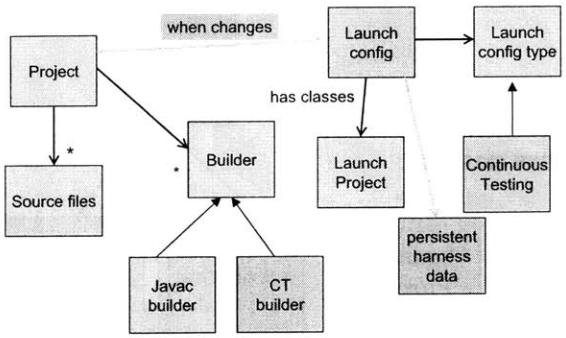

Figure 5-5: Static structure of continuous testing plug-in

5.3 Technical design for continuous testing

Continuous testing combines the auto-building and launching frameworks. It uses a modified JUnit launch type to run tests and create markers for tests that have failed, and uses the auto-build framework to, as much as possible, keep these markers up to date with the state of the code and tests.

Figure 5-5 shows the additions to the static structure made by the continuous testing plug-in. First, a new continuous testing builder type is added to handle keeping the failure markers up to date. For projects that have the continuous testing builder enabled, there must be a reference to a launch, of a new continuous testing launch type, that will be run whenever the project detects a delta. Also, to enable test prioritization (see Appendix E), a persistent dataspace is created attached to each continuous testing launch configuration for a harness to keep testing metadata. A harness that wanted to run the quickest tests first, for example, would use this dataspace to store the most recent observed duration for each test.

Figure 5-6 shows how everything works together when a source file is changed. As before (although unshown here), a delta is created, the build manager is notified, an autobuild thread is started, and each builder in turn is allowed to start. Assuming that Java compilation completes successfully, control is passed to the continuous testing builder, which uses the launch framework to launch the configuration attached to the changed project. The continuous testing configuration starts up modified versions of the remote test runner and local test runner client. It also uses a temporary file to

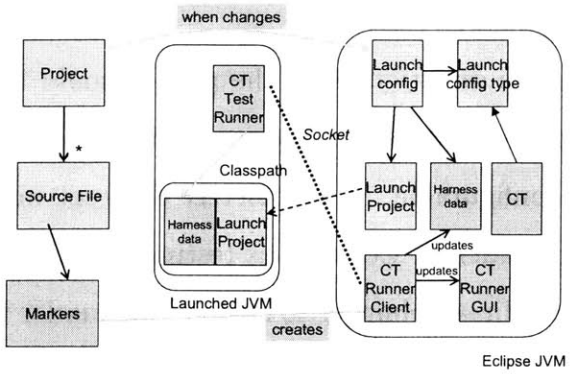

Figure 5-6: Dynamic behavior of continuous testing plug-in

pass the prioritization harness's metadata to a new instance of the harness within the launched JVM. The modified remote test runner asks the harness for the order in which tests should be run, rather than running them in the default order. The modified test runner client not only updates the test result GUI, it also feeds back changes to the harness metadata, and creates test failure markers, when necessary, to attach to the test project.

5.4 Future engineering work

It is a tribute to Eclipse that two mechanisms, each containing cooperating asynchronous processes, could be tied together in the intimate way we have done here, without any indication that such a feature was an original intent of Eclipse's designers and implementers. That said, there is remaining work to be done that will improve the design both of our plug-in, and of Eclipse, if plug-ins such as ours are considered valuable.

The most important next step for the continuous testing plug-in is to break it up into several component plug-ins that communicate through extension points, so that Eclipse users and extenders can pick and choose from the functionality that is currently bundled up into one plug-in. Namely, we imagine separate plug-ins for test prioritization within JUnit; creating markers based on test failures; associating test launches with projects; and using a builder to automatically start tests when a project

changes.

There are many future directions that the plug-in could take. The way in which test failure markers are associated with particular lines of code could be made somehow more general or configurable. A completely different metaphor for test failure notification could be considered. Continuous testing could make use of the hotswapping JVM used by the Eclipse debugger to change code under test on the fly, without having to stop and restart the entire test process. Prioritization and selection based on the actual content of the source delta just produced, and not just on the results of previous runs, should be considered, if the analysis required could be made fast enough. Eclipse's quick fix feature could be extended to provide quick fixes to common dynamic exceptions such as ClassCastExceptions. And it might be possible to use the fact that tests are often being run in the background to offer "instant variable watching": for example, one could image hovering a mouse over a variable to see what that variable's value is the next time that the test process executes that line of code.

Which future directions are taken depends heavily on where interesting research directions lie, and on feedback from users.

Chapter 6

Conclusion

6.1 Status and future work

In pursuing the work detailed in this thesis, we have integrated continuous testing plug-ins into both Eclipse and Emacs. We plan to use these plug-in in industrial case studies to obtain additional qualitative feedback. We will provide continuous testing to a company performing software development for real customers, then observe and interview the developers to learn how they use the feature, their impressions of it, and their suggestions regarding it.

We have also implemented an enhancement to the Perl `Test::Unit` test harness that has a real-time reporting mechanism and implements the Recent Errors test prioritization, based on the results in Section 3.4.3. This has been used in further development work on `delta-capture` and has proved very useful.

All of the studies conducted so far used test suites that could be run fairly quickly, never requiring more than two minutes. There are several ways to extend this to suites that take longer to run.

First, we intend to integrate our Eclipse plug-in with one provided by Andreas Zeller that performs Delta Debugging [40]. Continuous testing gives early indication that a program change has introduced a regression error. However, when test suites take a long time to run, there may have been multiple program changes between the last successful test run and the discovery of a regression error. Delta Debugging

can reduce the program changes to a minimum set that causes the regression error. Both continuous testing and this application of Delta Debugging reduce the number of program changes that a user must examine in order to understand the cause of a regression error; by using continuous testing, then Delta Debugging, the entire process can be made faster and more beneficial to the developer. (Delta Debugging can also simplify inputs [42] or cause-effect chains [41]; such capabilities are orthogonal.)

Second, we are investigating efficient test selection [21, 14, 31] and prioritization [39, 32, 38] for continuous testing, to improve performance on suites that contain a large number of small tests. Strategies based on code coverage seem particularly promising. After a developer edits a procedure, only tests that execute that procedure need to be re-run. This may greatly reduce the number of candidate tests, enabling getting to failing ones more quickly. Effective use of this technique will require evaluation of how long coverage information remains valid or useful as programs evolve.

Third, just as continuous compilation on large programs is infeasible without incremental compilation (see Section 2), continuous testing on large test suites will require some form of incremental testing. For test suites with many tests, test selection runs only those tests that are possibly affected by the most recent change, and test prioritization uses the limited time available to run the tests that are most likely to reveal a recently-introduced error. We have taken some first steps toward showing how prioritization can be combined with continuous testing [33]. However, for tests suites with long-running tests, prioritization is insufficient: it will be necessary to use data collected on previous test runs to run only those *parts* of tests that may reveal recently-introduced errors, a technique we call *test factoring*. For example, if an error is introduced in the file input component of a compiler, a full end-to-end test is not necessary to find it–it will suffice to run only the file-input part of the test, and test that the generated data structures match what was observed in the previous run of the full test. We are actively investigating the implementation of test factoring and its integration with continuous testing.

Finally, we introduced here a method for estimating the impact of a new software

engineering practice or technology without actually deploying the change, by using captured developer behavior to drive model-based simulations. We would like to increase our confidence in the predictions made by this method. First, we may be able to identify additional variables that might affect fix time, such as the impact of the regression error, in terms of number of tests or classes affected. Second, alternative methods of measuring effort, such as lines changed, rather than minutes spent typing, may prove more informative. Third, it may be that there are additional states of developer behavior that we have not captured, related to "life stages" of a project. This is suggested by the fact that the correlation of ignorance time to fix time is lower for the entire project taken as a whole than for the first or second halves of the project.

6.2 Contributions

We have introduced the idea of continuous testing — using excess CPU cycles to test programs while they are being edited — as a feedback mechanism that allows developers to know more confidently the outcome of their changes to a program. Most significantly, it can inform them at an early stage of errors that might otherwise be detected later during development, when they would be more difficult and expensive to correct.

We presented a conceptual model for investigating the usefulness of continuous testing with respect to wasted time and an experimental framework that implements the model. We ran experiments to evaluate the impact on development time of various continuous testing strategies. Together with additional results that verify that early detection of errors saves time overall, the results indicate that continuous testing has the potential to improve development, reducing overall development (programming and debugging) time by 8–15% in our case study. These are improvements to the critical resource in development: human time. Furthermore, the time saved is among the most annoying to developers: problems that could have been easily corrected earlier but grow in seriousness due to lack of attention. There is also promise that

by increasing the usefulness of test suites, continuous testing will allow for a greater return on the investment of time put into producing tests.

In addition to evaluating continuous testing strategies, we have evaluated the development time reduction that could be achieved with a variety of test prioritization strategies. We have also shown how to assess whether a developer is running tests too frequently, too infrequently, or at just the right frequency. These techniques are in themselves effective: increasing test frequency for the Java dataset could reduce wasted development time by 81%, and a more effective test prioritization could produce a 41% reduction in wasted time for the Perl dataset. Our experiments also show which test prioritization techniques are most effective for developers in their daily practice; previous studies typically considered running regression tests just once, when a new version of a software system was complete. Continuous testing was shown to combine the best aspects of both test frequency and test prioritization.

Our user study showed that developers with continuous testing were significantly more likely to complete the programming task than those without, without working for a significantly longer or shorter time. This effect could not be explained by other incidental features of the experimental setup, such as continuous compilation, regular testing, or differences in experience or preference.

A majority of users of continuous testing had positive impressions, saying that it pointed their attention to problems they would have overlooked and helped them produce correct answers faster and write better code. Staff said that students quickly built an intuitive approach to using the additional features. 94% of users said that they intended to use the tool on coursework after the study, and 90% would recommend the tool to others. Few users found the feedback distracting, and no negative effects on productivity were observed.

Students who used continuous compilation without continuous testing were statistically significantly more likely to complete the assignment than those without either tool, although the benefit was not as great as that of continuous testing. Continuous compilation has proved a very popular feature in modern IDE's, but ours is (to our knowledge) the first controlled experiment to assess its effects. The results lend

weight to the claim that it improves productivity, at least for some developers.

These positive results came despite some problems with the tools and despite continuous testing being used in a situation in which it does not necessarily give the most benefit: for initial development in a situation in which tests are easy to run and complete quickly. We have good reason to be hopeful that continuous testing will prove useful for many kinds of software developers.

Finally, we have designed and built two implementation of continuous testing, for the Emacs and Eclipse platforms. The Eclipse plug-in is currently in public beta testing, and can be downloaded from the author's research web page:

`http://pag.csail.mit.edu/~saff/continuoustesting.html`

The research already done on continuous testing gives us reasonable hope that many developers will find continuous testing useful in many different scenarios. Research will continue to enhance continuous testing and further quantify its usefulness.

Appendix A

User study, Emacs plug-in tutorial

All students in the Fall 2003 6.170 class (see Section 4) were encouraged to complete an optional tutorial before starting the first problem set. The section of this tutorial that was relevant to continuous testing has been extracted and reproduced here.

A.1 Handout

David Saff, a PhD student in CSAIL, is studying a new development environment feature called "continuous testing." (Learn more about the study)

If you want to participate in this study, the following exercise will help you understand continuous testing, and get started with Problem Set 1.

- Get your account and problem set directories set up (if you haven't already) by following the directions in the Problem set instructions.
- Remember to log out and log back in.
- Start up Emacs, pull down the Buffers menu, and select "*Messages*". Look for a line that starts with Continuous testing study enabled. your group: The completion of the line will be one of these three options:
  - no automatic help
  - just compilation errors
  - compilation and test errors
- Now, we'll look at convenient compilation. In Emacs, open

  6.170/ps1/playingcards/Card.java.
- Go to the method getSuit()
- Remove the line in the implementation: throw new RuntimeException("Method getSuit is not yet implemented!");. Save the file.

- At the keyboard, type "control-c control-c", and then "return". A *compilation* window should open up. It should tell you that you have a compilation error, and there should be a line partially highlighted in red and blue. Middle-click the red text, and you'll be taken straight to the source of the error.

- Implement getSuit correctly, and save the file. Type "control-c control-c" again, and everything should compile.

- Now, we'll look at convenient testing. Type "control-c control-t" in Emacs. A *compilation* window should open up, showing that you have a lot of test errors: you still have a problem set to complete! Middle-clicking on any red text will take you to the source of the error, or one of the methods that called the error-producing method.

If you are in the no automatic help group, this is the end of the exercise for you. Thanks!

- Remove the implementation of getSuit again, and save the file. Look in the modeline (the grey bar at the bottom of the Emacs window). You should see the message Compiling, and then the message Compile-errors (This might take a long time the very first time you work on a problem set). Note that you didn't have to explicitly run the compilation in order to find out about your compilation errors.

- Middle-click on the words Compile-errors. You should be taken directly to the source of the error. If you had multiple errors, repeatedly clicking Compile-errors would take you through all the errors.

- Fix the error, and save the file. Soon, the "Compile-errors" notification should go away, or be replaced by another message.

If you are in the just compilation errors group, this is the end of the exercise for you. Thanks!

- The mode line should tell you Unimpl-tests:45. This tells you how many tests you still have to complete. Don't worry! 45 sounds like a lot, but you'll find that many of the tests operate on the same method, and it won't be unusual to see 10 tests start working at once as you implement the right files. You can middle-click the Unimpl-tests:45 notification to be taken to the first test which fails. Repeated clicks will take you through the list of test failures.

- As you implement methods correctly, you'll see the number of unimplemented tests shrink. At some point, you might change your code so that a test that previously worked now fails. This is a regression error–your problem set is moving further away from being complete! The regression error will be reported in red in the mode line, to draw your attention to it. You might want to fix the regression error before moving on with your problem set.

- But don't finish the problem set now! Go on with the Tutorial Problems, and come back to it later. Thanks for your time.

Appendix B

User study, problem set 1

The following handout was used by all students in the Fall 2003 6.170 class (see Section 4), to guide them through completing problem set 1. The provided code for this problem set is omitted, but available on request. For an example of the kinds of provided code and test code that were given to students, see Section C.2.

B.1 Handout

6.170 Laboratory in Software Engineering
Fall 2003
Problem Set 1: Implementing an Abstract Data Type
Due: Tuesday, September 9, 2003 at noon

Handout P1

Quick links:

- Problem Set Submission guidelines
- HTML specifications for this problem set
- Provided source code and starter code

Contents:

- Introduction
- Continuous testing study
- Problems
  - Problem 1: CardValue and CardSuit (20 points)
  - Problem 2: Card (20 points)
  - Problem 3: Deck (25 points)
  - Problem 4: TwoPair and Straight (30 points)

  - Problem 5: Playing the Game (5 points)

- Getting started
  - Hints
- Errata
- Q & A

We recommend that you read the entire problem set before you begin work.

Introduction

In this problem set, you will practice reading and interpreting specifications, as well as reading and writing Java source code. You will implement four classes that will complete the implementation of a poker game, and answer questions about both the code you are given and the code you have written.

To complete this problem set, you will need to know

1. How to read procedural specifications (requires, modifies, effects)
2. How to read and write basic Java code
   - code structure and layout (class and method definition, field and variable declaration)
   - method calls
   - operators for:
     - object creation: new
     - field and method access: .
     - assignment: =
     - comparison: ==, !=, <, > , <=, >=
     - arithmetic: +, -, *, /
   - control structure: loops (while and for) and conditional branches (if and else)
   - the Java Collections framework: see Sun's overview or Liskov chapter 6.
3. How to run a Java compiler such as javac to create .class files
4. How to run the Java Virtual Machine (JVM) such as java to execute both your code and staff-provided libraries

Continuous testing study

Students completing this problem set are being asked to participate in a study of a new development environment feature called continuous testing. For more information on this study, please see here. The study's coordinators believe that you will find continuous testing helpful. You may choose to participate or not anonymously–the TA's and instructor will not be informed, and your participation will not affect your grade.

Problems

Problem 1: CardValue and CardSuit (20 points)

Read the specifications for CardSuit and CardValue, classes for playing card suits and numbers. Then read over the staff-provided implementations in CardSuit.java and CardValue.java. You may find it helpful to peruse the code in CardSuitTest.java and CardValueTest.java to see example usages of the classes (albeit in the context of a test driver, rather than application code).

Answer the following questions, writing your answers in the file problem1.txt.

1. Why are CLUBS, DIAMONDS, HEARTS, and SPADES in CardSuit static?

2. Why have a separate class for card suits, instead of just having them be integer constants in some other class? I.e., why not write:

```
public static final int CLUBS = 1;\\
public static final int DIAMONDS = 2;\\
public static final int HEARTS = 3;\\
public static final int SPADES = 4;
```

3. Notice that CardSuit constructor checks to see if the parameter suitName is null before using it. Why isn't this also checked against nullbefore it is used? (this is dereferenced implicitly when accessing, say, smallSuitIcon.)

4. What is the point of having the CardValue and CardSuit constructors be private?

5. Why don't CardValue or CardSuit need to override Object.equals?

6. What are the advantages of having CardValue and CardSuit implement the Comparable interface, as opposed to just creating a method with a different name and signature like boolean isLessThan(CardValue v)? (It might be helpful to follow the link to the Java API docs for Comparable.)

Problem 2: Card (20 points)

Read over the specifications for the Card class. Make sure that you understand the overview and specifications.

Read through the provided skeletal implementation of Card in Card.java. The most significant parts of the provided file are the comments describing how you are to use the provided fields to implement this class.

You'll notice that all of the methods that you need to implement currently have this method body: throw new RuntimeException("Method is not yet implemented!"); Try compiling your code and running the tests (see below). You'll notice that each of the tests that calls one of these methods prints out the error message "Method is not yet implemented", and then shows a *backtrace* . This shows the filenames and line numbers of the statement that threw the exception, the statement that called the method that contained the statement that threw the exception, and so on up to the main method that kicked everything off. When you implement a method, you should be sure to remove the throw new RuntimeException statement.

Fill in an implementation for the methods in the specification of Card. You may define new private helper methods if you want to, but you probably won't need to.

Also, we have provided a fairly rigorous test suite in CardTest.java. You can run the given test suite with JUnit as you program and evaluate your progress and the correctness of your code.

If you are participating in the continuous testing study, you can use the key combination "control-c control-t" in Emacs to run all of the provided tests, for Card and all other classes. You must first use "control-c control-c return" to compile the current version of your code. If you are in the **compilation and test errors** group of the study, the tests will run automatically every time you change and save a file, so explicitly running them should not be necessary. The tests are run in the order you are asked to implement the classes, so any problems with your Card implementation should show at the top of the list.

If you are not participating in the study, or not using Emacs, run the Card test suite, by typing the following command:

```
athena% java junit.swingui.TestRunner ps1.playingcards.CardTest\\
```

For a non-graphical alternative method of running JUnit, usejunit.textui.TestRunner instead ofjunit.swingui.TestRunner, as stated in the Hints.

Indicate clearly as a comment in your source code whether or not your code passes all the tests. We will assume that your code fails if you say nothing.

Problem 3: Deck (25 points)

Follow the same procedure given in Problem 2, but this time fill in the blanks for Deck. You can find the skeleton implementation in Deck.java. The same rules apply here (you may add private helper methods as you like).

We have provided a test suite in DeckTest.java. You can run the given test suite with JUnit as you program and evaluate your progress and the correctness of your code.

Indicate clearly as a comment in your source code whether or not your code passes all the tests. We will assume that your code fails if you say nothing.

Problem 4: TwoPair and Straight (30 points)

Now that you have completed the implementation of cards and decks, there are just a couple of pieces left to finish the programming of the PokerGame application. The class PokerRanking is used to compare hands to one another. You must fill in all unimplemented methods in TwoPair.java and Straight.java, which are two sub-classes of PokerRanking. You will need to look at the documentation for the Hand class. You will also probably find it helpful to look at the implementation of the PokerRanking class, since it contains helper methods like findNOfAKinds,removeNCardsWithValue, and isValidHand that could be useful.

We have provided test suites in TwoPairTest.java and StraightTest.java. You may also need to refer to the test suites' superclass GeneralRankingTest.java.

Indicate clearly as a comment in your source code whether or not your code passes all the tests. We will assume that your code fails if you say nothing.

Problem 5: Playing the Game (5 points)

Now that you have implemented all parts of the poker application, you can run the game by typing the following command:

```
athena% java ps1.PokerGame\\
```

Play a few hands of poker against the computer. See if you can increase your bankroll to at least $1200. When you're done, pull down the File menu and choose Save Score. Save the score file as /6.170/ps1/problem5.txt. Your grade will not depend on your score, but we may post the high scores for fun.

End of Problem Set 1. Please see Problem Set Submission instructions.

Getting started

Make sure you have followed the instructions on the tools handout relevant to directory setup and using Java before you begin development.

If you are doing the problem set on Athena, then you can easily get all the source files by running the command get-ps-files ps1. Fill in the skeleton source files, and test your implementation. You do not need to compile the other source code files that we have provided; compiled class files are already ready for your use in the 6.170 locker.

If you are not doing the problem set on Athena, follow the instructions above to copy the skeleton source code into your /6.170/ps1 directory, then download the files to your machine. Some of the provided code is in the package ps1.playingcards, and some is in ps1.hand. Java requires that the directory structure follows the package structure. So, when you copy the files, you will need to put them in the following directory structure on your machine:

```
ps1/
  playingcards/
    CardSuit.java
    CardSuitTest.java
    CardValue.java
    CardValueTest.java
    Card.java
    CardTest.java
    Deck.java
    DeckTest.java
  hand/
    TwoPair.java
    TwoPairTest.java
    Straight.java
    StraightTest.java
    GeneralRankingTest.java
```

If you are not doing the problem set on Athena, you will also need ps1-lib.jar and junit.jar from /mit/6.170/lib, which contain the rest of the poker game code and the JUnit test framework, respectively.

In any case, your code MUST work on Athena, so if you choose to work on a non-Athena machine, please take a few minutes before submission of your exercises to ensure that it runs correctly on Athena.

By the end of the problem set, you should have the following files in the /6.170/ps1/directory on Athena, ready to submit:

- ps1/problem1.txt
- ps1/problem5.txt
- ps1/playingcards/Card.java
- ps1/playingcards/Deck.java
- ps1/hand/TwoPair.java
- ps1/hand/Straight.java

Once everything is in order, read the Problem Set Submission instructions for how to run a final check of your code and turn it in.

Hints

See the problem set guidelines and advice. Think before you code!

JUnit reloads all of your classes each time you run a test, so you don't need to restart the JUnit application after making changes to your Java code (though you **do** need to recompile your code for the changes to take effect).

For a non-graphical alternative method of running JUnit, usejunit.textui.TestRunner instead ofjunit.swingui.TestRunner .

The provided test suites in problem set 1 are the same ones we will be using to grade your implementation; in later problem sets the staff will not provide such a thorough set of test cases to run on your implementations, but for this problem set you can consider the provided set of tests to be rigorous enough that you do not need to write your own tests.

We strongly encourage you to use the 6.170 Zephyr Instance. The instance can be used to get in touch with classmates and any lab assistants on duty.

Errata

All announcements (MOTDs) related to PS1 are periodically added here.

- 2003/09/05: Problem 4 incorrectly indicated that only evaluateHand and compareSameRankingHands needed to be implemented for Straight and TwoPair. The constructor and accessors for each class also need to be implemented to pass the tests and complete the assignment.

Q & A

This section will list clarifications and answers to common questions about problem sets. We'll try to keep it as up-to-date as possible, so this should be the first place to look (after carefully rereading the problem set handout and the specifications) when you have a problem.

1. **Q:** What is the definition of "two pair" and "straight"?

2. **A:** *Two pair* is a poker hand in which one pair of cards share the same value and another pair of cards share another (different) value. The remaining cards are arbitrary. An example is (Ace of Clubs, Ace of Hearts, 10 of Spades, 10 of Hearts, Jack of Diamonds) – the two pairs are Aces and 10s. Another example of two pair is (7, 7, 7, 2, 2), in which the two pairs are 7s and 2s. This hand is also a *full house* .

   A *straight* is a poker hand in which all the card values form an uninterrupted sequence, such as (2, 3, 4, 5, 6). Aces may be either low or high, so (Ace, 2, 3,

4, 5) and (10, Jack, Queen, King, Ace) are both straights.

For more information, see Poker Hand Ranking.

3. **Q:** Where can I find the JUnit API?
4. **A:** The JUnit API can be found here.
5. **Q:** Do suits matter when comparing two hands of equal rank?
6. **A:** No–only card values matter when comparing hands. Two straights with 4, 5, 6, 7, 8 cards are tied, regardless of the suit of the 8 in either hand.

Back to the Problem Sets.
For problems or questions regarding this page, contact: 6.170-webmaster@mit.edu.

Appendix C

User study, problem set 2

The following handout was used by all students in the Fall 2003 6.170 class (see Section 4), to guide them through completing problem set 2. A subset of the provided code for this problem set is shown in Section C.2.

C.1 Handout

6.170 Laboratory in Software Engineering
Fall 2003
Problem Set 2: Representation Invariants, Specifications, and Immutable Data Types
Due: Tuesday, September 16, at noon

Handout P2

Quick links:

- Specifications: HTML Documentation, Starter code
- Provided source code
- Problem set submission

Contents:

- Introduction
- Problems
  - Problem 1: RatNum (24 points)
  - Problem 2: RatPoly (40 points)
  - Problem 3: RatPolyStack (25 points)
  - Problem 4: PolyCalc (1 point)
  - Problem 5: Object Diagrams (10 points)

- Provided classes
- Getting started
  - Hints
  - Errata
  - Q & A

We recommend that you read the entire problem set before you begin work.

Introduction

In this problem set, you will implement a pair of classes that will complete the implementation of a graphing polynomial calculator, and answer questions about both the code you are given and the code you have written.

To complete this problem set, you will need to know:

1. Basic algebra (rational and polynomial arithmetic)
2. How to read procedural specifications (requires, modifies, effects)
3. How to read and write basic Java code
   - code structure and layout (class and method definition, field and variable declaration)
   - method calls
   - operators for:
     - object creation: new
     - field and method access: .
     - assignment: =
     - comparison: ==, !=, <, >, <=, >=
     - arithmetic: +, -, *, /
   - control structure: loops (while and for) and conditional branches (if, then, else)
4. How to run a Java compiler such as javac to create .class files
5. How to run the Java Virtual Machine (JVM) such as java to execute both your code and staff-provided libraries

Problems

Problem 1: RatNum (24 points)

Read the specifications for RatNum, a class representing rational numbers. Then read over the staff-provided implementation, RatNum.java.
You may find it helpful to peruse the code in RatNumTest.java to see example usages of the RatNum class (albeit in the context of a test driver, rather than application code).

Answer the following questions, writing your answers in the file problem1.txt

1. What is the point of the one-line comments at the start of the add, sub, mul, and div methods?

2. RatNum.div(RatNum) checks whether its argument is NaN (not-a-number). RatNum.add(RatNum) and RatNum.mul(RatNum) do not do that. Explain.

3. Why is RatNum.parse(String) a static method? What alternative to static methods would allow one to accomplish the same goal of generating a RatNum from an input String?

4. The checkRep method is called at the beginning and end of most methods. Why is there no checkRep call at the beginning of the constructor?

5. Imagine that the representation invariant were weakened so that we did not require that the numer and denom fields be stored in reduced form. This means that the method implementations could no longer assume this invariant held on entry to the method, but they also no longer were required to enforce the invariant on exit. The new rep invariant would then be:

   // Rep Invariant for every RatNum r: (r.denom >= 0)

   Which method or constructor implementations would have to change? For each changed piece of code, describe the changes informally, and indicate how much more or less complex the result would be (both in terms of code clarity, and also in terms of execution efficiency). Note that the new implementations must still adhere to the given spec; in particular, RatNum.unparse() needs to output fractions in reduced form.

6. add, sub, mul, and div all end with a statement of the form return new RatNum (numerExpr , denomExpr);. Imagine an implementation of the same function except the last statement is

   this.numer = numerExpr;
   this.denom = denomExpr;
   return this;

   Give two reasons why the above changes would not meet the specifications of the function.

Problem 2: RatPoly (40 points)

Read over the specifications for the RatTerm, RatTermVec, and RatPoly classes. Make sure that you understand the overview for RatPoly and the specifications for the given methods.

Read through the provided skeletal implementation of RatPoly.java. The most significant parts of the provided file are the comments describing how you are to use the provided fields to implement this class. The Representation Invariant is an especially important comment to understand, because what invariants you define can have a drastic effect on which implementations will be legal for the methods of RatPoly.

You'll notice that all of the methods that you need to implement currently have this method body: throw new RuntimeException("Method is not yet implemented!"); For more about what this means, see Problem Set 1.

Fill in an implementation for the methods in the specification of RatPoly. You may define new private helper methods as you like; we have suggested a few ourselves, complete with specifications, but you are not obligated to use them. (The staff believes that doing so will drastically simplify your implementation, however.)

We have provided a checkRep method in RatPoly that tests whether or not a RatPoly instance violates the representation invariant. We highly recommend you use checkRep in the code you write when implementing the unfinished methods.

We have provided a fairly rigorous test suite in RatPolyTest.java. You can run the given test suite with JUnit as you program and evaluate your progress and the correctness of your code.

The test suite depends heavily on the implementation of the RatPoly.unparse() method; if the test suite claims that all or many of the tests are failing, the source of the problem could be a bug in your implementation of unparse().

To run the RatPoly test suite, type the following command:

```
athena% java junit.swingui.TestRunner ps2.RatPolyTest
```

Problem 3: RatPolyStack (25 points)

Follow the same procedure given in Problem 2, but this time fill in the blanks for RatPolyStack.java. The same rules apply here (you may add private helper methods as you like). Since this problem depends on problem 2, you should not begin it until you have completed problem 2 (and the ps2.RatPolyTest test suite runs without any errors).

We have provided a test suite in RatPolyStackTest.java. You can run the given test suite with JUnit as you program and evaluate your progress and the correctness of your code. Again, the test suite relies heavily on RatPoly.unparse().

To run the RatPolyStack test suite, type the following command:

```
athena% java junit.swingui.TestRunner ps2.RatPolyStackTest
```

Problem 4: PolyCalc (1 point)

Now that you have implemented the two remaining classes in the system, you can run the PolyCalc application. This allows you to input polynomials and perform arithmetic operations on them, through a point-and-click user interface. The calculator graphs the resulting polynomials as well.

To run PolyCalc, type the following command:

```
athena% java ps2.PolyCalcFrame
```

A window will pop up with a stack on the left, a graph display on the right, a text area to input polynomials into, and a set of buttons along the bottom. Click the buttons to input polynomials and to perform manipulations of the polynomials on the stack. The graph display will update on the fly, graphing the top four elements of the stack.

Submit your four favorite polynomial equations, in the RatPoly.unparse format, in the file problem4.txt.

Problem 5: Object Diagrams (10 points)

a) Imagine that the following statements have been executed:

```
RatPolyStack s = new RatPolyStack();
RatPoly p1 = RatPoly.parse("-x^2");
RatPoly p2 = p1;
RatPoly p3 = RatPoly.parse("x^3+x^2");
s.push(p1);
s.push(p2);
s.push(p3);
```

Draw an object diagram showing all the objects currently in the system, and which variables point to which objects.

b) Now, imagine that an additional statement is executed:

```
s.add();
```

Draw a diagram showing all the objects now in the system, and which variables point to which objects.

Submit your drawing as a graphic file, with a filename like problem5.gif or problem5.jpg. You can use the Athena program dia to produce them, or your own software, or draw with pen and paper and use a scanner. We will accept GIF, JPG, PS, and PDF–please mail your TA *before Monday* if this is a problem.

Provided classes

The following classes are all provided for you, in compiled form, by the staff:
(If you have run the 6.170 setup scripts, the compiled code should already be in your classpath; you can just use the provided classes according to the HTML Documentation specification.)

With source code also provided: ps2.RatNum ps2.RatNumTest ps2.RatPolyTest ps2.RatPolyStackTest ps2.Cons (as part of the RatPolyStack.java starter code)

With source code *not* provided: ps2.PublicTest ps2.PolyGraph ps2.PolyCalcFrame ps2.RatTerm ps2.RatTermVec

Getting started

Make sure you have followed the instructions on the tools handout relevant to directory setup and using Java before you begin development.

Run get-ps-files ps2 to copy the provided files to your directory. Log out and log back in to reset your environment. Then, edit the specification files into your implementation, and test your implementation. You do not need to compile the other source code files that we have provided; compiled class files are already ready for your use in the 6.170 locker.

See the Specifications for the classes you will implement and those that are provided for you.

By the end of the problem set, you should have the following files ready to submit in your ps2 directory: problem1.txt, RatPoly.java, RatPolyStack.java, problem4.txt, and your Problem 5 graphical solution, along with the Manifest file listing your submission entries (be sure that the Manifest correctly identifies your Problem 5 submission!)

Hints

See the problem set guidelines and advice.

Think before you code! The polynomial arithmetic functions are not difficult, but if you begin implementation without a specific plan, it is easy to get yourself into a terrible mess.

JUnit reloads all of your classes each time you run a test, so you don't need to restart the JUnit application after making changes to your Java code. However, you **do** need to recompile your code with javac for the changes to take effect.

For a non-graphical alternative method of running JUnit, use junit.textui.TestRunner instead of junit.swingui.TestRunner.

The provided test suites in problem set 2 are the same ones we will use to grade your implementation; in later problem sets the staff will not provide such a thorough set of test cases to run on your implementations, but for this problem set you can consider the provided set of tests to be rigorous enough that you do not need to write your tests.

Errata

- The provided implementation for RatPolyStack.checkRep is incorrect. You may use the following implementation:

```
private void checkRep() throws RuntimeException {
  int countResult = 0;
  Cons current = polys;

  while (current != null) {
    countResult++;
    current = current.tail;
  }

  if (countResult != size)
    throw new RuntimeException("size field is not equal to " +
                               "Count(polys). " +
                               "Size constant is " + size +
                               ". Cons cells have length " + countResult);
}
```

Q & A

This section will list clarifications and answers to common questions about problem sets. We'll try to keep it as up-to-date as possible, so this should be the first place to look (after carefully rereading the problem set handout and the specifications) when you have a problem.

- **Q:** Why is the RatTermVec constructor in RatPoly.java not mentioned in the javadoc?
- **A:** Because it is a private constructor, and therefore not part of the public specification. Javadoc should only list public methods available to clients, not private helper methods and constructors. Flags to the javadoc command control which methods get added to the HTML

- **Q:** How do I represent the "0" polynomial?
- **A:** Look at the no-arg public constructor.
- **Q:** What is a representation invariant?
- **A:** A representation invariant is a statement that must be true at the exit of every public method and constructor. You can check that you meet this requirement by using checkRep(). Given that you meet this requirement, you can then assume that the representation invariant holds at the entrance of every public method. This might make writing methods in RatPoly easier. For instance, you can assume without checking that there are no terms with zero coefficients.

Back to the Problem Sets.
For problems or questions regarding this page, contact: 6.170-webmaster@mit.edu.

C.2 Provided code

A subset of the provided code is shown here, as an example of what students were given to work with.

C.2.1 RatPoly.java

```
package ps2;

import java.util.StringTokenizer;

/** <b>RatPoly</b> represents an immutable single-variate polynomial
    expression.  RatPolys have RatNum coefficients and integer
    exponents.
    <p>

    Examples of RatPolys include "0", "x-10", and "x^3-2*x^2+5/3*x+3",
    and "NaN".
*/
public class RatPoly {

    // holds terms of this
    private RatTermVec terms;

    // convenient zero constant
    private static final RatNum ZERO = new RatNum(0);

    // convenient way to get a RatPoly that isNaN
    private static RatPoly nanPoly() {
      RatPoly a = new RatPoly();
      a.terms.addElement(new RatTerm(new RatNum(1, 0), 0));
      return a;
    }

    // Definitions:
    //   For a RatPoly p, let C(p,i) be "p.terms.get(i).coeff" and
    //                        E(p,i) be "p.terms.get(i).expt"
    //                        length(p) be "p.terms.size()"
    // (These are helper functions that will make it easier for us
    // to write the remainder of the specifications.  They are not
    // executable code; they just represent complex expressions in a
    // concise manner, so that we can stress the important parts of
    // other expressions in the spec rather than get bogged down in
    // the details of how we extract the coefficient for the 2nd term
    // or the exponent for the 5th term.  So when you see C(p,i),
    // think "coefficient for the ith term in p".)

    // Abstraction Function:
    //   A RatPoly p is the Sum, from i=0 to length(p), of C(p,i)*x^E(p,i)
    // (This explains what the state of the fields in a RatPoly
    // represents: it is the sum of a series of terms, forming an
    // expression like "C_0 + C_1*x^1 + C_2*x^2 + ...".  If there are no
    // terms, then the RatPoly represents the zero polynomial.)

    // Representation Invariant for every RatPoly p:
    //   terms != null &&
    //   forall i such that (0 <= i < length(p), C(p,i) != 0 &&
    //   forall i such that (0 <= i < length(p), E(p,i) >= 0 &&
    //   forall i such that (0 <= i < length(p) - 1), E(p,i) > E(p, i+1)
    //   (This tells us four important facts about every RatPoly:
    //   * the terms field always points to some usable object,
    //   * no term in a RatPoly has a zero coefficient,
    //   * no term in a RatPoly has a negative exponent, and
    //   * the terms in a RatPoly are sorted in descending exponent order.)

    /** @effects Constructs a new Poly, "0".
     */
    public RatPoly() {
      terms = new RatTermVec();
    }

    /** @requires e >= 0
        @effects Constructs a new Poly equal to "c * x^e".
```

```
    If c is zero, constructs a "0" polynomial.
*/
public RatPoly(int c, int e) {
    throw new RuntimeException("Unimplemented method!");
}

/** @requires 'rt' satisfies clauses given in rep. invariant
    @effects Constructs a new Poly using 'rt' as part of the
    representation. The method does not make a copy of 'rt'.
*/
private RatPoly(RatTermVec rt) {
    throw new RuntimeException("Unimplemented method!");
}

/** Returns the degree of this.
    @requires !this.isNaN()
    @return the largest exponent with a non-zero coefficient, or 0
    if this is "0".
*/
public int degree() {
    throw new RuntimeException("Unimplemented method!");
}

/** @requires !this.isNaN()
    @return the coefficient associated with term of degree 'deg'.
    If there is no term of degree 'deg' in this poly, then returns
    zero.
*/
public RatNum coeff(int deg) {
    throw new RuntimeException("Unimplemented method!");
}

/** @return true if and only if this has some coefficient = "NaN".
 */
public boolean isNaN() {
    throw new RuntimeException("Unimplemented method!");
}

/** Returns a string representation of this.
    @return a String representation of the expression represented
    by this, with the terms sorted in order of degree from highest
    to lowest.
    <p>
    There is no whitespace in the returned string.
    <p>
    Terms with zero coefficients do not appear in the returned
    string, unless the polynomial is itself zero, in which case
    the returned string will just be "0".
    <p>
    If this.isNaN(), then the returned string will be just "NaN"
    <p>
    The string for a non-zero, non-NaN poly is in the form
    "(-)T(+|-)T(+|-)...",where for each
    term, T takes the form "C*x^E" or "C*x", UNLESS:
    (1) the exponent E is zero, in which case T takes the form "C", or
    (2) the coefficient C is one, in which case T takes the form
    "x^E" or "x"
    <p>
    Note that this format allows only the first term to be output
    as a negative expression.
    <p>
    Valid example outputs include "x^17-3/2*x^2+1", "-x+1", "-1/2",
    and "0".
    <p>
*/
public String unparse() {
    throw new RuntimeException("Unimplemented method!");
}
```

```
/** Scales coefficients within 'vec' by 'scalar' (helper procedure).
    @requires vec, scalar != null
    @modifies vec
    @effects Forall i s.t. 0 <= i < vec.size(),
    if (C . E) = vec.get(i)
    then vec_post.get(i) = (C*scalar . E)
    @see ps2.RatTerm regarding (C . E) notation
*/
private static void scaleCoeff(RatTermVec vec, RatNum scalar) {
    throw new RuntimeException("Unimplemented method!");

}

/** Increments exponents within 'vec' by 'degree' (helper procedure).
    @requires vec != null
    @modifies vec
    @effects Forall i s.t. 0 <= i < vec.size(),
    if (C . E) = vec.get(i)
    then vec_post.get(i) = (C . E+degree)
    @see ps2.RatTerm regarding (C . E) notation
*/
private static void incremExpt(RatTermVec vec, int degree) {
    throw new RuntimeException("Unimplemented method!");

}

/** Merges a term into a sequence of terms, preserving the
    sorted nature of the sequence (helper procedure).

    Definitions:
    Let a "Sorted RatTermVec" be a RatTermVec V such that
    [1] V is sorted in descending exponent order &&
    [2] there are no two RatTerms with the same exponent in V &&
    [3] there is no RatTerm in V with a coefficient equal to zero

    For a Sorted(RatTermVec) V and integer e, let cofind(V, e)
    be either the coefficient for a RatTerm rt in V whose
    exponent is e, or zero if there does not exist any such
    RatTerm in V.  (This is like the coeff function of RatPoly.)

    @requires vec != null && sorted(vec)
    @modifies vec
    @effects sorted(vec_post) &&
    cofind(vec_post,newTerm.expt)
    = cofind(vec,newTerm.expt) + newTerm.coeff
*/
private static void sortedAdd(RatTermVec vec, RatTerm newTerm) {
    throw new RuntimeException("Unimplemented method!");

}

/** @return a new Poly equal to "0 - this";
    if this.isNaN(), returns some r such that r.isNaN()
*/
public RatPoly negate() {
    throw new RuntimeException("Unimplemented method!");
}

/** Addition operation.
    @requires p != null
    @return a new RatPoly, r, such that r = "this + p";
    if this.isNaN() or p.isNaN(), returns some r such that r.isNaN()
*/
public RatPoly add(RatPoly p) {
    throw new RuntimeException("Unimplemented method!");
}
```

```
  /** Subtraction operation.
      @requires p != null
      @return a new RatPoly, r, such that r = "this - p";
      if this.isNaN() or p.isNaN(), returns some r such that r.isNaN()
  */
  public RatPoly sub(RatPoly p) {
      throw new RuntimeException("Unimplemented method!");
  }

  /** Multiplication operation.
      @requires p != null
      @return a new RatPoly, r, such that r = "this * p";
      if this.isNaN() or p.isNaN(), returns some r such that r.isNaN()
  */
  public RatPoly mul(RatPoly p) {
      throw new RuntimeException("Unimplemented method!");
  }

  /** Division operation (truncating).
      @requires p != null
      @return a new RatPoly, q, such that q = "this / p";
      if p = 0 or this.isNaN() or p.isNaN(),
      returns some q such that q.isNaN()
      <p>

      Division of polynomials is defined as follows:
      Given two polynomials u and v, with v != "0", we can divide u by
      v to obtain a quotient polynomial q and a remainder polynomial
      r satisfying the condition u = "q * v + r", where
      the degree of r is strictly less than the degree of v,
      the degree of q is no greater than the degree of u, and
      r and q have no negative exponents.
      <p>

      For the purposes of this class, the operation "u / v" returns
      q as defined above.
      <p>

      Thus, "x^3-2*x+3" / "3*x^2" = "1/3*x" (with the corresponding
      r = "-2*x+3"), and "x^2+2*x+15 / 2*x^3" = "0" (with the
      corresponding r = "x^2+2*x+15").
      <p>

      Note that this truncating behavior is similar to the behavior
      of integer division on computers.
  */
public RatPoly div(RatPoly p) {
  RatPoly result = new RatPoly();
  if (p.isNaN() || this.isNaN() || p.terms.size() == 0) {
    return nanPoly();
  }

  RatPoly thisCopy = new RatPoly(this.terms.copy());

  while (thisCopy.degree() >= p.degree() &&
         thisCopy.terms.size() > 0) {
    RatTerm tempTerm =
      new RatTerm(thisCopy.terms.get(0).coeff.div(p.terms.get(0).coeff),
                  thisCopy.terms.get(0).expt - p.terms.get(0).expt);
    sortedAdd(result.terms, tempTerm);
    RatPoly tempPoly = new RatPoly();
    tempPoly.terms.addElement(tempTerm);
    thisCopy = thisCopy.sub(p.mul(tempPoly));
  }

  return result;
}
```

```
/** @return a new RatPoly that, q, such that q = dy/dx,
    where this == y.  In other words, q is the
    derivative of this.  If this.isNaN(), then return
    some q such that q.isNaN()

    <p>The derivative of a polynomial is the sum of the
    derivative of each term.

    <p>Given a term, a*x^b, the derivative of the term is:
    (a*b)*x^(b-1)  for b > 0 and 0 for b == 0
*/
public RatPoly differentiate() {
    throw new RuntimeException("Unimplemented method!");
}

/** @requires integrationConstant != null
    @return a new RatPoly that, q, such that dq/dx = this
    and the constant of integration is "integrationConstant"
    In other words, q is the antiderivative of this.
    If this.isNaN() or integrationConstant.isNaN(),
    then return some q such that q.isNaN()

    <p>The antiderivative of a polynomial is the sum of the
    antiderivative of each term plus some constant.

    <p>Given a term, a*x^b,  (where b >= 0)
    the antiderivative of the term is:  a/(b+1)*x^(b+1)
*/
public RatPoly antiDifferentiate(RatNum integrationConstant) {
    throw new RuntimeException("Unimplemented method!");
}

/** @return a double that is the definite integral of this with
    bounds of integration between lowerBound and upperBound.

    <p>  The Fundamental Theorem of Calculus states that the definite integral
    of f(x) with bounds a to b is F(b) - F(a) where dF/dx = f(x)
    NOTE: Remember that the lowerBound can be higher than the upperBound
*/
public double integrate(double lowerBound, double upperBound) {
    throw new RuntimeException("Unimplemented method!");
}

/** @return the value of this polynomial when evaluated at 'd'.
    For example, "x+2" evaluated at 3 is 5, and "x^2-x"
    evaluated at 3 is 6.
    if (this.isNaN() == true), return Double.NaN
*/
public double eval(double d)  {
    throw new RuntimeException("Unimplemented method!");
}

/** @requires 'polyStr' is an instance of a string with no spaces
    that expresses a poly in the form defined in the
    unparse() method.
    @return a RatPoly p such that p.unparse() = polyStr
*/
public static RatPoly parse(String polyStr) {

    RatPoly result = new RatPoly();

    // First we decompose the polyStr into its component terms;
    // third arg orders "+" and "-" to be returned as tokens.
    StringTokenizer termStrings =
        new StringTokenizer(polyStr, "+-", true);

    boolean nextTermIsNegative = false;
```

```
while (termStrings.hasMoreTokens()) {
    String termToken = termStrings.nextToken();

    if (termToken.equals("-")) {
        nextTermIsNegative = true;
    } else if (termToken.equals("+")) {
        nextTermIsNegative = false;
    } else {
        // Not "+" or "-"; must be a term

        // Term is: "R" or "R*x" or "R*x^N" or "x^N" or "x",
        // where R is a rational num and N is a natural num.

        // Decompose the term into its component parts.
        // third arg orders '*' and '^' to act purely as delimiters.
        StringTokenizer numberStrings =
            new StringTokenizer(termToken, "*^", false);

        RatNum coeff;
        int expt;

        String c1 = numberStrings.nextToken();
        if (c1.equals("x")) {
            // ==> "x" or "x^N"
            coeff = new RatNum(1);

            if (numberStrings.hasMoreTokens()) {
                // ==> "x^N"
                String N = numberStrings.nextToken();
                expt = Integer.parseInt(N);

            } else {
                // ==> "x"
                expt = 1;
            }
        } else {
            // ==> "R" or "R*x" or "R*x^N"
            String R = c1;
            coeff = RatNum.parse(R);

            if (numberStrings.hasMoreTokens()) {
                // ==> "R*x" or "R*x^N"
                String x = numberStrings.nextToken();

                if (numberStrings.hasMoreTokens()) {
                    // ==> "R*x^N"
                    String N = numberStrings.nextToken();
                    expt = Integer.parseInt(N);
                } else {
                    // ==> "R*x"
                    expt = 1;
                }

            } else {
                // ==> "R"
                expt = 0;
            }
        }

        // at this point, coeff and expt are initialized.
        // Need to fix coeff if it was preceeded by a '-'
        if (nextTermIsNegative) {
            coeff = coeff.negate();
        }

        // accumulate terms of polynomial in 'result'
        if (!coeff.equals(ZERO)) {
            result.terms.addElement(new RatTerm(coeff, expt));
```

```
                }
            }
        }
        result.checkRep();
        return result;
    }

    /** Checks to see if the representation invariant is being violated and if so, throws RuntimeException
        @throws RuntimeException if representation invariant is violated
    */
    private void checkRep() throws RuntimeException {
        if (terms == null) {
            throw new RuntimeException("terms == null!");
        }
        for (int i=0; i < terms.size(); i++) {
            if ((terms.get(i).coeff.compareTo(ZERO))==0) {
                throw new RuntimeException("zero coefficient!");
            }
            if (terms.get(i).expt < 0) {
                throw new RuntimeException("negative exponent!");
            }
            if (i < terms.size() - 1) {
                if (terms.get(i+1).expt >= terms.get(i).expt) {
                    throw new RuntimeException("terms out of order!");
                }
            }
        }
    }
}
```

C.2.2 RatPolyTest.java

```
package ps2;

import junit.framework.*;

/** This class contains a set of test cases that can be used to
    test the implementation of the RatPoly class.
    <p>
*/
public class RatPolyTest extends TestCase {
  // get a RatNum for an integer
  private RatNum num(int i) {
    return new RatNum(i);
  }

  private RatNum nanNum = (new RatNum(1)).div(new RatNum(0));

  // convenient way to make a RatPoly
  private RatPoly poly(int coef, int expt) {
    return new RatPoly(coef, expt);
  }

  // Convenient way to make a quadratic polynomial, arguments
  // are just the coefficients, highest degree term to lowest
  private RatPoly quadPoly(int x2, int x1, int x0) {
    RatPoly ratPoly = new RatPoly(x2, 2);
    return ratPoly.add(poly(x1, 1)).add(poly(x0, 0));
  }

  // convenience for parse
  private RatPoly parse(String s) {
    return RatPoly.parse(s);
  }

  // convenience for zero RatPoly
  private RatPoly zero() {
    return new RatPoly();
  }

  public RatPolyTest(String name) { super(name); }

  // only unparse is tested here
  private void eq(RatPoly p, String target) {
    String t = p.unparse();
    assertEquals(target, t);
  }

  private void eq(RatPoly p, String target, String message) {
    String t = p.unparse();
    assertEquals(message, target, t);
  }

  // parses s into p, and then checks that it is as anticipated
  // forall i, parse(s).coeff(anticipDegree - i) = anticipCoeffForExpts(i)
  //   (anticipDegree - i) means that we expect coeffs to be expressed
  //   corresponding to decreasing expts
  private void eqP(String s, int anticipDegree, RatNum[] anticipCoeffs) {
    RatPoly p = parse(s);
    assertTrue(p.degree() == anticipDegree);
    for (int i=0; i<= anticipDegree; i++) {
      assertTrue("wrong coeff; \n"+
                 "anticipated: "+anticipCoeffs[i]+
                 "; received: "+p.coeff(anticipDegree - i)+"\n"+
                 " received: "+p+" anticipated:"+s,
                 p.coeff(anticipDegree - i).equals( anticipCoeffs[i] ));
    }
  }
```

```
// added convenience: express coeffs as ints
private void eqP(String s, int anticipDegree, int[] intCoeffs) {
  RatNum[] coeffs = new RatNum[intCoeffs.length];
  for (int i = 0; i < coeffs.length; i++) {
    coeffs[i] = num(intCoeffs[i]);
  }
  eqP(s, anticipDegree, coeffs);
}

// make sure that unparsing a parsed string yields the string itself
private void assertUnparseWorks(String s) {
  assertEquals(s, parse(s).unparse());
}

public void testParseSimple() {
  eqP("0", 0, new int[]{0});
  eqP("x", 1, new int[]{1, 0});
  eqP("x^2", 2, new int[]{1, 0, 0});
}

public void testParseMultTerms() {
  eqP("x^3+x^2", 3, new int[]{ 1, 1, 0, 0});
  eqP("x^3-x^2", 3, new int[]{ 1, -1, 0, 0});
  eqP("x^10+x^2", 10, new int[]{1,0,0,0,0,0,0,0,1,0,0});
}

public void testParseLeadingNeg() {
  eqP("-x^2",   2, new int[]{-1,0,0});
  eqP("-x^2+1", 2, new int[]{-1,0,1});
  eqP("-x^2+x", 2, new int[]{-1,1,0});
}

public void testParseLeadingConstants() {
  eqP("10*x", 1, new int[]{10, 0});

  eqP("10*x^4+x^2", 4, new int[]{10, 0, 1, 0, 0});

  eqP("10*x^4+100*x^2", 4, new int[]{10, 0, 100, 0, 0});

  eqP("-10*x^4+100*x^2", 4, new int[]{-10, 0, 100, 0, 0});
}

public void testParseRationals() {
  eqP("1/2", 0, new RatNum[]{num(1).div(num(2))});
  eqP("1/2*x", 1, new RatNum[]{num(1).div(num(2)), num(0)});
  eqP("x+1/3", 1, new RatNum[]{num(1) , num(1).div(num(3))});
  eqP("1/2*x+1/3", 1, new RatNum[]{num(1).div(num(2)), num(1).div(num(3))});
  eqP("1/2*x+3/2", 1, new RatNum[]{num(1).div(num(2)), num(3).div(num(2))});
  eqP("1/2*x^3+3/2", 3, new RatNum[]{num(1).div(num(2)), num(0), num(0), num(3).div(num(2))});
  eqP("1/2*x^3+3/2*x^2+1", 3, new RatNum[]{num(1).div(num(2)), num(3).div(num(2)), num(0), num(1)});
}

public void testParseNaN() {
  assertTrue(parse("NaN").isNaN());
}

public void testUnparseSimple() {
  assertUnparseWorks("0");
  assertUnparseWorks("x");
  assertUnparseWorks("x^2");
}

public void testUnparseMultTerms() {
  assertUnparseWorks("x^3+x^2");
  assertUnparseWorks("x^3-x^2");
  assertUnparseWorks("x^100+x^2");
```

```
}

public void testUnparseLeadingNeg() {
  assertUnparseWorks("-x^2");
  assertUnparseWorks("-x^2+1");
  assertUnparseWorks("-x^2+x");
}

public void testUnparseLeadingConstants() {
  assertUnparseWorks("10*x");
  assertUnparseWorks("10*x^100+x^2");
  assertUnparseWorks("10*x^100+100*x^2");
  assertUnparseWorks("-10*x^100+100*x^2");
}

public void testUnparseRationals() {
  assertUnparseWorks("1/2");
  assertUnparseWorks("1/2*x");
  assertUnparseWorks("x+1/3");
  assertUnparseWorks("1/2*x+1/3");
  assertUnparseWorks("1/2*x+3/2");
  assertUnparseWorks("1/2*x^10+3/2");
  assertUnparseWorks("1/2*x^10+3/2*x^2+1");
}

public void testUnparseNaN() {
  assertUnparseWorks("NaN");
}

public void testNoArgCtor() {
  eq(new RatPoly(), "0");
}

public void testTwoArgCtor() {
  eq(poly(0, 0), "0");
  eq(poly(0, 1), "0");
  eq(poly(1, 0), "1");
  eq(poly(-1, 0), "-1");
  eq(poly(1, 1), "x");
  eq(poly(1, 2), "x^2");
  eq(poly(2, 2), "2*x^2");
  eq(poly(2, 3), "2*x^3");
  eq(poly(-2, 3), "-2*x^3");
  eq(poly(-1, 1), "-x");
  eq(poly(-1, 3), "-x^3");
}

public void testDegree() {
  assertTrue("x^0 degree 0", poly(1, 0).degree() == 0);
  assertTrue("x^1 degree 1", poly(1, 1).degree() == 1);
  assertTrue("x^100 degree 100", poly(1, 100).degree() == 100);
  assertTrue("0*x^100 degree 0", poly(0, 100).degree() == 0);
  assertTrue("0*x^0 degree 0", poly(0, 0).degree() == 0);
}

public void testAdd() {
  RatPoly _XSqPlus2X = poly(1, 2).add(poly(1, 1)).add(poly(1, 1));
  RatPoly _2XSqPlusX = poly(1, 2).add(poly(1, 2)).add(poly(1, 1));

  eq(poly(1, 0).add(poly(1, 0)), "2");
  eq(poly(1, 0).add(poly(5, 0)), "6");
  eq(poly(1, 1).add(poly(1, 1)), "2*x");
  eq(poly(1, 2).add(poly(1, 2)), "2*x^2");
  eq(poly(1, 2).add(poly(1, 1)), "x^2+x");
  eq(_XSqPlus2X, "x^2+2*x");
  eq(_2XSqPlusX, "2*x^2+x");
  eq(poly(1, 3).add(poly(1, 1)), "x^3+x");
}
```

```
public void testSub() {
  eq(poly(1, 1).sub(poly(1, 0)), "x-1" );
  eq(poly(1, 1).add(poly(1, 0)), "x+1" );
}

public void testMul() {
  eq(poly(0, 0).mul( poly(0, 0)), "0");
  eq(poly(1, 0).mul( poly(1, 0)), "1");
  eq(poly(1, 0).mul( poly(2, 0)), "2");
  eq(poly(2, 0).mul( poly(2, 0)), "4");
  eq(poly(1, 0).mul( poly(1, 1)), "x");
  eq(poly(1, 1).mul( poly(1, 1)), "x^2");
  eq(poly(1, 1).sub(poly(1, 0)).mul( poly(1, 1).add(poly(1, 0))), "x^2-1");
}

public void testOpsWithNaN(RatPoly p) {
  RatPoly nan = RatPoly.parse("NaN");
  eq(p.add(nan), "NaN");
  eq(nan.add(p), "NaN");
  eq(p.sub(nan), "NaN");
  eq(nan.sub(p), "NaN");
  eq(p.mul(nan), "NaN");
  eq(nan.mul(p), "NaN");
  eq(p.div(nan), "NaN");
  eq(nan.div(p), "NaN");
}

public void testOpsWithNaN() {
  testOpsWithNaN(poly(0, 0));
  testOpsWithNaN(poly(0, 1));
  testOpsWithNaN(poly(1, 0));
  testOpsWithNaN(poly(1, 1));
  testOpsWithNaN(poly(2, 0));
  testOpsWithNaN(poly(2, 1));
  testOpsWithNaN(poly(0, 2));
  testOpsWithNaN(poly(1, 2));
}

public void testImmutabilityOfOperations() {
  // not the most thorough test possible, but hopefully will
  // catch the easy cases early on...
  RatPoly one = poly(1, 0);
  RatPoly two = poly(2, 0);

  one.degree(); two.degree();
  eq(one, "1", "Degree mutates receiver!");
  eq(two, "2", "Degree mutates receiver!");

  one.coeff(0); two.coeff(0);
  eq(one, "1", "Coeff mutates receiver!");
  eq(two, "2", "Coeff mutates receiver!");

  one.isNaN(); two.isNaN();
  eq(one, "1", "isNaN mutates receiver!");
  eq(two, "2", "isNaN mutates receiver!");

  one.eval(0.0); two.eval(0.0);
  eq(one, "1", "eval mutates receiver!");
  eq(two, "2", "eval mutates receiver!");

  one.negate(); two.negate();
  eq(one, "1", "Negate mutates receiver!");
  eq(two, "2", "Negate mutates receiver!");

  one.add(two);
  eq(one, "1", "Add mutates receiver!");
  eq(two, "2", "Add mutates argument!");
```

```
  one.sub(two);
  eq(one, "1", "Sub mutates receiver!");
  eq(two, "2", "Sub mutates argument!");

  one.mul(two);
  eq(one, "1", "Mul mutates receiver!");
  eq(two, "2", "Mul mutates argument!");

  one.div(two);
  eq(one, "1", "Div mutates receiver!");
  eq(two, "2", "Div mutates argument!");
}

public void testEval() {
  RatPoly zero = new RatPoly();
  RatPoly one = new RatPoly(1, 0);
  RatPoly _X = new RatPoly(1, 1);
  RatPoly _2X = new RatPoly(2, 1);
  RatPoly _XSq = new RatPoly(1, 2);

  assertEquals(" 0 at 0 ", 0.0, zero.eval(0.0), 0.0001);
  assertEquals(" 0 at 1 ", 0.0, zero.eval(1.0), 0.0001);
  assertEquals(" 0 at 2 ", 0.0, zero.eval(2.0), 0.0001);
  assertEquals(" 1 at 0 ", 1.0, one.eval(0.0), 0.0001);
  assertEquals(" 1 at 1 ", 1.0, one.eval(1.0), 0.0001);
  assertEquals(" 1 at 1 ", 1.0, one.eval(2.0), 0.0001);

  assertEquals(" x at 0 ", 0.0, _X.eval(0.0), 0.0001);
  assertEquals(" x at 1 ", 1.0, _X.eval(1.0), 0.0001);
  assertEquals(" x at 2 ", 2.0, _X.eval(2.0), 0.0001);

  assertEquals(" 2*x at 0 ", 0.0, _2X.eval(0.0), 0.0001);
  assertEquals(" 2*x at 1 ", 2.0, _2X.eval(1.0), 0.0001);
  assertEquals(" 2*x at 2 ", 4.0, _2X.eval(2.0), 0.0001);

  assertEquals(" x^2 at 0 ", 0.0, _XSq.eval(0.0), 0.0001);
  assertEquals(" x^2 at 1 ", 1.0, _XSq.eval(1.0), 0.0001);
  assertEquals(" x^2 at 2 ", 4.0, _XSq.eval(2.0), 0.0001);

  RatPoly _XSq_minus_2X = _XSq.sub(_2X);

  assertEquals(" x^2-2*x at 0 ", 0.0, _XSq_minus_2X.eval(0.0), 0.0001);
  assertEquals(" x^2-2*x at 1 ", -1.0, _XSq_minus_2X.eval(1.0), 0.0001);
  assertEquals(" x^2-2*x at 2 ", 0.0, _XSq_minus_2X.eval(2.0), 0.0001);
  assertEquals(" x^2-2*x at 3 ", 3.0, _XSq_minus_2X.eval(3.0), 0.0001);
}

public void testCoeff() {
  // coeff already gets some grunt testing in eqP; checking an interesting
  // input here...
  RatPoly _XSqPlus2X = poly(1, 2).add(poly(1, 1)).add(poly(1, 1));
  RatPoly _2XSqPlusX = poly(1, 2).add(poly(1, 2)).add(poly(1, 1));

  assertTrue(_XSqPlus2X.coeff(-1).equals(num(0)));
  assertTrue(_XSqPlus2X.coeff(-10).equals(num(0)));
  assertTrue(_2XSqPlusX.coeff(-1).equals(num(0)));
  assertTrue(_2XSqPlusX.coeff(-10).equals(num(0)));
  assertTrue(zero().coeff(-10).equals(num(0)));
  assertTrue(zero().coeff(-1).equals(num(0)));
}

public void testDiv() {
  // 0/x = 0
  eq(poly(0,1).div(poly(1,1)), "0");

  // x/x = 1
  eq(poly(1,1).div(poly(1,1)), "1");
```

```
  // -x/x = -1
  eq(poly(-1,1).div(poly(1,1)), "-1");

  // x/-x = -1
  eq(poly(1,1).div(poly(-1,1)), "-1");

  // -x/-x = 1
  eq(poly(-1,1).div(poly(-1,1)), "1");

  // -x^2/x = -x
  eq(poly(-1,2).div(poly(1,1)), "-x");

  // x^100/x^1000 = 0
  eq(poly(1, 100).div(poly(1, 1000)), "0");

  // x^100/x = x^99
  eq(poly(1, 100).div(poly(1, 1)), "x^99");

  // x^99/x^98 = x
  eq(poly(1, 99).div(poly(1, 98)), "x");

  // x^10 / x = x^9 (r: 0)
  eq(poly(1, 10).div(poly(1, 1)), "x^9");

  // x^10 / x^3+x^2 = x^7-x^6+x^5-x^4+x^3-x^2+x-1  (r: -x^2)
  eq(poly(1, 10).div(poly(1,3).add(poly(1,2))), "x^7-x^6+x^5-x^4+x^3-x^2+x-1");

  // x^10 / x^3+x^2+x = x^7-x^6+x^4-x^3+x-1 (r: -x)
  eq(poly(1,10).div(poly(1,3).add(poly(1,2).add(poly(1,1)))), "x^7-x^6+x^4-x^3+x-1");

  // x^10+x^5 / x = x^9+x^4 (r: 0)
  eq(poly(1,10).add(poly(1,5)).div(poly(1,1)), "x^9+x^4");

  // x^10+x^5 / x^3 = x^7+x^2 (r: 0)
  eq(poly(1,10).add(poly(1,5)).div(poly(1,3)), "x^7+x^2");

  // x^10+x^5 / x^3+x+3 = x^7-x^5-3*x^4+x^3+7*x^2+8*x-10 (r: 29*x^2+14*x-30)
  eq(poly(1,10).add(poly(1,5)).div
     (poly(1,3).add(poly(1,1)).add(poly(3,0))), "x^7-x^5-3*x^4+x^3+7*x^2+8*x-10");
}

public void testDivComplexI() {
  // (x+1)*(x+1) = x^2+2*x+1
  eq( poly(1,2).add(poly(2,1)).add(poly(1,0))
      .div( poly(1,1).add(poly(1,0))), "x+1");

  // (x-1)*(x+1) = x^2-1
  eq( poly(1,2).add(poly(-1,0))
      .div( poly(1,1).add(poly(1,0))), "x-1");
}

public void testDivComplexII() {
  // x^8+2*x^6+8*x^5+2*x^4+17*x^3+11*x^2+8*x+3 =
  // (x^3+2*x+1) * (x^5+7*x^2+2*x+3)
  RatPoly large = poly(1,8).add(poly(2,6)).add(poly(8,5)).add(poly(2,4))
    .add(poly(17,3)).add(poly(11,2)).add(poly(8,1)).add(poly(3,0));

  // x^3+2*x+1
  RatPoly sub1 = poly(1,3).add(poly(2,1)).add(poly(1,0));
  // x^5+7*x^2+2*x+3
  RatPoly sub2 = poly(1,5).add(poly(7,2)).add(poly(2,1)).add(poly(3,0));

  // just a last minute typo check...
  eq(sub1.mul(sub2), large.unparse());
  eq(sub2.mul(sub1), large.unparse());

  eq(large.div(sub2), "x^3+2*x+1");
```

```
  eq(large.div(sub1), "x^5+7*x^2+2*x+3");
}

public void testDivExamplesFromSpec() {
  // seperated this test case out because it has a dependency on
  // both "parse" and "div" functioning properly

  // example 1 from spec
  eq(parse("x^3-2*x+3").div(parse("3*x^2")), "1/3*x");
  // example 2 from spec
  eq(parse("x^2+2*x+15").div(parse("2*x^3")), "0");
}

public void testDivExampleFromPset() {
  eq(parse("x^8+x^6+10*x^4+10*x^3+8*x^2+2*x+8")
     .div(parse("3*x^6+5*x^4+9*x^2+4*x+8")), "1/3*x^2-2/9");
}

private void assertIsNaNanswer(RatPoly nanAnswer) {
  eq(nanAnswer, "NaN");
}
public void testDivByZero() {
  RatPoly nanAnswer;
  nanAnswer = poly(1, 0).div(zero());
  assertIsNaNanswer(nanAnswer);

  nanAnswer = poly(1, 1).div(zero());
  assertIsNaNanswer(nanAnswer);
}

public void testDivByPolyWithNaN() {
  RatPoly nan_x2 = poly(1,2).mul(poly(1,1).div(zero()));
  RatPoly one_x1 = new RatPoly(1, 1);

  assertIsNaNanswer(nan_x2.div(one_x1));
  assertIsNaNanswer(one_x1.div(nan_x2));
  assertIsNaNanswer(nan_x2.div(zero()));
  assertIsNaNanswer(zero().div(nan_x2));
  assertIsNaNanswer(nan_x2.div(nan_x2));
}

public void testIsNaN() {
  assertTrue(RatPoly.parse("NaN").isNaN());
  assertTrue(!RatPoly.parse("1").isNaN());
  assertTrue(!RatPoly.parse("1/2").isNaN());
  assertTrue(!RatPoly.parse("x+1").isNaN());
  assertTrue(!RatPoly.parse("x^2+x+1").isNaN());
}

public void testDifferentiate() {
  eq(poly(1, 1).differentiate(), "1");
  eq(quadPoly(7, 5, 99).differentiate(), "14*x+5");
  eq(quadPoly(3, 2, 1).differentiate(), "6*x+2");
  eq(quadPoly(1, 0, 1).differentiate(), "2*x");

  assertIsNaNanswer(RatPoly.parse("NaN").differentiate());
}

public void testAntiDifferentiate() {
  eq(poly(1, 0).antiDifferentiate(new RatNum(1)), "x+1");
  eq(poly(2, 1).antiDifferentiate(new RatNum(1)), "x^2+1");
  eq(quadPoly(0, 6, 2).antiDifferentiate(new RatNum(1)), "3*x^2+2*x+1");
  eq(quadPoly(4, 6, 2).antiDifferentiate(new RatNum(0)), "4/3*x^3+3*x^2+2*x");

  assertIsNaNanswer(RatPoly.parse("NaN").antiDifferentiate(new RatNum(1)));
  assertIsNaNanswer(poly(1, 0).antiDifferentiate(new RatNum(1, 0)));
}
```

```
  public void testIntegrate() {
    assertEquals("one from 0 to 1", 1.0, poly(1,0).integrate(0,1), 0.0001 );
    assertEquals("2x from 1 to -2", 3.0, poly(2,1).integrate(1,-2), 0.0001 );
    assertEquals("7*x^2+6*x+2 from 1 to 5", 369.33333333, quadPoly(7,6,2).integrate(1,5), 0.0001 );

  }

  // Tell JUnit what order to run the tests in
  public static Test suite()
  {
    TestSuite suite = new TestSuite();
    suite.addTest(new RatPolyTest("testParseSimple"));
    suite.addTest(new RatPolyTest("testParseMultTerms"));
    suite.addTest(new RatPolyTest("testParseLeadingNeg"));
    suite.addTest(new RatPolyTest("testParseLeadingConstants"));
    suite.addTest(new RatPolyTest("testParseRationals"));
    suite.addTest(new RatPolyTest("testParseNaN"));
    suite.addTest(new RatPolyTest("testUnparseSimple"));
    suite.addTest(new RatPolyTest("testUnparseMultTerms"));
    suite.addTest(new RatPolyTest("testUnparseLeadingNeg"));
    suite.addTest(new RatPolyTest("testUnparseLeadingConstants"));
    suite.addTest(new RatPolyTest("testUnparseRationals"));
    suite.addTest(new RatPolyTest("testUnparseNaN"));
    suite.addTest(new RatPolyTest("testNoArgCtor"));
    suite.addTest(new RatPolyTest("testTwoArgCtor"));
    suite.addTest(new RatPolyTest("testDegree"));
    suite.addTest(new RatPolyTest("testAdd"));
    suite.addTest(new RatPolyTest("testSub"));
    suite.addTest(new RatPolyTest("testMul"));
    suite.addTest(new RatPolyTest("testOpsWithNaN"));
    suite.addTest(new RatPolyTest("testCoeff"));
    suite.addTest(new RatPolyTest("testDiv"));
    suite.addTest(new RatPolyTest("testDivComplexI"));
    suite.addTest(new RatPolyTest("testDivComplexII"));
    suite.addTest(new RatPolyTest("testDivExamplesFromSpec"));
    suite.addTest(new RatPolyTest("testDivExampleFromPset"));
    suite.addTest(new RatPolyTest("testDivByZero"));
    suite.addTest(new RatPolyTest("testDivByPolyWithNaN"));
    suite.addTest(new RatPolyTest("testIsNaN"));
    suite.addTest(new RatPolyTest("testEval"));
    suite.addTest(new RatPolyTest("testImmutabilityOfOperations"));
    suite.addTest(new RatPolyTest("testDifferentiate"));
    suite.addTest(new RatPolyTest("testAntiDifferentiate"));
    suite.addTest(new RatPolyTest("testIntegrate"));
    return suite;
  }
}
```

C.2.3 RatPolyStack.java

```
package ps2;

/** Cons is a simple cons cell record type. */
class Cons {
    RatPoly head;
    Cons tail;
    Cons(RatPoly h, Cons t) { head = h; tail = t; }
}

/** <b>RatPolyStack</B> is a mutable finite sequence of RatPoly objects.
    <p>
    Each RatPolyStack can be described by [p1, p2, ... ], where [] is
    an empty stack, [p1] is a one element stack containing the Poly
    'p1', and so on.  RatPolyStacks can also be described
    constructively, with the append operation, ':'. such that [p1]:S
    is the result of putting p1 at the front of the RatPolyStack S.
    <p>
    A finite sequence has an associated size, corresponding to the
    number of elements in the sequence.  Thus the size of [] is 0, the
    size of [p1] is 1, the size of [p1, p1] is 2, and so on.
    <p>
    Note that RatPolyStack is similar to <i>vectors</i> like {@link
    RatTermVec} with respect to its abstract state (a finite
    sequence), but is quite different in terms of intended usage.  A
    stack typically only needs to support operations around its top
    efficiently, while a vector is expected to be able to retrieve
    objects at any index in amortized constant time.  Thus it is
    acceptable for a stack to require O(n) time to retrieve an element
    at some arbitrary depth, but pushing and popping elements should
    be O(1) time.

 */
public class RatPolyStack {

    private Cons polys; // head of list
    private int size;   // redundantly-stored list length

    // Definitions:
    //   For a Cons c, let Seq(c) be [] if c == null,
    //                               [c.head]:Seq(c.tail) otherwise
    //                     Count(c) be 0 if c == null,
    //                                 1 + Count(c.tail) otherwise
    //
    // (These are helper functions that will make it easier for us
    // to write the remainder of the specifications.  They are
    // seperated out because the nature of this representation lends
    // itself to analysis by recursive functions.)

    // Abstraction Function:
    //   RatPolyStack s models Seq(s.polys)
    // (This explains how we can understand what a Stack is from its
    // 'polys' field.  (Though in truth, the real understanding comes
    // from grokking the Seq helper function).)

    // RepInvariant:
    //   s.size == Count(s.polys)
    // (This defines how the 'size' field relates to the 'polys'
    // field.  Notice that s.polys != null is *not* a given Invariant;
    // this class, unlike the RatPoly class, allows for one of its
    // fields to reference null, and thus your method implementations
    // should not assume that the 'polys' field will be non-null on
    // entry to the method, unless some other aspect of the method
    // will enforce this condition.)

    /** @effects Constructs a new RatPolyStack, [].
     */
```

```
public RatPolyStack() {
    throw new RuntimeException("Unimplemented method!");
}

/** Pushes a RatPoly onto the top of this.
    @requires p != null
    @modifies this
    @effects this_post = [p]:this
*/
public void push(RatPoly p) {
    throw new RuntimeException("Unimplemented method!");
}

/** Returns the top RatPoly.
    @requires this.size() > 0
    @modifies this
    @effects If this = [p]:S
    then this_post = S && returns p
*/
public RatPoly pop() {
    throw new RuntimeException("Unimplemented method!");
}

/** Duplicates the top RatPoly on this.
    @requires this.size() > 0
    @modifies this
    @effects If this = [p]:S
    then this_post = [p, p]:S
*/
public void dup() {
    throw new RuntimeException("Unimplemented method!");
}

/** Swaps the top two elements of this.
    @requires this.size() >= 2
    @modifies this
    @effects If this = [p1, p2]:S
    then this_post = [p2, p1]:S
*/
public void swap()  {
    throw new RuntimeException("Unimplemented method!");
}

/** Clears the stack.
    @modifies this
    @effects this_post = []
*/
public void clear() {
    throw new RuntimeException("Unimplemented method!");
}

/** Returns the RatPoly that is 'index' elements from the top of
    the stack.
    @requires index >= 0 && index < this.size()
    @effects If this = S:[p]:T where S.size() = index, then
    returns p.
*/
public RatPoly get(int index)  {
    throw new RuntimeException("Unimplemented method!");
}

/** Adds the top two elements of this, placing the result on top
    of the stack.
    @requires this.size() >= 2
    @modifies this
    @effects  If this = [p1, p2]:S
    then this_post = [p3]:S
    where p3 = p1 + p2
```

```
*/
public void add()  {
    throw new RuntimeException("Unimplemented method!");
}

/** Subtracts the top poly from the next from top poly, placing
    the result on top of the stack.
    @requires this.size() >= 2
    @modifies this
    @effects  If this = [p1, p2]:S
    then this_post = [p3]:S
    where p3 = p2 - p1
*/
public void sub()  {
    throw new RuntimeException("Unimplemented method!");
}

/** Multiplies top two elements of this, placing the result on
    top of the stack.
    @requires this.size() >= 2
    @modifies this
    @effects  If this = [p1, p2]:S
    then this_post = [p3]:S
    where p3 = p1 * p2
*/
public void mul() {
    throw new RuntimeException("Unimplemented method!");
}

/** Divides the next from top poly by the top poly, placing the
    result on top of the stack.
    @requires this.size() >= 2
    @modifies this
    @effects  If this = [p1, p2]:S
    then this_post = [p3]:S
    where p3 = p2 / p1
*/
public void div()  {
    throw new RuntimeException("Unimplemented method!");
}

/** Integrates the top element of this, placing the result on top
    of the stack.
    @requires this.size() >= 1
    @modifies this
    @effects  If this = [p1]:S
    then this_post = [p2]:S
    where p2 = indefinite integral of p1 with integration constant 0
*/
public void integrate()  {
    throw new RuntimeException("Unimplemented method!");
}

/** Differentiates the top element of this, placing the result on top
    of the stack.
    @requires this.size() >= 1
    @modifies this
    @effects  If this = [p1]:S
    then this_post = [p2]:S
    where p2 = derivative of p1
*/
public void differentiate()  {
    throw new RuntimeException("Unimplemented method!");
}

/** @return the size of this sequence.
 */
public int size()  {
```

```
        throw new RuntimeException("Unimplemented method!");
    }

    /** Checks to see if the representation invariant is being violated and if so, throws RuntimeException
        @throws RuntimeException if representation invariant is violated
    */
    private void checkRep() throws RuntimeException {

      if(polys == null) {
        if(size != 0)
          throw new RuntimeException(
            "size field should be equal to zero when polys is null sine stack is empty");
      } else {

        int countResult = 0;
        RatPoly headPoly = polys.head;
        Cons nextCons = polys.tail;

        if(headPoly != null) {
          for (int i = 1;; i++) {
            if(nextCons != null) {
              countResult = i;
              nextCons = nextCons.tail;
            } else
              break;
          }
        }
        if(countResult != size)
          throw new RuntimeException("size field is not equal to Count(s.polys). Size constant is "
                                     +size+" Cons cells have length "+countResult);
      }
    }
}
```

C.2.4 RatPolyStackTest.java

```
package ps2;

import junit.framework.*;

public class RatPolyStackTest extends TestCase {
  // create a new poly that is a constant (doesn't depend on x)
  private RatPoly constantPoly(int constant) {
    return new RatPoly(constant, 0);
  }

  // create a new poly that is a constant (doesn't depend on x)
  // taking a char allows us to represent stacks as strings
  private RatPoly constantPoly(char constant) {
    return constantPoly(Integer.valueOf("" + constant).intValue());
  }

  //* @return a new RatPolyStack instance
  private RatPolyStack stack() { return new RatPolyStack(); }

  // create stack of single-digit constant polys
  private RatPolyStack stack(String desc) {
    RatPolyStack s = new RatPolyStack();

    // go backwards to leave first element in desc on _top_ of stack
    for(int i = desc.length() - 1; i >= 0; i--) {
      char c = desc.charAt(i);
      s.push(constantPoly(c));
    }
    return s;
  }

  // RatPoly equality check
  // (getting around non-definition of RatPoly.equals)
  private boolean eqv( RatPoly p1, RatPoly p2 ) {
    return p1.unparse().equals(p2.unparse());
  }

  // compares 's' to a string describing its values
  // thus stack123 = "123".  desc MUST be a sequence of
  // decimal number chars
  //
  // NOTE: THIS CAN FAIL WITH A WORKING STACK IF RatPoly.unparse IS BROKEN!
  private void assertStackIs( RatPolyStack s, String desc ) {
    assertTrue(s.size() == desc.length());

    for(int i=0; i < desc.length(); i++) {
      RatPoly p = s.get(i);
      char c = desc.charAt(i);
      String asstr =
        "Elem("+i+"): "+p.unparse()+
        ", Expected "+c+
        ", (Expected Stack:"+desc+")";

      assertTrue(asstr, eqv(p, constantPoly(c)));
    }
  }

  public RatPolyStackTest( String name) { super(name); }

  public void testCtor() {
    RatPolyStack stk1 = stack();
    assertTrue( stk1.size() == 0 );
  }

  public void testPush() {
    RatPolyStack stk1 = stack();
```

```
  stk1.push(constantPoly(0));

  assertStackIs( stk1, "0" );

  stk1.push(constantPoly(0));
  assertStackIs( stk1, "00" );

  stk1.push(constantPoly(1));
  assertStackIs( stk1, "100");

  stk1 = stack("3");
  assertStackIs( stk1, "3" );

  stk1 = stack("23");
  assertStackIs( stk1, "23" );

  stk1 = stack("123");
  assertStackIs( stk1, "123" );
}

public void testPushCheckForSharingTwixtStacks() {
  RatPolyStack stk1 = stack();
  RatPolyStack stk2 = stack("123");
  assertStackIs( stk1, "" );
  assertStackIs( stk2, "123" );

  stk1.push(constantPoly(0));
  assertStackIs( stk1, "0" );
  assertStackIs( stk2, "123" );

  stk1.push(constantPoly(0));
  assertStackIs( stk1, "00" );
  assertStackIs( stk2, "123" );

  stk1.push(constantPoly(1));
  assertStackIs( stk1, "100" );
  assertStackIs( stk2, "123" );

  stk2.push(constantPoly(8));
  assertStackIs( stk1, "100" );
  assertStackIs( stk2, "8123" );
}

public void testPop() {
  RatPolyStack stk1 = stack("123");

  RatPoly poly = stk1.pop();
  assertTrue( eqv( poly, constantPoly(1)));
  assertStackIs( stk1, "23" );

  poly = stk1.pop();
  assertTrue( eqv( poly, constantPoly(2) ));
  assertStackIs( stk1, "3" );

  poly = stk1.pop();
  assertStackIs( stk1, "" );
}

public void testDup() {
  RatPolyStack stk1 = stack("3");
  stk1.dup();
  assertStackIs( stk1, "33" );

  stk1 = stack("23");
  stk1.dup();
  assertStackIs( stk1, "223" );
  assertTrue( stk1.size() == 3 );
  assertTrue( eqv( stk1.get(0), constantPoly(2) ));
```

```
  assertTrue( eqv( stk1.get(1), constantPoly(2) ));
  assertTrue( eqv( stk1.get(2), constantPoly(3) ));

  stk1 = stack("123");
  stk1.dup();
  assertStackIs( stk1, "1123" );

}

public void testSwap() {
  RatPolyStack stk1 = stack("23");
  stk1.swap();
  assertStackIs( stk1, "32" );

  stk1 = stack("123");
  stk1.swap();
  assertStackIs( stk1, "213" );

  stk1 = stack("112");
  stk1.swap();
  assertStackIs( stk1, "112" );
}

public void testClear() {
  RatPolyStack stk1 = stack("123");
  stk1.clear();
  assertStackIs( stk1, "" );
  RatPolyStack stk2 = stack("112");
  stk2.clear();
  assertStackIs( stk2, "" );
}

public void testAdd() {
  RatPolyStack stk1 = stack("123");
  stk1.add();
  assertStackIs( stk1, "33" );
  stk1.add();
  assertStackIs( stk1, "6" );

  stk1 = stack("112");
  stk1.add();
  assertStackIs( stk1, "22" );
  stk1.add();
  assertStackIs( stk1, "4" );
  stk1.push( constantPoly(5) );
  assertStackIs( stk1, "54" );
  stk1.add();
  assertStackIs( stk1, "9" );

}

public void testSub() {
  RatPolyStack stk1 = stack("123");
  stk1.sub();
  assertStackIs( stk1, "13" );
  stk1.sub();
  assertStackIs( stk1, "2" );

  stk1 = stack("5723");
  stk1.sub();
  assertStackIs( stk1, "223" );
  stk1.sub();
  assertStackIs( stk1, "03" );
  stk1.sub();
  assertStackIs( stk1, "3" );
}

public void testMul() {
```

```
  RatPolyStack stk1 = stack("123");
  stk1.mul();
  assertStackIs( stk1, "23" );
  stk1.mul();
  assertStackIs( stk1, "6" );

  stk1 = stack("112");
  stk1.mul();
  assertStackIs( stk1, "12" );
  stk1.mul();
  assertStackIs( stk1, "2" );
  stk1.push( constantPoly(4) );
  assertStackIs( stk1, "42" );
  stk1.mul();
  assertStackIs( stk1, "8" );
}

public void testDiv() {
  RatPolyStack stk1 = stack("123");
  stk1.div();
  assertStackIs( stk1, "23" );

  stk1.push( constantPoly(6) );
  stk1.push( constantPoly(3) );
  assertStackIs( stk1, "3623" );
  stk1.div();
  assertStackIs( stk1, "223" );
  stk1.div();
  assertStackIs( stk1, "13" );
  stk1.div();
  assertStackIs( stk1, "3" );
}

public void testDifferentiate() {
  RatPolyStack stk1 = stack("123");
  stk1.differentiate();
  stk1.differentiate();
  stk1.differentiate();
  stk1.differentiate();
  assertTrue ("Test if stack size changes", stk1.size() == 3);
  assertStackIs (stk1, "023");

  RatPoly rp1 = new RatPoly(3, 5);
  RatPoly rp2 = new RatPoly(7, 0);
  RatPoly rp3 = new RatPoly (4,1);
  stk1.push (rp1);
  stk1.push (rp2);
  stk1.push (rp3);

  stk1.differentiate();
  assertTrue ("Test simple differentiate1", stk1.pop().unparse().equals ("4"));
  stk1.differentiate();
  assertTrue ("Test simple differentiate2", stk1.pop().unparse().equals ("0"));
  stk1.differentiate();
  assertTrue ("Test simple differentiate3", stk1.pop().unparse().equals ("15*x^4"));
}

public void testIntegrate() {
  RatPolyStack stk1 = stack("123");
  stk1.integrate();
  stk1.integrate();
  stk1.integrate();
  stk1.integrate();
  assertTrue ("Test if stack size changes", stk1.size() == 3);
  assertTrue ("Test simple integrate1", stk1.pop().unparse().equals ("1/24*x^4"));
  RatPoly rp1 = new RatPoly(15, 4);
  RatPoly rp2 = new RatPoly(7, 0);
  RatPoly rp3 = new RatPoly (4,0);
```

```
    stk1.push (rp1);
    stk1.push (rp2);
    stk1.push (rp3);

    stk1.integrate();
    assertTrue ("Test simple integrate1", stk1.pop().unparse().equals ("4*x"));
    stk1.integrate();
    assertTrue ("Test simple integrate2", stk1.pop().unparse().equals ("7*x"));
    stk1.integrate();
    assertTrue ("Test simple integrate3", stk1.pop().unparse().equals ("3*x^5"));
  }


  // Tell JUnit what order to run the tests in
  public static Test suite()
  {
    TestSuite suite = new TestSuite();
    suite.addTest(new RatPolyStackTest("testCtor"));
    suite.addTest(new RatPolyStackTest("testPush"));
    suite.addTest(new RatPolyStackTest("testPushCheckForSharingTwixtStacks"));
    suite.addTest(new RatPolyStackTest("testPop"));
    suite.addTest(new RatPolyStackTest("testDup"));
    suite.addTest(new RatPolyStackTest("testSwap"));
    suite.addTest(new RatPolyStackTest("testClear"));
    suite.addTest(new RatPolyStackTest("testAdd"));
    suite.addTest(new RatPolyStackTest("testSub"));
    suite.addTest(new RatPolyStackTest("testMul"));
    suite.addTest(new RatPolyStackTest("testDiv"));
    suite.addTest(new RatPolyStackTest("testDifferentiate"));
    suite.addTest(new RatPolyStackTest("testIntegrate"));
    return suite;
  }
}
```

Appendix D

Questionnaire for user study

All students in the Fall 2003 6.170 class (see Section 4), regardless of whether they volunteered for the continuous testing study, were encouraged to complete the following questionnaire. Results gathered from this data are included in Section 4.4.

D.1 Questionnaire

Thank you for answering this survey. Your answers will help us to evaluate and improve continuous testing. It should take about 5 minutes to complete. All answers will be kept strictly confidential. All respondents who answer before Friday, September 19, will be entered into a drawing for $50. Please mail saff@mit.edu if you have any problems.

Prior Experience

All students (even those who did not participate in the study) should answer this section.

Username:

Before 6.170, how many years have you been programming?

- Please select an answer: 0, Less than 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, more than 20

Before 6.170, how many years have you been programming in Java?

- Please select an answer: 0, Less than 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, more than 10

When you are programming, do you primarily work with tools for:

- Windows
- Unix

- Both

Before 6.170, how many years have you used Emacs?

- Please select an answer: 0, Less than 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, more than 20

Have you previously used Emacs to write Java code?

- yes
- no

Have you previously used Emacs to compile code or run tests (using M-x compile)?

- yes
- no

Before 6.170, how many years have you been using a graphical Java IDE (such as Eclipse, JBuilder, or VisualCafe)?

- Please select an answer: 0, Less than 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, more than 20

Before 6.170, were you already familiar with the notions of test cases and regression errors?

- yes
- no

While solving the problem set, how would you evaluate your testing frequency?

- tested too frequently
- tested about the right amount
- tested too infrequently
- only used continuous testing feedback

Approximately how often did you run tests while you were programming?

- Once every ___ minutes.
- I only used continuous testing feedback.
- I only tested after everything was written.

On what system(s) did you complete problem set 1?

- Entirely on Athena.

- Entirely off Athena.
- A mixture of both.

On what system(s) did you complete problem set 2?

- Entirely on Athena.
- Entirely off Athena.
- A mixture of both.

Participation

Did you agree to participate in the study?

- yes
- no

If "yes", skip to the perceptions of tools section.
If "no", why not? (Check all that apply.)

- I don't use Emacs (what IDE do you use?)
- I don't use Athena
- I didn't want the hassle of participating
- I feared that the experiments would hinder my work
- I had privacy concerns about the study.
- other (please explain below)

Comments:

If you did not participate in the study, you are now finished with the survey. Thank you. Please go to the bottom of the page to submit.

Perceptions of tools

Which tools were provided to you (in **either** problem set)?

- no tool
- compilation error notification
- compilation and test error notification

You should only answer questions in the remainder of the survey about the tools that were provided to you.
If you were in the control group (no tools were enabled for you) for both problem sets, then you are now done with the survey. Thank you. Please go to the bottom of the page to submit.

Here, you will be given a series of statements. You should give each statement a score based on the tools that were enabled for you. The choice "strongly disagree" means that you strongly agree with the *opposite* of the provided statement. If you feel that neither the statement nor its opposite were true, choose "neutral".

If test error notification was not provided to you, please leave that column blank.
Statements:

- When the tool indicated an error, the error was often a surprise to me, rather than an error I already knew about.
- The tool helped me discover problems with my code more quickly.
- The tool helped me complete the assignment faster.
- The tool helped me write better code.
- The tool was distracting.
- I enjoyed using the tool.

After addressing an error or unimplemented test that you discovered through continuous testing, were you more likely to hide the window that contained the errors, or leave it open, to watch for new errors?

- Hide it
- Leave it open
- Something else (please explain in comments below)

How could continuous testing be improved?
What did you like least about it?
What tasks could it have supported better?

Did use of continuous testing change the way that you worked (either on the same problem set, or on a subsequent problem set)?

- Definitely
- Somewhat
- A little
- No

How?

Would you use the tool in the remainder of 6.170?

- yes
- no

Would you use the tool in your own programming?

- yes
- no

Would you recommend the tool to other users?

- yes
- no

Please explain why or why not, or explain under what circumstances you would answer yes.

Please use this space for any additional comments.

Thank you for your feedback! Please click this button to submit your survey (and be entered in the drawing for a prize).

Appendix E

Continuous testing tutorial for Eclipse plug-in

The following tutorial is included in the on-line help for the continuous testing plug-in for Eclipse (see Section 5).

E.1 Installing Continuous Testing

The continuous testing plugin should only be used on Eclipse versions 3.0M6 and later (download new version). To install continuous testing using Eclipse's own built-in Software Updates facility:

1. Start Eclipse
2. From the main menubar, select Help > Software Updates > Find and Install
3. Choose "Search for new features to install", and press "Next"
4. Press the button labelled "Add Update Site..."
5. For name, type "Continuous Testing Site". For URL, type `http://pag.csail.mit.edu/~saff/eclipse`
6. You should get a new checkbox in "Sites to include in Search". Check it, and press "Next"
7. Check the box for Continuous Testing, and press "Next"
8. You may get notifications for the license and signature of the plugin. Accept to finish installation.
9. Allow Eclipse to restart.
10. After restarting, select Help > Welcome from the main menubar
11. Scroll to the section on Continuous Testing, and get started!

E.2 What is Continuous Testing?

Continuous testing builds on the automated developer support in Eclipse to make it even easier to keep your Java code well-tested, if you have a JUnit test suite. If continuous testing is enabled, Eclipse runs your tests in the background as you edit your code, and notifies you if any of them fail or cause errors. It is most useful in situations where you already have a test suite while you are changing code: when performing maintenance, bug fixing, refactoring, or using test-first development.

Continuous testing builds on other features of Eclipse:

- **JUnit integration** : Test suites run under continuous testing give you the same information, in the same format, that you already get from Eclipse's JUnit integration. However, continuous testing also helps to automate the choice of when to run tests, which tests to run, and in which order. Continuous testing does not interfere with your current use of JUnit.

- **Compile error notification** : As soon as you write code that contains a compilation error, Eclipse highlights the error in the text and the margin, and creates a task in the Problems table. This makes it easy both to quickly find out if what you wrote was wrong, and to step through and fix problems in an orderly fashion. With continuous testing, you get the same kind of notification when you write or edit code that causes one of your tests to fail. However, test failures are different from compile errors in several ways: test failures can sometimes not easily be tracked to a single line of code, and test failures also can provide more information, such as a backtrace, than compile errors.

Continuous testing was first developed by David Saff at MIT as part of his PhD research. E-mail is always appreciated. Let me know how you're using continuous testing, and if you find any bugs, or have ideas for new features.

E.3 Getting Started

This tutorial assumes that you already have some experience developing Java code in Eclipse and using the built-in JUnit integration. It also assumes that you have already followed the on-line instructions to install the continuous testing feature into Eclipse.

This is not a comprehensive reference to continuous testing features, but should give you enough information to feel comfortable getting started on your own. You can use the provided example code if you like, or follow along using an existing or new project of your own. The example given is contrived to show off several features of continuous testing in a short time, and is not intended to be a model of good development practice.

Creating a project

First, create a new java project called "topten". For this example, we are developing a rudimentary software library to help under-imaginative journalists and television writers come up with "Top Ten" lists of anything imaginable. Create three classes as follows. You should add the JUnit libraries to your classpath now. (Or, if you're familiar with the technique, use Quick Fix to add them once you've created the files.)

- TopTen.java: A utility class for picking the top N objects out of any list.

```
import java.util.ArrayList;
import java.util.List;

public class TopTen {
  public static List getTopN(int i, List list) {
    List returnThis = new ArrayList();
    for (int j = 0; j < i; j++) {
      returnThis.add(list.get(j));
    }
    return returnThis;
  }
}
```

- Joke.java: A simple class for storing the punchline to a joke.

```
public class Joke {
  private String _punchline;

  public Joke(String punchline) {
    _punchline = punchline;
  }

  public String toString() {
    return _punchline;
  }
}
```

- TopTenTest.java: A rudimentary test suite for the TopTen class.

```
import java.util.Arrays;

import junit.framework.TestCase;

public class TopTenTest extends TestCase {
  public void testTopThreeInts() {
```

```
    Integer one = new Integer(1);
    Integer two = new Integer(2);
    Integer three = new Integer(3);
    Integer four = new Integer(4);
    Integer five = new Integer(5);
    Integer six = new Integer(6);

    assertEquals(
      Arrays.asList(new Integer[] { one, two, three }),
      TopTen.getTopN(
        3,
        Arrays.asList(new Integer[]
          { one, two, three, four, five, six })));
  }

  public void testTopTwoStrings() {
    assertEquals(
      Arrays.asList(new String[] { "a", "b" }),
      TopTen.getTopN(2, Arrays.asList(new String[]
        { "a", "b", "c", "d" })));
  }

  public void testTopJoke() {
    Joke chicken =
      new Joke("Why did the chicken cross the road?");
    Joke grapenuts =
      new Joke("What is up with grape nuts?");
    Joke sausage =
      new Joke("Ack!  A talking sausage!");
    Joke bar =
      new Joke("Two men walk into a bar.  The third ducks.");
    assertEquals(
      Arrays.asList(new Joke[] { chicken }),
      TopTen.getTopN(
        1,
        Arrays.asList(new Joke[]
          { chicken, grapenuts, sausage, bar })));
    }
}
```

If you wish, remind yourself of how compile error notification and JUnit integration work in Eclipse. Introduce a compile error into one of your files, and notice how it is flagged immediately with an icon in the margin, with a wavy line in the code text, and with an entry in the Problems view at the bottom of the screen. Fix the compile error. Now, select your topten project, and from the main menu, choose "Run As >

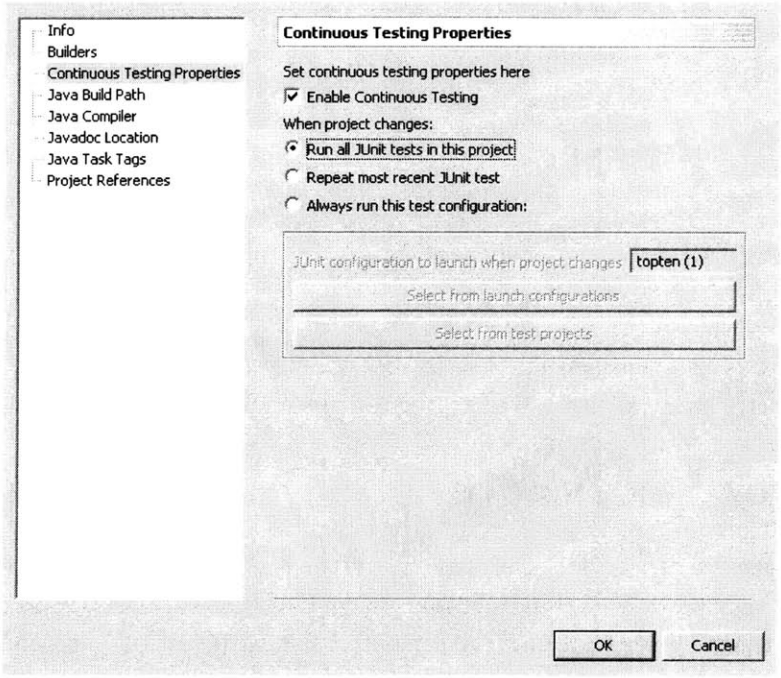

Figure E-1: Continuous Testing tab of the project Properties dialog

JUnit Test". The JUnit view should pop up, and a green bar run across the top as your tests run. Now close the JUnit view.

Since you've already installed continuous testing, it's time to enable it for our project. Right-click the topten project in the Navigator view, and select "Properties" (Figure E-1). Now, in the selection list on the left, choose "Continuous Testing Properties". A number of controls will appear. For now, the only one of interest is the checkbox labelled Enable Continuous Testing. Check the box, and select OK.

At this point, all the tests in your topten project will run. You may see a view appear whose tab is labelled "Continuous Testing". This will look very much like the JUnit results view, with a green bar creeping across the top. If you ever want to keep a close eye on what the continuous testing feature is doing, you could leave this view on top. But for now, it's likely to be more distracting than useful. Hide it by bringing another view to the front on top of it; perhaps the Problems view. You should see that nothing else appears much different than it did before you enabled continuous testing. Congratulations! This means that the tests in your suite all pass.

There is one last thing you need to do to make continuous testing useful. In the Problems view, click the "filters" button on the upper right: it has three arrows pointing to the right. A dialog box will appear. Look for the checkboxes labelled "Show items of type:" Make sure that testFailureMarker is checked (by default, it most likely is not). Now, go on to the next step, where we'll see continuous testing in action.

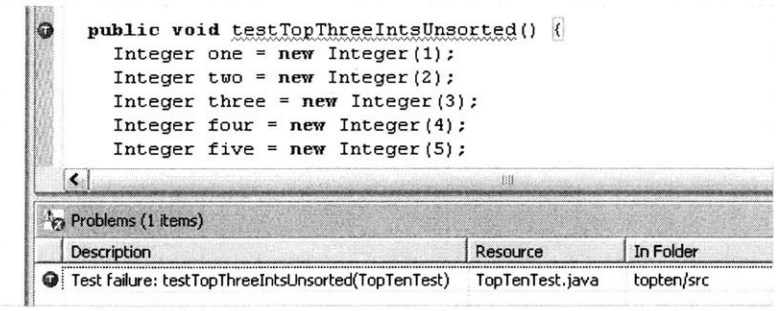

Figure E-2: A test failure notification is shown in three ways: in the Problems view, in the source code margin, and as a wavy red underline under the method name.

E.4 Error Notification

In our imaginary scenario, let's assume that we have a new feature request. Users want getTopN to return the *best* n items, even if the list passed in is not sorted. So far, none of our tests uses a obviously unsorted list as input to getTopN. So let's add one to TopTenTest.java:

```
public void testTopThreeIntsUnsorted() {
  Integer one = new Integer(1);
  Integer two = new Integer(2);
  Integer three = new Integer(3);
  Integer four = new Integer(4);
  Integer five = new Integer(5);
  Integer six = new Integer(6);

  assertEquals(
      Arrays.asList(new Integer[] { one, two, three }),
      TopTen.getTopN(
          3,
          Arrays.asList(new Integer[] { four, one, two, three, five, six })));
}
```

We don't expect this test to pass. Save the file, and soon, continuous testing springs into action. If you watch the status bar at the bottom of the Eclipse window, you'll see a notification of each test as it's run, and then a note that they have passed. More importantly, you'll see several indications of the failing test, many of them similar to what you see when there is a compile error in your code: (Figure E-2)

- A problem has been added to the problems view. The icon for this problem is a red ball, just like a compile error. However, this one has a T for "test failure". (Figure E-3)

Figure E-3: Icon for a test failure (red)

- Double-clicking the failure in the problems view will open and highlight the test method that failed. Notice that the method name is marked with a wavy red underline, and a red T in the margin.

Although the same kind of notifications are used for compile errors and test failures, the two kinds of problems are different. Most importantly, compile errors are usually indicated near the code that needs to be changed, whereas test failures are indicated on the test method that failed. To fix a test failure will often require making a change in the code being tested, somewhere far away from the failure notification.

E.5 What You Don't See

Before we go farther, it's worth quickly pointing out a few things that you might expect to see, but don't. Continuous testing tries hard not to bother you unless there's something you really need to know.

First, notice that the tests being run by continuous testing are not shown in the standard JUnit output view, nor do they appear on the submenu under Run > Run History. These places are reserved for launches that you initiate manually. If you want to see details on what tests continuous testing is running right now, you can open the Continuous Testing view, which we hid in the Getting Started section. However, we recommend that you generally use just the information provided in the status bar and problems view, and leave the Continuous Testing view hidden, to minimize distraction. Hiding the Continuous Testing view is as simple as stacking it with another view (like the Navigator), and then clicking on the tab for the other view.

There are other subtle ways that continuous testing tries to stay out of your way. First, if there is a compile error anywhere in the project you are editing or the project containing the tests, or in those projects' dependencies, no tests are run (you can try this now by introducing a compile error and watching the status line). Second, we'll see later that even if two projects are associated with the same test, only one failure marker per test is ever created.

E.6 Context Menu Options

Right click on the test failure that has currently been created in the Problems View. You'll see some standard options, including "Go To", "Copy", and "Properties", as well as four options specific to continuous testing: (Figure E-4)

- **Rerun Tests**: This will rerun all tests in the suite that generated the selected test failure. The effect is the same as introducing a trivial change to your project code, like slightly rearranging the whitespace. This can be useful if you think

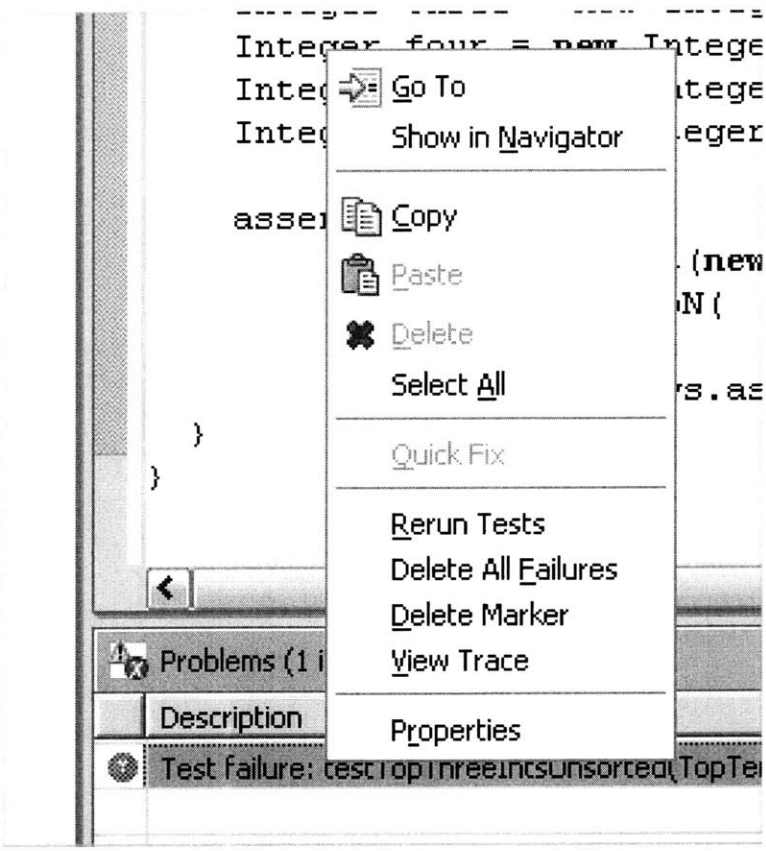

Figure E-4: The right-click context menu for a test failure marker

Figure E-5: Icon for a test that failed the most recent time it was run, and is scheduled to be run again (gray)

that Eclipse has missed an event that could affect your tests, such as relevant resources changing on-disk.

Try selecting Rerun now. Notice that the "T" icon at the left of the Problems View briefly changes from red (Figure E-3) to gray (Figure E-5)), then back to red. If it's too quick to see, add the following line to the beginning of getTopN:

```
Thread.sleep(1000);
```

The gray means that continuous testing is currently planning to rerun the indicated test, and doesn't yet know the result. If the re-run test fails, the icon is switched back to red. If the re-run test passes, the problem is deleted.

- **Delete Marker**: If continuous testing has failed to remove a test failure marker, for whatever reason, that it should have, you can use this option to remove it.

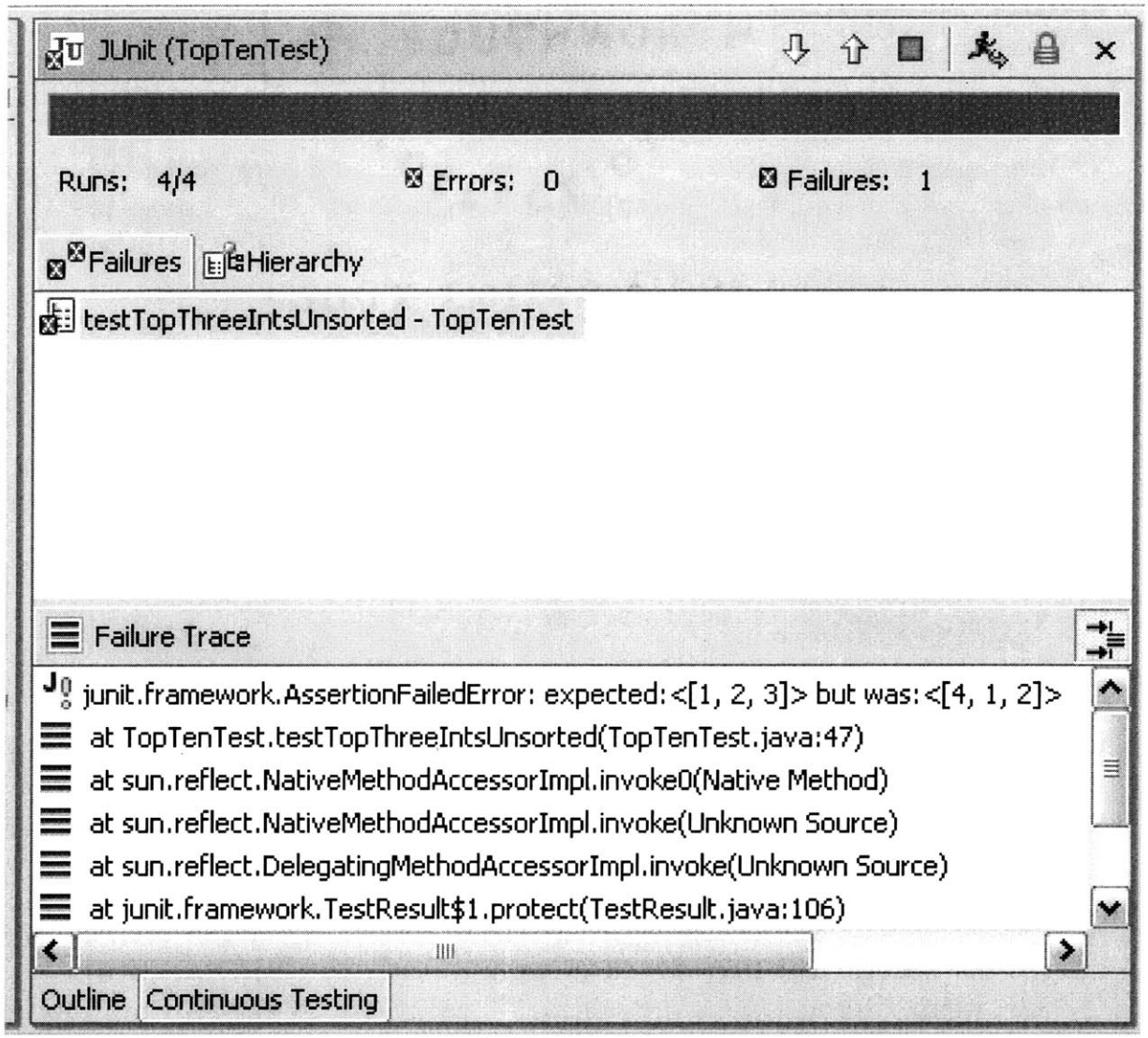

Figure E-6: The continuous testing view, showing a stack trace when requested

However, if the test fails the next time the project is changed, the marker will be re-created.

- **Delete All**: Delete all test failure markers, for now. As tests fail in the future, some of the markers may be recreated.

- **View trace**: This will open the Continuous Testing results view to show the stack trace of the failed assertion or exception that caused the selected test to fail (Figure E-6). You can double-click on elements of the stack trace to visit the source code location indicated.

E.7 Catching an Unexpected Error

In this section, we'll see how continuous testing can come in handy at unexpected times, and learn a clever debugging trick that it enables.

It's finally time to fix the test failure that we introduced several sections ago. The problem, was that getTopN could not handle unsorted lists. We'll fix this problem by creating a sorted copy of the input list, and then using that to grab the top N elements. Let's redefine getTopN like this:

```
public static List getTopN(int i, List list) {
  List sorted = new ArrayList(list);
  Collections.sort(sorted);
  List returnThis = new ArrayList();
  for (int j = 0; j < i; j++) {
    returnThis.add(sorted.get(j));
  }
  return returnThis;
}
```

You'll see that testTopThreeIntsUnsorted now passes, but testTopJoke fails. Why? Right-click the marker in the Problems view and select View Trace. You'll see that a ClassCastException is thrown in Arrays.mergeSort(). If you are an experienced Java programmer, you may realize that this is likely because some element type in a sorted collection does not implement Comparable. This is the kind of error that can sometimes take you by surprise. Continuous testing catches it quickly, and notifies you so you can deal with it before you move on.

But which element type is the culprit? In this case, it's not hard to figure it out, but in more complicated real-life scenarios, it might be difficult to figure out the behavior of the callers of the sort method. You could imagine examining the source, or firing up a debugger, but Continuous Testing allows a third useful option. Simply insert a debugging print statement in the getTopN method to see what element types are being sorted over:

```
public static List getTopN(int i, List list) {
  System.out.println("getTopN element class: " + list.get(0).getClass());
  ...
}
```

As soon as you save, your tests will be re-run, and the console will pop up with the answer to your question (Figure E-7). The element types in question are Integer, String, and Joke. A moment's reflection shows that we will need to have Joke implement Comparable. (Doing this correctly is left as an exercise for the reader, but don't do it yet. We need the failure to stay as we work through the rest of the tutorial.)

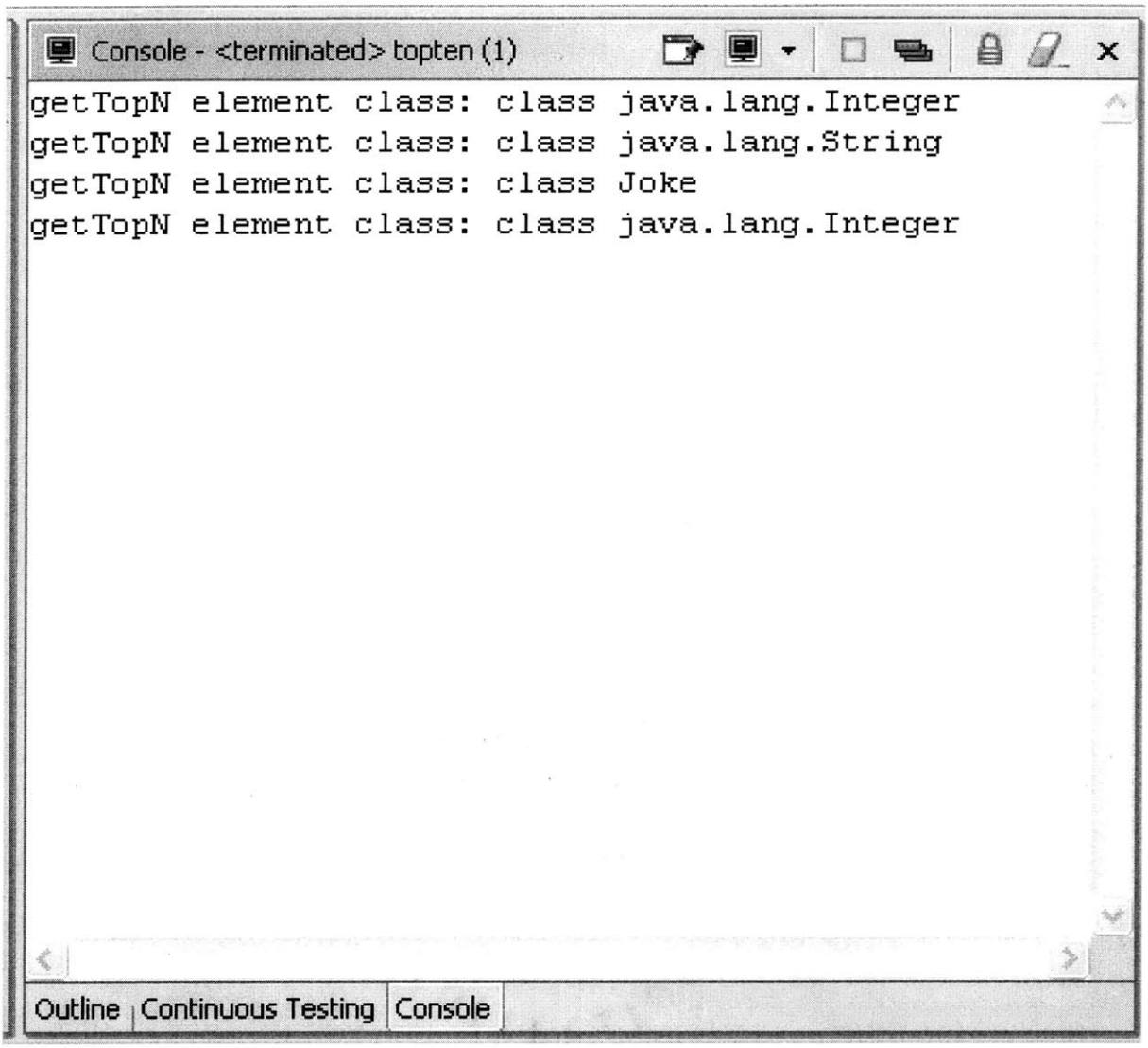

Figure E-7: The console view after adding a System.out.println statement in the tested code

E.8 Prioritizing Tests

By default, the Eclipse JUnit integration always runs the tests in a suite in the same order (although it is sometimes difficult in advance of the first run to predict what that order will be). For continuous testing, this can be frustrating. Consider our current failure, on testTopJoke. As you try to fix the error, the most interesting thing to you will be whether testTopJoke finally passes. However, every time continuous testing restarts, it runs up to three passing tests before trying testTopJoke. To verify this, open the Continuous Testing view, and choose the hierarchy tab. (Figure E-8) Since these tests are pretty fast, it doesn't seem too annoying at this point. However, you can imagine that the frustration would build if each test required 5 or 10 seconds of

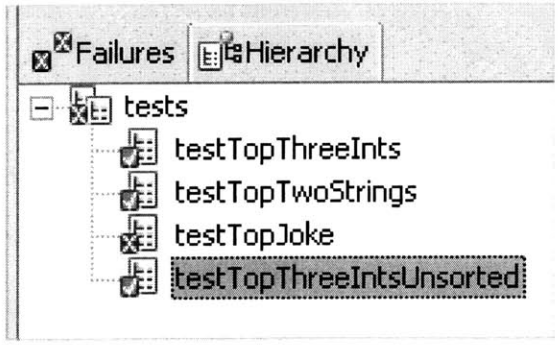

Figure E-8: Hierarchy tab in the Continuous Testing view. The failing test is the third in the test ordering.

time. To help alleviate this issue, continuous testing allows for several kinds of test prioritization:

1. Open the properties dialog for topten, and choose Continuous Testing Properties.

2. We are now going to edit the default continuous testing launch configuration for this project, so select the "Always run this test configuration" dialog, and then press "Select from launch configurations". (Figure E-9)

3. On the right side of the resulting dialog box, you have the option of choosing from six different test prioritization strategies. Select "Most recent failures first": we have found that this is often the most helpful strategy, since the most interesting tests are often those that are currently failed, or that we've recently fixed.

4. Press OK.

To see prioritization in action, leave the Continuous Testing view open. Right-click the marker in the Problem view, and Rerun this test. You'll see that the tests run in exactly the same order as before. Why? The "Most recent failures first" has to observe at least one test run in order to see which tests have failed recently. If you now Rerun the test again, you'll see that testTopJoke, the failing test, has moved up to the top of the test run order.

E.9 Multiple Projects

Each project in Eclipse that has continuous testing enabled can have only one associated test launch configuration. However, a project can reference tests defined in a different project, and two or more projects can share the same launch configuration.

To see why this might be desirable, consider that currently our topten project references the JUnit libraries. While this is convenient for our testing, it pollutes the

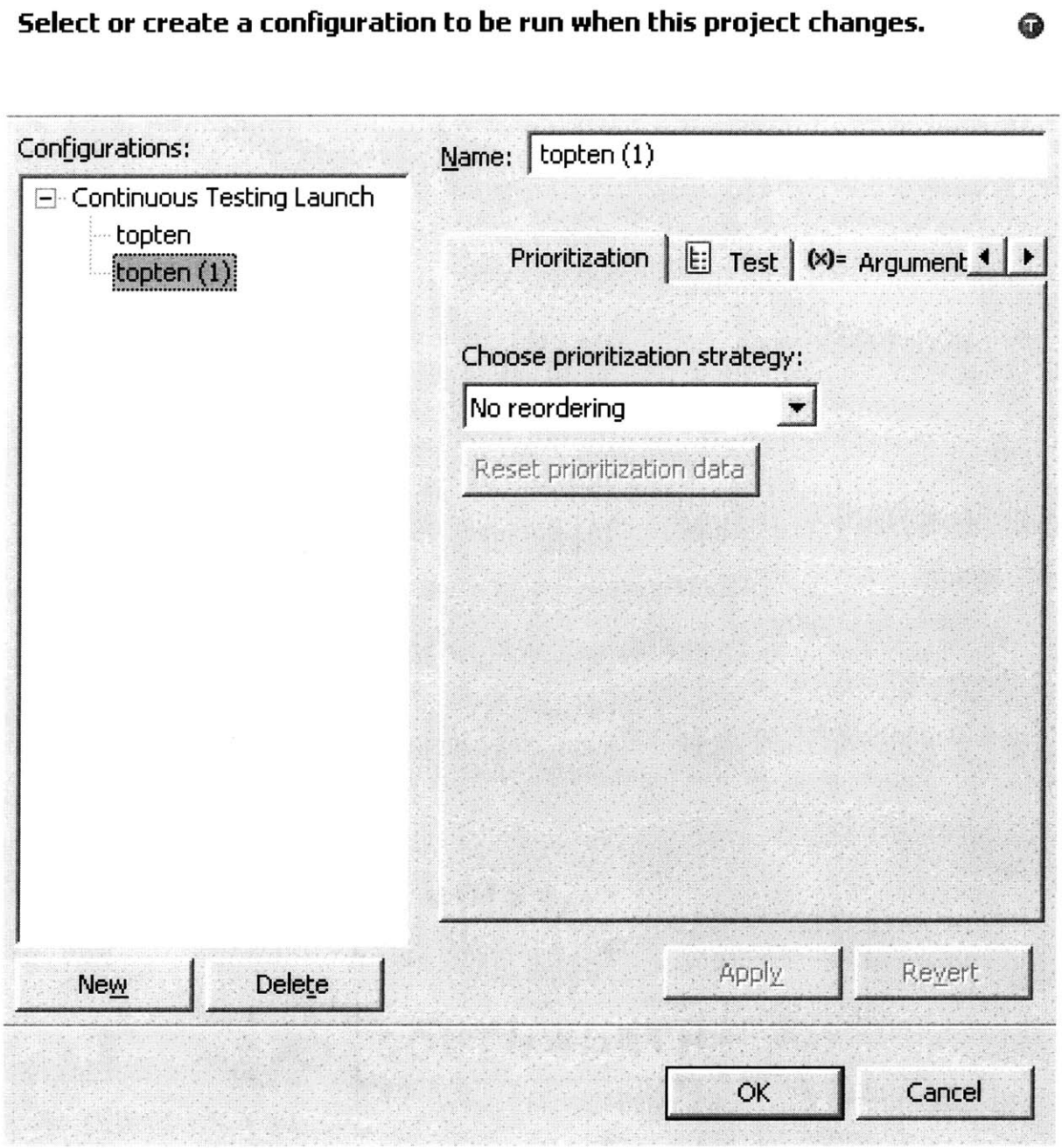

Figure E-9: Dialog for setting the test prioritization strategy on a continuous testing launch configuration

classpath of our library: we don't want to force our clients to have JUnit installed. So, it makes sense to split the tests into their own project.

1. Create a new Java project called "topten.test"
2. Open the properties view on topten.test, and under Java Build Path > Projects, choose "topten"
3. Add the JUnit library to the classpath of topten.test
4. Drag TopTenTest.java from the source folder of topten into the source folder of topten.test.
5. Remove the JUnit library from the classpath of topten.

While you follow the above steps, you may occasionally see errors in the Progress View from continuous testing trying to run tests that suddenly are no longer there. We now need to adjust the continuous testing properties of the two new projects.

First, topten's tests are no longer in the project itself, but in topten.test:

1. Open the Continuous Testing Properties for topten, and choose "Always run this test configuration".
2. Press the "Select from test projects" button, and choose topten.test. We could also have chosen "Select from launch configurations", and created a new configuration by hand, but this is an easier shortcut.
3. Take note of the configuration name that is filled in the "JUnit configuration to launch when project changes" text box: sometimes there may be several configurations created for the same test project.
4. Press OK

Second, we want to be sure that any changes made to the tests in topten.test trigger a test re-run:

1. Open the Continuous Testing Properties for topten, and choose "Always run this test configuration".
2. Press the "Select from launch configurations", and choose the configuration name that you took note of above.
3. Press OK

Making both projects point to the same configuration allows them to share prioritization data, and it ensures that continuous testing handles the markers for the launch configuration correctly. For example, continuous testing will not create two markers for the same test failure if the test was run on behalf of two different projects sharing the same configuration. Try introducing test failures in both topten and topten.test to get comfortable with the features.

E.10 Forward Pointers

The following features of the continuous testing plug-in have not been discussed in this tutorial:

- **Repeat most recent JUnit test** . If you find yourself regularly running different test configurations for the same project at different times, depending on what part of the code you're working on, this option on the Continuous Testing Properties page may be useful. Rather than establishing a permanent link between a project and a launch configuration, this option causes continuous testing, when enabled, to look to the most recent manual JUnit test to decide which launch configuration to use.
- **Source plugin** . If you decide to write an Eclipse plugin that builds on continuous testing, a source plugin has been provided for importing, editing, and debugging out of the box.

Bibliography

[1] Victor R. Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th international conference on Software engineering*, pages 442–449. IEEE Computer Society, 1996.

[2] Walter Baziuk. BNR/NORTEL: Path to improve product quality, reliability, and customer satisfaction. In *Sixth International Symposium on Software Reliability Engineering*, Toulouse, France, October 24–27, 1995.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[4] Kent Beck. *Test-Driven Development: By Example.* Addison-Wesley, Boston, 2002.

[5] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, 1976.

[6] Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the 25th international conference on Software engineering*, pages 93–103. IEEE Computer Society, 2003.

[7] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates, Inc., 2000.

[8] Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In

International Parallel and Distributed Processing Symposium (IPDPS'03), pages 205–214, Nice, France, April 22–26, 2003.

[9] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.

[10] Alastair Dunsmore, Marc Roper, and Murray Wood. Further investigations into the development and evaluation of reading techniques for object-oriented code inspection. In *Proceedings of the 24th international conference on Software engineering*, pages 47–57. ACM Press, 2002.

[11] Eclipse. `http://www.eclipse.org`.

[12] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-ins.* Addison Wesley Longman, 2003.

[13] Stephen José Hanson and Richard R. Rosinski. Programmer perceptions of productivity and programming tools. *Communications of the ACM*, 28(2):180–189, February 1985.

[14] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[15] Peter Henderson and Mark Weiser. Continuous execution: The VisiProg environment. In *Proceedings of the 8rd International Conference on Software Engineering*, pages 68–74, London, August 28–30, 1985.

[16] Idea. `http://www.intellij.com/idea/`.

[17] Philip M. Johnson, Hongbing Kou, Joy M. Agustin, Christopher Chan, Carleton A. Moore, Jitender Miglani, Shenyan Zhen, and William E. Doane. Beyond the Personal Software Process: Metrics collection and analysis for the differently

disciplined. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 641–646, Portland, Oregon, May 6–8, 2003.

[18] Michael Karasick. The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler. In *FSE '98, Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 131–142, Lake Buena Vista, FL, USA, November 3–5, 1998.

[19] Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test application frequency. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 126–135. ACM Press, 2000.

[20] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *International Conference on Machine Learning*, pages 527–534, Stanford, CA, June 2000.

[21] Hareton K. N. Leung and Lee White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, Miami, FL, October 16–19, 1989.

[22] George A. Miller, Eugene Galanter, and Karl H. Pribram. *Plans and the Structure of Behavior.* Holt, Rinehart and Winston, Inc., 1960.

[23] Robert C. Miller. *Lightweight Structure in Text.* PhD thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 2002. Also available as CMU Computer Science technical report CMU-CS-02-134 and CMU Human-Computer Interaction Institute technical report CMU-HCII-02-103.

[24] Daniel L. Moody, Guttorm Sindre, Terje Brasethvik, and Arne Sølvberg. Evaluating the quality of information models: empirical testing of a conceptual model quality framework. In *Proceedings of the 25th international conference on Software engineering*, pages 295–305. IEEE Computer Society, 2003.

[25] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.

[26] Robert P. Nix. Editing by example. *ACM Transactions on Programming Languages and Systems*, 7(4):600–621, October 1985.

[27] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Helsinki, Finland, September 3–5, 2003.

[28] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: Continuous evolution of software after deployment. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 65–69, Rome, Italy, July 22–24, 2002.

[29] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 277–284, Los Angeles, CA, USA, May 19–21, 1999.

[30] Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late than never"? In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 120–131, Paris, France, January 15–17, 1997.

[31] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[32] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[33] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, Denver, CO, November 17–20, 2003.

[34] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development, January 2004.

[35] Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. Incremental compilation in Magpie. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 122–131, Montreal, Canada, June 17–22, 1984.

[36] Shel Siegel. *Object-Oriented Software Testing: A Hierarchical Approach.* John Wiley & Sons, 1996.

[37] Mary Lou Soffa. Continuous testing. Personal communication, February 2003.

[38] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 97–106, Rome, Italy, July 22–24, 2002.

[39] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal. A study of effective regression testing in practice. In *Eighth International Symposium on Software Reliability Engineering*, pages 264–274, Albuquerque, NM, November 2–5, 1997.

[40] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–267, Toulouse, France, September 6–9, 1999.

[41] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 1–10, Charleston, SC, November 20–22, 2002.

[42] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(3):183–200, February 2002.

[43] Marc K. Zimmerman, Kristina Lundqvist, and Nancy Leveson. Investigating the readability of state-based formal requirements specification languages. In *Proceedings of the 24th international conference on Software engineering*, pages 33–43. ACM Press, 2002.