An experimental evaluation of continuous testing during development

David Saff Michael D. Ernst

MIT Computer Science & Artificial Intelligence Lab
200 Technology Square
Cambridge, MA 02139 USA
{saff,mernst}@csail.mit.edu

Abstract

Continuous testing 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 reduces the time and energy required to keep code well-tested, and prevents regression errors from persisting uncaught for long periods of time. This paper experimentally evaluates the promise of continuous testing, based on a controlled human experiment.

The experiment indicates that the tool has a statistically significant effect on developer success in completing a programming task, without affecting time worked. Developers using continuous testing were three times more likely to complete the task before the deadline than those without. Most participants found continuous testing to be useful and believed that it helped them write better code faster, and 90% would recommend the tool to others. The participants were more resilient to distraction than we had feared, and intuitively developed ways of incorporating the feedback into their workflow.

1 Introduction

Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background as the developer edits code. Continuous testing provides developers rapid feedback regarding errors that they have inadvertently introduced. This paper experimentally evaluates whether the feedback assists programmers in a programming task.

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 can trade these two sources of wasted time off against one another by testing more or less frequently. Test case selection [8, 12] and prioritization [17, 13] can reduce the time spent waiting for tests. Editing while tests run can also help, but further complicates reproducing and tracking down errors.

Continuous testing reduces both varieties of wasted time by using real-time integration with the development environment to asynchronously run tests against the current version of the code and notify the developer of regression errors. It can be combined with other approaches for further optimization.

Previous work prospectively evaluated continuous testing [14]. Developer behavior was monitored at a fine granularity, including the state of all editor buffers and all on-disk files, and when tests were run. These observations permitted determination of the *ignorance time* between introduction of each regression error (in the developer's editor) and the developer becoming aware of the error (by running the test suite), and the *fix time* between the developer becoming aware of the error and fixing the error. The ignorance time and fix time are correlated: larger ignorance times yield larger fix times. This suggests that reducing ignorance time should reduce fix time; this observation, along with the development history, permit prediction of how much time could have been saved by use of a tool that reduced ignorance time (and indirectly reduced fix time). The previous work 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 paper experimentally evaluates the promise of continuous testing, based on a controlled human experiment with 17 developers, each of whom performed two unrelated development projects. The experimental results indicate that a continuous testing tool has a statistically significant effect on developer success in completing a programming task. This paper presents both the quantitative and the qualitative results of the experiment.

2 Tools

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

Our JUnit extensions (used by both plug-ins) persistently record whether a test has ever succeeded in the past. This permits it to both change 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". The mode line indicates whether the code cannot be compiled; regression errors have been introduced (tests that used to give correct answers no longer do); or 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 first error. Additional clicks navigate to lines corresponding to additional errors and/or to different 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. The Emacs plug-in indicates errors that would occur if the developer were to save all modified buffers, then compiled and tested 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. 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 5.

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

3 Experimental design

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

3.1 Participant demographics

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 this sample, participants were not rewarded in any way. For logistical reasons, data could be obtained from only 22 participants. There are 17 datapoints for the second problem set because 5 participants turned off monitoring or, more frequently, switched to a different development environment. 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 1 gives demographic details regarding the study participants.

Figure 2 summarizes the reasons given by students who chose not to participate; 31 of the non-participants an-

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

swered a questionnaire regarding their decision. 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 generally chose not to 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. Student use of Emacs or of Athena was not a statistically significant predictor of any measure of success (see Section 4.1.1).

Only two factors that we measured predicted participation to a statistically significant degree. First, students who had more Java experience were less likely to participate. Many 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, and neither variety of programming experience predicted any measure of success for participants. 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. Differences between participants and non-participants do not affect our results, because we chose a control group (that was supplied with no experimental tool) from among the participants who had volunteered for the study.

	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 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.

3.2 Task

During the experiment, the participants completed the first two assignments (problem sets) for the course. Participants were treated no differently than other students. 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. (In fact, our results are all the more impressive because a few students in the treatment groups ignored the tools and thus gained no benefit from them.) All students were given a 20-minute tutorial on the tools and had access to webpages explaining their use.

Each problem set provided students with a partial implementation of a simple program. Students were also provided with a complete test suite (see Section 3.2.1). 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. This meant that the code compiled and 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.

The first problem set required implementing four Java classes to complete a poker game. The second problem set 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 3 gives statistics regarding the participant solutions to the problem sets.

3.2.1 Test suites

Students were provided with test suites prepared by the course staff (see Figure 4). Passing these test suites correctly accounted for 75% of the grade for the programming problems in the problem set.

	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. 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.

	participants	non-participants
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 5. Student use of test suites, self-reported. "Participants" includes only participants without continuous testing. Only students who tested regularly throughout development reported test frequencies.

The suites are optimized for grading, not performance, coverage, or usability. 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).

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 5 gives more detail about student use of the provided test suites, ignoring for now participants who used continuous testing.

3.3 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 2.

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 Quantitative results

Data collected from questionnaires, problem set solutions, and by monitoring the participants during development allowed us to evaluate several hypotheses. Section 4.1 reports which variables predicted participant success on the problem sets. Section 4.2 confirms the hypothesis that manually discovering errors more quickly leads to fixing errors more quickly. Section 4.3 investigates at a finer-grained level what effects continuous testing and continuous compilation had on the way that participants progressed through the problem set solution.

4.1 Statistical tests

This paper reports all, and only, effects that are statistically significant at the $p = .05$ level. All of our tests properly accounted 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).

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.

4.1.1 Variables compared

As predictors, we used experimental treatment, problem set, and all quantities of Figures 1 and 9.

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.
- grade, as assigned by TAs. This is available only for PS1, as PS2 had not yet been graded at the time of writing.
- errors. Number of tests that the student submission failed.
- correct. True if the student solution passed all tests.

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, manual runs of the test suite, and clicking the mode line to see errors.

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

4.1.2 Statistical results

Overall, we found relatively few statistically significant effects. We hypothesize that the main reason for this is the well-known large variation among individual programmers. This section lists all the statistically significant effects.

1. Treatment predicts correctness (see Figure 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 than those who were provided with no tool. Even provision of continuous compilation doubled the success rate.
2. Problem set predicts time worked (PS2 took 13.2 hours of programming time on average, compared to 9.4 hours for PS1). 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).
3. For PS1 only, years of Java experience predicted correctness. For the first problem set, participants with previous Java experience had an advantage. By one week later, the others had caught up (or at least were no longer statistically significantly behind).
4. For PS1 participants with correct programs, years of Java IDE experience predicts time worked: those who had more previous Java IDE experience spent less time.

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

Figure 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.

It is worth emphasizing that we found no other statistically significant effects. In particular, none of the predictors of section 4.1.1 predicted number of errors, time worked (except years of Java IDE experience, for PS1 participants with correct programs), grade, or the variables measuring various development events such as number of regression errors.

4.2 Effect of ignorance on fix time

This section reports a correlation between ignorance time and fix time. This suggests that continuous testing could not only help keep code correct and reduce time wasted manually testing, but also reduce the amount of time spent fixing regression errors, thus shortening the overall development process.

For students who used manual testing, we could monitor when regression errors were introduced, when they were discovered through testing, and when they were fixed. Of course, not all times when a previously-passing test is caused to fail are due to inadvertent errors. It may also be that the developer has knowingly, temporarily made a change that breaks the test suite, in expectation of soon making another change that will bring it back to passing. To summarize the model of developer behavior put forward in [14], we infer that developers do not generally run tests when they are in the midst of such an editing cycle. Thus,

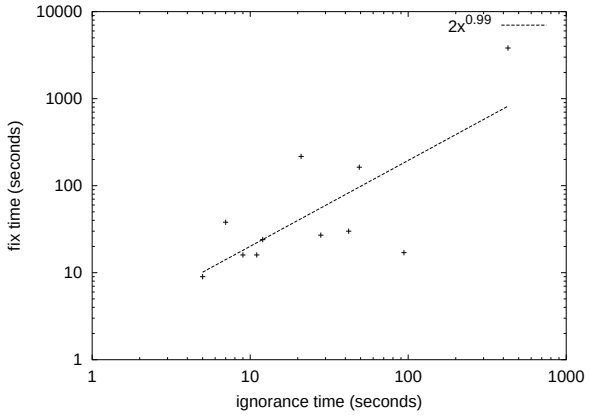

Figure 7. Scatter plot and best-fit line for fix time vs. ignorance time. Axes are log-scale, and the best-fit line is plotted.

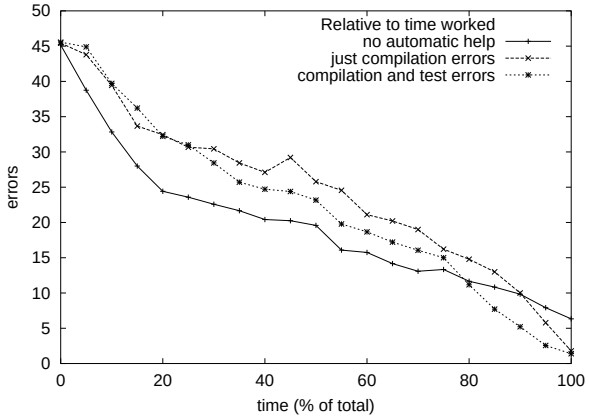

Figure 8. Failing tests remaining after a given proportion of working time has passed. Not all submitted solutions (time = 100%) passed all tests.

only when a test is run at a time when the suite is failing do we count it as a caught regression error.

Figure 7 plots ignorance time and fix time for a single student (the one who was measured to have introduced and fixed the most regression errors). 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. The best fit line indicates that fix time f and ignorance time i are related by the formula $f = 34i^{0.37}$. This helps to confirm the idea that reducing ignorance time by providing more frequent feedback may reduce the time spent fixing regression errors.

Section 4.1.2 reported that use of continuous testing has no statistically significant effect on any variety of time that we measured in our experiment. Given the correlation, and the fact that continuous testing should speed discovery and correction of regression errors, this was not the result we expected. One possible reason for the result is that participants spent only 4% of their overall development time fixing regression errors, so that is the largest amount of time that continuous testing could possibly have saved. This differs from previous projects where the proportion of regression fix time was higher [14].

4.3 Progress patterns

Since students were provided with test suites that represented a significant part of their grade, it is natural to measure their progress over time in terms of the number of tests passed by their solution. Figure 8 graphs the progress of participants from when they started working on the problem set to when they stopped working and turned it in. The graph normalizes across problem sets and participant variation by reporting, on the x axis, percentages of time worked.

For the majority of development, students in the control group appear to be making better progress than those in the other groups. They make quick gains at the beginning, likely by quickly completing simple methods without pausing to test. However, after about 20% of time has passed, the control students begin to slow down, while those with continuous compilation and continuous testing continue on a steadier pace. Then, near the end, students with continuous testing begin to go more quickly, perhaps because the tools help them move more quickly and confidently when debugging the last few persistent bugs. In the end, although the average number of errors is very similar for those with compilation and test error notification, those with test error notification are significantly more likely to have removed all bugs and completed the assignment (see Figure 6).

5 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 5.1 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.

5.1 Multiple choice results

Figure 9 summarizes the multiple-choice questions about experiences with the tools. Participants appear to have felt that continuous compilation provided somewhat more benefit than continuous testing (though the statistical evidence of Section 4.1.2 indicates the opposite), but 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 con-

	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 9. 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").

tinuous testing and continuous compilation participants reported leaving the continuous testing window open. 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).

5.2 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 3.2.1): "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."

5.3 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 mentioned they enjoyed watching unimplemented tests disappear as they coded correctly. 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's "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 also 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 weren't sure they would use continuous testing without a provided test suite, because they were uncomfortable writing their own testing code. 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. It's likely that the tools will become even more useful to students as they learn these concepts.

5.4 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 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.

5.5 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 9 proposes integrating continuous testing with Delta Debugging [18], to provide some of these hints.

5.5.1 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.

Another problem was that when a test caused an infinite loop in the code under test, no output would be provided to the student to this effect. This is identical to the behavior of standard JUnit, but since students did 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 environments, 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.

6 Threats to validity

Our experiment has produced statistically significant results showing that a continuous compilation tool doubles the success rate of developers in creating a correct program, and 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. The level of volunteerism for the study was relatively low; thus, our findings may only apply to users who are open to new development tools and methodologies.

We may have failed to measure some other quantity that predicts success; such an effect could conceivably swamp the predictors that we did measure. The tool may have really had a positive effect, but for a reason that neither we, nor the students we obtained feedback from, understood properly.

There are several factors that suggest that the actual results may be even better than measured by this experiment.

We identified a number of problems with the tools (Section 5.5.1). Many of these could be corrected relatively easily, which would only improve the results for the participants who were provided the tool.

The experiment tested three rather than two different treatments (a control group and two different tools). Had we omitted the continuous compilation group, more effects would be statistically significant, but it would not have been clear whether the benefits came from the compilation or the testing portion of the tool; we have demonstrated that both are significant. Likewise, the experiment involved relatively few participants, so only relatively large differences in effects are statistically significant. There may be other effects that are less important (smaller in magnitude) yet are still statistically significant. 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 rather weak predictor of experience — we are forced to this conclusion by the statistics that are available to us.)

The experiment provided in some ways a worst-case scenario for a continuous testing tool. Continuous testing is most useful when it relieves developers of difficult, lengthy, or easily forgotten tasks. We believe continuous testing to be of most benefit for developers who are performing maintenance (or other tasks that are likely to introduce regression errors) on a codebase whose test suite takes a substantial amount of time to run. By contrast, the participants in our study were performing initial development to a given test suite, the test suites completed in seconds (see Figure 4), and even members of the control and continuous compilation groups could run the test suite with a single keystroke.

Our results demonstrate that even when not matched against a strawman, continuous testing is valuable, and that it does not hinder developer performance — as might be feared by someone who has never used the tool — even in a situation in which it would not be expected to have a large positive impact.

7 Related work

We previously introduced the notion of continuous testing during development to reduce wasted development time [14]. The previous work also presented a model of developer belief that, along with a detailed record of a prior development project, enabled estimation of what the effects would have been, had the developer used a different testing tool in the prior project. A case study with one developer indicated that savings of 10–15% of development time should be possible. This research extends the previous research by implementing the continuous testing tool and performing a controlled experiment in order to measure rather than estimate the effect of the tool, and in order to obtain qualitative feedback regarding developer perceptions of the tool.

Boehm [3] and Baziuk [1] 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, and this paper represents the beginning of an investigation into the veracity of our belief.

Continuous testing can be viewed as a natural extension of modern IDEs (integrated development environments). These IDEs 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. Continuous testing can also be viewed as a natural extension of Extreme Programming [2], 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.

Continuous execution [5], Programming by Example [4, 7], and Editing by Example [10, 9] all provide continuous feedback to developers about the results of their program on one or more inputs as the program changes. Our work abstracts from the entire output to the boolean result of each indvidual test case.

Johnson, et al. [6] 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.

Several other authors use terms similar to our uses of continuous compilation, continuous execution, and continuous testing. Siegel advocates "continuous testing", by which he means frequent synchronous testing during the development process by pairs of developers [15]. Perpetual testing or residual testing [11] (also known as "continuous testing" [16]) 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.

8 Future work

As noted in Section 6, our tools and our experiment could be improved in several ways. However, the experiment cannot be repeated: given the finding that users of continuous testing are better able to complete the assignment successfully before the deadline, it would be unethical to deny it to the control group. It would also be unpopular, since students enjoyed using it. However, future experiments can be performed in new situations, where continuous testing is as yet unproven.

We plan to conduct industrial case studies to provide additional qualitative information regarding continuous testing. We will provide continuous testing tools to a company performing software development for real customers, then observe and interview the developers to learn how they use the tools, their impressions of it, and their suggestions regarding it.

This experiment relied on the fact that the test suites being used could be run very quickly, easily providing real-time notification. 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 [18]. 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.

Second, we are investigating efficient test prioritization for continuous testing [14], to improve performance on suites that contain a large number of small tests.

Third, some individual test cases may take a long time to run; this is particularly true of system or end-to-end tests. An environment that must wait for such tests to complete will not give the impression of instantaneous testing, even if it prioritizes test cases perfectly. We are actively investigating *test factoring* to introduce new test cases that are smaller and faster. Test factoring determines how a system test uses a particular component, then creates unit tests for the component based on that usage. If (only) the component has recently changed, the unit test is just as effective as the system test, but more efficient. The unit tests can be made yet more efficient by eliminating redundancies.

9 Conclusion

We have shown that continuous testing is more than just a good idea by implementing a continuous testing tool and experimentally evaluating its effectiveness at helping students complete programming assignments. The key, statistically significant, result is that participants who used continuous testing were three times more likely to have successfully completed their problem sets than those who did not.

However, there was no statistically significant effect on time worked. Previous work [14] had suggested that by reducing the amount of time spent waiting for tests and fixing regression errors, continuous testing held the promise of speeding software development. This result indicates that students were not burdened by the tools. We believe the lack of improvement resulted from the fact that students were working not just against a functional specification, but also against a deadline. Many students likely budgeted a certain amount of time to the problem set before the deadline, and turned in whatever they had when time ran out. It's impossible to know for unsuccessful students how much longer it would have taken for them to develop a complete solution.

Furthermore, students liked continuous testing and believed it helped them; their criticisms were minor by comparison to their praise. 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 perform best: for initial development in a situation in which tests are easy to run and complete quickly. It is a substantial success of continuous testing that despite these obstacles, users of the tool were still significantly helped. This encourages us to extend our research on continuous testing, so it can be used in additional academic and industrial settings.

References

[1] W. Baziuk. BNR/NORTEL: Path to improve product quality, reliability, and customer satisfaction. In *ISSRE*, Oct. 1995.

[2] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

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

[4] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.

[5] P. Henderson and M. Weiser. Continuous execution: The VisiProg environment. In *ICSE*, pages 68–74, Aug. 1985.

[6] P. M. Johnson, H. Kou, J. M. Agustin, C. Chan, C. A. Moore, J. Miglani, S. Zhen, and W. E. Doane. Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined. In *ICSE*, pages 641–646, May 2003.

[7] T. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, Stanford, CA, June 2000.

[8] H. K. N. Leung and L. White. Insights into regression testing. In *ICSM*, pages 60–69, Oct. 1989.

[9] R. 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.

[10] R. P. Nix. Editing by example. *ACM Trans. Prog. Lang. Syst.*, 7(4):600–621, Oct. 1985.

[11] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *ICSE*, pages 277–284, May 1999.

[12] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE TSE*, 22(8):529–551, Aug. 1996.

[13] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE TSE*, 27(10):929–948, Oct. 2001.

[14] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, Nov. 2003.

[15] S. Siegel. *Object-Oriented Software Testing: A Hierarchical Approach*. John Wiley & Sons, 1996.

[16] M. L. Soffa. Continuous testing. Personal communication, Feb. 2003.

[17] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *ISSRE*, pages 264–274, Nov. 1997.

[18] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE*, pages 253–267, Sept. 1999.