An experimental evaluation of continuous testing during development

David Saff, Michael D. Ernst
MIT CSAIL
ISSTA 2004
Overview

• Continuous testing runs tests in the background to provide feedback as developers code.
• A controlled human experiment revealed that students with continuous testing:
  – Were significantly more likely to complete a class assignment
  – Took no longer to finish
  – Would recommend the tool to others
Outline

Introduction

• Experimental Design
• Quantitative Results
• Qualitative Results
• Conclusion
Continuous Testing

- Continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background as the developer edits code.
- Developer no longer thinks about what to test when.
Continuous testing: inspired by continuous compilation

- Continuous compilation, as in Eclipse, notifies the developer quickly when a **syntactic error** is introduced:

  ![Syntactic error example]

- Continuous testing notifies the developer quickly when a **semantic error** is introduced:

  ![Semantic error example]
Previous work

• Single-developer case study [ISSRE 03]
• Upgrades of *existing software* with regression test suites.
• Test suites took *minutes*: test prioritization needed for best results
• Focus on reduced development time (10-15%) through quick discovery of *regression errors*
This work

• Controlled human experiment: 22 students
• Each subject performed two unrelated development tasks.
• *Initial development*: regressions not a factor, test suite provided in advance.
• Test suites took *seconds*: prioritization unnecessary
• Focus on productivity effects of *automatic testing*
• “What happens when the computer thinks about testing for us?”
Experimental Questions

1. Does continuous testing improve productivity?
   - Yes

2. Are productivity benefits due to continuous testing, or:
   a. Continuous compilation - Yes
   b. Frequent testing - Yes
   c. Demographics

3. Does asynchronous feedback distract users?
   - No
Outline

• Introduction
• Experimental Design
• Quantitative Results
• Qualitative Results
• Conclusion
Participants

- Students in MIT’s 6.170 Laboratory in Software Engineering class.

107 total students

34 volunteers

14.5 worked outside monitored environment

73 non-volunteers

19.5 monitored

25% (5.5) no tools

25% (5) compilation notification only

50% (9) compilation and test error notification
## Experience

<table>
<thead>
<tr>
<th>Years...</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>...programming</td>
<td>2.8</td>
</tr>
<tr>
<td>...using Emacs</td>
<td>1.3</td>
</tr>
<tr>
<td>...using Java</td>
<td>0.4</td>
</tr>
<tr>
<td>...using IDE</td>
<td>0.2</td>
</tr>
</tbody>
</table>

- Relatively inexperienced group of participants
Programming Tasks

- Participants completed (PS1) a poker game and (PS2) a graphing polynomial calculator.
- Test suites provided by course staff.
- To compile and run tests took < 5 secs.
- The provided code failed most tests.

<table>
<thead>
<tr>
<th></th>
<th>PS1</th>
<th>PS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>participants</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td>written lines of code</td>
<td>150</td>
<td>135</td>
</tr>
<tr>
<td>written methods</td>
<td>18</td>
<td>31</td>
</tr>
<tr>
<td>time worked (hours)</td>
<td>9.4</td>
<td>13.2</td>
</tr>
<tr>
<td>tests</td>
<td>49</td>
<td>82</td>
</tr>
</tbody>
</table>
Emacs plug-in

• Compile and test
  – on file save
  – after 15-second pause
• Display results in modeline:
  – “Compilation Errors”
  – “Unimplemented Tests: 45”
  – “Regressions: 2”
• Clicking on modeline brings up stack backtrace of indicated errors.
Modeline screenshots

```java
public CardValue getValue() {
    throw new RuntimeException("Method getValue is not yet implemented!");
}
```

Middle-click "Unimpl-tests:45" in mode line to see errors.

```java
public CardValue getValue() {
    throw new RuntimeException("Method getValue is not yet implemented!");
}
```

Middle-click "Compile-errors" in mode line to see compilation errors.

```java
public CardValue getValue() {
    return value;
}
```

Middle-click "Regressions:1" in mode line to see errors.
Sources of data

• Quantitative:
  – Monitored development history
  – Submitted problem set solutions
  – Grades

• Qualitative:
  – Questionnaire from all students
  – E-mail feedback from some students
  – Interviews and e-mail from staff
Outline

• Introduction
• Experimental Design
Quantitative Results
• Qualitative Results
• Conclusion
Productivity measures

- *time worked*: Time spent editing source files.
- *grade*: On each individual problem set.
- *correct program*: True if the student solution passed all tests.
- *failed tests*: Number of tests that the student submission failed.
# Treatment predicts correctness (Question 1)

<table>
<thead>
<tr>
<th>Treatment</th>
<th>N</th>
<th>Correct programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>No tool</td>
<td>11</td>
<td>27%</td>
</tr>
<tr>
<td>Continuous compilation</td>
<td>10</td>
<td>50%</td>
</tr>
<tr>
<td>Continuous testing</td>
<td>18</td>
<td>78%</td>
</tr>
</tbody>
</table>

\[ p < .03 \]
Can other factors explain this? (Question 2)

- Continuous testing: 78% vs. 27% success
- Continuous compilation: no
  - Just continuous compilation: 50% success
- Frequent testing: no
  - Just frequent manual testing: 33% success
- Easy testing: no
  - All students could run tests with a keypress
- Demographics: no
  - No significant differences between groups
No significant effect on other productivity measures

<table>
<thead>
<tr>
<th>Treatment</th>
<th>N</th>
<th>Time worked</th>
<th>Failed tests</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>No tool</td>
<td>11</td>
<td>10.1 hrs</td>
<td>7.6</td>
<td>79%</td>
</tr>
<tr>
<td>Cont. comp.</td>
<td>10</td>
<td>10.6 hrs</td>
<td>4.1</td>
<td>83%</td>
</tr>
<tr>
<td>Cont. testing</td>
<td>18</td>
<td>10.7 hrs</td>
<td>2.9</td>
<td>85%</td>
</tr>
</tbody>
</table>

*only for correct programs*
Other effects seen

• Students spent longer on PS2 than PS1.
• On PS1 only, Java experience improved correctness and grade.
• For PS1 participants with correct programs, previous experience with a Java IDE reduced time worked.
• Only effects seen at the $p < .05$ level.
Outline

• Introduction
• Experimental Design
• Quantitative Results
• Qualitative Results
• Conclusion
Do developers enjoy the tool?  
(Question 3)

<table>
<thead>
<tr>
<th>(scale: +3 = strongly agree, -3 = strongly disagree)</th>
<th>Continuous compilation</th>
<th>Continuous testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>The reported errors often surprised me</td>
<td>1.0</td>
<td>0.7</td>
</tr>
<tr>
<td>I discovered problems more quickly</td>
<td>2.0</td>
<td>0.9</td>
</tr>
<tr>
<td>I completed the assignment faster</td>
<td>1.5</td>
<td>0.6</td>
</tr>
<tr>
<td>I enjoyed using the tool</td>
<td>1.5</td>
<td>0.6</td>
</tr>
<tr>
<td>The tool changed the way I worked</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>I was not distracted by the tool</td>
<td>0.5</td>
<td>0.6</td>
</tr>
</tbody>
</table>
Did continuous testing win over users?

<table>
<thead>
<tr>
<th>I would use the tool...</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>...for the rest of the class</td>
<td>94%</td>
</tr>
<tr>
<td>...for my own programming</td>
<td>80%</td>
</tr>
<tr>
<td>I would recommend the tool to others</td>
<td>90%</td>
</tr>
</tbody>
</table>
Participant comments, part 1

• “I got a small part of my code working before moving on to the next section, rather than trying to debug everything at the end.”

• “It was easier to see my errors when they were only with one method at a time.”

• “Once I finally figured out how it worked, I got even lazier and never manually ran the test cases myself anymore.”
Participant comments, part 2

• “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.”

• “I suppose that, if I did not already have a set way of doing my coding, continuous testing could have been more useful.”
Outline

• Introduction
• Experimental Design
• Quantitative Results
• Qualitative Results

Conclusion
Threats to validity

• Participants were undergraduates
  – 2.8 years programming experience, 0.4 with Java
  – Standard practice for controlled human experiments in software engineering
  – Can’t predict the effect of more experience

• Tests existed a priori

• Small programs

• Some problems with provided tools
  – scalability
  – user confusion
Future Work

• Case studies in with larger projects
  – We’ve built an industrial-strength implementation in Eclipse, including test prioritization and selection

• Extend to bigger test suites:
  – Help developers understand failures: Integrate with Delta Debugging (Zeller)
  – Run the right tests: Better test prioritization
  – Run the right parts of tests: Test factoring: making unit tests from system tests [PASTE 2004]
Conclusion

• Continuous testing has a significant effect (78% vs. 27%) on developer success in completing a programming task
  – without affecting time worked
• Most developers enjoy using continuous testing, and find it helpful
• Download Eclipse plug-in for continuous testing
  – Google “continuous testing”
The End

• Thanks to:
  – 6.170 staff
  – Participants
  – ISSTA reviewers
Pedagogical usefulness

• Several students mentioned that continuous testing was most useful when:
  – Code was well-modularized
  – Specs and tests were written before development

• These are important goals of the class
Introduction: Previous Work: Findings

- Finding 2: Continuous testing is more effective at reducing wasted time than:
  - changing test frequency
  - reordering tests

- Finding 3: Continuous testing reduces total development time 10 to 15%
Reasons cited for not participating

Students could choose as many reasons as they wished.

<table>
<thead>
<tr>
<th>Reason</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don't use Emacs</td>
<td>45%</td>
</tr>
<tr>
<td>Don't use Athena</td>
<td>29%</td>
</tr>
<tr>
<td>Didn't want the hassle</td>
<td>60%</td>
</tr>
<tr>
<td>Feared work would be hindered</td>
<td>44%</td>
</tr>
<tr>
<td>Privacy concerns</td>
<td>7%</td>
</tr>
</tbody>
</table>

Other IDE’s cited, in order of popularity:

- Eclipse
- text editors (vi, pico, EditPlus2)
- Sun ONE Studio
- JBuilder
Variables that predicted participation

- Students with more Java experience were *less* likely to participate
  - already had work habits they didn’t want to change
- Students with more experience compiling programs in Emacs were *more* likely to participate
- We used a control group *within* the set of voluntary participants—results were not skewed.
Demographics: Experience (1)

<table>
<thead>
<tr>
<th>Years…</th>
<th>Mean</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>...programming</td>
<td>2.8</td>
<td>0.5</td>
<td>14.0</td>
</tr>
<tr>
<td>...using Java</td>
<td>0.4</td>
<td>0.0</td>
<td>2.0</td>
</tr>
<tr>
<td>...using Emacs</td>
<td>1.3</td>
<td>0.0</td>
<td>5.0</td>
</tr>
<tr>
<td>...using IDE</td>
<td>0.2</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Problem Sets

- Participants completed several classes in a skeleton implementation of (PS1) a poker game and (PS2) a graphing polynomial calculator.

<table>
<thead>
<tr>
<th></th>
<th>PS1</th>
<th>PS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>participants</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td>total lines of code</td>
<td>882</td>
<td>804</td>
</tr>
<tr>
<td>skeleton lines of code</td>
<td>732</td>
<td>669</td>
</tr>
<tr>
<td>written lines of code</td>
<td>150</td>
<td>135</td>
</tr>
<tr>
<td>written classes</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>written methods</td>
<td>18</td>
<td>31</td>
</tr>
<tr>
<td>time worked (hours)</td>
<td>9.4</td>
<td>13.2</td>
</tr>
</tbody>
</table>
Test Suites

- Students were provided with test suites written by course staff.
- Passing tests correctly was 75% of grade.

<table>
<thead>
<tr>
<th></th>
<th>PS1</th>
<th>PS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>tests</td>
<td>49</td>
<td>82</td>
</tr>
<tr>
<td>initial failing tests</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>lines of code</td>
<td>3299</td>
<td>1444</td>
</tr>
<tr>
<td>running time (secs)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>compilation time (secs)</td>
<td>1.4</td>
<td>1.4</td>
</tr>
</tbody>
</table>
JUnit wrapper

- Reorder tests
- Time individual tests
- Remember results
- Output failures immediately
- Distinguish regressions from unimplemented tests
- Reorder and filter result text

Test Suite → Wrapper → Results

JUnit

Saff, Ernst: Continuous Testing
Demographics: Experience (2)

Usual environment: Unix 29%, Windows 38%, both 33%
More variables: where students spent their time

- All time measurements used *time worked*, at a five-minute resolution:

  - Some selected time measurements:
    - Total time worked
    - Ignorance time
      - between introducing an error and becoming aware of it
    - Fixing
      - between becoming aware of an error and fixing it

  x = source edit
Ignorance and fix time

- Ignorance time and fix time are correlated, confirming previous result.
- Chart shown for the single participant with the most regression errors
Errors over time

- Participants with no tools make progress faster at the beginning, then taper off; may never complete.
- Participants with automatic tools make steadier progress.
Previous Work

• Monitored two single-developer software projects
• A model of developer behavior interpreted results and predicted the effect of changes on *wasted time*:
  – Time waiting for tests to complete
  – Extra time tracking down and fixing regression errors
Previous Work: Findings

- Delays in notification about regression errors correlate with delays in fixing these errors.
- Therefore, quicker notification should lead to quicker fixes.
- Predicted improvement: 10-15%
Other comments

• Head TA: “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.”

• “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.”

• Some didn’t understand the modeline, or how shadowing worked.
Test Suites

- Students were provided with test suites written by course staff.
- Passing tests correctly was 75% of grade.

<table>
<thead>
<tr>
<th></th>
<th>PS1</th>
<th>PS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>tests</td>
<td>49</td>
<td>82</td>
</tr>
<tr>
<td>initial failing tests</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>running time (secs)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>compilation time (secs)</td>
<td>1.4</td>
<td>1.4</td>
</tr>
</tbody>
</table>
Suggestions for improvement

• More flexibility in configuration
• More information about failures
• Smarter timing of feedback
• Implementation issues
  – JUnit wrapper filtered JUnit output, which was confusing.
  – Infinite loops led to no output.
  – Irreproducible failures to run.
  – Performance not acceptable on all machines.
## Test Suites: Usage

<table>
<thead>
<tr>
<th></th>
<th>Participants</th>
<th>Non-participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>waited until end to test</td>
<td>31%</td>
<td>51%</td>
</tr>
<tr>
<td>tested regularly throughout</td>
<td>69%</td>
<td>49%</td>
</tr>
</tbody>
</table>

Test frequency (minutes) for those who tested regularly

<table>
<thead>
<tr>
<th></th>
<th>Participants</th>
<th>Non-participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>min</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>max</td>
<td>40</td>
<td>60</td>
</tr>
</tbody>
</table>
Shadow directory

- The developer’s code directory is “shadowed” in a hidden directory.
- Shadow directory has state as it would be if developer saved and compiled right now.
- Compilation and test results are filtered to appear as if they occurred in the developer’s code directory.
Monitoring

• Developers who agree to the study have a monitoring plug-in installed at the same time as the continuous testing plug-in.

• Sent to a central server:
  – Changes to the source in Emacs (saved or unsaved)
  – Changes to the source on the file system
  – Manual test runs
  – Emacs session stops/starts
Error buffer screenshot

```java
java -cp /mit/6.170/delta-capture/ctrunner.jar:/afs/athena.mit.edu/user/s/a/saff/.delta-ca
1) NEW REGRESSION ERROR: testValue(ps1.playingcards.CardTest)
   junit.framework.AssertionFailedError: expected:<Jack> but was:<null>
   at ps1.playingcards.CardTest.testValue(CardTest.java:56)
   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
   at sun.reflect.NativeMethodAccessorImpl.invoke()
   at sun.reflect.DelegatingMethodAccessorImpl.invoke()
   at edu.mit.lcs.pag.ct.junit.CtTestRunner.start()
   at edu.mit.lcs.pag.ct.junit.CtTestRunner.run()
   at edu.mit.lcs.pag.ct.junit.CtTestRunner.run()
   at edu.mit.lcs.pag.ct.junit.CtTestRunner.main()

=== UNIMPLEMENTED TESTS ===

1) testCompareTo(ps1.playingcards.CardTest)
   junit.framework.AssertionFailedError: Should raise a NullPointerException: java.lang.Runn
   at ps1.playingcards.CardTest.testCompareTo(CardTest.java:75)
   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
   at sun.reflect.NativeMethodAccessorImpl.invoke()
   at sun.reflect.DelegatingMethodAccessorImpl.invoke()
   at edu.mit.lcs.pag.ct.junit.CtTestRunner.start()
   at edu.mit.lcs.pag.ct.junit.CtTestRunner.run()
   at edu.mit.lcs.pag.ct.junit.CtTestRunner.run()
   at edu.mit.lcs.pag.ct.junit.CtTestRunner.run()

2) testEquals(ps1.playingcards.CardTest)
   java.lang.RuntimeException: Method equals is not yet implemented!
   at ps1.playingcards.Card.equals(Card.java:137)
   at ps1.playingcards.CardTest.testEquals(CardTest.java:118)
   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```
• Preview of results:
  – Continuous testing has a significant effect on success *completing* a task.
  – This effect cannot be attributed to other factors.
  – Developers enjoy using continuous testing, and find it helpful, not distracting.