An experimental evaluation of continuous testing during development

David Saff, Michael D. Ernst

MIT CSAIL

ISSTA 2004

Overview

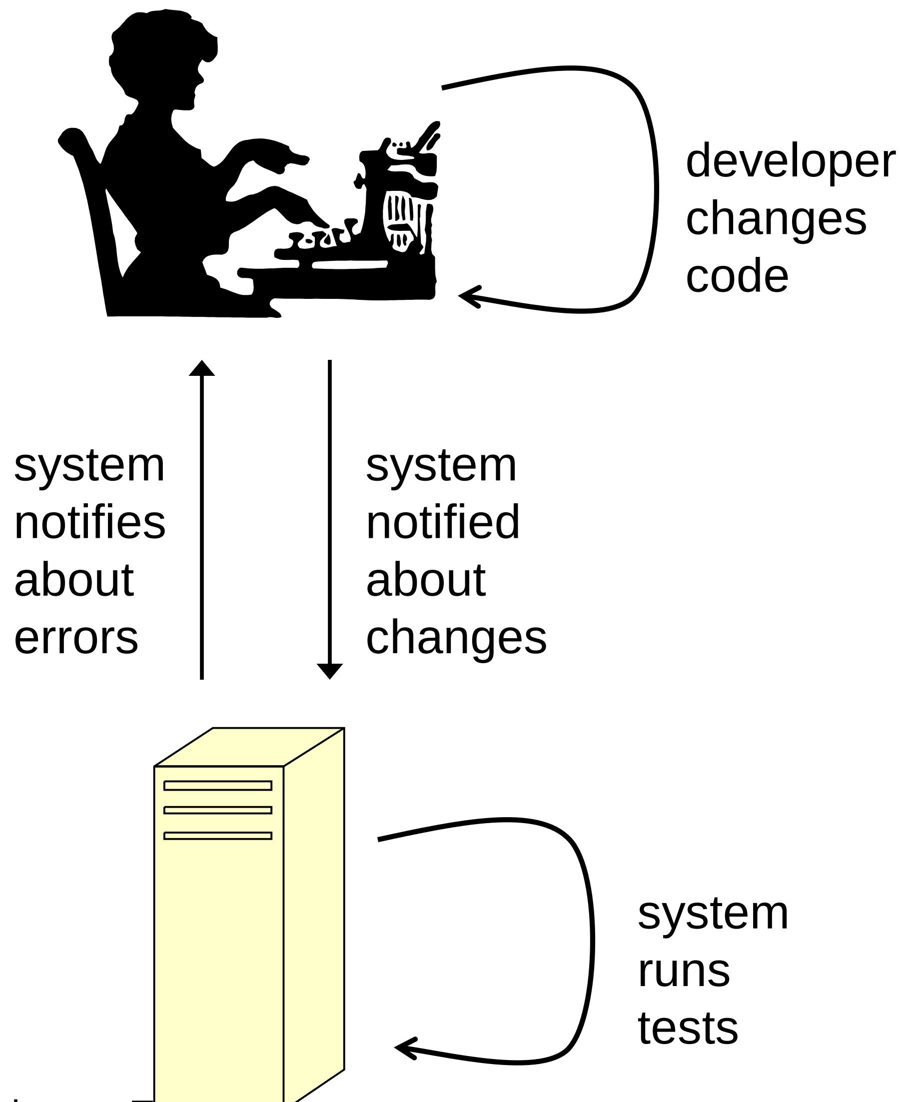
- Continuous testing runs tests in the background to provide feedback as developers code.
- A controlled human experiment revealed that students with continuous testing:
  - Were significantly more likely to complete a class assignment
  - Took no longer to finish
  - Would recommend the tool to others

Outline

- Introduction
- Experimental Design
- Quantitative Results
- Qualitative Results
- Conclusion

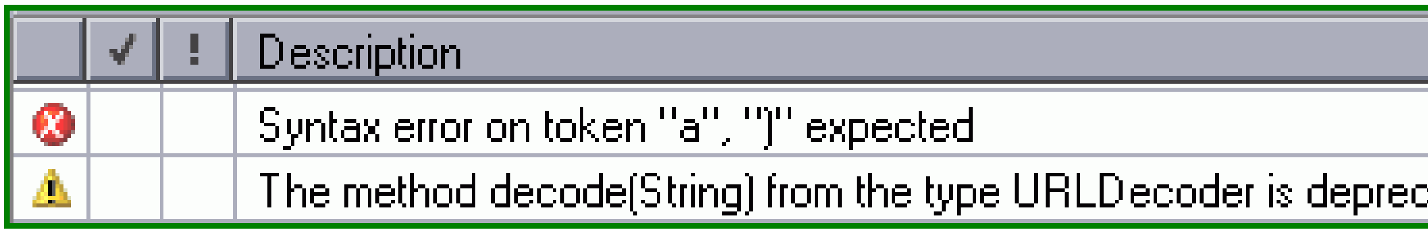
Continuous Testing

- 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.

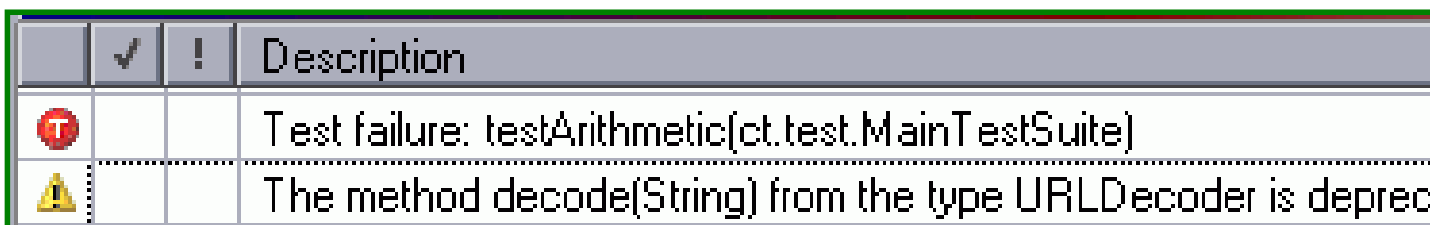


Continuous testing: inspired by continuous compilation

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



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



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

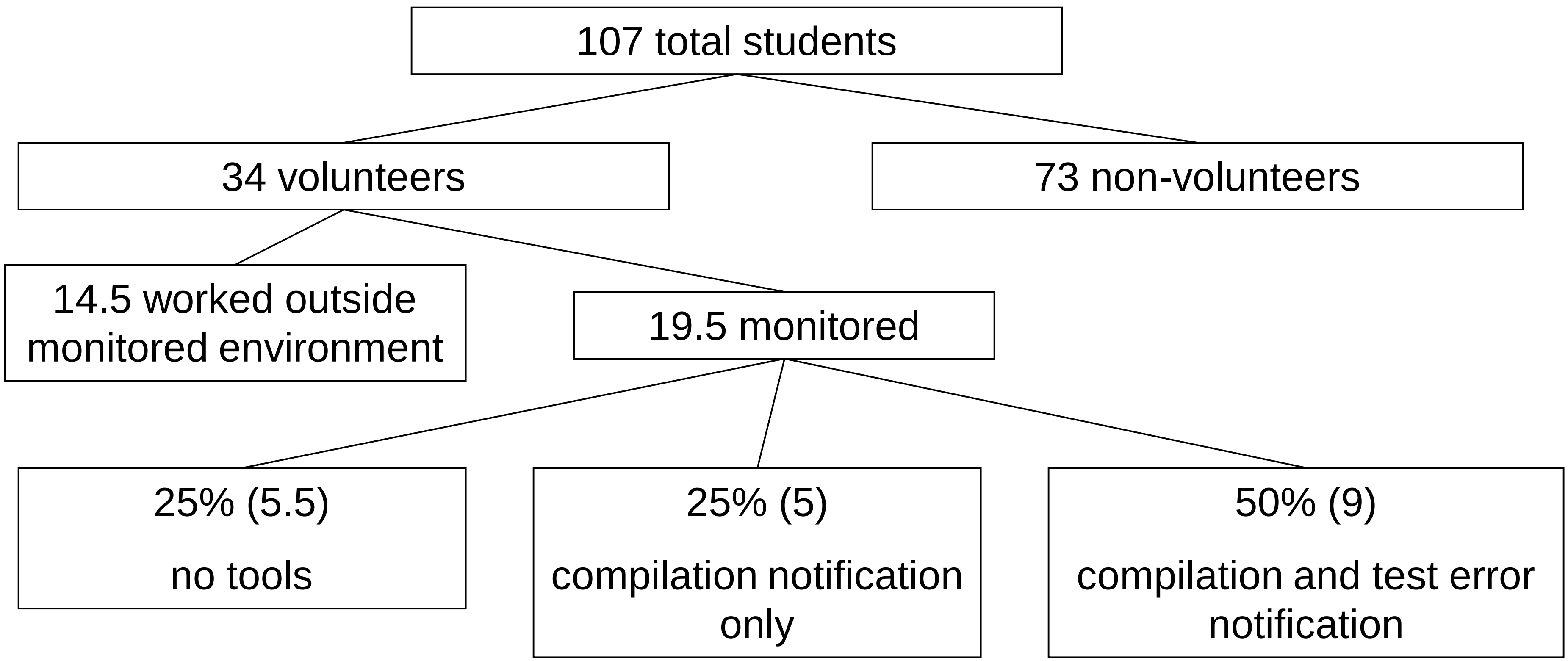
1. Does continuous testing improve productivity? **Yes**
2. Are productivity benefits due to continuous testing, or: **Yes**
   a. Continuous compilation
   b. Frequent testing
   c. Demographics
3. Does asynchronous feedback distract users? **No**

Outline

- Introduction
- Experimental Design
- Quantitative Results
- Qualitative Results
- Conclusion

Participants

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



Experience

Years…	Mean
…programming	2.8
…using Emacs	1.3
…using Java	0.4
…using IDE	0.2

- 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.

	PS1	PS2
participants	22	17
written lines of code	150	135
written methods	18	31
time worked (hours)	9.4	13.2
tests	49	82

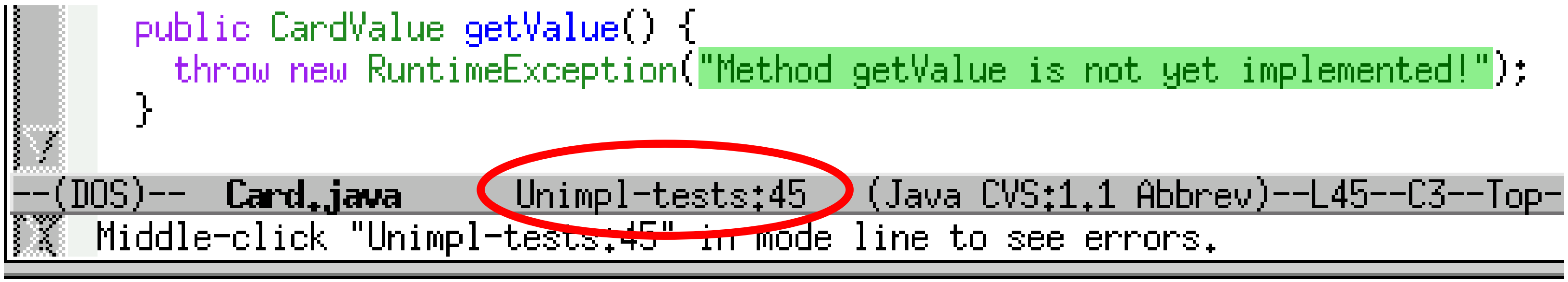
Emacs plug-in

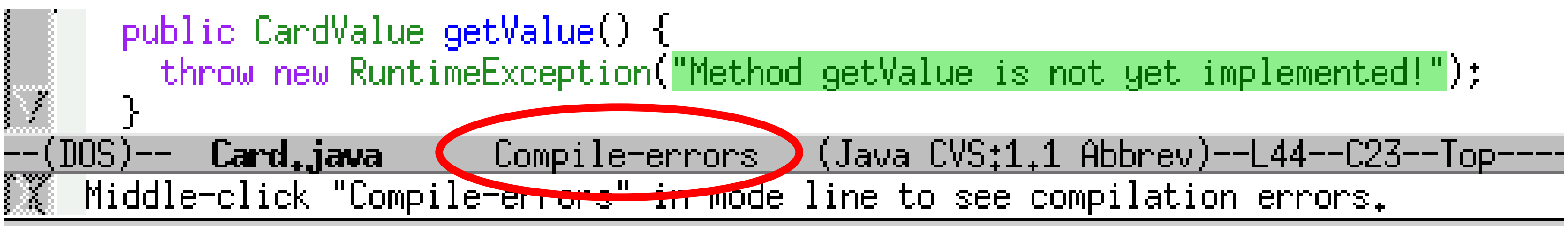
- Compile and test
  - on file save
  - after 15-second pause
- Display results in modeline:
  - "Compilation Errors"
  - "Unimplemented Tests: 45"
  - "**Regressions: 2**"
- Clicking on modeline brings up stack backtrace of indicated errors.

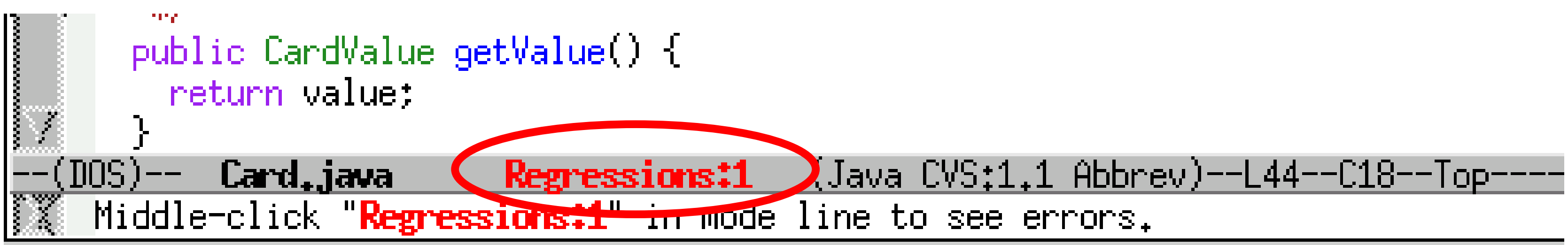
Never passed

Once passed, Now failing

Modeline screenshots







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)

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

$p < .03$

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

Treatment	N	Time worked	Failed tests	Grade
No tool	11	10.1 hrs	7.6	79%
Cont. comp.	10	10.6 hrs	4.1	83%
Cont. testing	18	10.7 hrs	2.9	85%

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)

(scale: +3 = strongly agree, -3 = strongly disagree)	Continuous compilation	Continuous testing
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 enjoyed using the tool	1.5	0.6
The tool changed the way I worked	1.7	1.7
I was not distracted by the tool	0.5	0.6

Did continuous testing win over users?

I would use the tool…	Yes
…for the rest of the class	94%
…for my own programming	80%
I would recommend the tool to others	90%

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

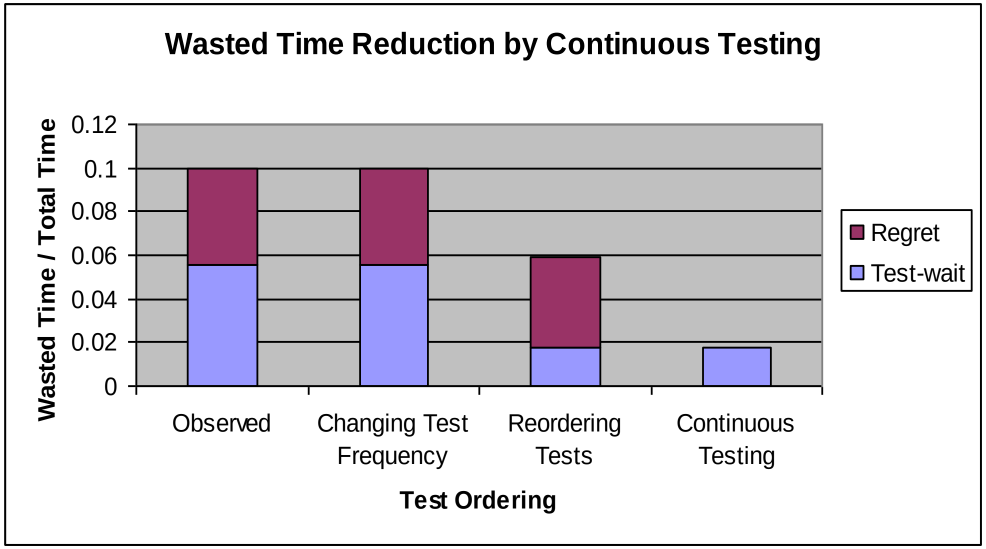
- Thanks to:
  - 6.170 staff
  - Participants
  - ISSTA reviewers

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.

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%

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)

Years…	Mean	Min	Max
…programming	2.8	0.5	14.0
…using Java	0.4	0.0	2.0
…using Emacs	1.3	0.0	5.0
… using IDE	0.2	0.0	1.0

Problem Sets

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

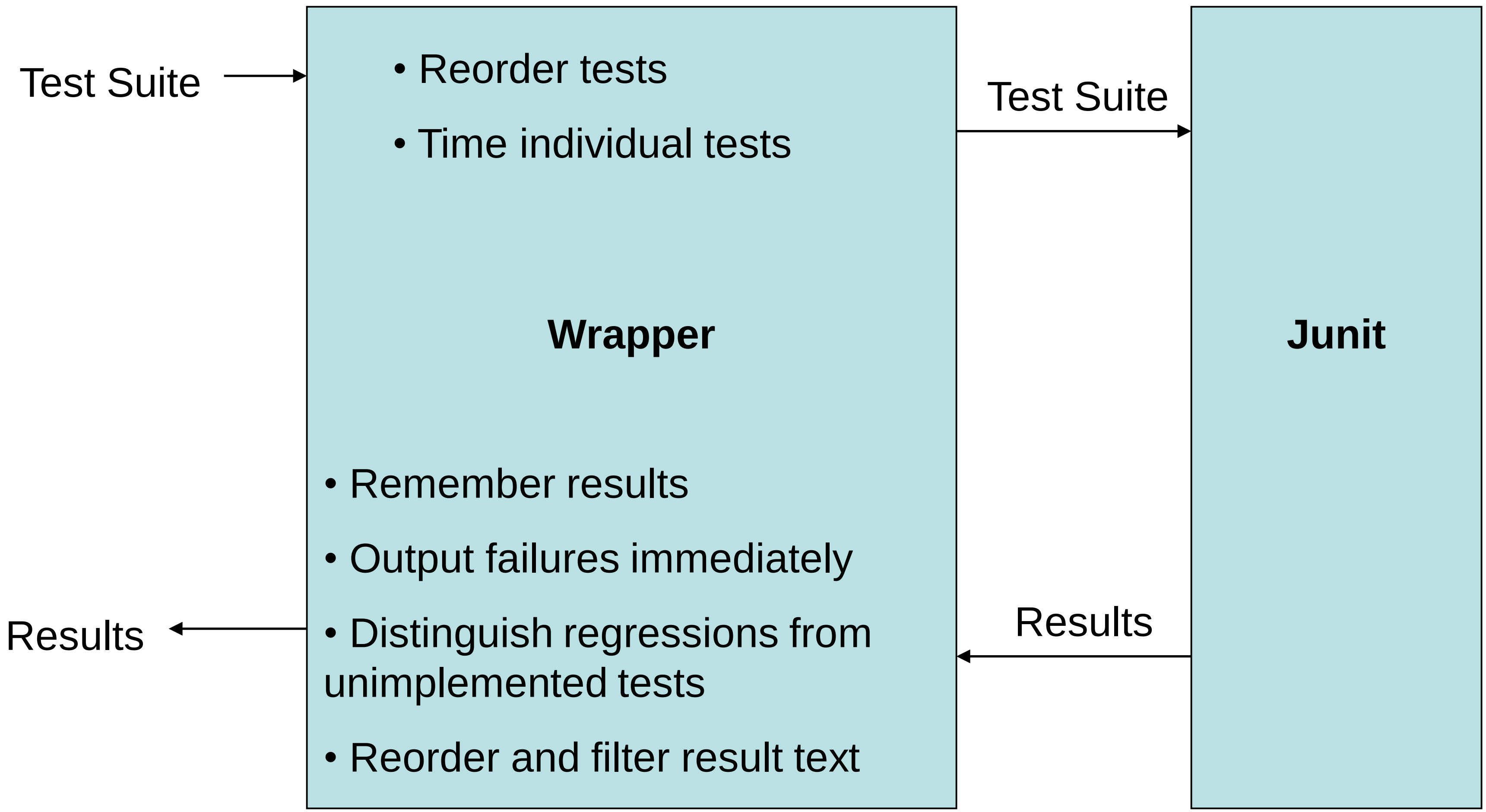
	PS1	PS2
participants	22	17
total lines of code	882	804
skeleton lines of code	732	669
written lines of code	150	135
written classes	4	2
written methods	18	31
time worked (hours)	9.4	13.2

Test Suites

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

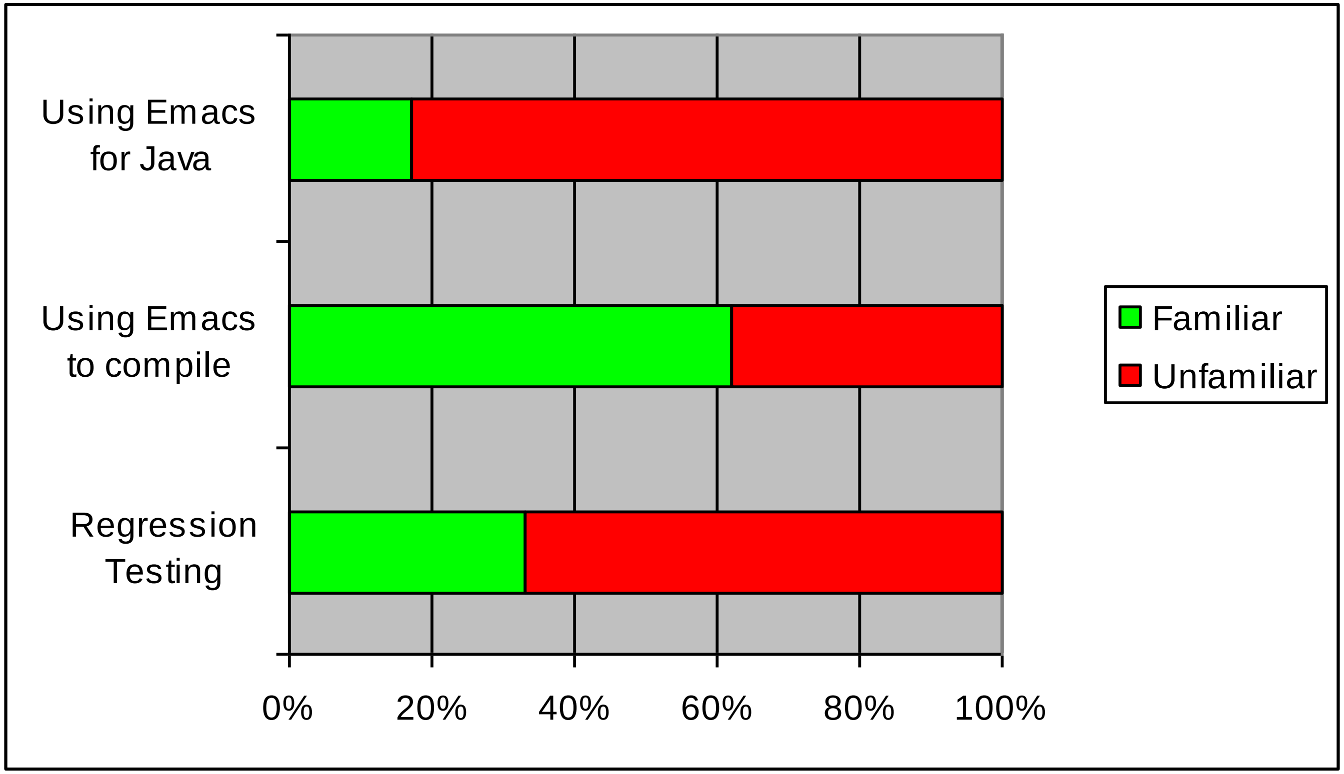
	PS1	PS2
tests	49	82
initial failing tests	45	46
lines of code	3299	1444
running time (secs)	3	2
compilation time (secs)	1.4	1.4

JUnit wrapper



Demographics: Experience (2)

Usual environment: Unix 29%, Windows 38%, both 33%



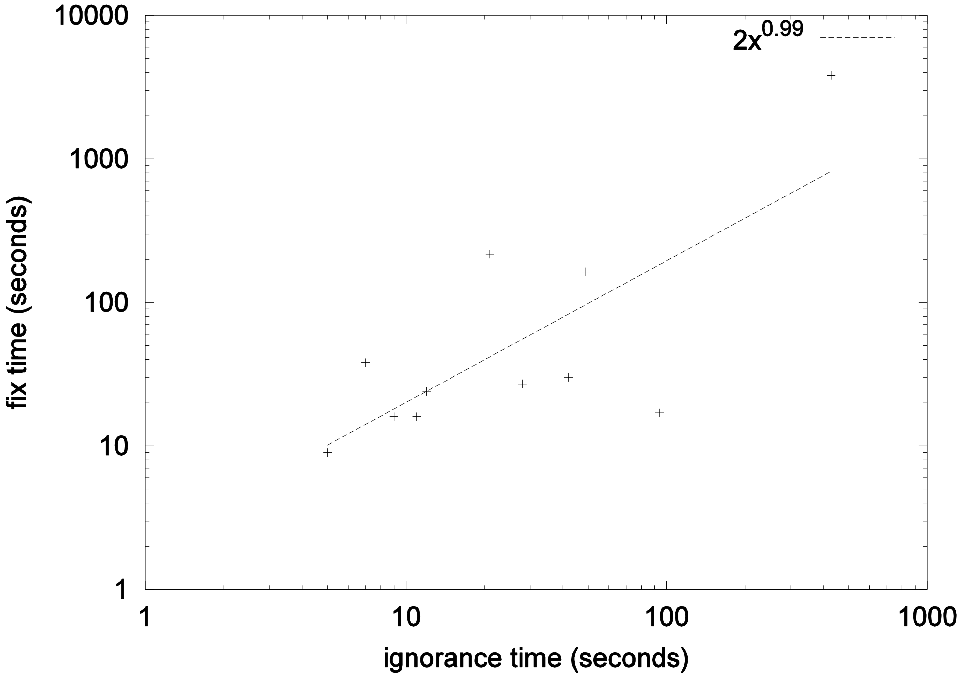
More variables: where students spent their time

- 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

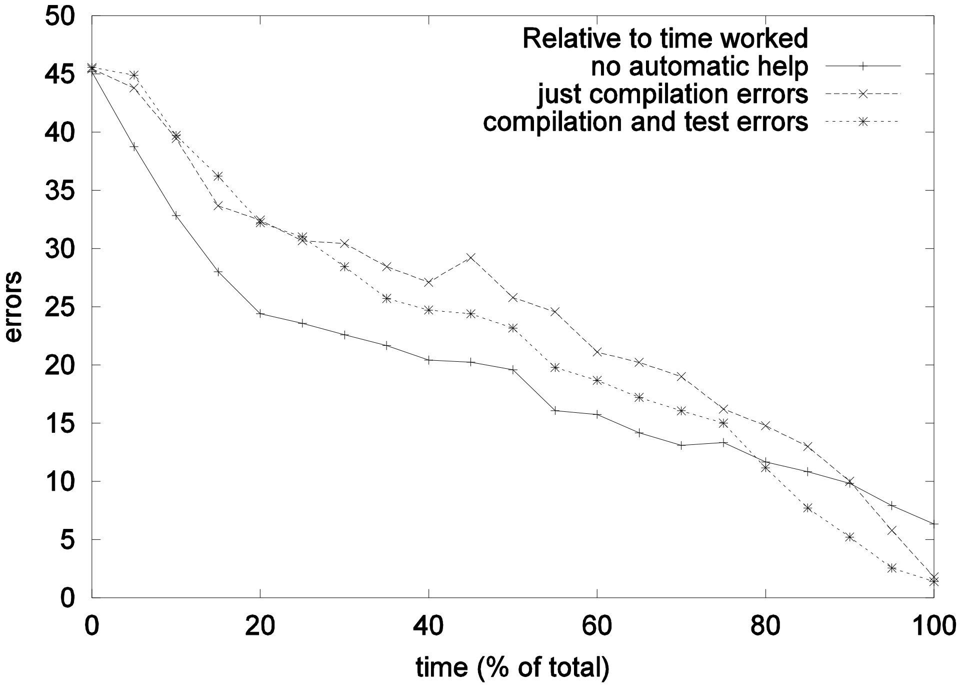
Ignorance and fix time

- 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.

	PS1	PS2
tests	49	82
initial failing tests	45	46
running time (secs)	3	2
compilation time (secs)	1.4	1.4

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

	Participants	Non-participants
waited until end to test	31%	51%
tested regularly throughout	69%	49%

Test frequency (minutes) for those who tested regularly		
mean	20	18
min	7	3
max	40	60

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 -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.Runt
 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.main()

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.