Test Factoring:  
Focusing test suites on the task at hand

David Saff, MIT  
ASE 2005
The problem: large, general system tests

How can I get: Quicker feedback? Less wasted time?

[Saff, Ernst, ISSRE 2003]
The problem: large, general system tests

My test suite

Test selection
The problem: large, general system tests

My test suite

Test selection

Test prioritization
The problem:
large, general system tests

My test suite

Test selection

Test prioritization

Test factoring
Test factoring

• Input: large, general system tests
• Output: small, focused unit tests
• Work with Shay Artzi, Jeff Perkins, and Michael D. Ernst
A factored test…

- exercises less code than system test
- should be faster if a system test is slow
- can eliminate dependence on expensive resources or human interaction
- isolates bugs in subsystems
- provides new opportunities for prioritization and selection
Test Factoring

• What?
  – Breaking up a system test

• How?
  – Automatically creating mock objects

• When?
  – Integrating test factoring into development

• What next?
  – Results, evaluation, and challenges
There’s more than one way to factor a test!

Basic strategy:
- *Capture* a subset of behavior beforehand.
- *Replay* that behavior at test time.
System Test

Provided

Tested Code

Checked

PayrollCalculator
- Fast
- Is changing

Environment

Database Server
- Expensive
- Not changing

Fast

Is changing

Expensive

Not changing

Daniel Saff
Introduce Mock:

- simulate *part* of the functionality of the original environment
- validate the unit’s interaction with the environment

[Saff, Ernst, PASTE 2004]
Test Factoring

• **What?**
  – Breaking up a system test

• **How?**
  – Automatically creating mock objects

• **When?**
  – Integrating test factoring into development

• **What next?**
  – Results, evaluation, and challenges
How? Automating *Introduce Mock*

- **PayrollCalculator**
  - `calculatePayroll()`
  - `getResult()`
  - `getResult()`

- **ResultSet**
  - `getResult()`
  - `getResult()`

- **Database**
  - `addResult(String)`
  - `addResult(String)`
  - `addResult(String)`

- **Tested Code Environment**

---

David Saff
Interfacing: separate type hierarchy from inheritance hierarchy

IPayrollCalculator

getResult() getResult() getResult() getResult()

IDatabase

addResult(String) addResult(String) addResult(String)

IResultSet

calculatePayroll()

addResultsTo(IResultSet)
Capturing: insert recording decorators where capturing must happen

calculatePayroll()

IPayrollCalculator

PayrollCalculator

IResultSet

Callback ResultSet

getResult() getResult() getResult() getResult() getResult() getResult() getResult() getResult() getResult() getResult()

Tested Code

Environment

15
Replay: simulate environment’s behavior

calculatePayroll()

IPayrollCalculator

PayrollCalculator

getResult()
getResult()
getResult()

addResult(String)
addResult(String)

IResultSet

ResultSet

addResultsTo(IResultSet)

verified

replayed

IResultSet

Replaying Database

IDatabase

Database

addResult(String)
addResult(String)
addResult(String)

Tested Code

Environment

David Saff
Test Factoring

• What?
  – Breaking up a system test

• How?
  – Automatically creating mock objects

• When?
  – Integrating test factoring into development

• What next?
  – Results, evaluation, and challenges
When? Test factoring life cycle:

- **Slow system tests**
  - **Capture**
  - **Transcript**
  - **Replay**

- **Fast unit tests**

- **Developer changes tested unit**

- **Run factored tests**
  - Success
  - Failure
  - Replay exception

- **Run system tests for replay exceptions**

David Saff
Time saved:

- Slow system tests
- Run factored tests
- Run system tests for replay exceptions
Time saved:

- Slow system tests
- Factored tests

Time until first error

Time to complete tests
Test Factoring

• What?
  – Breaking up a system test

• How?
  – Automatically creating mock objects

• When?
  – Integrating test factoring into development

• What next?
  – Results, evaluation, and challenges
Implementation for Java

• Captures and replays
  – Static calls
  – Constructor calls
  – Calls via reflection
  – Explicit class loading

• Allows for shared libraries
  – i.e., tested code and environment are free to use disjoint ArrayLists without verification.

• Preserves behavior on Java programs up to 100KLOC
Case study

• Daikon: 347 KLOC
  – Uses most of Java: reflection, native methods, JDK callbacks, communication through side effects

• Tests found real developer errors

• Two developers
  – Fine-grained compilable changes over two months: 2505
  – CVS check-ins over six months (all developers): 104
Evaluation method

• Retrospective reconstruction of test factoring’s results during real development
  – Test on every change, or every check-in.
• Assume capture happens every night
• If transcript is too large, don’t capture
  – just run original test
• If factored test throws a ReplayException, run original test.
Measured Quantities

- *Test time*: total time to find out test results
- *Time to failure*: If tests fail, how long until first failure?
- *Time to success*: If tests pass, how long until all tests run?

- ReplayExceptions are treated as giving the developer no information
## Results

<table>
<thead>
<tr>
<th>How often?</th>
<th>Test time</th>
<th>Time to failure</th>
<th>Time to success</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dev. 1</td>
<td>Every change</td>
<td>.79 (7.4 / 9.4 min)</td>
<td>1.56 (14 / 9 s)</td>
</tr>
<tr>
<td>Dev. 2</td>
<td>Every change</td>
<td>.99 (14.1 / 14.3 min)</td>
<td>1.28 (64 / 50 s)</td>
</tr>
<tr>
<td>All devs.</td>
<td>Every check-in</td>
<td>.09 (0.8 / 8.8 min)</td>
<td>n/a</td>
</tr>
</tbody>
</table>

David Saff
Discussion

• Test factoring dramatically reduced testing time for checked-in code (by 90%)
• Testing on every developer change catches too many meaningless versions
• Are ReplayExceptions really not helpful?
  – When they are surprising, perhaps they are
Future work: improving the tool

• Generating automated tests from UI bugs
  – Factor out the user
• Smaller factored tests
  – Use static analysis to distill transcripts to bare essentials
Future work: Helping users

• How do I partition my program?
  – Should ResultSet be tested or mocked?

• How do I use replay exceptions?
  – Is it OK to return null when "" was expected?

• Can I change my program to make it more factorable?
  – Can the tool suggest refactorings?
Conclusion

• Test factoring uses large, general system tests to create small, focused unit tests
• Test factoring works now
• How can it work better, and help users more?
• saff@mit.edu
Challenge: Better factored tests

• Allow more code changes
  – It’s OK to call toString an additional time.

• Eliminate redundant tests
  – Not all 2,000 calls to calculatePayroll are needed.
Evaluation strategy

1) *Observe*: minute-by-minute code changes from real development projects.

2) *Simulate*: running the real test factoring code on the changing code base.

3) *Measure*:
   - Are errors found faster?
   - Do tests finish faster?
   - Do factored tests remain valid?

4) *Distribute*: developer case studies
Conclusion

• Rapid feedback from test execution has measurable impact on task completion.
• Continuous testing is publicly available.
• Test factoring is working, and will be available by year’s end.
• To read papers and download:
  – Google “continuous testing”
Case Study

- Four development projects monitored
- Shown here: Perl implementation of delta tools.
- Developed by me using test-first development methodology. Tests were run often.
- Small code base with small test suite.

<table>
<thead>
<tr>
<th>lines of code</th>
<th>5714</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time worked (hours)</td>
<td>59</td>
</tr>
<tr>
<td>total test runs</td>
<td>266</td>
</tr>
<tr>
<td>average time between tests (mins)</td>
<td>5</td>
</tr>
</tbody>
</table>
We want to reduce wasted time

Test-wait time:
If developers test often, they spend a lot of time waiting for tests to complete.

Regret time:
If developers test rarely, regression errors are not found quickly. Extra time is spent remembering and fixing old changes.
Results predict: continuous testing reduces wasted time.

Best we can do by changing frequency.

Best we can do by changing order.

Continuous testing drastically cuts regret time.

Wasted Time Reduction by Continuous Testing

Without ct

With ct

Regret

Test-wait

Observed

Best Reorder

Random

Recent Errors
A small catalog of test factorings

- Like refactorings, test factorings can be catalogued, reasoned about, and automated

Separate Sequential Code:

Also “Unroll Loop”, “Inline Method”, etc. to produce sequential code
A small catalog of test factorings

Introduce Mock:
Unit test

Provided

Checked
Always tested: Continuous Testing and Test Factoring

David Saff
MIT CSAIL
IBM T J Watson, April 2005
Overview

• **Part I: Continuous testing**
  Continuous testing runs tests in the background to provide feedback as developers code.

• **Part II: Test factoring**
  Test factoring creates small, focused unit tests from large, general system tests.
Part I: Continuous testing

- **Continuous testing** runs tests in the background to provide feedback as developers code.
- Work with Kevin Chevalier, Michael Bridge, Michael D. Ernst
Part I: Continuous testing

• Motivation

• Students with continuous testing:
  – Were more likely to complete an assignment
  – Took no longer to finish

• A continuous testing plug-in for Eclipse is publicly available.

• Demo!
“Traditional” testing during software maintenance (v2.0 → v2.1)

- Developer has v2.0 test suite
  - Changes the code
  - Runs the tests
  - Waits for completion
  - Repeats…
Continuous Testing

- Continuous testing uses excess cycles on a nearby 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:

<table>
<thead>
<tr>
<th>✓</th>
<th>!</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>!</td>
<td>Syntax error on token &quot;a&quot;, &quot;,&quot;)&quot; expected</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>✓</th>
<th>!</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>!</td>
<td>Test failure: testArithmetic(ct.test.MainTestSuite)</td>
</tr>
</tbody>
</table>
Case study

• Single-developer case study [ISSRE 03]
• Maintenance of *existing software* with regression test suites
• Test suites took *minutes*: test prioritization needed for best results
• Focus: *quick discovery of regression errors* to reduce development time (10-15%)
Controlled human experiment

- 22 undergraduate students developing Java in Emacs
- Each subject performed two 1-week class programming assignments
  - Test suites provided in advance
- *Initial development*: regressions less important
- Test suites took *seconds*: prioritization unnecessary
- Focus: “What happens when the computer thinks about testing for us?”
Experimental Questions

1. Does continuous testing improve productivity?  
   - Yes

2. Does continuous compilation improve productivity?  
   - Yes

3. Can productivity benefits be attributed to other factors?  
   - No

4. Does asynchronous feedback distract users?  
   - No
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 (Questions 1 and 2)

<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 3)

- **Frequent testing: no**
  - Frequent manual testing: 33% success

- **Easy testing: no**
  - All students could test with a keystroke

- **Demographics: no**
  - No significant differences between groups

<table>
<thead>
<tr>
<th>Treatment</th>
<th>correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>No tool</td>
<td>27%</td>
</tr>
<tr>
<td>Cont. comp.</td>
<td>50%</td>
</tr>
<tr>
<td>Cont. testing</td>
<td>78%</td>
</tr>
</tbody>
</table>
### 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>
Did continuous testing win over users? (Question 4)

<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>
Eclipse plug-in for continuous testing

• Upgrades current Eclipse JUnit integration:
  – Remember and display results from several test suites
  – Pluggable test prioritization and selection strategies.
  – Remote test execution
  – Associate test suites with projects
Eclipse plug-in for continuous testing

- Adds continuous testing:
  - Tests run with every compile
  - Can run as low-priority process
  - Can take advantage of hotswapping JVMs
  - Works with plug-in tests, too.

- Demo!
Future Work: Continuous testing

• Incorporate JUnit and continuous testing features from plug-in directly into Eclipse
• Encourage test prioritization researchers to implement JUnit plug-ins
• Industrial case studies
There’s more than one way to factor a test!

Basic strategy:
- *Capture* a subset of behavior beforehand.
- *Replay* that behavior at test time.
Separate Sequential:

• Before each stage, recreate state
• After each stage, confirm state is correct