Mock Object Creation for Test Factoring

David Saff, Michael D. Ernst
MIT CSAIL
PASTE, 2004 June
Motivation

• Continuous testing plug-in for the Eclipse IDE*

• Test suite:

  - Problem: find out about errors faster
  - Solution: *mock objects* to replace Eclipse framework

* Saff, Ernst, ETX 2004: Continuous testing in Eclipse
Outline

• Mock objects introduced
• Test factoring introduced
• Mock object creation for test factoring
• Conclusion
Outline

• Mock objects introduced
• Test factoring introduced
• Mock object creation for test factoring
• Conclusion
Unit test for plug-in

Provided

Checked

Plug-in
System Test for plug-in

Provided

Checked

Plug-in

Eclipse
A mock object:

- provides part of the functionality of the original object(s)
- is focused on allowing the test to proceed
Mock objects for our example

• Using a debugger, determined:
  – 147 static calls from plug-in to framework
    • Defined on 49 classes
  – 8 callbacks from framework to plug-in
• Substantial work to define mock objects.
• How well can we automate this process without additional manual effort?
Outline

• Mock objects introduced
• Test factoring introduced
• Mock object creation for test factoring
• Conclusion
What is a factored test?

• Split a system test into several smaller *factored* tests that
  – exercise less code than system test
  – can be added to the suite and prioritized
    • Find out about errors faster
  – embody assumptions about future code changes
Pros and cons of factored tests

- **Pro**: factored test should be faster if system test
  - is slow
  - requires an expensive resource or human interaction
- **Pro**: isolates bugs in subsystems
- **Con**: if assumptions about how developer will change the code are violated, can lead to:
  - false negatives: OK, some delay
  - false positives: bad, distract developer
When change language is violated, factored test must be discarded and re-created

- Can detect violation through analysis, or incorrect result.
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:
Related work

This work (static + dynamic)

- Produce factored tests
- Run factored tests
- Run original tests

Binkley '97 (static)

- Slice based on change
- Run factored tests

Developer makes change

Correct test results

Early warning if assumptions hold
Outline

• Mock objects introduced
• Test factoring introduced
• Mock object creation for test factoring
• Conclusion
Basic Procedure: Trace Capture

MockExpectations

“Boundary”

Mocked Realm

Tested Realm

Params, Returns, Callbacks
Basic Procedure: code generation

- MockExpectations encodes a state machine:

```
MockExpectations

0
DebugPlugin.getDefault() → [object id 347]

1
[object id 347].getLaunchManager() → [object id 78]

2
[object id 78].uniqueLaunchNameFrom("a") → "a134"
```
Expanding the change language

• Current tolerated change language includes:
  – Extract method
  – Inline method

• Using static analysis on mocked code, improve the procedure to include:
  – Reorder calls to independent objects
  – Add or remove calls to pure methods
Reorder calls to independent objects

• Group objects that share state into state sets

• One MockExpectations per state set:
Add or remove pure method calls

• Allow reordering, addition, removal of calls to pure methods:

```java
DebugPlugin.getDefault() → [object id 347]
[object id 347].getLaunchManager() → [object id 78]
[object id 78].uniqueLaunchNameFrom("a") → "a134"

[object id 78].removeLaunch("a134") → NEXT STATE
```
Outline

• Mock objects introduced
• Test factoring introduced
• Mock object creation for test factoring
• Conclusion
Future work

• Develop a framework for test factoring
• Implement the “Implement Mock” factoring
• Analytic evaluation of framework
  – Capture real-project change data*
  – Measure notification time, false positives
• Case studies of test factoring in practice
  – How do developers feel about the feedback they receive?

* Saff, Ernst, ISSRE 2003: Reducing wasted development time via continuous testing
Conclusion

• Test factoring can indicate errors earlier
• “Introduce Mock” is an important test factoring for complicated systems
• We propose:
  – Dynamic analysis for building mock objects
  – Static analysis for increasing the change language
• Mail: saff@mit.edu
A small catalog of test factorings

• Separate Sequential Test:
  – [graphic]

• Unroll Loop:
  – [graphic]

• Introduce Mock:
  – [graphic]
Frequent testing is good:

• Frequent *manual* testing in agile methodologies

• Frequent *automatic* testing in continuous testing.

• A testing framework should minimize the cost of frequent testing
  
  – *Suite* completes rapidly
  
  – *First failing test* completes rapidly
Getting faster to the first failing test

- Default:

- Test selection:

- Test prioritization:

- Test factoring:
Dynamic, change-independent test factoring

- Dynamic: instrument and run the original test
- Change-independent: factoring happens before any changes are made.
  - Requires a hypothesized change language
- Binkley ’97: Static, change-dependent test factoring
Automatic test factoring: change-dependence

• Change-dependent test factoring:
  – After tested code is changed, generate new tests with same result as old tests for that change.

• Change-independent test factoring:
  – Before tested code is changed, generate new tests that have the same result as old tests for some set of changes.
Automatic test factoring: static vs. dynamic analysis

• Static analysis (Binkley ’97)
  – Analyze code to determine mock object behavior
  – Well-suited for change-dependent factoring
  – May fail
    • without source
    • when dependent on file system or user interaction
  – Guaranteed change language may be restrictive

• Dynamic analysis (this work)
  – Instrument and run the original test, gather logs
  – May run original test after factored test fails