Mock Object Creation for Test Factoring

David Saff Michael D. Ernst
MIT Computer Science & Artificial Intelligence Lab
{saff,mernst}@csail.mit.edu

Abstract

Test factoring creates fast, focused unit tests from slow system-wide tests; each new unit test exercises only a subset of the functionality exercised by the system tests. Augmenting a test suite with factored unit tests, and prioritizing the tests, should catch errors earlier in a test run.

One way to factor a test is to introduce *mock objects*. If a test exercises a component A, which is designed to issue queries against or mutate another component B, the implementation of B can be replaced by a *mock*. The mock has two purposes: it checks that A's calls to B are as expected, and it simulates B's behavior in response. Given a system test for A and B, and a record of A's and B's behavior when the system test is run, we would like to automatically generate unit tests for A in which B is mocked. The factored tests can isolate bugs in A from bugs in B and, if B is slow or expensive, improve test performance or cost.

This paper motivates test factoring with an illustrative example, proposes a simple procedure for automatically generating mock objects for factored tests, and gives examples of how the procedure can be extended to produce more robust factored tests.

Categories and Subject Descriptors: D.2.5 (Testing and Debugging): Testing tools

General Terms: Algorithms, Design, Performance, Verification

Keywords: test factoring, mock objects, unit testing

1. Introduction

Frequent execution of a test suite during software maintenance can catch regression errors early and bolster the developer's confidence that steady progress is being made. However, if the test suite takes a long time to produce feedback, the developer is slowed down, and the benefit of frequent testing is reduced, whether the testing is manual (as in agile methodologies such as Extreme Programming [1]) or automated (as in continuous testing [4]).

Test selection and prioritization can reduce the cost of frequent testing for large test suites containing many small tests. Test selection [3] runs only those tests that are possibly affected by the most recent change, and test prioritization [7] runs first the tests that are most likely to reveal a recently-introduced error.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASTE'04, June 7–8, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-910-1/04/0006 ...$5.00.

For test suites with long-running tests (or expensive tests that require human attention or the use of a limited resource), selection and prioritization are insufficient. We propose augmenting them with *test factoring*, which from each large test generates multiple unit tests that can be run individually. Dynamic test factoring takes as input an original test and a trace collected from running an instrumented version of the original test, and produces one or more new tests, each of which should run more quickly than the original, while testing less functionality than the original — perhaps exercising only a single component of the code. Static test factoring uses sound static analysis on just the source of the original test; it introduces a different set of tradeoffs than dynamic test factoring, and is not considered here. Test factoring occurs ahead of time, not at test time.

This paper proposes an automated system for applying the Introduce Mock test factoring to a test suite. The paper defines the goals of this test factoring (Section 2) and describes an example test suite that we are using for inspiration and benchmarking (Section 3). It then describes a procedure for generating the factored tests (Section 4) and an experimental strategy for evaluating the procedure (Section 5).

2. Introduce Mock

A test is factored by applying one or more *test factorings*. We believe that test factorings can be cataloged, shared, and automated, just as code refactorings [2] are. Three of the test factorings we have defined are Separate Sequential Test (which separates functionality exercised at different times), Unroll Loop (which separates control structures and their contents), and Introduce Mock (which separates functionality defined in different classes). This paper focuses on Introduce Mock.

Introduce Mock operates on a codebase that is divided into two different realms. The *tested realm* is code that is being changed, into which the developer is concerned regression errors may be introduced. The *mocked realm* is code that should be simulated for the purposes of testing. Either the mocked realm is not changing, and may therefore be skipped to improve test performance, or it is likely to change independently of the tested realm, and the developer would like to isolate errors between the realms.

3. Example

To focus our efforts, we have chosen an example application from our own experience. We previously developed a plug-in for the Eclipse IDE that implements continuous testing [5]. Continuous testing utilizes excess cycles on a developer's workstation to run tests in the background, providing faster notification of errors [4, 6].

The plug-in has an automated test suite that uses JUnit and Eclipse's Plug-in Development Environment (PDE). Using the PDE

to test a plug-in requires loading and initializing all of Eclipse before tests can be run against the specific plug-in. The test suite is excellent for integration testing, but has become increasingly unwieldy for frequent manual invocation or continuous testing. The suite contains 49 programmatic tests that exercise the GUI, backend, and utility classes. The tests themselves average less than a second each, and are executed according to an effective test prioritization algorithm; errors, if they exist, are usually found very early in the suite execution. However, initializing the Eclipse runtime requires 20-30 seconds before the first test can run. If this overhead could be eliminated, the developer could be notified of failures an order of magnitude faster than they are now. This may improve development productivity [4, 6].

We would like to generate a factored version of the test suite that mocks all packages under `org.eclipse`. This should allow us to run the tests much more quickly. Our tested realm is all of the plug-in and our mocked realm is the Eclipse runtime.

This example is sufficiently complex to require automation, but simple enough to permit some manual analysis. We used a debugger to step through the execution of a single test case for the important basic functionality of the plug-in, and noted every executed line of code that contained a method call across the boundary between the tested and mocked realms. The tested code called the mocked code in 147 places, and the mocked code called back to the tested code in 8 places. The majority of these static calls were executed only once, but a few were executed dozens of times with different parameters. The methods called on the mocked realm were defined on 49 different classes.

4. Procedure

The Introduce Mock procedure can be generally outlined as follows:

1. **Transformation:** The code undergoes a semantics-preserving transformation that replaces constructor calls, array writes, and field accesses across the realm boundary with method calls. The reason for this transformation is that method calls are easier to instrument.
2. **Trace capture:** The original test is executed (we assume it passes), and traces are collected of calls from the tested realm into the mocked realm and vice versa.
3. **Mock code generation:** The traces are analyzed and code generated for the mock objects; this *mock realm* replaces the mocked realm in the final factored test.

Section 4.1 gives more details on a basic implementation of Introduce Mock. Section 4.2 discusses limitations in the basic implementation, and Section 4.3 discusses how to eliminate some of those limitations.

4.1 Basic Procedure

4.1.1 Trace capture

The original test is instrumented and run. The instrumentation captures all method calls from the tested realm into the mocked realm, and vice versa. The actual parameters are captured, along with the method's *behavior*, which consists of the return value, if any, and any callbacks across the boundary. For example, if a list object in the tested realm is passed as a parameter to a method in the mocked realm, any methods on the list called from the mocked realm are captured.

The trace is analyzed to produce a `MockExpectations` table for use in the next step. This table encodes a transition function that specifies for each state of the mocked realm, the expected next call (including the receiver and parameters), the behavior to be exhibited in response to that call, and the next state to enter. The states are modeled as simple integers starting at 0 and counting upward, without any branches or loops.

4.1.2 Mock code generation

In the factored test, each object in the mocked realm that is visible to the tested realm is replaced by a mock object that satisfies the same interface. In addition, a singleton mock object for handling constructors and static methods is created for each class in the mock realm that is visible to the tested realm. Two global objects are shared by all mock objects: the static `MockExpectations` table generated by the table creation step of Section 4.1.1, and a dynamic `MockState` that points to the current state of the mock realm.

For each call received by a mock object, the `MockExpectations` is consulted to determine whether the received call matches the expected next call for the current `MockState`. If so, the behavior for that call is played back, and the `MockState` is updated. If not, the test fails. At the end of the test, the test fails if the current `MockState` does not match the expected final state in the `MockExpectations`.

4.2 Limitations of the Basic Procedure

A factored test introduces assumptions about the implementation details of the functionality being tested; if those assumptions are violated during program evolution, the factored test becomes useless. For example, consider a test for a method that inserts several records into a database. If making calls against the database is slow, the test may be factored to use a mock object that simulates the behavior of the database and ensures that the expected calls are made to the database API. If the code under test is modified to use a database API call that inserts records all at once rather than one at a time, then the original test will still pass, but the factored test will likely fail, because the mock object receives unexpected calls.

A test factoring procedure can always be extended to eliminate an erroneous assumption. For example, with knowledge of the semantics of the database API, the database mock object could be extended to accept all valid data input call sequences. However, the only way to eliminate *all* assumptions is to turn the factored tests into exact replicas of the original test, eliminating the speed and bug-isolation advantages of test factoring.

We believe that a developer facing a maintenance task consciously or unconsciously uses a *change language*, a kind of pattern language that breaks down a complex maintenance goal into a plan for making a series of simple code changes. If this is the case, then understanding a developer's change language would allow prediction of which changes they are likely to make. This in turn would help to maximize the "lifespan" of factored tests before their assumptions are violated and they become useless. This paper describes a change language as a set of program transformations and refactorings [2]. The basic procedure defined in Section 4.1 assumes that the order of calls across the boundary will not change; this assumption is compatible with a change language including refactorings like Inline Method and Extract Method [2], but in general it is too strict.

4.3 Expanding the Basic Procedure

This section discusses three functionality-preserving changes that are common in the authors' practice but violate the assumptions of the basic procedure, leading to false failures. In each case, sound static analysis could be used to identify exactly when those assumptions can be correctly relaxed. However, there are good reasons to look for simple unsound heuristics that may err on the side of relaxing assumptions too much. First, keeping the analyses simple allows quicker (and therefore more frequent) test factoring, and quicker implementation of new test factoring procedures. Second, overly lax assumptions carry a smaller penalty than overly strict assumptions. If assumptions are too lax, a factored test may pass when its corresponding system test would have failed (a *false success*). This failure will eventually be caught when running the original system test, so only the opportunity for early notification is lost. If assumptions are too strict, a factored test may fail when the system test would have passed (a *false failure*). The developer may lose time and attention investigating the failure before discovering that it is false.

4.3.1 Reordering calls to independent objects

The basic procedure of Section 4.1 assumes that the expectations and behavior of the entire mocked realm depend on the order of all calls to mocked objects. However, this is an oversimplification; sometimes, two objects are independent of each other's state, and calls to these objects can be intermixed in any order without affecting overall behavior. The developer may choose to reorder such calls to improve the clarity or structure of the code; or, the order of calls may be nondeterministic.

This problem could be solved by introducing separate `MockExpectations` for each and every mock object, but this would oversimplify in the other direction, treating every mocked object as independent. Instead, the mocked objects are grouped into *state sets*, each with a single `MockExpectations` and `MockState`. Our heuristic for determining state sets is that if one object is passed to the constructor for another, both belong to the same state set. We believe that this heuristic may be idiosyncratic of Eclipse coding practices, and further research will help to fine-tune it.

4.3.2 Adding or removing calls to accessors

The basic procedure assumes that all calls to mocked objects are significant state-changing operations, and therefore cannot be added or deleted. However, accessor methods, which do not change the receiver's state, are often added, deleted, and reordered during maintenance. For example, multiple calls to an accessor might be replaced with a single call to improve efficiency.

To accommodate such changes, each method in the mocked realm that is called from the tested realm can be labeled *read-only*, *write-only*, or *read-write* with respect to each state set, by extending the trace to record reads and writes to the state of mocked objects. `MockExpectations` can then be modified to neither fail nor advance the state when a read-only method is called.

This introduces a new complication, however. It is now possible for a change to introduce a call to a read-only method in a state for which the mocked object's behavior was not recorded in the trace. There are several possibilities for handling this situation: the test could fail, the method could return a value unlikely to occur in real execution, or the mock object could examine the trace to return a value that is likely to occur in real execution, in hopes of minimizing false failures. Further research is needed to know when each should be used.

4.3.3 Changing the order of simple mutators

Some classes have multiple options that must be set through individual setter methods before an action method can be called. For example, before a `Button` object in the `SWT` framework is displayed, methods `setText`, `setFont`, `addSelectionListener`, and `setSelection` may be called in any order. Reordering calls to these methods should not cause a factored test to fail.

To address this problem, the `MockExpectations` object can be further modified to allow an unordered set, or *clump*, of method calls to be the trigger to advance to a new state, rather than changing state on every method call that is not read-only. Choosing the right method calls for advancing the state is important. In our example, methods that read-write tend to be important state-changing operations like `open`, `close`, or `addListener`. This suggests the heuristic of allowing write-only method calls to clump, but advancing the state on read-write method calls.

5. Evaluation

The success of test factoring depends on how much less time the factored tests take to run than the originals, how much faster failures are indicated to the user, and how many false failures are generated as the program changes. We intend to perform an evaluation using high-resolution change data captured from third-party development projects [4], followed by case studies of developers using test factoring. These evaluations should allow us to measure the cost and benefit of test factoring, and to improve our change language catalog and test refactoring procedures.

6. Conclusion

In this paper, we have introduced the idea of test factoring, and described how it might be useful for enabling more frequent, and more useful, regression testing during development. We considered one test factoring, Introduce Mock Object, in detail, in the context of a real test suite to be factored. Our research on this topic is still in its early stages, but we believe that the problems are of interest not only for their practical impact, but also for their potential to open new research questions in testing, software engineering, and program analysis.

7. REFERENCES

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

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

[4] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, Nov. 2003.

[5] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, Spain, Mar. 2004.

[6] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, July 2004.

[7] 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.