Test Factoring with amock: Generating Readable Unit Tests from System Tests

by

David Samuel Glasser

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

Author . . . .
Department of Electrical Engineering and Computer Science
August 21, 2007

Certified by. . . . .
Michael D. Ernst
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Test Factoring with `amock`: Generating Readable Unit Tests from System Tests

by

David Samuel Glasser

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Automated unit tests are essential for the construction of reliable software, but writing them can be tedious. If the goal of test generation is to create a lasting unit test suite (and not just to optimize execution of system tests), it is essential that generated tests can be understood by the developers that will be running them, so that they can tell the difference between real and spurious failures. `amock` is a system which automatically generates human-readable JUnit regression tests that use mock objects to simulate the behavior of individual objects dynamically observed during a system test execution.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Acknowledgments

Professor Michael Ernst, my research adviser, consistently expects nothing but the best from his entire lab. Without his high standards, detailed feedback, and constant encouragement, I cannot imagine completing this thesis project.

I appreciate the support of the rest of the Program Analysis Group, especially David Saff (whose project inspired `amock`), Adam Kieżun and Shay Artzi (who guided me through my first PAG project), Stephen McCamant (who keeps the lab reading group and all of our workstations running), and Maria Rebelo (who keeps grouchy grad students fueled with a stream of smiles and sugar).

Thanks to the jMock development team for their responsiveness and guidance in using their excellent testing tool.

Two years ago I was a theoretical mathematician; today I am an open source software developer. I would not be where I am today without the encouragement and mentorship of Jesse Vincent and Chia-Liang Kao of Best Practical Solutions.

My sanity in my time as a grad student is mostly due to my housemates at pika, who teach me that life is a continuing experiment, and my senseis and fellow students at MIT Isshinryu Karate-do, who teach me the difference between my perceived and actual limits.

Finally, I am forever grateful to my parents and sister, who have always let me set my own path through life and given me loving support along the way.

Contents

List of Figures

11

Chapter 1

Introduction

1.1 Motivation

Automated testing is essential for the construction of reliable software. Computer systems are complex enough that even small changes can have far-reaching consequences. Both macroscopic *system tests* and microscopic *unit tests* are required to ensure both that changes do not affect the overall operation of the system and that internal interfaces continue to function as expected. However, writing tests can be one of the most tedious parts of the software development process. While most software engineers recognize the importance of automated testing, many projects fail to achieve a desirable level of testing.

Unit tests complement system tests; a suite with both unit and system tests has advantages over one with just system tests. If a developer changes one small part of a program, the unit tests for just that part may run faster than an entire system test. System test failures often only reveal whether or not the test passed; when a system test starts to fail, it is often difficult to find which subsystem the error originated in. Unit tests are more focused, and so when a unit test fails, it is generally easier to tell which module or even method is responsible for the failure. Finally, unit tests enable testing the internal logic of programs which deal with complex resources such as databases, network connections, or graphical user interfaces without needing to set up the resource.

Unfortunately, writing unit tests can be tedious, especially when you are retroactively adding them to a legacy project. It is relatively easy to write a few system tests for a working program — simply run the program with some specified input and check that the output is as expected. Writing unit tests is much more time-consuming, since a program typically has a very large number of individual units (classes, methods, etc.) that would profit from independent unit testing. Writing tests while writing the program itself, perhaps via test-driven development [4], makes this much easier. But given a large legacy system without any unit tests, the task of writing unit tests by hand for every module can be daunting.

Because writing tests is important but tedious, the possibility of automating any part of the test creation process is very attractive. Many different approaches to automatic test generation have been studied. One common method is **capture-replay**[27]. Here, a developer runs the program in a special mode which **captures** the inputs and outputs to the program; the system later **replays** the program with the recorded inputs, checking that the outputs and other behavior are the same. Capture-replay is commonly used for system testing of programs with graphical user interfaces: the system captures and replays the mouse clicks. Unfortunately, these capture-replay tools generally create fragile and obscure tests; they generally can only interact with the external user interface of the program, and only verify externally-visible results. Thus they can generally only create system tests, not unit tests.

A test factoring [35, 13] system generates an entire suite of *unit* tests from a single system test execution. The key insight of test factoring is that a system test (which is easier to create than an exhaustive suite of unit tests) exercises the units of a program in a "typical" way. Test factoring extends the idea of capture-replay to three phases: capture, factor, and replay. A test factoring system first runs a capture phase on a system test (or any execution); instead of just recording the externally visible inputs and outputs, `amock` instruments the running program to record much of its internal interactions. The second phase is the factoring phase: `amock` slices a single recorded trace of the entire system into many descriptions of the interaction of single units with their environments. Finally, the replay phase runs all of the generated tests

separately; it only runs code in the unit being tested, relying on the description of the expected interactions to skip executing the rest of the system.

We have designed and implemented a new test factoring system for Java, named `amock`. `amock` differs from previous capture-factor-replay systems in that the replay phase contains no custom infrastructure: the factored tests are ordinary Java code that uses the JUnit unit testing framework [23, 5] and the jMock mock object generation library [22, 17]. Any developer who is familiar with jMock can read and understand the generated unit test suite. Developers can incorporate these generated tests directly into their test suites as unit regression tests.

If the goal of test generation is to create a lasting unit test suite (and not just to optimize execution of system tests), it is essential that generated tests can be understood by the developers that will be running them, so that they can tell the difference between real and spurious failures. Many automated test generation systems (including `amock`!) generate tests that are too brittle: they are tightly bound to a specific implementation. That is, the conditions verified by brittle tests are overconstrained; semantically irrelevant changes to the tested code can cause spurious test failures. However, a test that is too brittle might still be salvageable if the developer can fix it by hand, perhaps by relaxing an expected output constraint. Because the tests generated by `amock` use the standard JUnit and jMock testing framework, `amock`'s tests can be comprehended by developers with no special knowledge of `amock` itself. If `amock` generates a test with expectations that are too constrained, the developer can edit it to relax the expectation and allow it to correctly pass. `amock`'s generated tests are thus more comprehensible and malleable than those of previous test factoring systems [35, 13].

1.2   Overview of test factoring technique

Test factoring with `amock` consists of three phases:

- **Capture**: A developer runs a system test with the `amock` **instrumentation agent** loaded into their JVM. The instrumentation writes a **trace** of method

calls, field accesses, and other events that occur during the execution. This phase is described in Section 3.1.

- **Factor**: The developer chooses which object from the trace to test and runs the `amock` **test factorizer** on the trace. The factorizer produces a JUnit test simulating the effect of the system test on the chosen object. The tested object is isolated from the rest of the system through the use of **mock objects**; mock objects are described in detail in Chapter 2. The factorizer is implemented as a simple state machine that makes a single pass over the trace. This phase is described in Section 3.2.

- **Replay**: The developer compiles the generated JUnit tests and adds it to the project's JUnit test suite. This phase is described in Section 3.3.

A single trace can be factored into many unit tests, one for each object in the trace.

1.3   Example

We show an example of `amock` in action. We would like to create unit tests for the library shown in Figure 1-1. We already have a system test, shown in Figure 1-2. The system test creates a cookie jar, loads it with some cookies, creates a cookie monster, and tells the cookie monster to eat all the cookies in the jar. We write the specification shown in Figure 1-3, and run it by typing "`rake bakery`".

Capture phase

The build system first runs the `BakerySystemTest` test using `amock`'s instrumentation agent, by executing "`java -javaagent:amock.jar=--tracefile=trace BakerySystemTest`"[1]. This creates a trace of the system test execution in the file `trace`; this file logs every method call and field access during the execution.

---

[1]Some details of the trace generation are glossed over here; it is described in more detail in Section 3.1.

```
1  public class CookieMonster {
2      public int eatAllCookies(CookieJar jar) {
3          int cookiesEaten = 0;
4          for (Cookie k = jar.getACookie();
5               k != null;
6               k = jar.getACookie()) {
7              k.eat();
8              cookiesEaten++;
9          }
10         return cookiesEaten;
11     }
12 }
13 public class CookieJar {
14     private List<Cookie> myCookies;
15     public Cookie getACookie() {
16         if (myCookies.isEmpty()) {
17             return null;
18         } else {
19             return myCookies.remove(0);
20         }
21     }
22 }
```

Figure 1-1: A library lacking a unit test suite.

```
1  public class BakerySystemTest {
2      public static void main(String[] args) {
3          CookieJar j = new CookieJar();
4          Cookie oatmeal = new OatmealCookie();
5          j.add(oatmeal);
6          loadMoreCookies(j);
7          assertThat(new CookieMonster().eatAllCookies(j),
8                     is(2));
9      }
10     private static void loadMoreCookies(CookieJar j) {
11         j.add(new ChocolateCookie());
12     }
13 }
```

Figure 1-2: A system test for the library in Figure 1-1.

```
1  require 'amock_tasks' # defines amock_test declaration.
2  amock_test(:bakery) do |a|
3    a.system_test = 'edu.mit.csail.pag.amock.subjects.Bakery'
4
5    a.unit_test('cookiemonster') do |u|
6      u.package = 'edu.mit.csail.pag.amock.subjects.bakery'
7      u.tested_class = "CookieMonster"
8    end
9
10   a.unit_test('cookiejar') do |u|
11     u.package = 'edu.mit.csail.pag.amock.subjects.bakery'
12     u.tested_class = "CookieJar"
13   end
14 end
```

Figure 1-3: A specification for generating two unit tests from the system test in Figure 1-2 (as a Ruby Rakefile [32]).

Factor phase

The build system then runs the `amock` factorizing processor twice on the trace file: once specifying the `CookieMonster` as the class to generate unit tests for, and once specifying the `CookieJar`[2]. The first execution produces the test shown in Figure 1-4; the second produces the test shown in Figure 1-5.

The `CookieMonster` test (Figure 1-4) creates a new `CookieMonster` and tells it to eat all the cookies in a "mocked" cookie jar (in the assertion on line 39), and verifies that the method returns 2. Instead of actually running the code in `CookieJar`, the mocked `CookieJar` specifies exactly how the jar should act: when the cookie monster tries to get a cookie from the mocked jar, it will first return `mockCookie`, then `mockCookie1`, and finally `null`. Note also that the `CookieMonster` test does not need to know about the precise implementations of the `Cookie` interface that it is dealing with (`OatmealCookie` and `ChocolateCookie` in the system test); the mock objects in lines 15 and 16 are just mocking `Cookie`. The test also specifies (in lines 24 and 31) that the cookie monster must call `eat` on both cookies.

The `CookieJar` test (Figure 1-5) represents the same system test, but from the jar's point of view. The existence of the `CookieMonster` is irrelevant to this test. From the perspective of the jar, all that happened during the system test was that two cookies were added to it and then removed again. None of the methods invoked on the jar called out to the mocked objects, so no expectations need to be set up. Thus we see that `amock` factors out the interaction of each individual object with its environment during the test.

Let us imagine that in a future version of the library, an optimization to `CookieMonster` changes its behavior to take all of the cookies out of the jar before eating any of them. This will cause the test in Figure 1-4 to fail, because it explicitly specifies that the expected methods have to occur in the order specified. However, the developer can fix this easily: just remove the `inSequence` calls in lines 25 and 32. Thus, `amock`'s generated tests can be repaired instead of just scrapped when they are too brittle.

---

[2]Again, some internal stages are glossed over; the factorization process is described in Section 3.2.

```
1  // Generated by amock.
2  package edu.mit.csail.pag.amock.subjects.bakery;
3
4  import edu.mit.csail.pag.amock.jmock.Expectations;
5  import edu.mit.csail.pag.amock.jmock.MockObjectTestCase;
6  import static org.hamcrest.MatcherAssert.assertThat;
7  import static org.hamcrest.core.Is.is;
8
9  public class AutoCookieMonsterTest extends MockObjectTestCase {
10   public void testCookieEating() throws Throwable {
11     // Set up primary object.
12     final CookieMonster testedCookieMonster = new CookieMonster();
13
14     // Set up expectations and run the test.
15     final Cookie mockCookie = mock(Cookie.class);
16     final Cookie mockCookie1 = mock(Cookie.class);
17     final CookieJar mockCookieJar = mock(CookieJar.class);
18
19     verifyThenCheck(new Expectations() {{
20       one (mockCookieJar).getACookie();
21       inSequence(s);
22       will(returnValue(mockCookie));
23
24       one (mockCookie).eat();
25       inSequence(s);
26
27       one (mockCookieJar).getACookie();
28       inSequence(s);
29       will(returnValue(mockCookie1));
30
31       one (mockCookie1).eat();
32       inSequence(s);
33
34       one (mockCookieJar).getACookie();
35       inSequence(s);
36       will(returnValue(null));
37     }});
38
39     assertThat(testedCookieMonster.eatAllCookies(mockCookieJar),
40       is(2)
41     );
42   }
43 }
```

Figure 1-4: An automatically generated test for the `CookieMonster` object from the system test in Figure 1-2. The test uses the jMock library, described in Chapter 2.

```
 1 // Generated by amock.
 2 package edu.mit.csail.pag.amock.subjects.bakery;
 3
 4 import edu.mit.csail.pag.amock.jmock.Capture;
 5 import edu.mit.csail.pag.amock.jmock.MockObjectTestCase;
 6 import java.lang.Object;
 7 import static org.hamcrest.MatcherAssert.assertThat;
 8 import static org.hamcrest.core.Is.is;
 9 import static org.hamcrest.core.IsNull.nullValue;
10
11 public class AutoCookieJarTest extends MockObjectTestCase {
12   public void testCookieEating() throws Throwable {
13     // Set up primary object.
14     final CookieJar testedCookieJar = new CookieJar();
15
16     // Set up expectations and run the test.
17     final Cookie mockCookie = mock(Cookie.class);
18
19     testedCookieJar.add(mockCookie);
20
21     final Cookie mockCookie1 = mock(Cookie.class);
22
23     testedCookieJar.add(mockCookie1);
24
25
26     assertThat(testedCookieJar.getACookie(),
27       is((Cookie) mockCookie)
28     );
29
30
31     assertThat(testedCookieJar.getACookie(),
32       is((Cookie) mockCookie1)
33     );
34
35
36     assertThat(testedCookieJar.getACookie(),
37       is(nullValue())
38     );
39   }
40 }
```

Figure 1-5: An automatically generated test for the `CookieJar` object from the system test in Figure 1-2. The test uses the jMock library, described in Chapter 2.

1.4 Contributions

This thesis presents the following contributions:

- A new approach to test factoring that produces human-readable JUnit tests

- `amock`: An implementation of this approach

- Case studies showing the applicability of `amock` to real-world projects

- `smock`: An extension to the jMock library allowing developers to mock static methods

1.5 Thesis outline

The rest of this thesis is structured as follows.

Chapter 2 gives background on writing unit tests that use behavior verification with the library jMock. These tests are what `amock` creates; this chapter explains why one would write behavior-based tests and how jMock helps in doing so. Chapter 3 describes the overall architecture and implementation of `amock`: how it captures a system test and factors it into unit tests. We also describe conceptual and implementation limitations of `amock`. Chapter 4 motivates and describes several heuristics that improve the quality of `amock`'s output.

While creating `amock`, we discovered that the tests it generates were not as well isolated from the environment as they would have been if jMock allowed developers to mock static method calls. Chapter 5 describes a modest extension to jMock called `smock`, which can be used (completely independently of `amock`) to mock static method calls during jMock tests.

Chapter 6 describes case studies of real-world programs for which `amock` can generate test suites. Our examples include several programs that are often considered difficult to test, such as GUI programs and network clients. Chapter 7 describes the results of efficiency, robustness, and sensitivity experiments performed on `amock`.

Finally, Chapter 8 describes future work, gives an overview of related work, and summarizes the contributions of this thesis.

Chapter 2

Background: Behavior-based testing with jMock

`amock` creates behavior-based unit tests that use the jMock library [17, 22] to isolate the tested class from its environment. Section 2.1 describes state and behavior verification for unit tests and the concept of mock objects for behavior verification. Section 2.2 demonstrates how to write tests using jMock.

2.1 Testing state and behavior

Unit tests complement system tests by focusing on smaller modules than the entire system. In an object-oriented environment, a unit test generally tests just one class or method; we refer to the specific objects instantiated by the test whose behavior is verified as the "system under test" or SUT[28]. A typical unit test consists of four phases:

1. **Set up** the **test fixture**.

2. **Exercise** the SUT: call the methods that are being tested on the SUT, checking return values along the way.

3. **Verify** that the expected outcome has occurred.

4. **Tear down** the test fixture.

The test fixture consists of the SUT itself as well as all other objects that it needs to collaborate with during the test. The other objects can be "real" implementations of their type from the project itself, but often, complex resources such as databases or network connections are replaced with **test doubles** that allow the tests to be run without needing to set up real instances of the resources. These doubles can be simpler implementations of the same interface (such as an in-memory database instead of an on-disk one) or new objects created solely for the test suite. Most modern programming environments provide a standard framework for creating unit tests, such as Java's JUnit [5], which gives Java developers a common framework for writing and running unit tests.

Unit tests can decide whether or not they should pass using **state verification** or **behavior verification** [15]. State-based tests call the methods that are being tested on the SUT, and then examine the SUT and other fixture objects to ensure that they are in the expected state. State verification ignores the interactions of the tested methods with their environment during their execution, as long as the program does not crash: tests using state-based verification just check that at the end of the day, the world is in the right state.

Behavior verification allows tests to ensure that the "communication" between objects is as expected. The primary elements of an object-oriented program are the objects and the messages they pass to each other by calling methods on each other. Behavior verification monitors these messages to verify that the correct messages are passed. Instead of focusing on the state of objects *after* a tested method is called, behavior-based tests pay attention to the method calls made by the tested method.

To see the difference between state-based and behavior-based tests, consider testing a banking system's transfer operation: a method, `transfer`, which withdraws money from one account and deposits it into a second account. A state-based test, as shown in Figure 2-1, would set up two accounts with a given amount of money, call `transfer`, and verify that the balances of both accounts have changed by the correct amount. A behavior-based test would set up two accounts, run `transfer`, and check

24

```
1 public class PortfolioTest extends TestCase {
2     public static void testTransfer() {
3         // Set up fixtures.
4         Account from = new CheckingAccount(40);
5         Account to = new CheckingAccount(150);
6
7         // Set up tested object.
8         Portfolio portfolio = new Portfolio(from, to);
9
10        // Call a method and check its return value.
11        assertTrue(portfolio.transfer(from, to, 20));
12
13        // Check the state of the fixtures to make sure they're what
14        // is expected.
15        assertThat(from.getBalance(), is(20));
16        assertThat(to.getBalance(), is(170));
17    }
18 }
```

Figure 2-1: A JUnit unit test using state verification.

that the `withdraw` and `deposit` methods are called on the accounts with the correct amount of money.

It is possible to create behavior-based tests using manually-written stubs, but it is rather laborious. Figure 2-2 shows how this could be accomplished for the bank account example. In order to write just one test, the developer needed to define two stub classes which implement all of the `Account` methods and painstakingly check that the right methods are called up them by the `transfer` method. Manual stubs for behavior verification become even more complicated to write as the sequence of expectations grows.

A mock object generation library allows a developer to write behavior-based tests without having to manually track the expected method calls in lengthy stubs. Using a library such as jMock [17, 22], the developer can create **mock objects** that implement a given interface or subclass a given class and use the jMock API to concisely describe the expected method sequence. Figure 2-3 tests the same property as Figure 2-2, but with much less code and no need to define custom `Account` implementations. The test first creates two mock objects (`from` and `to`). The `checking` block[1] defines two expectations: a `deposit` call on `from` and a `withdraw` call on `to`, both with

---

[1]jMock consists of an underlying API for defining expectations and a syntactic sugar layer for describing them. The sugar layer includes unusual constructs such as the double-{ block seen on line 13 of Figure 2-3: the expectations are actually being defined inside the initializer of an anonymous subclass of `Expectations`.

```
1 public interface Account {
2     public void deposit(Money m);
3     public void withdraw(Money m);
4 }
5
6 public class PortfolioTest extends TestCase {
7     static class InteractionBasedDepositAccountStub implements Account {
8         private boolean gotExpectedDeposit = false;
9
10         public void deposit(Money m) {
11             assertFalse("deposit_called_more_than_once", gotExpectedDeposit);
12
13             assertThat(m, equals(20));
14             gotExpectedDeposit = true;
15         }
16
17         public void withdraw(Money m) {
18             fail("unexpected_method_call_'withdraw'");
19         }
20
21         public void assertSatisfied() {
22             assertTrue("deposit_not_called", gotExpectedDeposit);
23         }
24     }
25
26     static class InteractionBasedWithdrawAccountStub implements Account {
27         // [ implementation elided ]
28     }
29
30     public static void testTransfer() {
31         InteractionBasedWithdrawAccountStub from
32             = new InteractionBasedWithdrawAccountStub();
33         InteractionBasedDepositAccountStub to
34             = new InteractionBasedDepositAccountStub();
35
36         Portfolio portfolio = new Portfolio(from, to);
37
38         portfolio.transfer(from, to, 20);
39         from.assertSatisfied();
40         to.assertSatisfied();
41     }
42 }
```

Figure 2-2: A behavior-based JUnit test using manually-written stubs instead of mock objects.

```
1 public interface Account {
2     public void deposit (Money m);
3     public void withdraw (Money m);
4 }
5
6 public class PortfolioTest extends MockObjectTestCase {
7     public static void testTransfer () {
8         Account from = mock (Account.class);
9         Account to = mock (Account.class);
10
11        Portfolio portfolio = new Portfolio (from, to);
12
13        checking (new Expectations () {{
14            one (from).withdraw (20);
15            one (to).deposit (20);
16        }});
17
18        portfolio.transfer (from, to, 20);
19    }
20 }
```

Figure 2-3: A behavior-based JUnit test equivalent to Figure 2-2, using jMock instead of custom stubs.

20 as the sole argument. The "one" indicates that each method call is expected exactly one time. Finally, the test runs the tested method `transfer` itself. When the implementation of `transfer` calls `withdraw` and `deposit` on the mock objects, jMock checks that the argument is 20, throwing an exception otherwise. jMock complains if an unexpected call is made on a mock object, and at the end of the test method automatically checks that all expectations have been satisfied. It is much easier and clearer to define a behavior-based test using jMock than by hand.

State-based tests require the use of a concrete implementation of the fixture objects, whether a standard implementation or a stub. The state-based test in Figure 2-1 relies on the correct operation of both `Portfolio` and `CheckingAccount`. If the test fails, the bug could potentially be in either class. Additionally, the test could erroneously pass because of a pair of defects in the two classes which cancel each other out; and it could pass even if the `transfer` method also inappropriately calls extra methods on one of the accounts which don't affect the balance. It may also be the case that all of the "real" implementations of a fixture type are complex and require much overhead to set up; for example, it's hard to believe that a `CheckingAccount` could really be created with just an initial balance, since a real checking account has many other attributes that vary from one customer to another. A developer writing

27

a state-based test must either go through the effort to initialize the complex implementation in each test, or write a custom stub `Account` implementation. The latter choice has the downside that custom stubs are yet another place where bugs can hide.

Behavior-based testing allows the developer to test each class independently of its environment. The jMock test in Figure 2-3 does not run any code except for `Portfolio` and the expectations are defined succinctly in the test method itself. It is impossible for a bug in an `Account` implementation to cause `testTransfer` to fail or to erroneously pass, because no `Account` implementation is used. And we can verify that `transfer` does not affect the two accounts in any other way than the expected `withdraw` and `deposit` calls.

It can be difficult to write clean behavior-based tests for all Java programs; modules with complicated and ad hoc ways of interacting with their environment require equally complicated mock objects. In fact, some proponents of behavior-based testing claim that its strongest advantage is encouraging clean OO design, as it works best with programs that follow OO design guidelines such as the "tell, don't ask" principle and the "Law of Demeter" [20, 26, 18]. These developers view behavior-based testing and the use of mocks as a design tool as much as a testing tool: classes that are intertwined enough to be difficult to test with mock objects are often difficult to extend in other ways.

Because behavior verification does not require knowledge of the internal state representation of the objects in the test fixture, unit tests with behavior verification make an ideal target for automated test factoring. The entire purpose of test factoring is to separate testing a single object from testing an entire system, which meshes nicely with the fact that mock-based tests run no code outside of the SUT. Guessing how the fields of the fixture objects represent the abstract state of the environment is difficult if not impossible, but observing the method calls that the SUT makes on its environment is relatively straightforward. The behavior of the SUT during the execution to be factored is much more well-defined than the state of the entire fixture.

2.2   Writing tests with jMock

In addition to being more succinct than manually-defined stub objects, jMock provides simple syntax to define more complex expectations. jMock allows test writers to customize when an expectation matches a given invocation; define actions that occur when an expectation is matched; and provides higher-level mechanisms for tying multiple expectations together. Because jMock consists of a syntactic layer on top of an extensible API for defining expectations, developers can add more possibilities themselves; some of the features described here come with jMock, and others are extensions provided by `amock`. More information about jMock can be found in [16, 18, 17, 22].

2.2.1   Expectations

Figure 2-3 only showed the simplest form of jMock expectation declaration: single independent calls with fixed arguments. jMock allows test writers to state how many times methods should be called, customize how arguments are matched, and enforce ordering constraints on arguments. These features are demonstrated in Figure 2-4.

Test writers can declare that expected methods should be invoked any number of times. Expectations can be declared as having `one` invocation, `atLeast` or `atMost` a given number of times, or `between` two amounts. They can occur any number of times including zero using either the `allowing` or `ignoring` declaration (depending on taste). Finally, they can be declared to `never` occur, though this is mostly a matter of documentation, because invocations that are not explicitly expected result in test failure anyway. Test writers can extend jMock to provide custom cardinalities such as `anEvenNumberOfTimes`.

In addition to just listing method arguments in the expectation (which are compared to the received values using `equals`), test writers can specify other ways to compare arguments by writing **matchers**. The matcher interface is provided by the Hamcrest[2] library [19]. To define an expectation with parameter matchers, each ar-

---

[2] "Hamcrest" is an anagram of "matchers".

```
1 public class ExpectationsTest extends MockObjectTestCase {
2     public static void testCooking () {
3         Bowl bowl = mock(Bowl.class);
4         BakingDish pan = mock(Pan.class);
5         Sequence baking = sequence("baking");
6         // [ other mocks elided ]
7         Cook cook = new Cook(/* ... */);
8
9         checking(new Expectations() {{
10             atLeast(2).of (bowl).add(milk);
11             between(4, 6).of (bowl).stir ();
12
13             allowing (bowl).smell ();
14             ignoring (bowl).spin ();
15             never (bowl).spill ();
16
17             one (bowl).add(with(any(Flour.class)));
18
19             one (bowl).pourInto(pan);
20             inSequence(baking);
21
22             one (pan).bake();
23             inSequence(baking);
24         }});
25
26         cook.makeCookies();
27     }
28 }
```

Figure 2-4: Examples of jMock expectation declarations.

gument in the declaration must be a matcher, enclosed in a `with` call, as in line 17 of
Figure 2-4. Here, the `any` method returns a matcher which matches any object of the
`Flour` class or a subclass. Hamcrest and jMock come with several useful matchers
and matcher combinators, and test writers can define their own custom matchers as
well.

By default, expected invocations may occur in any order, but jMock provides
several ways to specify otherwise. If a series of expectations must be satisfied in a
particular order, tests can create a sequence (line 5 of Figure 2-4) and declare that the
expectations lie in it (lines 19 through 23). The expectations may even have different
receivers. jMock also allows the test writer to state that certain expectations can
only happen during certain "states" and to define a post-invocation for expectations.

2.2.2   Actions

As well as making sure that the right method calls are made, jMock allows the test
writer to declare *what happens* when methods are invoked on mock objects. Tests can

30

```
1 public class ExpectationsTest extends MockObjectTestCase {
2     public static void testCooking() {
3         Bowl bowl = mock(Bowl.class);
4         // [ more setup elided ]
5
6         checking(new Expectations() {{
7             allowing (bowl).getVolume();
8             will(returnValue(1000));
9
10            allowing (bowl).getContents();
11            will(returnIterator(flour , milk , eggs));
12
13            allowing (bowl).pour(with(any(BakingDish.class)));
14            will(onConsecutiveCalls(returnValue(true),
15                                    throwException(new EmptyException())));
16
17            one (bowl).emptyIngredientsIntoCollection(collection);
18            will(doAll(new CallBack() { public void go() {
19                collection.add(flour);
20                collection.add(milk);
21                collection.add(eggs);
22            }), returnValue(3)));
23        }});
24
25        cook.makeCookies();
26    }
27 }
```

Figure 2-5: Examples of jMock action declarations.

specify one of several predefined **actions** that may occur when a mocked method is invoked, or can trigger custom actions. Actions are declared inside a `will` call, after the expectation declaration itself. Various actions are demonstrated in Figure 2-5.

The simplest actions are to return a given value or throw an exception. If no action at all is given for a non-`void` method, it will return 0, `null`, `false`, or the empty string, as appropriate (and the test writer can customize default results by class). The `returnValue` action specifies a specific return value for the method to return. The `returnIterator` method is a convenient shorthand for returning an iterator over a sequence of values (and a new iterator will be created each time the expected method is invoked). The `throwException` action makes the method throw the specified exception.

If the mocked method needs to make callbacks on the SUT, a custom callback action can be written. An example is shown on lines 18 through 21 of Figure 2-5; here, when the SUT calls `bowl.emptyIngredientsIntoCollection(collection)`, the test driver will `add` three items to the `collection`. (The syntax is a little clunky

31

because Java does not support closures well.) There is no need for special support for "nested" callbacks. Say that the SUT calls `A` on a mock, which needs to call `B` on the SUT, which will call `C` on a mock, which itself needs to call `D` on the SUT. This can be expressed as a pair of expectations: one expecting `A` and with a callback for `B`, and the second expecting `C` and with a callback for `D`; ordering constraints can be used to make sure they invoked at the right time.

Multiple actions can be combined into one single action. The `onConsecutiveCalls` method creates an action that runs a different action on each invocation of an expectation. The `doAll` action creates an action that executes a series of actions in order; this is useful for making an expectation that first invokes callbacks and then returns a value, for example.

2.2.3   Test structure

Expectations are declared with calls to the `checking` method in the body of the unit test. If a test has multiple `checking` blocks, the expectations are just combined, and all expectations are automatically verified at tear-down time. "Verify" in this context means to check for unsatisfied expectations ("errors of omission"); unexpected invocations ("errors of commission") result in errors as soon as they occur. The exercise phase in the tests that `amock` generates can contain several method calls on the SUT. Ideally, expectations should be verified after each exercising method call, so a single tear-down verification stage is not appropriate. `amock` thus provides an alternative to `checking` blocks called `verifyThenCheck` that verifies the previous set of expectations before adding more.

When expectations fail, the user needs to be able to understand the failure. All of the important interfaces in jMock and Hamcrest implement a `SelfDescribing` interface, which gives a method for concisely describing themselves. So when a failure occurs, jMock outputs a compact legible description of all of the passing and failed expectations, including how many times they were invoked.

While it's easy to refer to the elements of the test fixture in expectation declarations, it's slightly trickier to refer to objects created by the SUT and passed as argu-

```
1 public void testCaptures() {
2     final Capture<String> seenTwice = capture(String.class);
3     final Receiver r = mock(Receiver.class);
4
5     checking(new Expectations() {{
6         one (r).getIt(with(a(String.class)));
7         will(seenTwice.capture(0));
8
9         one (r).getAgain(with(valueCapturedBy(seenTwice)));
10
11        one (r).whatWasIt();
12        will(returnValueCapturedBy(seenTwice));
13    }});
14
15    // Exercise phase.
16    String random = "I chose: " + new Random().nextInt();
17    r.getIt(random);
18    r.getAgain(random);
19    assertThat(r.whatWasIt(), is(random));
20 }
```

Figure 2-6: Example usage of amock's Capture feature.

ments to methods in the fixture. It is impossible to directly say `one (obj).method(fooCreatedBySUT)` because there's no variable `fooCreatedBySUT` in the test method! One attempt to solve this problem would be to write `one (obj).method(with(any(Foo.class)))`, but this does not work if the same object is used more than once: passed to a mocked method twice, or passed to a mocked method and later returned from another mocked method, say.

amock provides a class `Capture` that allows test writers to deal with this situation. To capture an object of class `Foo`, declare a `Capture<Foo>` object, and use its `capture` method which returns an action that captures the argument at the specified position. Later expectations can use the `valueCapturedBy` matcher to ensure that the same value is passed to the mocked method, or use the `returnValueCapturedBy` action to return the captured value back to the SUT.

Chapter 3

Test factoring with `amock`

`amock` is a three-phase system: capture, factor, and replay. Section 3.1 describes describe how `amock` implements the capture phase of test factoring. Section 3.2 describes the factorizing processor which produces a suite of JUnit tests from each trace. Section 3.3 briefly describes `amock`'s replay phase. Finally, in Section 3.4 we describe the limitations of `amock`'s design and implementation.

This chapter contains the core of `amock`'s technique. However, we found that several additional heuristics were necessary to produce tests of a reasonable quality. To not distract from the main concepts, these enhancements are described later, in Chapter 4.

3.1 Capturing a system test

The first phase of test factoring is producing a transcript, or **trace**, of a system test. `amock` instruments the bytecode of the system test using the Java programming language agent framework and the ASM bytecode manipulation library [3]. The instrumented version of the system test produces a trace of events and serializes the trace to disk[1].

---

[1] `amock` can serialize objects either as XML (using XStream [42]) for ease of debugging or using Java's native serialization for efficiency.

3.1.1 Data harvested during the capture phase

The capture phase outputs three pieces of data: the trace, the instance information database, and the hierarchy file. The instance information database is only used for the extra heuristics described in Chapter 4 and is described there; this section describes the other two files.

The `amock` trace

The trace file is a list of events. Each event may refer to several **trace objects**; a trace object is either a primitive or an instance. Primitives (including strings) are recorded in the trace file with their actual value. Instances are recorded with their (dynamic) class name and a serial number; the serial numbers are tracked inside the trace mechanism with a identity-based hash table which weakly references its keys.

The follow events are logged:

Pre- and post-method invocations All method call sites are instrumented to produce pre-call and post-call events. Each call receives a unique serial number which is shared by the pre-call and post-call event. Both types of event contain the method name, descriptor, declaring class, and receiver. The pre-call event contains the arguments, and the post-call event contains the return value. The receiver, arguments, and return value are all stored as trace objects.

Method entry and exit Method entry and exit are also instrumented. The trace events contain the same information as at the call sites; however, the class name logged is of the *implementation* of the method, not of the static type it is invoked on. Note that while entry and exit events share a call serial number (just like the pre-/post-call events), there is no connection between the two types of serial number. The reasons for using both types of log entry are described in Section 3.1.2.

Field reads Every field read (instance or static) is logged; the event contains the receiver, field name and type, and value. (Both receiver and value are stored as

trace objects.)

Static initialization Static initialization methods are not instrumented (we have
no interest in trying to generate unit tests that trigger class initialization at
specific times), but the entry and exit from static initialization methods are
logged so that any other code run between them can be ignored.

After the trace has been generated, it is post-processed to add some information
that allows the factorization phase to only make one pass over the trace per generated
test. First, all entries which take place during static initialization are removed[2]. Next,
pre-call and method-entry events for constructors have their receivers fixed: when
constructors are called, the uninitialized receiver cannot be examined directly by the
trace code without halting the JVM, so there is no way to log it as a trace object (with
serial number) until it has been constructed. This pass matches up constructor post-
call and method-exit events (which do contain trace object records for the receiver)
with their pre-call and method-entry events and inserts the correct instance record
into it. Finally, method-entry and pre-call events are matched up, as will be described
in Section 3.1.2.

The hierarchy file

The `amock` processor avoids using reflection on the subject code, because loading the
subject code could run static initialization in the subject code or have other side
effects. In order to get the information about the loaded classes which the processor
needs, the instrumentation also outputs a file containing information (superclasses,
interfaces implemented, access control, etc.) about the class hierarchy of all of the
instrumented code.

---

[2]This should probably be implemented by just not logging them in the first place, saving time
and disk space; the choice to trim in a later pass made it easier to debug when the logic which
matched up pairs of static initialization enter/exit methods failed, but all such bugs have long been
fixed.

3.1.2 Dealing with uninstrumentable code

The `java.lang.instrument` framework does not allow `amock` to instrument the standard JDK libraries. Additionally, instrumentation is impossible for libraries of native code (in the JDK or elsewhere). We assume that `amock` users are trying to test their own Java code, not the JDK, so this is not a major drawback. `amock` relies mostly on pre- and post-call events, so calls from the (instrumented) developer's project to the (uninstrumented) JDK are logged appropriately. However, when JDK code calls methods in the developer's project, no pre-/post-call events will be generated. For example, if a developer has written a custom handler for one of the XML parser libraries in the JDK, the methods on the custom handler will only be called from the JDK and thus pre-/post-callback events would be lost. While one could require the user to instrument the Java runtime environment's "bootstrap" jars ([35] does require this), this would make the tracer much more complicated to run: currently it just requires adding a single `-javaagent` argument to the command line, instead of replacing part of the installed JRE on the filesystem. Additionally, not instrumenting the JDK makes it easier to write the `amock` tracer without having to worry about using instrumented code inside the instrumentation itself.

In order to deal with callbacks from uninstrumented code, `amock` logs method entry and exit events as well as pre-/post-call events. After the trace is created, `amock` identifies method entry and exit events that do not have corresponding pre-/post-call events, and removes all others. From this point on, the remaining entry/exit events are treated identically to pre-/post-call events.

3.1.3 Dealing with reflection

`amock` has limited support for recognizing the use of the Java reflection API. Reflection is often a source of difficulty for Java program analyses, because it allows the operands of standard operations (instance creation, method invocation, field access, etc) to be resolved at run time instead of at compile time. However, a dynamic analysis like `amock` can handle reflection intelligently.

`amock` currently recognizes reflective instance creation through the `newInstance` method on `Class` objects, and logs the appropriate constructor event. It would be straightforward to extend `amock` to similarly recognize reflective method invocation, field accesses, and so on.

3.2   Factoring into unit tests

3.2.1   The factorizing finite state machine

`amock` creates tests in a high-level intermediate representation tailored specifically to behavior-based unit tests. The use of this domain-specific representation (as opposed to immediately rendering everything to strings of Java statements, say) allows `amock` to generate a unit test with a single pass through the trace, and then optimize them if necessary.

When processing the trace, `amock` translates each instance encountered in the trace into a **program object**. In doing so, `amock` determines which instances belong to the system under test (**SUT objects**) and which belong to the environment (**mock objects**). Note that this is more complicated than just declaring that objects in a certain class or package are SUT objects and the others are mocks (like in [35]). Specifically, the SUT objects are the original object specified by the user to factor out a test for, and any other object constructed by a `new` expression or returned from a JDK static method (which is unmockable) during the execution of a method on any SUT object. Every other instance is in the environment, and if it needs to be referred to in the generated unit test, it is represented by a mock object.

The factorizer is a state machine which reads through the trace, processing each entry, and builds up a test as it goes. The internal representation of the test tracks all of the **program objects** which the test refers to, all of the descriptions, and all of the invocations exercising the system under test (and the assertions on their return values). The states of the processor FSM and the transitions between them are shown in Figure 3-1.

START ➡ Environment (outermost)

initial SUT object constructed OR method call on SUT object

constructor call

method call on mocked object OR non-JDK static method call

SUT

JDK static method call

Unmockable static method

Environment (mocked)

method call on SUT object

Legend

FSM state

description of event

pre-call transition

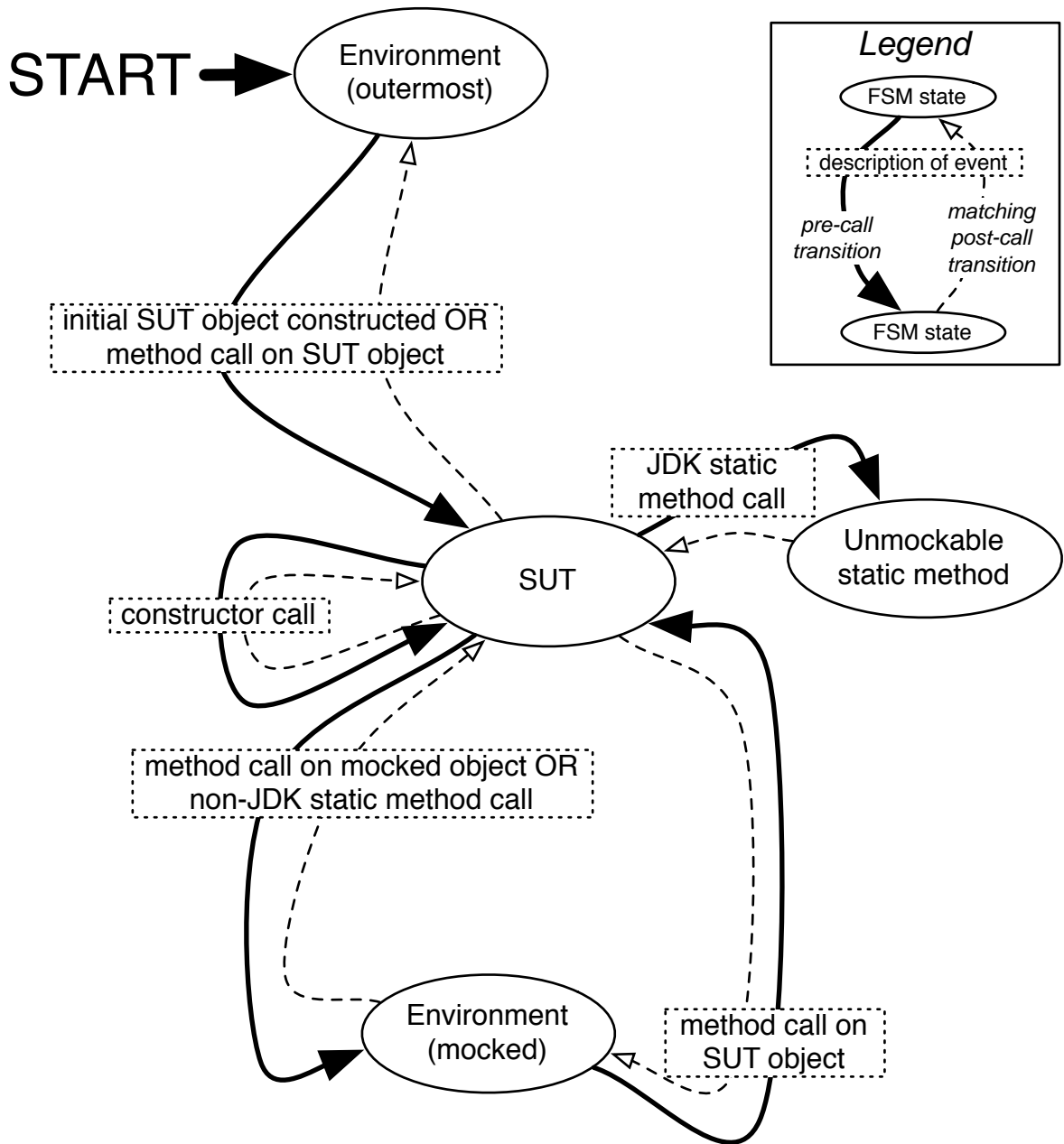matching post-call transition

FSM state

Figure 3-1: The amock factorizing processor can be viewed as a finite state machine. The figure shows its states and transitions. When the processor reads a pre-call event from the log, the FSM may transition along one of the solid edges; when its matching post-call event is read, the FSM transitions along the matching dashed edge.

The processor processes one event at a time. Pre-call events may trigger the transitions between states shown by solid edges in Figure 3-1. The matching post-call event triggers the transition shown by the corresponding dashed edge. For example, in the "Environment (outermost)" state, the pre-call for the constructor of the specified tested object or the pre-call for any method call on a SUT object will cause a transition to the "SUT" state. Assuming that the trace is well-formed, when `amock` reaches the post-call that matches the pre-call which caused the "Environment (outermost)" to "SUT" transition, it will have returned to the "SUT" state (perhaps without having left); the post-call then triggers `amock` to follow the dashed edge back to the "Environment (outermost)" state. Essentially, the states are on a stack: relevant pre-calls push a new state onto the stack, and their matching post-call pops it off again. (Note that this is even true for the constructor call event in the "SUT" state: when the self-loop transition is taken, a second "SUT" is pushed onto the stack, and popped off again when the constructor finishes.)

3.2.2 Example

Consider a partial trace containing the following events (we use the notation `ClassName@id` to denote a specific instance of class `ClassName`):

1. pre-call: `new CookieJar@42()`

2. pre-call: `new ArrayList@43()`

3. post-call: `new ArrayList@43()`

4. post-call: `new CookieJar@42()`

5. pre-call: `CookieJar@42.add(Cookie@44)`

6. pre-call: `ArrayList@42.add(Cookie@44)`

7. post-call: `ArrayList@42.add(Cookie@44)`

8. post-call: `CookieJar@42.add(Cookie@44)`

41

9. pre-call: `CookieJar@42.getACookie()`

10. pre-call: `ArrayList@43.isEmpty()`

11. post-call: `ArrayList@43.isEmpty()` returns `false`

12. pre-call: `ArrayList@43.remove(0)`

13. post-call: `ArrayList@43.remove(0)` returns `Cookie@44`

14. post-call: `CookieJar@42.getACookie()` returns `Cookie@44`

A developer invokes `amock` to construct a test for `CookieJar@42`. The finite state machine starts in the "Environment (outermost)" state, and performs the following operations:

pre-call: `new CookieJar@42()` This is the initial SUT object constructor, so `amock` transitions to the "SUT" state and creates a SUT-object program object to represent `CookieJar@42`.

pre-call: `new ArrayList@43()` This is a constructor call, so `amock` follows the self-loop from "SUT" to itself and marks `ArrayList@43` as an internal SUT object.

post-call: `new ArrayList@43()` This is the matching post-call event to the previous event, so `amock` follows the corresponding dashed self-loop edge (and remains in "SUT").

post-call: `new CookieJar@42()` This matches the first pre-call event, so `amock` transitions back to "Environment (outermost)".

pre-call: `CookieJar@42.add(Cookie@44)` This is a method call on a SUT object, so `amock` transitions to the "SUT" state and begins building a SUT method execution for the method call `add`. `amock` determines that `Cookie@44` is in the environment and represents it by a mock object.

pre-call: `ArrayList@42.add(Cookie@44)` This is neither a constructor call, a static method call, or a call on a mock object: it is just the internal implementation of `CookieJar.add` that is being tested. `amock` makes no transitions.

post-call: `ArrayList@42.add(Cookie@44)` Similarly, no transitions are necessary.

post-call: `CookieJar@42.add(Cookie@44)` `amock` follows the dashed edge back to "Environment (outermost)" and finishes building the SUT method execution; because `CookieJar.add` is a `void` method, it does not need to create an assertion for the SUT method execution.

pre-call: `CookieJar@42.getACookie()` This is a method call on a SUT object, so `amock` transitions back to the "SUT" state and begins building a SUT method execution for the method call `getACookie`.

pre-call: `ArrayList@43.isEmpty()` This does not cause a transition.

post-call: `ArrayList@43.isEmpty()` **returns** `false` This does not cause a transition.

pre-call: `ArrayList@43.remove(0)` This does not cause a transition.

post-call: `ArrayList@43.remove(0)` **returns** `Cookie@44` This does not cause a transition.

post-call: `CookieJar@42.getACookie()` **returns** `Cookie@44` `amock` transitions back to "Environment (outermost)", and creates an assertion verifying that the `getACookie` execution returns the mock object associated with `Cookie@44`

3.2.3 State details

The processor starts in the "Environment (outermost)" state. It waits until it sees that the object specified by the user as the initial SUT object is being constructed. `amock` then transitions to the "SUT" state, gathering data for what arguments the object's constructor takes as it goes. Once the constructor finishes, `amock` returns

to "Environment (outermost)" and waits for method calls on any SUT object; these method calls are turned into "SUT executions" with assertions on their return value.

The events inside the "SUT" state correspond to the code that is actually being tested. Constructors cause `amock` to transition to a nested "SUT" state (as shown in Figure 3-1 by a self-loop) and mark the object being constructed as a SUT object. Method calls on mocked objects (that is, any object not known to be a SUT object) or static methods in neither the current class or the JDK cause `amock` to transition to the "Environment (mocked)" state. This state represents the code that should *not* be executed during test replay, so `amock` ignores everything until the method returns, and use the arguments and return value to create an expectation. The one exception is that method invocations on SUT objects during "Environment (mocked)" cause `amock` to transition to a nested "SUT" state; these methods are recorded as callbacks in the expectation. Finally, static method calls that cannot be mocked (see Section 5.1 for details) go to a special state, ignoring the events that occur during it but making sure that the value returned from the static method is marked as being a SUT object.

When a field read event on a mock object is encountered during the "SUT" state, `amock` needs to make sure that the field has the appropriate state during the unit test execution. `amock` handles this by adding a "callback"-type action (called `TweakState`) to the *previous* expectation which manually sets the field to the expected value. This is only effective if the field has a consistent value over the course of the SUT method, though. A related heuristic is described in Section 4.2.

When the processor finishes its single pass over the trace, it has generated a unit test. Each pass through the "Environment (mocked)" state creates an expectation; each pass through the "SUT" state creates an assertion (top-level if it came from "Environment (outermost)", and a callback if it came from "Environment (mocked)"). The internal representation of the test tracks all of the mock object and SUT objects required, with their declarations at the appropriate spots. Capture declarations (see Section 2.2.3) are inserted for any objects created by the SUT, passed to the environment, and referred to again. One pass over the generated test eliminates unnecessary

44

declarations of objects that are only used once.

3.2.4   Writing the test

After constructing the internal representation of the generated test, `amock` analyzes each mock object to decide what class it should be declared as in the Java code. It takes the least upper bound of all the static types that the object is used as during the generated test, and sets the class of the mock object to be that class, instead of using the actual class of the instance that it is based on. This is so that the generated test will refer to (for example) `Iterator`, not `ArrayList$Iterator`.

Finally, the processor prints out the generated test as Java source code. It resolves class and variable names at this point, so that classes used in the code can be `import`ed and the main text of the test can be free of long package names. Care is taken to appropriately indent code, separate expectations from each other with blank lines, and generally to create legible code.

The generated test is by default in the same package as the primary SUT object, though the user can specify a different package. All methods exercised by the test case and all methods expected to be invoked must be accessible from the test case. This choice of package placement allows it to access protected and package-private methods on the primary SUT object. Because the generated test is a unit test and mostly focuses on the single object, it is likely that there is only one package that the test needs to access protected members of. That is, being in the same package as the initial SUT object allows you to call protected methods on it (and private methods won't be exercised by the environment anyway), and it shouldn't be able to make callbacks on any objects it does not have access to. The main hole here is that the SUT object may make protected method calls (either static, or on mock objects of the same class) on protected superclass methods in a different package; the generated test will not be able to call them.

3.3   Replaying and enhancing generated unit tests

amock's replay phase is straightforward: compile the generated tests with a standard Java compiler, and replay them with any JUnit test harness (command line test runner, IDE plug-in, etc). Note that if the test needs to mock any static methods, the test must be run with the smock instrumentation; see Section 5.1 for more details. If the project already has a JUnit test suite, the tests may be added to it; nothing about amock-generated tests requires them to be separated from manually written tests.

Because the tests are just standard Java code, they can be modified and refactored just like ordinary JUnit tests. For example, a developer might use amock to create a unit regression test for a method foo when called with a certain argument. Now that amock has taken care of writing a test with all of the setup required to test foo(0), the developer can refactor the test (either by hand or with an automatic refactoring tool) to make it easy to write tests for foo(1), foo(-1), foo(Integer.MAX_VALUE), and so on. (Such refactoring would not be possible for tests that are not expressed as code, at least not without requiring domain-specific tools.)

It may be possible to automatically enhance amock-generated tests, either with source-level or higher-level analysis. For example, a system test may use an enormous number of instances of the same class during its execution, but only exercise them in a small number of essentially different ways. amock can be directed to generate one test case for each instance, but this will create a massively redundant test suite, and a single code change will lead to a huge number of failures, making it difficult for a developer to know where to start investigating. Code similarity analyses could help to weed out duplicates, or to automatically refactor common parts out of nearly-redundant tests.

3.4 Limitations

While `amock` successfully creates tests for real-world projects (see Chapters 6 and 7 for more details), it can not always successfully factor every test with respect to every run-time object. This is due to a combination of conceptual limitations with the concept of factoring executions into JUnit tests, and implementation deficiencies in the current `amock` prototype implementation.

3.4.1 Conceptual limitations

Java access control features can prevent both developers and `amock` from creating mock-based tests for some classes. If reproducing the behavior of the system test requires running protected methods or accessing protected classes in multiple packages, there is no straightforward way to write a single test which can access all of the methods and classes. If the test would require mocking a `final` class, jMock will not be able to properly mock the class. (There is a relatively straightforward workaround for this: the attempt to mock `final` classes fails at run time, not compile time, and so you can use the `java.lang.instrument` framework to remove the `final` modifier from classes and methods as they are loaded. The library Definalizer [12] implements this in 39 lines of code (including `import` statements, blank lines, etc). `amock` could choose to run tests under Definalizer when necessary.)

Arrays present a difficulty for behavior-based testing. Arrays have some characteristics that make them more like objects than like primitives, such as being mutable and not necessarily having a literal representation (if their base type is not primitive). However, they are not actually objects, and there is no way to mock them. `amock` would need to create an actual array of the proper length and set its entries to the expected values ahead of time. There is no way to declare that it expects certain elements of the array to be modified, except for by checking that they end up in the expected state at the end of the method call.

`amock` does not handle all cases of field access across the factorization boundary: if the SUT needs to access fields in the mocked environment and the fields change values

frequently, the generated test may not be able to present the correct field values to the SUT. By, default jMock only allows you to create mock objects based on interface types (which do not have fields); the assumption of the jMock developers is that clean code communicates over well-defined method interfaces, not via direct field access. In a concession to the need to test legacy code, though, jMock does allow you to mock concrete classes (using the Objenesis library [30]), but even here it's clear that they frown on such uses, as the code to mock concrete classes is distributed in a separate `jmock-legacy.jar` archive. Of course, `amock` needs to be able to mock concrete classes, and so it is possible that the SUT code does try to access the fields of the mock objects. When manually writing tests for a SUT which directly accesses fields in its environment, the technically best solution is most likely to change the field access into a (mockable) accessor method call. `amock` cannot change the tested code, so instead it works around it by trying to set up fields to have the right value before the SUT accesses it. This is effective (albeit ugly) if the value does not change more often than the test can update it, but fails under more complex access patterns. In addition, the fact that this is implemented by attaching a callback action to the *previous* expectation means that relaxation of ordering constraints can fire the callback at the wrong time.

Not all code is testable, so in general, we can not expect `amock` to be able to create fully-isolating tests for every instance in every execution. Much code is difficult to write tests for by hand, let alone automatically. In fact, the authors of jMock recognize this [18]; they argue that jMock should be thought of as being a design tool as much as a testing tool. That is, they believe designing classes with testability in mind (and specifically, testability via behavior verification) leads to better code *and* better tests. (This explains why, for example, the jMock authors have no interest in allowing users to mock static methods[3]: code that relies on the behavior of static methods should in their view be rewritten to not rely on hard-coded global methods.)

---

[3]We implemented this, as described in Chapter 5.

3.4.2 Implementation limitations

The current implementation of `amock` has some additional limitations which have not been solved at this point. Some of these would be relatively straightforward to fix; others would require deeper changes to the implementation.

When generating tests that use arrays, `amock` may create incorrect code which fails to compile. Specifically, if an array must pass over the boundary between the SUT and the mocked environment, and thus must be explicitly mentioned in the generated test code, `amock` currently generates code which does not even compile. (An array that is used entirely within the SUT or entirely within the mocked environment causes no problems.)

`amock` does not recognize the use of multiple threads. The trace writer is synchronized so that multi-threaded programs will not corrupt the format of the trace, but events from multiple threads will be overlapped. This can cause the finite state machine to end up in an inconsistent state. It would be straightforward to add a thread identifier to each event; the processor could then only pay attention to events in the thread in which the original SUT object was constructed, or use a similar heuristic. This would enable it to generate correct unit tests if the SUT is only being accessed in one of the threads. (For example, if there is a UI thread and a logic thread, this would effectively create tests for UI elements or internal logic objects even if unrelated events are occurring in the other thread, as long as the specific code being tested doesn't depend on inter-thread communication.)

If the SUT code runs an `instanceof` check or a checked cast on the object, it may behave differently from the instance it is simulating. As described in Section 3.2, mock objects are declared as the most general type which is a subtype of all types that the mock is used as during the generated test. This means that, for example, mocks will be declared as mocking a public interface instead of as mocking a private implementing class. This means that the mock object will not necessarily be a subtype of the type of the instance that it is simulating. We have not found this to cause problems in practice. This could be solved by instrumenting `instanceof`

checks and checked casts, and including the types seen there in the set of types which the declared type must be a least upper bound of.

While `amock` does support writing tests which capture objects created by the SUT and passed as arguments to expected invocations on mock objects (as described in Section 2.2.3), it only handles the most straightforward cases. If the captured object must then be expected by a later expectation, or is returned to the SUT code, `amock` can write the appropriate code. However, if (for example) an object *found in a field* of the captured object must be returned to the SUT, `amock` fails to recognize this condition, and can not generate a passing test. In order to fix this, `amock` would need a data flow analysis that remembers how to access each SUT object from the SUT objects that the test has immediate access to (i.e., the initially created SUT object, any captured SUT object, and any SUT object returned from a SUT method).

`amock` ignores the Java 1.5 generic type system; the generated tests will use raw types (and have compile-time warnings because of this). Because ASM does give access to generic information, it should be possible to dynamically discover the generic signature of many variables, but this is not implemented.

If the initial SUT object is created via a public static method which calls a private constructor, the generated test will erroneously try to use the private constructor to create the SUT object, which will fail to compile. This could be fixed by making the initial FSM state track static method calls in order to deal with this special case explicitly, instead of only looking for constructor calls.

Chapter 4

Heuristics to improve generated tests

The technique described in Chapter 3 will create JUnit tests that simulate the effect of system tests on individual objects. However, the basic technique does not always generate concise and readable tests. The generated tests could be lengthy, fragile, and hard to understand, because they express in minute detail concepts that would be rendered at a higher level of abstraction in manually-written tests. To improve the quality of generated tests, `amock` implements several heuristics that generate better tests for certain patterns in the code. This chapter describes the patterns recognized by `amock`: iterators (Section 4.1), record classes (Section 4.2), and final static fields (Section 4.3).

The heuristics rely on an instance information database, which is built from the trace before running the factorizing processor. In order for the processor to only make a single pass over the trace per generated unit test, it needs to be able to decide if a given instance matches one of the heuristic patterns as soon as it is first encountered. However, it may not be able to tell whether it fits the pattern until it has seen all of its interactions. Thus, `amock` creates a database with information about all of the instances seen in the trace; then during each execution of the processor (for each generated unit test), it can simply refer to the database to see what each instance does. Each instance info entry contains the set of methods invoked on the

instance, the set of fields read from the instance, and the list of static fields that the instance is ever read as a value from. (The instance information database is currently implemented as an in-memory hash table which is serialized to disk, but could easily be changed to an on-disk database for reasons of scalability.)

The classes to which the heuristics apply are configurable by the developer. In the current implementation, the developer must explicitly specify the classes that should be considered for heuristics, as well as some extra pattern-specific information. However, it would be possible for that configuration file to be automatically generated as the results of an analysis, or for the heuristic decision to be made automatically on the fly.

4.1   The iterator pattern

If an expected invocation returns an iterator which the SUT uses, the naïve algorithm of Chapter 3 will create a mocked iterator object and a long series of repetitive expectations. For example, if the iterator is over a sequence of three strings, `amock` would generate expectations like those shown in Figure 4-1. This is lengthy, repetitive, and much harder to tell what is actually happening than the equivalent (but fifteen times shorter) expectation in Figure 4-2.

Under this heuristic, when `amock` finds an iterator object that is not a SUT object and on which *only* the `hasNext` and `next` (or the equivalent for types other than `Iterator`) are invoked, then, instead of translating the instance into a mock object, it represents it by a special iterator-class program object. `amock` uses the instance information database to make this decision as soon as the iterator object is used in the generated test.

When in the "SUT" FSM state (of Figure 3-1) and methods are invoked on an iterator-class program object, the state machine goes to a new "Iterator invocation" state; if the call is `next`, its return value is remembered and added to an internal list which is use to construct the argument to `returnIterator` (as shown in Figure 4-2).

52

```
 1 will ( returnValue ( mockIterator ) ) ;
 2 inSequence ( s ) ;
 3
 4 one ( mockIterator ) . hasNext ( ) ;
 5 will ( returnValue ( true ) ) ;
 6 inSequence ( s ) ;
 7
 8 one ( mockIterator ) . next ( ) ;
 9 will ( returnValue ( " foo " ) ) ;
10 inSequence ( s ) ;
11
12 one ( mockIterator ) . hasNext ( ) ;
13 will ( returnValue ( true ) ) ;
14 inSequence ( s ) ;
15
16 one ( mockIterator ) . next ( ) ;
17 will ( returnValue ( " bar " ) ) ;
18 inSequence ( s ) ;
19
20 one ( mockIterator ) . hasNext ( ) ;
21 will ( returnValue ( true ) ) ;
22 inSequence ( s ) ;
23
24 one ( mockIterator ) . next ( ) ;
25 will ( returnValue ( " baz " ) ) ;
26 inSequence ( s ) ;
27
28 one ( mockIterator ) . hasNext ( ) ;
29 will ( returnValue ( false ) ) ;
30 inSequence ( s ) ;
```

Figure 4-1: Expectations generated by the naïve algorithm of Chapter 3 for a method returning an iterator over three strings.

```
 1 will ( returnIterator ( " foo " , " bar " , " baz " ) ) ;
 2 inSequence ( s ) ;
```

Figure 4-2: Expectation generated by the iterator pattern heuristic for the same test as Figure 4-1.

4.2   Record classes

The jMock authors [18] advise developers to only mock objects with interesting logic. A "record" class does not actively respond to messages, but is rather just a convenient aggregate for multiple pieces of data. Such a class doesn't have much logic, and should not be isolated from the test. `amock` does not mock these kinds of objects, leading to shorter, clearer code.

For example, when testing GUI code, there are many uses of `java.awt.Rectangle`. We feel confident trusting that this class is relatively bug-free, and so there is no good reason to mock it; it is much more straightforward to just include `new Rectangle(1, 2, 3, 4)` in a test than to mock the `Rectangle` and include many expectations on `getX`, `getWidth`, etc. Thus `amock` has a heuristic to decide to treat instances as record classes. Like with the iterator pattern, there is a configuration file that specifies which classes are record classes; it also specifies mappings between fields and constructor parameters, and between method return values and constructor parameters. When a method is invoked on a record-class program object (in the "SUT" state) it goes into a special record-class data gathering state; when that call returns, `amock` stores the return value in the slot of the constructor that it called. Similarly, when values are read from fields that correspond to constructor parameters, those values are used as parameters in the constructor.

4.3   Naming static fields

Java's equivalent of symbolic constants is `final` static fields. One would expect a unit test to use the same constants as the code it is testing; when these constants are objects, it would be unnatural and confusing to represent these constants by mock objects. For example, SVNKit (see Section 6.2) has a class `SVNRevision` to represent revision numbers; whenever a non-specific revision needed in the API, the static field `SVNRevision.UNDEFINED` is used. Without any special treatment of symbolic constants, `amock` would represent any use of `SVNRevision.UNDEFINED` with a mock

object and explicit expectations about the return values of `isValid` and `getNumber` on the mock object. It would make the test more natural and readable if it simply referred to the object as `SVNRevision.UNDEFINED`.

On the other hand, not all static fields should be referenced literally in the generated test. For example, if the environment passes `System.out` to the SUT code, then in the interests of isolation and testability, the unit test really should pass a mock `PrintStream` to the SUT instead of passing the real `System.out`.

`amock` could implement this feature using a data-flow analysis which tracks values from the time they are fetched from a static field until they are actually used in assertions and expectations, but instead it uses a much simpler heuristic. When `amock` builds the instance information database, it records the list of static fields that each instance is ever fetched from. When `amock` first translates the instance into a program object, it checks to see if any of the static fields that it ever is in is acceptable for this heuristic; if so, the generated test will refer to the value by its static field name instead of by value.

Like the other heuristics, there is a configuration file listing classes and fields that the heuristic applies to, and one could certainly automatically populate this file as the results of an analysis. A potentially useful guideline would be that this should be used for `final` fields whose values have an immutable type; it may be helpful to pay attention to the recommended Java style that static fields representing constants are named with all capital letters.

Chapter 5

Mocking static methods with `smock`

jMock provides no support for mocking static methods. This chapter describes a tool that we have created, `smock`, which extends jMock to allow test writers to mock static methods. `smock` can be used independently of `amock` when manually writing unit tests, and `amock` takes advantage of it when generating tests. `amock` was first implemented without the ability to mock static methods, but we found that this vastly decreased the quality of the generated unit tests. The entire purpose of `amock` is to generate tests which isolate the tested class from its environment. We discovered that without being able to mock static classes, this was impossible.

For example, when generating tests for the business logic of the Subversion network client in SVNKit [39], one would hope that the test could use a mocked version of the interface used to connect over the network to the server. However, the client code gets its repository access object by calling the static method `SVNClientManager.newInstance`; without somehow intercepting that call, there is no way to prevent the test from actually accessing the network. Before we wrote `smock`, the test generated for this high-level class would declare expectations referencing the low-level textual commands sent to the server! With `smock`, `amock` is now able to generate tests for this class whose expectations are at the same conceptual level as the code being tested.

In Section 5.1, we demonstrate how `smock` is used, and in Section 5.2 we describe its implementation. In Section 5.3 we consider `smock`'s efficiency. In Section 5.4, we explain why jMock does not itself support mocking static methods.

5.1 smock usage

Some extra setup is required to use smock. The smock jar must be loaded into the JVM as the Java programming language agent; with the command-line java program, pass the -javaagent:smock.jar argument. Additionally, instead of using org.jmock.Expectations and org.jmock.integration.junit3.MockObjectTestCase, you must use subclasses with the same names provided by smock (by changing the import declarations).

Declaring static method expectations with smock is very similar to declaring methods on mock objects. Wherever you would specify the mock object, you just write a class literal, like one (SVNClientManager.class).newInstance(). Everything else works just like jMock: arguments can be literals or matchers, you can declare an action for the expectation, and so on. The one major difference is the default behavior: if a static method is invoked which corresponds to no (unsatisfied) expectation, the method is executed as if smock does not exist instead of throwing an "unexpected invocation" exception. This allows the test writer to decided which static methods give too much access to the real environment and must be mocked out, and which are benign. One drawback to smock is that you can only mock static methods which are in instrumented classes; specifically, this means that you cannot mock static methods in JDK classes (and an attempt to do so currently causes undefined behavior).

5.2 smock implementation

smock is implemented as a layer on top of jMock. It does not need to reimplement any of jMock's expectation matching or dispatching logic; it simply builds expectations which have a different type of receiver from jMock's normal expectations (a new CapturingClass type instead of mock objects). Unfortunately, our implementation does require using a few classes which are part of jMock's internals, not its stable documented API.

The instrumentation adds a single call, Smock.maybeMockStaticMethod, to the

beginning of every static method in the program. The parameters are the names of the class and method, the method's descriptor, and the arguments passed to the function. `maybeMockStaticMethod` returns a result object with a "short circuit" flag and a short circuit return value. If the flag is set, the static method immediately returns the short circuit return value; otherwise it continues to the actual method code.

The point of communication between `smock` and jMock is the `CapturingClass` class. There is at most one `CapturingClass` object for each Java `Class` object. During the evaluation of expectation declarations, a call like `one(Foo.class)` looks up the `CapturingClass` for `Foo.class` and passes it to the jMock implementation of `one` as if it were a mock object. The jMock internals then pass an expectation builder to the `CapturingClass`, which it saves. Finally, `one(Foo.class)` returns null. The method `one` is declared with the generic type `public <T> T one(Class<T> cls)`. That is, the return type of `one(Foo.class)` is `Foo`! Because static methods can be invoked with an expression on the left-hand side of the dot (whose value is evaluated and ignored), this means that it is in fact legal to write `one(Foo.class).someStaticFooMethod()`. When `someStaticFooMethod` gets executed during this declaration, `Smock.maybeMockStaticMethod` notices that the `CapturingClass` for `Foo.class` is currently capturing expectations, and passes the current invocation to the expectation builder saved in the `CapturingClass`.

When `Smock.maybeMockStaticMethod` is called *not* during expectation declarations, it creates a fake `Invocation` on the `CapturingClass` and passes it to the main jMock invocation dispatcher. Because the expectations were registered with the `CapturingClass` as "mock object", the standard jMock expectation matcher will invoke the correct expectation; `Smock.maybeMockStaticMethod` then returns a short-circuit result with whatever the invoked action returns. `smock` traps the "unexpected invocation" exception and return a non-short-circuit result instead; as described in Section 5.1, unexpected static methods should just be executed.

5.3 Efficiency

`smock` instruments every static method in a program. `smock`'s overhead comes from two sources: every class must be instrumented when it is loaded, and every static method invocation triggers a call to the expectation dispatcher. Calls which do not match expectations cause an exception to be thrown and caught, and exception handling tends to be a relatively slow operation in most JVMs [6].

In order to analyze the slowdown caused by using `smock`, we ran the SVNKit [39] test suite with and without `smock` instrumentation. The experiment was performed on a 1800GhZ AMD Opteron with 8GB of RAM. The SVNKit test suite invokes the SVNKit command-line executables 239 times, causing the `smock` instrumentation to occur each time. SVNKit makes heavy use of static methods[1]. The slowdown in this experiment should thus be a conservative estimate. The uninstrumented test suite took 636 seconds to execute and the instrumented test suite took 782 seconds, revealing a 23% slowdown in test suite execution.

5.4 jMock and static methods

jMock does not support mocking static methods for a combination of philosophical and technical reasons. However, the philosophical reasons do not apply to automated test generation, and the technical issues are minor compared to the improvement in the quality of `amock`'s output when static methods can be mocked.

The authors of jMock view mock-based testing as a design tool as well as a testing tool [18], and allowing test writers to mock static methods is counterproductive in that view. jMock is designed to work best within the test-driven development mindset [4]. Here, the discovery that the SUT makes a static method call that should not be executed during testing should lead to eliminating the static method and replacing it with an instance method on some new type of object, perhaps provided to the SUT when it is constructed. This way, the test can simply mock the refactored object. A

---

[1]In fact, we were convinced that `amock` needed `smock` to creative effective tests while performing a case study on SVNKit.

second benefit is that a user of the SUT can determine by inspection that it needs to access the troublesome resource, instead of having the dependency hidden away. When jMock is used to write tests during the design process, its inability to mock static methods helps to improve the code being tested, at least in the philosophy of the jMock team. However, this is not much help for an automatic test generation tool such as `amock`: although the ability to mock static methods does not help test-driven development, it does help automatic test generation, where refactoring the SUT may not be an option.

Mocking static methods seems to require load-time bytecode instrumentation, which is significantly more invasive than the techniques used by jMock. jMock's mock objects are dynamically generated subclasses of the interface or class that they are mocking (created either with `java.lang.reflect.Proxy` or with Objenesis [30]), so all of the special mock code is in the subclass and does not need to be "injected" into the mocked class. This strategy won't work for static methods, because there is no straightforward way to change what code is run by an `INVOKESTATIC` instruction without changing the instruction itself or the body of the method which is invoked. `smock` solves this problem by instrumenting static methods at load time using the `java.lang.instrument` framework; however, using this framework requires running your JVM with a special "Java agent" argument, and may be inconvenient or impossible to use with test harnesses that are embedded in other programs, such as a IDE plug-in.

Chapter 6

Case Studies

In this chapter, we describe the results of applying `amock` to several real-world Java programs. We describe the characteristics of the individual programs, display examples of successfully generated tests, and explain what limitations (conceptual and implementation) of `amock` prevent it from complete success in all cases. (We discuss issues of brittleness, sensitivity, and efficiency later, in Chapter 7.)

For each subject library, we took a sample system test provided with the project or constructed our own sample execution. We ran `amock`'s capture phase, and attempted to factor and replay the trace for the first instance of every class observed in the trace. We then analyzed the results and subjectively classified the factored tests into the following categories:

Potentially useful A test that appears to usefully test non-trivial behavior of the specified instance. (Because we are not intimately familiar with the details of the subject programs, we cannot claim that a given test is definitely useful or not.)

Accessor-driven A test that constructs the specified instance, but only exercises it by calling getter and setter methods. While this does test real properties of the underlying objects, it is likely that a much simpler static analysis could produce similar tests.

Assertion-free A test that constructs the specified instance and calls one or more

project	observed classes	useful	accessor	no asserts	no exercise	invalid
JHotDraw	47	19	4	2	7	15
SVNKit	33	5	1	0	5	22
Esper	178	17	13	2	18	128

Figure 6-1: Summary of tests generated for case studies.

`void` methods on it which require no expectations. The only thing that such tests actually verify is that it does not crash.

Exercise-free A test that just creates the specified instance and does nothing else (because the given instance in the factored execution had no other method calls invoked on it). Such tests only verify that the constructor does not crash.

Invalid The factor phase fails to produce a test, the generated test fails to compile and pass, or the generated test is empty (because the construction of the specified instance is not recognized). The limitations which cause `amock` to generate invalid tests are described in Section 3.4.

We note that the three least useful categories of generated tests (assertion-free, exercise-free, and invalid) can be trivially automatically recognized, so `amock` could choose to not generate them and only generate tests of the first two types. Additionally, given some knowledge about what methods are accessors (based on name-based heuristics or a program analysis[8, 34, 29, 33, 37, 36, 2]), `amock` could attempt to separate the first and second classes. Thus, `amock` could produce only the factored tests that are most likely to be meaningful.

We performed the case studies on three open-source Java programs: JHotDraw, SVNKit, and Esper. Figure 6-1 shows the number of tests of each kind generated for each project. These include several application domains that are traditionally considered "hard to test", such as graphical user interfaces and network connections. JHotDraw and SVNKit were used as experimental subjects throughout the development of `amock`, and so many of the heuristics described in Chapter 4 were implemented as a direct response to deficiencies in the tests generated by early versions of `amock`. This does mean that positive results could represent overfitting to these particular

projects. On the other hand, `amock` has not (yet) been changed based on our experience with Esper, so this criticism would not apply to the Esper case study. We found in general that two types of tests were created: **general** tests, which exercise a part of the library in a relatively generic way, and **simulation** tests, which essentially replicate the high-level execution in JUnit form.

6.1 GUI: JHotDraw

JHotDraw [21] is a Java GUI framework for creating drawing applications. We performed experiments on the application JModeller [25] which uses the JHotDraw framework to create a UML-style object design application. JHotDraw and JModeller combined are 10K lines of Java code[1]. Neither JHotDraw nor JModeller comes with a test suite (unit or system). This means that we were unable to compare our generated unit tests to "real" unit tests; on the other hand, it means it is a legacy project without unit tests, where automated generation may be very useful.

Because this project had no system tests, we manually ran a sample execution of JModeller. The execution consisted of launching the application, creating a new file, adding two "class" figures to the canvas, connecting the figures with a line, and closing the application. This was selected as a typical use of the tools in the application.

`amock` generated 19 potentially useful unit tests, both general and simulation. Figure 6-2 shows a general test; it tests a `ReverseFigureEnumerator`, which wraps a vector and returns its elements in reverse order. (It is typesafe like a Java 1.5 generic `Iterator`, but was written before Java 1.5, which is why it does not just implement `Iterator<Figure>`.) It is a correct test, and just about the minimal amount that one could exercise a `ReverseFigureEnumerator` in order to verify its behavior. Its one obvious deficiency is that it is unlikely that a human writing a test of this behavior would choose to mock the standard `Vector` class; an equivalent manually-written test might look something like Figure 6-3. Note that the exercising statements are identical to the automatically generated ones[2]. This suggests that implementing heuristics for

---

[1]All LOC calculations are generated using David A. Wheeler's 'SLOCCount'.

[2]The only difference is that the manually-written test does not contain some superfluous casts,

```
 1 public class AutoGeneratedTest extends MockObjectTestCase {
 2   public void testSomethingGenerated() throws Throwable {
 3     final Vector mockVector = mock(Vector.class);
 4
 5     verifyThenCheck(new Expectations() {{
 6       one (mockVector).size();
 7       inSequence(s);
 8       will(returnValue(2));
 9     }});
10
11     final ReverseFigureEnumerator testedReverseFigureEnumerator = new
          ReverseFigureEnumerator(mockVector);
12
13     assertThat(testedReverseFigureEnumerator.hasMoreElements(),
14       is(true)
15     );
16
17     final Figure mockFigure = mock(Figure.class);
18
19     verifyThenCheck(new Expectations() {{
20       one (mockVector).elementAt(1);
21       inSequence(s);
22       will(returnValue(mockFigure));
23     }});
24
25     assertThat(testedReverseFigureEnumerator.nextFigure(),
26       is((Figure) mockFigure)
27     );
28
29     assertThat(testedReverseFigureEnumerator.hasMoreElements(),
30       is(true)
31     );
32
33     final Figure mockFigure1 = mock(Figure.class);
34
35     verifyThenCheck(new Expectations() {{
36       one (mockVector).elementAt(0);
37       inSequence(s);
38       will(returnValue(mockFigure1));
39     }});
40
41     assertThat(testedReverseFigureEnumerator.nextFigure(),
42       is((Figure) mockFigure1)
43     );
44
45     assertThat(testedReverseFigureEnumerator.hasMoreElements(),
46       is(false)
47     );
48   }
49 }
```

Figure 6-2: A amock-generated unit test for JHotDraw's reverse figure enumerator.

```
1  public class AutoGeneratedTest extends MockObjectTestCase {
2    public void testSomethingGenerated() throws Throwable {
3      Vector someVector = new Vector();
4      Figure mockFigure0 = mock(Figure.class);
5      Figure mockFigure1 = mock(Figure.class);
6
7      someVector.add(mockFigure0);
8      someVector.add(mockFigure1);
9
10     ReverseFigureEnumerator testedReverseFigureEnumerator = new
           ReverseFigureEnumerator(someVector);
11
12     assertThat(testedReverseFigureEnumerator.hasMoreElements(),
13       is(true)
14     );
15
16     assertThat(testedReverseFigureEnumerator.nextFigure(),
17       is(mockFigure)
18     );
19
20     assertThat(testedReverseFigureEnumerator.hasMoreElements(),
21       is(true)
22     );
23
24     assertThat(testedReverseFigureEnumerator.nextFigure(),
25       is(mockFigure1)
26     );
27
28     assertThat(testedReverseFigureEnumerator.hasMoreElements(),
29       is(false)
30     );
31   }
32 }
```

Figure 6-3: A manually-written unit test similar to the automatically-generated test in Figure 6-2.

standard `java.util` collections, like the iterator heuristic described in Section 4.1, could improve generated test quality; we have not yet had time to implement these additional heuristics.

An example of a generated simulation test is shown below. This is a test of the `ConnectionTool` which the user clicked on. The test first constructs the `ConnectionTool` (line 6) and calls `activate` on it (line 13). It then informs the tool about the mouse's motion: a `mouseMove` in the empty part of the canvas (line 56), a `mouseMove` over one of the figures (line 114), a `mouseDown` on the figure (line 186), a `mouseDrag` (line 249), and finally a `mouseUp`. The actual test had three more `mouseMove` calls and two more `mouseDown` calls which were redundant with the ones given and can be elided without changing the effect of the test. One can imagine that this redundancy could

---

which are necessary for generic type inference in some cases but which `amock` could try to leave out when unnecessary.

67

be found automatically (see Section 8.2.2).

```
1  public class AutoGeneratedTest extends MockObjectTestCase {
2    public void testSomethingGenerated() throws Throwable {
3      final ConnectionFigure mockConnectionFigure = mock(ConnectionFigure.class);
4      final DrawingView mockDrawingView = mock(DrawingView.class);
5
6      final ConnectionTool testedConnectionTool = new ConnectionTool(mockDrawingView,
           mockConnectionFigure);
7
8      verifyThenCheck(new Expectations() {{
9        one (mockDrawingView).clearSelection();
10       inSequence(s);
11     }});
12
13     testedConnectionTool.activate();
14
15     final Drawing mockDrawing = mock(Drawing.class);
16     final Figure mockFigure = mock(Figure.class);
17     final Figure mockFigure1 = mock(Figure.class);
18
19     verifyThenCheck(new Expectations() {{
20       one (mockDrawingView).drawing();
21       inSequence(s);
22       will(returnValue(mockDrawing));
23
24       one (mockDrawing).figuresReverse();
25       inSequence(s);
26       will(returnValue(new FigureEnumerationIteratorWrapper(mockFigure, mockFigure1))
           );
27
28       one (mockFigure).includes(null);
29       inSequence(s);
30       will(returnValue(false));
31
32       one (mockFigure).canConnect();
33       inSequence(s);
34       will(returnValue(true));
35
36       one (mockFigure).containsPoint(128, 18);
37       inSequence(s);
38       will(returnValue(false));
39
40       one (mockFigure1).includes(null);
41       inSequence(s);
42       will(returnValue(false));
```

```
43
44        one (mockFigure1).canConnect();
45        inSequence(s);
46        will(returnValue(true));
47
48        one (mockFigure1).containsPoint(128, 18);
49        inSequence(s);
50        will(returnValue(false));
51
52        one (mockDrawingView).checkDamage();
53        inSequence(s);
54     }});
55
56     testedConnectionTool.mouseMove(mock(MouseEvent.class), 128, 18);
57
58     // [Two other mouseMove calls elided.]
59
60     final MouseEvent mockMouseEvent = mock(MouseEvent.class);
61
62     verifyThenCheck(new Expectations() {{
63       one (mockDrawingView).drawing();
64       inSequence(s);
65       will(returnValue(mockDrawing));
66
67       one (mockDrawing).figuresReverse();
68       inSequence(s);
69       will(returnValue(new FigureEnumerationIteratorWrapper(mockFigure, mockFigure1))
             );
70
71       one (mockFigure).includes(null);
72       inSequence(s);
73       will(returnValue(false));
74
75       one (mockFigure).canConnect();
76       inSequence(s);
77       will(returnValue(true));
78
79       one (mockFigure).containsPoint(343, 97);
80       inSequence(s);
81       will(returnValue(false));
82
83       one (mockFigure1).includes(null);
84       inSequence(s);
85       will(returnValue(false));
86
```

```
87        one (mockFigure1).canConnect();
88        inSequence(s);
89        will(returnValue(true));

90

91        one (mockFigure1).containsPoint(343, 97);
92        inSequence(s);
93        will(returnValue(true));

94

95        one (mockFigure1).connectorVisibility(true);
96        inSequence(s);

97

98        one (mockMouseEvent).getX();
99        inSequence(s);
100       will(returnValue(343));

101

102       one (mockMouseEvent).getY();
103       inSequence(s);
104       will(returnValue(97));

105

106       one (mockFigure1).connectorAt(343, 97);
107       inSequence(s);
108       will(returnValue(mock(Connector.class)));

109

110       one (mockDrawingView).checkDamage();
111       inSequence(s);
112     }});

113

114     testedConnectionTool.mouseMove(mockMouseEvent, 343, 97);

115

116     // [One mouseMove call elided.]

117

118     final ConnectionFigure mockConnectionFigure1 = mock(ConnectionFigure.class);
119     final Connector mockConnector = mock(Connector.class);
120     final MouseEvent mockMouseEvent2 = mock(MouseEvent.class);

121

122     verifyThenCheck(new Expectations() {{
123       one (mockMouseEvent2).getX();
124       inSequence(s);
125       will(returnValue(344));

126

127       one (mockMouseEvent2).getY();
128       inSequence(s);
129       will(returnValue(94));

130

131       one (mockDrawingView).drawing();
```

```
132        inSequence ( s ) ;
133        will ( returnValue ( mockDrawing ) ) ;
134
135        one ( mockDrawing ) . figuresReverse ( ) ;
136        inSequence ( s ) ;
137        will ( returnValue ( new FigureEnumerationIteratorWrapper ( mockFigure , mockFigure1 ) )
               ) ;
138
139        one ( mockFigure ) . includes ( null ) ;
140        inSequence ( s ) ;
141        will ( returnValue ( false ) ) ;
142
143        one ( mockFigure ) . canConnect ( ) ;
144        inSequence ( s ) ;
145        will ( returnValue ( true ) ) ;
146
147        one ( mockFigure ) . containsPoint ( 344 , 94 ) ;
148        inSequence ( s ) ;
149        will ( returnValue ( false ) ) ;
150
151        one ( mockFigure1 ) . includes ( null ) ;
152        inSequence ( s ) ;
153        will ( returnValue ( false ) ) ;
154
155        one ( mockFigure1 ) . canConnect ( ) ;
156        inSequence ( s ) ;
157        will ( returnValue ( true ) ) ;
158
159        one ( mockFigure1 ) . containsPoint ( 344 , 94 ) ;
160        inSequence ( s ) ;
161        will ( returnValue ( true ) ) ;
162
163        one ( mockFigure1 ) . canConnect ( ) ;
164        inSequence ( s ) ;
165        will ( returnValue ( true ) ) ;
166
167        one ( mockFigure1 ) . connectorAt ( 344 , 94 ) ;
168        inSequence ( s ) ;
169        will ( returnValue ( mockConnector ) ) ;
170
171        one ( mockConnectionFigure ) . clone ( ) ;
172        inSequence ( s ) ;
173        will ( returnValue ( mockConnectionFigure1 ) ) ;
174
175        one ( mockConnectionFigure1 ) . startPoint ( 344 , 94 ) ;
```

```
176        inSequence(s);
177
178        one (mockConnectionFigure1).endPoint(344, 94);
179        inSequence(s);
180
181        one (mockDrawingView).add(mockConnectionFigure1);
182        inSequence(s);
183        will(returnValue(mockConnectionFigure1));
184     }});
185
186     testedConnectionTool.mouseDown(mockMouseEvent2, 344, 94);
187
188     final MouseEvent mockMouseEvent3 = mock(MouseEvent.class);
189
190     verifyThenCheck(new Expectations() {{
191        one (mockMouseEvent3).getX();
192        inSequence(s);
193        will(returnValue(194));
194
195        one (mockMouseEvent3).getY();
196        inSequence(s);
197        will(returnValue(212));
198
199        one (mockDrawingView).drawing();
200        inSequence(s);
201        will(returnValue(mockDrawing));
202
203        one (mockDrawing).figuresReverse();
204        inSequence(s);
205        will(returnValue(new FigureEnumerationIteratorWrapper(mockConnectionFigure1,
              mockFigure, mockFigure1)));
206
207        one (mockConnectionFigure1).includes(mockConnectionFigure1);
208        inSequence(s);
209        will(returnValue(true));
210
211        one (mockFigure).includes(mockConnectionFigure1);
212        inSequence(s);
213        will(returnValue(false));
214
215        one (mockFigure).canConnect();
216        inSequence(s);
217        will(returnValue(true));
218
219        one (mockFigure).containsPoint(194, 212);
```

```java
220        inSequence(s);
221        will(returnValue(false));
222
223        one (mockFigure1).includes(mockConnectionFigure1);
224        inSequence(s);
225        will(returnValue(false));
226
227        one (mockFigure1).canConnect();
228        inSequence(s);
229        will(returnValue(true));
230
231        one (mockFigure1).containsPoint(194, 212);
232        inSequence(s);
233        will(returnValue(false));
234
235        one (mockConnector).owner();
236        inSequence(s);
237        will(returnValue(mockFigure1));
238
239        one (mockFigure1).connectorVisibility(false);
240        inSequence(s);
241
242        one (mockDrawingView).checkDamage();
243        inSequence(s);
244
245        one (mockConnectionFigure1).endPoint(194, 212);
246        inSequence(s);
247    }});
248
249    testedConnectionTool.mouseDrag(mockMouseEvent3, 194, 212);
250
251    // [Two more mouseDrag calls elided.]
252
253    final Connector mockConnector3 = mock(Connector.class);
254    final DrawingEditor mockDrawingEditor = mock(DrawingEditor.class);
255    final MouseEvent mockMouseEvent6 = mock(MouseEvent.class);
256
257    verifyThenCheck(new Expectations() {{
258        one (mockMouseEvent6).getX();
259        inSequence(s);
260        will(returnValue(165));
261
262        one (mockMouseEvent6).getY();
263        inSequence(s);
264        will(returnValue(271));
```

```
265
266        one (mockDrawingView).drawing();
267        inSequence(s);
268        will(returnValue(mockDrawing));

269
270        one (mockDrawing).figuresReverse();
271        inSequence(s);
272        will(returnValue(new FigureEnumerationIteratorWrapper(mockConnectionFigure1,
               mockFigure)));

273
274        one (mockConnectionFigure1).includes(mockConnectionFigure1);
275        inSequence(s);
276        will(returnValue(true));

277
278        one (mockFigure).includes(mockConnectionFigure1);
279        inSequence(s);
280        will(returnValue(false));

281
282        one (mockFigure).canConnect();
283        inSequence(s);
284        will(returnValue(true));

285
286        one (mockFigure).containsPoint(165, 271);
287        inSequence(s);
288        will(returnValue(true));

289
290        one (mockConnector).owner();
291        inSequence(s);
292        will(returnValue(mockFigure1));

293
294        one (mockFigure).canConnect();
295        inSequence(s);
296        will(returnValue(true));

297
298        one (mockFigure).includes(mockFigure1);
299        inSequence(s);
300        will(returnValue(false));

301
302        one (mockConnectionFigure1).canConnect(mockFigure1, mockFigure);
303        inSequence(s);
304        will(returnValue(true));

305
306        one (mockMouseEvent6).getX();
307        inSequence(s);
308        will(returnValue(165));
```

```
309
310        one ( mockMouseEvent6 ) . getY ( ) ;
311        inSequence ( s ) ;
312        will ( returnValue (271) ) ;
313
314        one ( mockFigure ) . connectorAt (165 , 271) ;
315        inSequence ( s ) ;
316        will ( returnValue ( mockConnector3 ) ) ;
317
318        one ( mockConnectionFigure1 ) . connectStart ( mockConnector ) ;
319        inSequence ( s ) ;
320
321        one ( mockConnectionFigure1 ) . connectEnd ( mockConnector3 ) ;
322        inSequence ( s ) ;
323
324        one ( mockConnectionFigure1 ) . updateConnection ( ) ;
325        inSequence ( s ) ;
326
327        one ( mockDrawingView ) . editor ( ) ;
328        inSequence ( s ) ;
329        will ( returnValue ( mockDrawingEditor ) ) ;
330
331        one ( mockDrawingEditor ) . toolDone ( ) ;
332        inSequence ( s ) ;
333        will ( new Callback ( ) { public void go ( ) {
334            testedConnectionTool . deactivate ( ) ;
335          } } ) ;
336
337        one ( mockDrawingView ) . setCursor ( with ( a ( Cursor . class ) ) ) ;
338        inSequence ( s ) ;
339
340        one ( mockFigure ) . connectorVisibility ( false ) ;
341        inSequence ( s ) ;
342      } } ) ;
343
344      testedConnectionTool . mouseUp ( mockMouseEvent6 , 165 , 271) ;
345   }
346 }
```

This test directly verifies the behavior of the connection tool without requiring the GUI toolkit to be set up. While it is superficially similar to standard GUI capture-replay tools, note that because it mocks out the actual GUI, it mocks out all details of the GUI toolkit implementation, functioning at the application logic level.

```
1  public class AutoGeneratedTest extends MockObjectTestCase {
2    public void testSomethingGenerated() throws Throwable {
3      final SVNConfigFile mockSVNConfigFile = mock(SVNConfigFile.class);
4
5      final SVNCompositeConfigFile testedSVNCompositeConfigFile = new
          SVNCompositeConfigFile(mock(SVNConfigFile.class), mockSVNConfigFile);
6
7      verifyThenCheck(new Expectations() {{
8        one (mockSVNConfigFile).setPropertyValue("auth", "store-auth-creds", "yes",
            false);
9        inSequence(s);
10     }});
11
12     testedSVNCompositeConfigFile.setPropertyValue("auth", "store-auth-creds", "yes",
          false);
13
14     verifyThenCheck(new Expectations() {{
15       one (mockSVNConfigFile).getPropertyValue("auth", "store-auth-creds");
16       inSequence(s);
17       will(returnValue("yes"));
18     }});
19
20     assertThat(testedSVNCompositeConfigFile.getPropertyValue("auth", "store-auth-
          creds"),
21       is((String) "yes")
22     );
23   }
24 }
```

Figure 6-4: An `amock`-generated unit test for SVNKit's composite configuration file.

6.2   Network: SVNKit

SVNKit [39] is a Java (60K LOC) implementation of a client for the Subversion
version control system [9, 38]. SVNKit comes with very few unit tests, some system
tests, and a custom test harness for running Subversion's Python-scripted command-
line system tests. The system tests all run the SVNKit command-line client multiple
times; because `amock` runs on the output of a single execution (see Section 3.2), we
could not directly use an entire system test. We chose to instrument an execution of
the Subversion command `svn ls http://svn.collab.net/repos/svn/` as a typical
network-intensive use of SVNKit.

Of the 33 SVNKit classes used in this example, we decided that 5 factored
tests were potentially useful. As with JHotDraw, we found that `amock` generated
both general and simulation tests. Figure 6-4 shows a general test. It tests the
`SVNCompositeConfigFile` class, which combines two configuration file objects (one
for per-user configuration and one for system-wide configuration) into one composite
object. The generated test verifies that setting a value on the composite object will

76

```
1  public class AutoGeneratedTest extends MockObjectTestCase {
2    public void testSomethingGenerated() throws Throwable {
3      final ISVNRepositoryPool mockISVNRepositoryPool = mock(ISVNRepositoryPool.class);
4      final SVNLogClient testedSVNLogClient = new SVNLogClient(mockISVNRepositoryPool,
           mock(ISVNOptions.class));
5
6      final ISVNDirEntryHandler mockISVNDirEntryHandler = mock(ISVNDirEntryHandler.
           class);
7      // [ More mock declarations elided. ]
8      final Capture<Collection> capturingCollection = capture(Collection.class);
9
10     verifyThenCheck(new Expectations() {{
11       one (mockISVNRepositoryPool).createRepository(mockSVNURL, true);
12       inSequence(s);
13       will(returnValue(mockSVNRepository));
14       // [ ... ]
15       one (mockSVNRepository).getDir(with(equal("")), with(equal(25884L)), with(aNull
             (java.util.Map.class)), with(valueCapturedBy(capturingCollection)));
16       inSequence(s);
17       will(doAll(capturingCollection.capture(3),
18         new Callback() { public void go() {
19           assertThat(capturingCollection.getCapturedValue().add(mockSVNDirEntry),
20             is(true)
21           );
22           assertThat(capturingCollection.getCapturedValue().add(mockSVNDirEntry1),
23             is(true)
24           );
25           assertThat(capturingCollection.getCapturedValue().add(mockSVNDirEntry2),
26             is(true)
27           );
28           assertThat(capturingCollection.getCapturedValue().add(mockSVNDirEntry3),
29             is(true)
30           );
31           assertThat(capturingCollection.getCapturedValue().add(mockSVNDirEntry4),
32             is(true)
33           );
34           assertThat(capturingCollection.getCapturedValue().add(mockSVNDirEntry5),
35             is(true)
36           );
37         }},
38         returnValueCapturedBy(capturingCollection)));
39
40       one (mockSVNDirEntry1).compareTo(mockSVNDirEntry);
41       inSequence(s);
42       will(returnValue(-16));
43       // [ More compareTos elided.]
44
45       one (mockSVNDirEntry2).getName();
46       inSequence(s);
47       will(returnValue("branches"));
48
49       one (SVNPathUtil.class).append("", "branches");
50       inSequence(s);
51       will(returnValue("branches"));
52       // [ ... ]
53
54       one (mockISVNDirEntryHandler).handleDirEntry(mockSVNDirEntry2);
55       inSequence(s);
56       // [ handleDirEntry for the other entries elided. ]
57     }});
58
59     testedSVNLogClient.doList(mockSVNURL, SVNRevision.UNDEFINED, SVNRevision.
           UNDEFINED, false, false, mockISVNDirEntryHandler);
60   }
61 }
```

Figure 6-5: Excerpts from an amock-generated unit test for SVNKit's log client.

set the same value on the per-user configuration object, and that reading a value from the composite object which is in the per-user object will use the per-user value.

Figure 6-5 shows an excerpt from a simulation test; it tests the internal "log client" which is used to fetch information about the repository. It constructs a log client (line 4, passing in a mocked repository pool (a factory object)). It then exercises the client with a `doList` invocation (line 59). The bulk of the actual network interaction is mocked out by the `getDir` call on the repository object (line 15). This method adds six `SVNDirEntry` objects to the collection passed in as its last argument, using the capture facility described in Section 2.2.3 to allow the callback to refer to the collection constructed by the tested code. Because the added `SVNDirEntry` objects are mocks and the collection constructed by the tested code is a `TreeSet`, amock has to define `compareTo` expectations for the inserted mocks (line 40). Finally, the `SVNDirEntrys` are fed to the `handleDirEntry` methods on a directory handler.

The log client example shows us some of the strengths and weaknesses of `amock`. The fact that the log client uses a factory object (the `SVNRepositoryPool`) instead of directly constructing the `SVNRepository` allows `amock` to inject a mock repository (line 11); the test would not have been able to isolate the log client from the network without this. The ability to capture arguments and to write custom callback actions (line 18) is clearly essential for this particular test. On the other hand, the generation of many `compareTo` expectations (only one shown in the excerpt, at line 40) is annoying; perhaps `amock` should have a custom `compareTo` heuristic which inserts a single line of code like `comparesInThisOrder(mockSVNDirEntry1, mockSVNDirEntry2, mockSVNDirEntry3, ...);` which automatically sets up all of the relevant `compareTo` expectations. Also, while it is useful in some cases to mock out static methods, the simple choice (described in Section 3.2) to mock out *every* static method call outside of the current class leads to unnecessary expectations like the one on line 49, which mocks out a simple string manipulation call. Even given these shortcomings, though, this automatically-generated test successfully tests the log client as used in the `svn ls` command without needing to make any network connections.

6.3 Esper

Esper [14] is a Java (47K LOC) event stream processing framework. Esper is distributed with a large JUnit test suite and a set of example programs. We chose the `net.esper.examples.transaction.sim` example for our case study: a transaction stream simulator. Unlike with the previous case studies, we made no changes to `amock` based on the results of the case studies. This section describes tests for two data structures and for two more active objects.

Figure 6-6 shows the generated test for a `SortedRefCountedSet` object. The test adds numbers to the set, and checks the minimum value in the set at various times. Note that no expectations are necessary here because there are no callbacks. It is a perfectly effective test for the sorted set. It is, however, very redundant: the actual generated test has 10 calls to `add` and 20 to `minValue`. This is yet another case where automated minimization or refactoring of the generated test could yield improvements to the test's readability and maintainability. Figure 6-7 is the generated test for an `OuterInnerDirectionalGraph` data structure. It sets up a graph structure, and then queries it repeatedly with `isInner` and `isOuter`. It similarly requires no expectations, and could profit from automated minimization.

Figure 6-8 shows the generated test for a `DispatchServiceImpl` object. The test adds `Dispatchable`s to the dispatcher with the `addExternal` method; the `dispatch` method calls `execute` on each registered `Dispatchable`, and then forgets about the `Dispatchable`s. This test clearly takes advantage of mock objects and expectations. It also places an expectation on the static method `ExecutionPathDebugLog.isEnabled`; it appears that these calls could probably be safely executed instead of mocking them out. Again, there is enormous redundancy in the generated test: the full generated test adds 49 `Dispatchable`s and calls `dispatch` 42 times.

Figure 6-9 shows the generated test for a `FieldGenerator` object. This object uses a random-number generator to generate values for various fields (names, numbers, etc.) in the example's transaction model. This test shows that `amock` is capable of deterministically testing randomized code; this would be difficult for a capture-replay

79

```
1 public class AutoGeneratedTest extends MockObjectTestCase {
2   public void testSomethingGenerated() throws Throwable {
3     final SortedRefCountedSet testedSortedRefCountedSet = new SortedRefCountedSet();
4
5     assertThat(testedSortedRefCountedSet.minValue(),
6       is(nullValue())
7     );
8
9     testedSortedRefCountedSet.add(929L);
10
11     assertThat(testedSortedRefCountedSet.minValue(),
12       is((Object) 929L)
13     );
14
15     assertThat(testedSortedRefCountedSet.minValue(),
16       is((Object) 929L)
17     );
18
19     testedSortedRefCountedSet.add(1297L);
20
21     assertThat(testedSortedRefCountedSet.minValue(),
22       is((Object) 929L)
23     );
24
25     // [More add and minValue calls elided.]
26
27     testedSortedRefCountedSet.add(684L);
28
29     assertThat(testedSortedRefCountedSet.minValue(),
30       is((Object) 684L)
31     );
32
33     // [More add and minValue calls elided.]
34   }
35 }
```

Figure 6-6: An `amock`-generated unit test for an Esper sorted set.

```
1 public class AutoGeneratedTest extends MockObjectTestCase {
2   public void testSomethingGenerated() throws Throwable {
3     final OuterInnerDirectionalGraph testedOuterInnerDirectionalGraph = new
          OuterInnerDirectionalGraph (3);
4
5     assertThat (testedOuterInnerDirectionalGraph.add (0, 1),
6       is ((OuterInnerDirectionalGraph) testedOuterInnerDirectionalGraph)
7     );
8
9     assertThat (testedOuterInnerDirectionalGraph.add (1, 0),
10      is ((OuterInnerDirectionalGraph) testedOuterInnerDirectionalGraph)
11    );
12
13    assertThat (testedOuterInnerDirectionalGraph.add (2, 1),
14      is ((OuterInnerDirectionalGraph) testedOuterInnerDirectionalGraph)
15    );
16
17    assertThat (testedOuterInnerDirectionalGraph.add (1, 2),
18      is ((OuterInnerDirectionalGraph) testedOuterInnerDirectionalGraph)
19    );
20
21    assertThat (testedOuterInnerDirectionalGraph.isOuter (1, 0),
22      is (true)
23    );
24
25    assertThat (testedOuterInnerDirectionalGraph.isInner (0, 1),
26      is (true)
27    );
28
29    assertThat (testedOuterInnerDirectionalGraph.isOuter (2, 1),
30      is (true)
31    );
32
33    assertThat (testedOuterInnerDirectionalGraph.isInner (1, 2),
34      is (true)
35    );
36
37    // [More isInner and isOuter calls elided.]
38  }
39 }
```

Figure 6-7: An amock-generated unit test for an Esper graph.

```
1  public class AutoGeneratedTest extends MockObjectTestCase {
2    public void testSomethingGenerated() throws Throwable {
3      final DispatchServiceImpl testedDispatchServiceImpl = new DispatchServiceImpl();
4
5      verifyThenCheck(new Expectations() {{
6        one (ExecutionPathDebugLog.class).isEnabled();
7        inSequence(s);
8        will(returnValue(false));
9      }});
10
11     testedDispatchServiceImpl.dispatch();
12
13     // [More dispatch calls elided.]
14
15     final Dispatchable mockDispatchable = mock(Dispatchable.class);
16
17     testedDispatchServiceImpl.addExternal(mockDispatchable);
18
19     verifyThenCheck(new Expectations() {{
20       one (ExecutionPathDebugLog.class).isEnabled();
21       inSequence(s);
22       will(returnValue(false));
23
24       one (mockDispatchable).execute();
25       inSequence(s);
26     }});
27
28     testedDispatchServiceImpl.dispatch();
29
30     // [More addExternal and dispatch calls elided.]
31
32     final Dispatchable mockDispatchable46 = mock(Dispatchable.class);
33
34     testedDispatchServiceImpl.addExternal(mockDispatchable46);
35
36     final Dispatchable mockDispatchable47 = mock(Dispatchable.class);
37
38     testedDispatchServiceImpl.addExternal(mockDispatchable47);
39
40     final Dispatchable mockDispatchable48 = mock(Dispatchable.class);
41
42     testedDispatchServiceImpl.addExternal(mockDispatchable48);
43
44     verifyThenCheck(new Expectations() {{
45       one (ExecutionPathDebugLog.class).isEnabled();
46       inSequence(s);
47       will(returnValue(false));
48
49       one (mockDispatchable46).execute();
50       inSequence(s);
51
52       one (mockDispatchable47).execute();
53       inSequence(s);
54
55       one (mockDispatchable48).execute();
56       inSequence(s);
57     }});
58
59     testedDispatchServiceImpl.dispatch();
60   }
61 }
```

Figure 6-8: An `amock`-generated unit test for an Esper dispatch service.

```
 1 public class AutoGeneratedTest extends MockObjectTestCase {
 2   public void testSomethingGenerated() throws Throwable {
 3     final Random mockRandom = mock(Random.class);
 4
 5     verifyThenCheck(new Expectations() {{
 6       one (RandomUtil.class).getNewInstance();
 7       inSequence(s);
 8       will(returnValue(mockRandom));
 9     }});
10
11     final FieldGenerator testedFieldGenerator = new FieldGenerator();
12
13     verifyThenCheck(new Expectations() {{
14       one (mockRandom).nextInt(6);
15       inSequence(s);
16       will(returnValue(2));
17     }});
18
19     assertThat(testedFieldGenerator.getRandomCustomer(),
20       is((String) "YELLOW")
21     );
22
23     verifyThenCheck(new Expectations() {{
24       one (mockRandom).nextInt(1000);
25       inSequence(s);
26       will(returnValue(717));
27     }});
28
29     assertThat(testedFieldGenerator.randomLatency(1185490766152L),
30       is(1185490766869L)
31     );
32
33     verifyThenCheck(new Expectations() {{
34       one (mockRandom).nextInt(1000);
35       inSequence(s);
36       will(returnValue(580));
37     }});
38
39     assertThat(testedFieldGenerator.randomLatency(1185490766869L),
40       is(1185490767449L)
41     );
42
43     // [More field generation elided.]
44   }
45 }
```

Figure 6-9: An amock-generated unit test for an Esper randomized data generator.

system without any sort of factoring or explicit random-number generator seeding. On the other hand, the test does rely on the particular algorithm used to translate from the randomly generated numbers to the field values, which is unlikely to be semantically significant. So while it is notable that this test isolates the SUT from its source of randomness, it is unlikely that this particular test is verifying important properties of the `FieldGenerator`.

Chapter 7

Experimental evaluation

Chapter 6 described the qualitative applicability of `amock` to a variety of real Java programs. This chapter quantitatively analyzes the performance of `amock` on one subject program, the SVNKit Subversion client (as described in Section 6.2). We performed all experiments on a 1800GhZ AMD Opteron with 8GB of RAM.

7.1   Capture phase slowdown

The first step in generating a test suite is executing a system test with the trace instrumentation enabled. Because `amock` traces every method call within the system test, the instrumented system test will run slower than the original test. (This overhead is only for tracing the system test, not for running the generated test suite.) Additionally, the generated trace takes up disk space. The factoring phase also takes time to execute. These stages can be performed automatically and without developer interaction, so large constant factors do not necessarily prevent the project from being useful.

We measured that time required to run the "capture" phase on the SVNKit system test used in the case studies (Section 6.2), and analyzed the sources of inefficiency. It took 254 seconds to run the system test 50 times, and 651 to run it 50 times under the `amock` tracer. This was thus a slowdown factor of 2.6. The three output files described in Section 3.1.1 consumed 12 MB of disk space: the transcript was 12 MB,

the instance information database was 290 KB, and the hierarchy file was 17 KB.

7.2   Unit test speedup

One of the advantages that unit tests can have over system tests is that they generally run much faster, because they do not need to set up complex resources and they only test a small part of the code. The tests that `amock` generates should similarly be noticeably faster (or at least no slower than) the system tests they are factored out of.

We measured the run time of the (uninstrumented) SVNKit system test used in the case studies (Section 6.2) and compared it to the run time of the unit tests factored from it. It took 254 seconds to run the system test 50 times, and 54 seconds to run the generated unit test suite 50 times. (Note that these figures include the startup time for the JVM, as well as the time for `smock` to instrument all non-JDK code.) This is a speedup factor of 4.7.

7.3   Robustness: resistance to false failures

Automatically generated tests can be brittle. If tests fail spuriously when the tested code changes, the effort to understand and fix the failing tests is a distraction from development, and if a tool has too many false failures, a "boy who cried wolf" effect will prevent developers from paying attention to actual defects revealed by failing tests.

We used `amock` to generate a passing test suites for an older version of SVNKit (version 1.1.0). We performed the same test generation process as in the case studies, as described in Section 6.2. We then ran the test suite against the four later re-leased versions of the library which spanned eight months of active development (443 changesets). We observed how many of the generated tests continued to pass and which failed. We subjectively classified the failing tests into two categories: spurious failures, where the behavior of the underlying code did not change in way imme-

version	date	pass	spurious failure	true failure
1.1.0	2006-11-14	11	0	0
1.1.1	2007-02-01	11	0	0
1.1.2	2007-03-29	10	1	0
1.1.3	2007-06-22	9	1	1
1.1.4	2007-07-20	9	1	1

Figure 7-1: Results of robustness experiments

diately relevant to the behavior being verified, and true failures, where the actual documented behavior of the tested code or API changed incompatibly.

As Figure 7-1 shows, the generated tests mostly continued to pass under the later versions. There was enough code churn between the tested versions that two tests began to fail, but the other nine tests correctly continued to pass. The test which begins to fail in SVNKit 1.1.2 was a test generated for a `DefaultSVNRepositoryPool`. The SVNKit 1.1.2 code calls `getLocation` and `getProtocol` methods one time fewer than the SVNKit 1.1.0 code did, so the test fails due to unsatisfied expectations. This spurious failure could be avoided if `amock` used a purity analysis, as described in Section 8.2.1. The test which begins to fail in SVNKit 1.1.3 is a test of the `SVNLogClient` class. SVNKit 1.1.3 started to implement user cancellation support on the `SVNRepository` class, so the SUT code began to call a new method `setCanceller` on a mocked `SVNRepository`, causing an unexpected invocation error. Any jMock-based test of that method would require adding an expectation to make the test continue to pass, so this is a less spurious failure than the previously described one.

Chapter 8

Conclusion

8.1   Related work ∂

Automatic generation of software tests has been studied from many different angles. The techniques range from random testing [40, 31, 10] to systematic static analyses [11, 41] to model-based dynamic analyses [1]. Two previous projects exist which attempt a "test factoring" strategy of turning dynamic traces of large system tests into smaller unit tests are [35] and [13].

The main goal of Saff's test factoring project [35] is allowing developers to run a slow system test much more efficiently when only a small part of the system has changed. Saff's tool captures a transcript of a long system test or other program execution, and then replays it in a special instrumented Java run-time environment where all objects other than those of the class under test are mocks following the transcript. The bodies of method in classes not in the SUT are skipped during playback, leading to a much faster execution of the original test suite which only exercises code from one class. If the class under test attempts to make different method calls to the rest of the system than it did during the original execution, the replay stops and tells the user that the full system test should be run instead; otherwise, the factored test succeeds or fails according to whether the program that it is replaying succeeds or fails. This project uses behavior-based verification, like `amock`. Factored tests in Saff's project obey the following property: if a factored test

for the only class whose code has changed passes, then the system test would have passed as well.

In Saff's project, many benign changes to the class can cause the factored test to fail. For example, the method under test could call external methods in a slightly different order or with slightly different parameters; test factoring will consider this to be too different to continue the replay, but a human could determine that both versions are acceptable. A developer cannot easily take a transcript made by test factoring and make it less brittle by relaxing these constraints; tests are recorded in a transcript which is not meant for human consumption.

Additionally, test factoring slices up the program state based on class or package names, not based on time or individual object lifetimes: the generated tests consist of replaying an entire system test on the target class. Thus, even if the tests were human-readable, they would be very long; even if a typical use of an instance of the class under test only involves a few method calls, each test includes all of the method calls ever made on *any object* of that class.

Finally, test factoring relies on the ability to instrument all classes (including the JDK system libraries) even just to replay the tests, which makes it non-trivial to integrate into a pre-existing unit testing process. While the `smock` library used by `amock` also requires instrumentation, `smock` does not require the JDK libraries to be pre-instrumented, and only modifies the bodies of methods, whereas Saff's instrumentation changes the entire class, creating new versions of each class and method.

In test carving [13], during the execution of a long system test, all reachable objects are frequently serialized to disk. Pieces of the long test can then be independently "played back" by loading the state before an action, executing that action, and comparing the actual post-state to the serialized post-state. This method fundamentally uses state-based verification (unlike `amock`), and relies on the internal data structures of the objects not changing significantly. Test carving produces tests which only work in the context of a custom serialization framework — carved tests look nothing like tests that a programmer would have designed by hand, and it is unclear how much

information a developer can get about why a carved test failed.

Agitar Software's AgitarOne and JUnit Factory products can automatically gener-
ate mock-based JUnit tests[7, 24]. Like `amock`, the mock objects are defined explicitly
in the test source (as opposed to being an implicit part of the Java runtime environ-
ment); the mock objects use Agitar's proprietary "Mockingbird" library. The tests
are generated using Agitar's "agitation testing" methodology, a combination of dy-
namic invariant detection and test-input generation. As it relies on dynamic analysis
to discover "typical" uses of objects, this is in a sense similar in concept to test
factoring.

Test factoring can be thought of as a three-phase extension of a two-phase capture-
replay system. There exist several commercial capture-replay tools, mostly operating
at the user-interface level, though not as much academic research in the field[27].
Recorded tests tend not to be very legible or comprehensible, and often suffer from
interface sensitivity[28]. Test factoring moves the capture-replay boundary from the
user interface to internal APIs, and separates tests of different modules from each
other.

8.2 Future work

There is much potential for future improvement to the `amock` concept and implemen-
tation.

8.2.1 Purity

The current `amock` algorithm ensures that the SUT code calls precisely the same
methods on the environment is exactly the same order. By default, every expectation
is declared to be expected exactly once, and every expectation is threaded into a single
sequence. When these method calls represent active manipulation of the environment,
this can be appropriate. But method calls which passively access the environment
do not need to be as constrained. It should be acceptable for the tested code to call
"pure" methods which have no side effects in a different order than in the observed

execution, or a different number of times, or not at all [35].

There is a rich history of research in analyzing programs to determine which methods have side effects [8, 34, 29, 33, 37, 36, 2]. `amock` could use the results of such an analysis to decide which methods are side-effect-free, and relax the ordering and uniqueness constraints on them. While developers can already do this manually by editing the JUnit tests (already an improvement on previous test factoring implementations), it is reasonable to believe (based on observations of `amock`'s generated tests) that automating this step would significantly decrease the brittleness of generated tests.

8.2.2   Refactoring tests

The current `amock` implementation makes only minor improvements to the test after initially generating it. However, there are many potential refactorings that could be applied to the generated test suite to improve its overall quality. Some examples include:

- If tests are generated for every instance in the execution, many of them will be identical or isomorphic. `amock` could detect and suppress redundant tests.

- Each test simulates the entire lifetime of the targeted instance. However, the object may be exercised repeatedly in essentially the same way. For example, when generating tests for a GUI handler in JHotDraw, we found that an object might be exercised by a `mouseDown` method call, followed by several dozen `mouseMove` calls, followed by a `mouseUp` call. The test would have been equally valid if it only had one of the `mouseMove` calls. `amock` could detect and remove intra-test redundancy.

- When `amock` generates many tests for instances of the same class, the tests may be similar in structure but still test slightly different behavior. `amock` could automatically factor out the common code into a helper method, reducing each individual test to the parts that actually vary.

All of these analyses could be based either on domain-specific knowledge of behavior-based testing, or through methods that work on any Java code.

8.2.3   Backtracking

The `amock` factorizing processor makes many choices during its single pass over the trace which affect the generated test; for example, it must decide whether to apply each of the heuristics described in Chapter 4, and it must choose a package to place the test in. If the choice made is in error, it could lead to a test that does not pass or perhaps even compile. `amock` has no way to fix such mistakes. `amock` could potentially make its choices explicit, and backtrack and try again if the generated test is not suitable.

8.2.4   Pattern recognition

More heuristics along the lines of those in Chapter 4 could help the generated tests use a higher level of abstraction in describing expectations. For example, if mock objects are inserted into a `SortedSet`, the current `amock` implementation will need many `compareTo` expectations between various pairs of mock objects; `amock` could instead replace the explicit `compareTo` expectations with a declaration like `sortedInThisOrder(mockFoo, mockBar, mockBaz)`. Additionally, support for recognizing (small) collections from the `java.util` package and using them directly instead of mocking them (like we do now with iterators and record types) could potentially be useful.

8.2.5   Efficiency

`amock` could be made more efficient in several ways. It is very disk-intensive; the raw trace is very long, and the various stages of post-processing that occur before generating tests are done in series with a complete deserialization and serialization on each stage. It would be relatively straightforward to do more of this processing at once.

If it is already known at trace time which classes will have tests generated for them, it may be possible to run a simplified version of the state machine during trace and refrain from logging events that will be mocked out by any generated test. This may reduce the overhead of tracing, though it would also make the tracer more complex.

8.3   Contributions

This thesis presents the following contributions:

- A new approach to test factoring that produces human-readable JUnit tests with user-level mock objects. Our approach is extensible with heuristics that more succinctly describe common Java patterns.

- `amock`: An implementation of this approach. While `amock` has some limitations (most notably in its handling of arrays), it successfully factors tests for several real-world programs.

- Case studies showing the applicability of `amock` to real-world projects.

- `smock`: An extension to the jMock library allowing developers to mock static methods. `smock` can assist jMock users interested in testing legacy code, independently of whether they use `amock`.

Bibliography

[1] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*, Portland, OR, USA, October 23, 2006.

[2] Shay Artzi, Adam Kieżun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2007-020, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, March 23, 2007.

[3] ASM. http://asm.objectweb.org/.

[4] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, Boston, 2002.

[5] Kent Beck and Erich Gamma. JUnit test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[6] Joshua Bloch. *Effective Java Programming Language Guide*. Addison Wesley, Boston, MA, 2001.

[7] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 169–179, Portland, ME, USA, July 18–20, 2006.

[8] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive inter-procedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, SC, January 1993.

[9] B. Collins-Sussmann, B.W. Fitzpatrick, and C.M. Pilato. *Version Control with Subversion*. O'Reilly, 2004.

[10] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, September 2004.

[11] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE'05, Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, St. Louis, MO, USA, May 18–20, 2005.

[12] Definalizer. `http://www.sixlegs.com/blog/java/definalizer.html`.

[13] Sebastian Elbaum, Hui Nee Chin, Mathew Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the International Symposium Foundations of Software Engineering*. ACM, november 2006.

[14] Esper. `http://esper.codehaus.org/`.

[15] Martin Fowler. Mocks aren't stubs. `http://www.martinfowler.com/articles/mocksArentStubs.html`, 2004.

[16] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in Java. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 855–865, 2006.

[17] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. jMock: supporting responsibility-based design with mock objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 4–5, New York, NY, USA, 2004. ACM Press.

[18] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. Mock roles, not objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246, New York, NY, USA, 2004. ACM Press.

[19] Hamcrest. `http://code.google.com/p/hamcrest/`.

[20] A. Hunt and D. Thomas. Tell, don't ask. `http://www.pragmaticprogrammer.com/ppllc/papers/1998_05.html`, 1998.

[21] JHotDraw. `http://www.jhotdraw.org/`.

[22] jMock. `http://www.jmock.org/`.

[23] JUnit. `http://www.junit.org`.

[24] JUnit Factory. `http://www.junitfactory.com/`.

[25] Wolfram Kaiser. Become a programming Picasso with JHotDraw. *JavaWorld*, 2001.

[26] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[27] G. Meszaros. Agile regression testing using record & playback. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 353–360, 2003.

[28] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, 2007.

[29] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 1–11, Rome, Italy, July 22–24, 2002.

[30] Objenesis. `http://code.google.com/p/objenesis/`.

[31] Parasoft Corporation. *Jtest version 4.5.* `http://www.parasoft.com/`.

[32] Rake. `http://rake.rubyforge.org/`.

[33] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *ICSM 2004, Proceedings of the International Conference on Software Maintenance*, pages 82–91, Chicago, Illinois, September 12–14, 2004.

[34] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Compiler Construction: 10th International Conference, CC 2001*, pages 20–36, Genova, Italy, April 2001.

[35] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, November 9–11, 2005.

[36] Alexandru Sălcianu. *Pointer analysis for Java programs: Novel techniques and applications.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 2006.

[37] Alexandru Sălcianu and Martin C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI'05, Sixth International Conference on Verification, Model Checking and Abstract Interpretation*, pages 199–215, Paris, France, January 17–19, 2005.

[38] Subversion. `http://subversion.tigris.org/`.

[39] SVNKit. `http://svnkit.com/`.

[40] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006 — Object-Oriented Programming, 20th European Conference*, pages 380–403, Nantes, France, July 5–7, 2006.

[41] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, Edinburgh, UK, April 4–8, 2005.

[42] XStream. `http://xstream.codehaus.org/`.