amock

Automated generation of readable unit tests from system tests

David Glasser

July 16, 2007

System tests

- show you the big picture
- don't test internal interfaces

SVNKit System Test

```
1 $ jsvn checkout http://amock.googlecode.com/svn/trunk
2 $ jsvn update -r245
3 $ jsvn diff
```

Unit tests

- focused
- tedious to write retroactively

Generated SVNKit Unit Test

```
1 one (mockSVNLogClient).doList(mockSVNURL, SVNRevision.UNDEFINED,
2                                SVNRevision.UNDEFINED, false,
3                                false, testedSVNLsCommand);
4 inSequence(s);
5 will(new Callback() { public void go() {
6     testedSVNLsCommand.handleDirEntry(mockSVNDirEntry);
7     testedSVNLsCommand.handleDirEntry(mockSVNDirEntry1);
8     testedSVNLsCommand.handleDirEntry(mockSVNDirEntry2);
9     testedSVNLsCommand.handleDirEntry(mockSVNDirEntry3);
10    testedSVNLsCommand.handleDirEntry(mockSVNDirEntry4);
11    testedSVNLsCommand.handleDirEntry(mockSVNDirEntry5);
12 }});
13
14 one (mockSVNDirEntry).getRelativePath();
15 inSequence(s);
16 will(returnValue("branches"));
17
18 one (mockPrintStream).print("branches");
19 inSequence(s);
```

My goal

Automatically generate readable and malleable unit tests.

Capture-replay approach to test generation

- Capture phase: developer runs the program in a special mode which records input and output
- Replay phase: automatically run the program with the given inputs, checking the outputs

Capture-replay approach to test generation

- Capture phase: developer runs the program in a special mode which records input and output and internal interactions
- Factor phase: slice the recording into many smaller descriptions of the interactions of individual units with their environments
- Replay phase: run each unit independently, using the sliced trace to fill in the blanks

Test Factoring

Definition

Test factoring: a process which analyzes an execution of a program and generates unit tests which exercise its modules in the same way that the original program did.

David Saff's test factoring project

- Capture: creates a transcript of all method calls in a system test
- Factor: specify which class to exercise
- Replay: play back the transcript
  - special Java run-time environment
  - skip over any methods not in the class being tested
  - verifies that the methods being tested:
    - make the expected calls to the rest of the code
    - return the expected values

Problems with generated tests from test factoring

- Automatically generated tests can be brittle (spuriously fail)
  - Expects the code to call `getName()` exactly four times
  - Expects the code to call `getFirstName()` before `getLastName()`
  - Expects the code to return exactly 1.48392593333
- Constraints may need to be relaxed
  - automatically (ideal, but hard)
  - manually (only if the factored tests are in a developer-friendly format)

Generating JUnit

Why generate JUnit:

- The most developer-friendly format for defining unit tests
- The replay phase is trivial: compile and run
- Generated tests can be improved by human developers

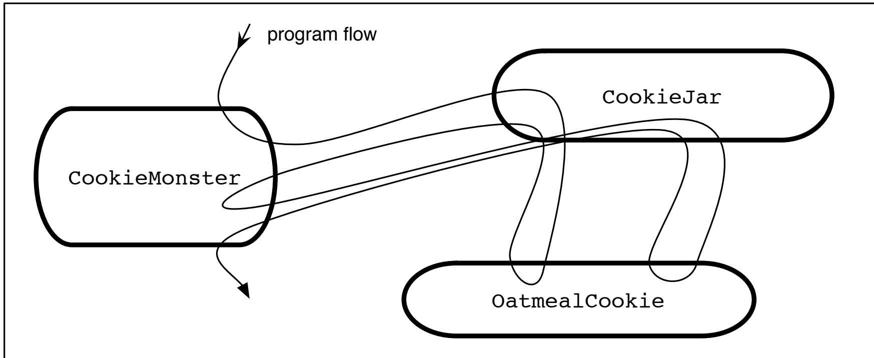
amock

- amock implements test factoring for Java
- Capture phase: Instrument a system test to create a transcript (similar to Saff)
- Factor phase: Slice transcript into a suite of JUnit tests that simulate the experience of individual objects in the transcript
- Replay phase: Compile and run in JUnit test harness

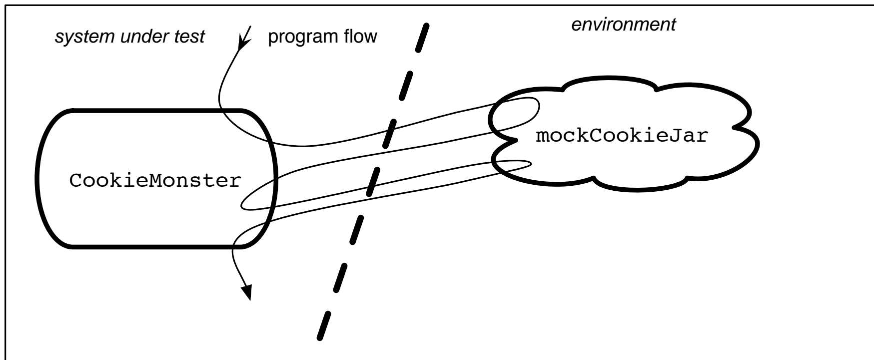
Testing Java classes in isolation

- We only want to test the code in class `CookieMonster`
- We don't want to set up the entire environment (GUIs, databases, network connections, etc)
- Methods in class `CookieMonster` take an argument of type `CookieJar` and call methods on it
- For unit tests, we just want to verify `CookieMonster`, not `CookieJar`!

Flow of control during system test



Isolated flow of control during unit test



Testing with Mocks

- A solution: make a "mock `CookieJar`"!
- The mocked version of `CookieJar` implements the same interface as `CookieJar`
- Instead of running `CookieJar`'s code, it is specially rigged to just return whatever `CookieJar`'s methods would return
- ... or make whichever calls into the tested class it would make, etc.
- Several libraries exist for defining mock-based tests
  - `amock` uses jMock

Example Library

```
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 }
```

Example System Test

```
1 public class Bakery {
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         new CookieMonster().eatAllCookies(j);
8     }
9     private static void loadMoreCookies(CookieJar j) {
10        j.add(new ChocolateCookie());
11    }
12 }
```

Generated Unit Test: Setup

```
1 public class AutoCookieMonsterTest extends MockObjectTestCase {
2   public void testCookieEating() throws Throwable {
3     // Set up primary object.
4     final CookieMonster testedCookieMonster = new CookieMonster();
5
6     // Set up expectations and run the test.
7     final Cookie mockCookie = mock(Cookie.class);
8     final Cookie mockCookie1 = mock(Cookie.class);
9     final CookieJar mockCookieJar = mock(CookieJar.class);
```

Generated Unit Test: Expectations

```
11      verifyThenCheck(new Expectations() {{
12        one (mockCookieJar).getACookie();
13        inSequence(s);
14        will(returnValue(mockCookie));
15
16        one (mockCookie).eat();
17        inSequence(s);
18
19        one (mockCookieJar).getACookie();
20        inSequence(s);
21        will(returnValue(mockCookie1));
22
23        one (mockCookie1).eat();
24        inSequence(s);
25
26        one (mockCookieJar).getACookie();
27        inSequence(s);
28        will(returnValue(null));
29      }});
```

Generated Unit Test: Assertions

```
31        assertThat(testedCookieMonster.eatAllCookies(mockCookieJar),
32          is(2)
33        );
34    }
35 }
```

Tests are Brittle

- `CookieMonster` isn't specified to eat each cookie before it takes the next one
- Two months later, an optimization makes it eat them all at the end
- The test starts failing
- With previous work, such as Test Factoring, all you can do is throw out the test (if you can even understand why it's failing)
- With `amock`, you can just relax the expectations by hand!

Failing Unit Test

```
11      verifyThenCheck(new Expectations() {{
12        one (mockCookieJar).getACookie();
13        inSequence(s);
14        will(returnValue(mockCookie));
15
16        one (mockCookie).eat();
17        inSequence(s);
18
19        one (mockCookieJar).getACookie();
20        inSequence(s);
21        will(returnValue(mockCookie1));
22
23        one (mockCookie1).eat();
24        inSequence(s);
25
26        one (mockCookieJar).getACookie();
27        inSequence(s);
28        will(returnValue(null));
29      }});
```

Relaxed Unit Test

```
11    verifyThenCheck(new Expectations() {{
12      one (mockCookieJar).getACookie();
13      inSequence(s);
14      will(returnValue(mockCookie));
15
16      one (mockCookie).eat();
17      // no inSequence any more
18
19      one (mockCookieJar).getACookie();
20      inSequence(s);
21      will(returnValue(mockCookie1));
22
23      one (mockCookie1).eat();
24      // no inSequence any more
25
26      one (mockCookieJar).getACookie();
27      inSequence(s);
28      will(returnValue(null));
29    }});
```

Recognizing Patterns

- Amock can recognize several patterns in the tested code
- Can write more concise and natural test code
- Configurable

Iterators

- We need to specify that an expected method call returns an iterator over 3 objects
- Naïvely, we would have it return a mock `Iterator` with seven expectations (4 `hasNext` calls, 3 `next`)
- Hard to follow and redundant
- Instead, write `will(returnIterator(foo, bar, baz));`

Record classes

- Some classes just aggregate data without logic
- We can trust that they work
- Would rather read `new Rectangle(1,2,3,4)` than `mock(Rectangle.class)` and many expectations on `getX`, `getWidth`, etc.

Naming static fields

- Java programmers use "static fields" for constants
- `SVNRevision.UNDEFINED`, `SVNArgument.PASSWORD`, etc.
- Tests should be able to refer to them by name
- `amock` detects static field reads

Purity (future work)

- There are many side-effect analyses for Java
- `amock` should be able to use purity information to be more lax about order and repetitions of pure methods

JHotDraw

- Framework for creating drawing programs
- 10K LOC
- Graphical user interface

JHotDraw: Testing the connection tool

```
23      // Create mocks.
24      final ConnectionFigure mockConnectionFigure
25          = mock(ConnectionFigure.class);
26      final DrawingView mockDrawingView
27          = mock(DrawingView.class);
28
29      // Set up primary object.
30      final ConnectionTool testedConnectionTool
31          = new ConnectionTool(mockDrawingView,
32                               mockConnectionFigure);
33      final Rectangle testedRectangle = new Rectangle(0, 0, 0, 0);
34      final Rectangle testedRectangle1 = new Rectangle(0, 0, 0, 0);
35
36      // Set up expectations and run the test.
37
38      verifyThenCheck(new Expectations() {{
39        one (mockDrawingView).clearSelection();
40        inSequence(s);
41      }});
42
43      testedConnectionTool.activate();
```

SVNKit

- Java client for Subversion
- 60K LOC
- Network and filesystem

Results of case studies

- amock successfully creates passing unit tests for multiple classes generated from multiple executions for both case studies
- Some areas where amock fails would be difficult to write tests for even manually
  - Unmockable dependencies
  - Private code worthy of testing

Efficiency

- Measure time and space overhead of trace
- Analyze sources of inefficiency

Robustness

- Resistance to false failures
- Generate passing test suites for older version of SVNKit
- Run against newer versions, observing how many pass and how many fail
- Classify:
  - Failure due to changes that would affect even hand-written tests
  - Spurious failures that are trivially correctable
  - Spurious failures that are not easily salvageable

Sensitivity

- Resistance to false passes
- Effectiveness in creating targeted unit regression tests
- Using change log, create system test for bugs fixed in SVNKit
- Generate unit tests against the fixed version of SVNKit
- Run against buggy version and see if they (correctly) fail

Contributions

- A new approach to test factoring which produces human-readable JUnit tests
- `amock`: An implementation of this approach
- Case studies showing the applicability of `amock` to real-world projects