Pseudo-Classes: Very Simple and Lightweight MockObject-like Classes for Unit-Testing

Geoff Sobering
Isthmus Group
222 State St Suite 300
Madison, WI
001-608-661-1234
geoff.sobering@isthmusgroup.com

Levi Cook
Isthmus Group
222 State St Suite 300
Madison, WI
001-608-661-1234
levi.cook@isthmusgroup.com

Steve Anderson
Berbee
5200 Research Park Drive
Madison, WI
001-608-298-1117
steve.anderson@berbee.com

ABSTRACT

A simple alternative to MockObjects is presented. Given the interface of an object required by a class-under-test, a Pseudo-Class is created implementing all methods such that they immediately fail. A test-specific sub-class of the Pseudo-Class is created locally in the test (ex. as an anonymous inner-class in Java), over-riding only the methods required by the interaction between the object and the class-under-test for the test-scenario. Typically, the method implementations are extremely simple (a few lines, at most), and the number of methods overridden is small. This mechanism was found adequate for more than 90% of our unit-tests (in a 1000-class system with over 2000 test methods, we finally ended up with about four real MockObject classes and more than 40 Pseudo-Classes).

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Design, Reliability, Verification.

Keywords

Unit-Tests, MockObjects, PseudoObjects, Test-Driven-Development, TDD, Test-First-Design, TFD

1.INTRODUCTION

During a recent Test-Driven Development (TDD) Project, we ran into the well-known problem of needing to supply instances of various classes that the "Class Under Test" required for a specific test-sequence. We first investigated MockObjects [1] but found the formalism more complex than most of our tests required. In particular, we found the separation of the mock-class from the test-class reduced the clarity of the test. We observed that each test in a good unit-test suite usually covers only a small portion of the behavior of the class-under-test. Thus, only simple "stand-in" implementations are required to support the class-under-test's interactions.

The major problem was finding a way to explicitly, simply, and compactly define the exact nature of the interaction between the class-under-test and its associated classes.

Note that pseudo-classes are not intended as a replacement for full-blown MockObjects. In some cases with complex interactions between a class-under-test and its associates we did find that a MockObject (with it's `verify()` method) was the best way to capture/express the intent of a test. However, we did find that there was about a 100:1 ratio of tests where a pseudo-class was sufficient vs. the need for a real mock.

OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

2.IMPLEMENTATION

In keeping with the desire for simplicity and clarity, the implementation of Pseudo-Classes is nearly trivial. The starting point is a type-declaration interface (Java™ terminology and semantics will be used for examples). The example we'll use here is a commissioning system for insurance agents. The basic behavior is that an agent gets a commission for every sale they make. That commission calculation is different for full-time and part-time agents.

As always, we start with a test first (JUnit here). In this case the test is nearly trivial:

```
public void testFullTime()
{
   CommissionCalculator calculatorUT =
      new CommissionCalculator();

   Money saleAmount =
      new Money( "1000.00" );

   Money expectedCommissionAmount =
      new Money( "100.0" );

   Agent fullTimeAgent =
      new FullTimePseudoAgent();

   Money actualCommissionAmount =
      calculatorUT.calculateCommission
         ( saleAmount, fullTimeAgent );

    assertEquals( expectedCommissionAmount,
                  actualCommissionAmount );
}
```

The method we are testing is `CommissionCalculator.calculateCommission()`. It takes a sale-amount and the selling-agent as parameters and returns the commission amount. The sale-amount is easy to supply, since the `Money` class is a simple value-type, we can just construct one. The selling-agent is tougher. In the real system I adapted this example from, the production `Agent` implementation is fairly expensive to create, and requires some state from a data-base. Instead of trying to manage all that, we supply an instance of `FullTimePseudoAgent`:

```
private static class FullTimePseudoAgent extends
PseudoAgent
{
    public boolean isFullTime() { return true; }
}
```

The definition is easy to read and understand, and it can live adjacent to the test-method as well (it could even be defined as an anonymous inner-class inside the test-method, but in this instance we thought the test was clearer this way). An important thing to note is that, for a simple interaction like the one in this test (which is quite representative of many of the 2000+ tests we wrote), the complexity of the test-code is independent of the complexity of the `Agent` interface; even if `Agent` has 100 methods on it, we still need only override one of them for this test.

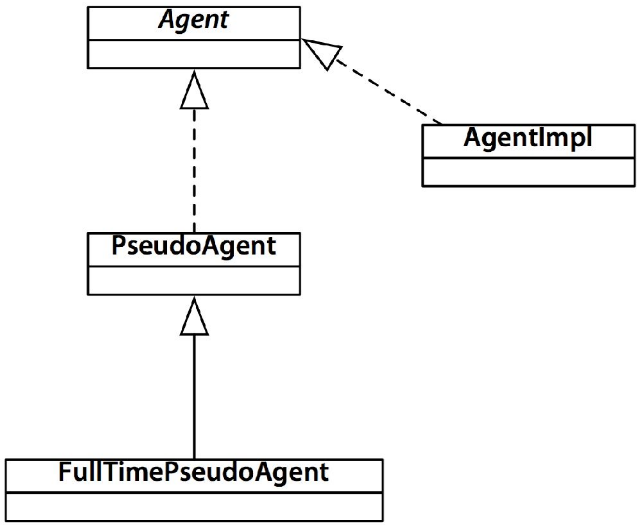

Figure 1 -The abstract relationships between the classes.

Now that we've seen how a typical test can be simply coded, and the relationship between classes in our example `Agent` hierarchy (figure 1), it's time to look in a bit more detail and see how the pseudo-classes are defined.

Our first code example is the `Agent` interface (note that in a real system there would probably be many more methods):

```
public interface Agent
{
    public boolean isFullTime();

    public Account getAccountFor(AccountType key);
}
```

Next, we show the `PseudoAgent` used as the super-class for our `FullTimePseudoAgent` in the test class we defined on the previous page. This would also be the superclass of all other pseudos of the `Agent` interface used anywhere in tests.

```
public class PseudoAgent implements Agent
{
    public boolean isFullTime()
    {
        throw new UnimplementedPseudoClassError();
    }

    public Account getAccountFor(AccountType key)
    {
        throw new UnimplementedPseudoClassError();
    }
}
```

Note that, the implementation is trivial (in fact, easily automatable). Every method in the interface is implemented to immediately throw an uncaught exception unique to the pseudo-framework.

3.DISCUSSION

The benefits of using pseudo-classes in unit-tests have been alluded to above. In this section we will describe then in more detail.

The first benefit is the one shown in the example above. For a simple "getter-like" interaction between the class-under-test and its associates, the test values are easy to insert, and the nature of the interaction is clear. In the example above, for instance, it is impossible for the `calculateCommission()` method to call anything other than `isFullTime()` on the supplied `Agent` instance (without throwing an exception).

The second situation where pseudo-classes are very helpful is as "filler arguments" in a test-scenario that requires a parameter be present (ex. to satisfy the method signature), but where the scenario has no interaction with the parameter. Previously, this case was often handled by passing in a `null` reference. Once one is familiar with the pseudo-pattern, supplying an instance of the un-extended pseudo-class is a clear statement that no methods will be called.

Third, supplying a specifically typed instance for a dependent object will also catch any "hidden" casting inside the class-under-test. For example, if one aggressively applies the "interface segregation principle" [2], then one implementation class may implement a number of different interfaces. It is not uncommon for a class to cast an argument from its supplied type to another one that it "is known" to implement (sometimes in an attempt to mimic dynamic-typing behavior in a statically-typed language). The pseudo-class framework addresses this in two ways. First, since a typical pseudo-class implements only one type, any cast away from one of its super-types will cause the test to fail immediately. Second, if such casting is necessary, one can expose it more clearly in the test by supplying a pseudo-class instance that *explicitly* implements all the types it is cast to in the test-scenario.

Finally, in a test-driven-design (TDD) development environment, it is common in the early stages of the development of a new class to require an instance of an existing associated class. The test-code can first supply an un-extended pseudo-class. As behavior is added to the class-under-test, the test will fail with the unique pseudo-class framework exception whenever a new method on the supplied instance is used. The developer can quickly implement the required behavior in the pseudo, and continue work on the new feature.

4.CONCLUSION

We have found the creation of pseudo-classes for many of the core objects in a medium-sized system (ca. 120,000 lines-of-code and 1500 classes) helped simplify and clarify many of the unit-test scenarios. The pseudo-classes were easy to implement, manage and maintain.

5.REFERENCES

[1] Mackinnon, T., Freeman, S., Craig, P., *Endo-Testing: Unit Testing with Mock Objects*, Proc. eXtreme Programming and Flexible Processes in Software Engineering (2000)

[2] Martin, R., *The Interface Segregation Principle*, C++ Report (1996)