ObjectMother
Easing Test Object Creation in XP

Peter Schuh
ThoughtWorks, Inc.
Suite 600
651 W. Washington
Chicago, IL 60661
+1 312 373 8433
peschuh@thoughtworks.com

Stephanie Punke
ThoughtWorks, Inc.
Suite 600
651 W. Washington
Chicago, IL 60661
+1 312 373 8429
spunke@thoughtworks.com

ABSTRACT
One of the most time-consuming and unfulfilling activities in testing is coding and maintaining test objects. The ObjectMother pattern addresses this issue by proposing a simple, scalable framework whose sole purpose is to construct and facilitate the customization of test objects. The pattern's primary responsibilities are: to create test objects, to tailor those objects at any point during the testing process, and to terminate those objects once testing is complete. When the process of test-writing is made easier and the quality of test data more consistent, developers are likely to write more and better tests.

Keywords
eXtreme Programming, ObjectMother, test data creation, automated tests, unit testing

1 INTRODUCTION
What developer hasn't stared at an empty unit test intended to ensure the proper functioning of getThis() on object That, dismayed at the prospect of building test objects A through Zed and every one of their distant Kind or Kin in order to test that one little innocent method? In such cases, possibly, the only thing more frustrating than weaving such a tangled object structure is maintaining that same structure as the application-in-development wends its way toward completion. This problem, unfortunately, is only exacerbated by Extreme Programming, which encourages organic growth, iterative development and just-in-time code.

Nonetheless, an exhaustive unit-test suite is a necessary requirement in any proper development process. By quickly and efficiently passing verdict on the suitability of any given build, a test suite can save countless hours of debugging, user-acceptance testing and managerial nail-biting. Even with these obvious benefits, however, developers often have to be cajoled, bribed or otherwise induced to both write new tests and maintain the old. As applications and their object structures grow ever larger and more complicated, even advocates of unit-testing can be put off by the thought of testing a particularly tricky bit of functionality perched atop a heap of varied and pesky business objects. "Well, I suppose I didn't expect to actually write any real code today."

There is a more intelligent way to unit test. The ObjectMother pattern provides developers with a fabrication plant for testable business objects. It can deliver objects-to-order via only a handful of method calls. The delivered objects are customizable and updateable, so they can fit the needs of most any test. Because specific classes are dedicated solely to the task of test data generation—and little or none of this activity happens outside of those classes—the process of locating and updating specific test object creations is greatly simplified. Finally, the pattern allows database-dependent applications to run a "clean" test suite; that is, one that neither relies upon set-up data nor leaves its bought-and-forgotten business objects scattered about like toys in a child's bedroom.

2 OBJECTMOTHER AND TESTING
ObjectMother is intended to compliment a xunit testing framework[1]. By relieving test classes of the burden of constructing test objects, ObjectMother allows them to be dedicated to their intended function—testing. The result is better test objects and cleaner test code.

The pattern covers an application in much the same way as an xunit framework, and the application needs no special architecture to support the pattern. Therefore, while it is best to begin writing an ObjectMother at the same time you begin writing tests—that is, before you write any real code—ObjectMother can be easily retrofit onto an existing application.

The ObjectMother pattern is superior to back-ending test

[1] For a good introduction to the xunit framework check out its archetype, junit, at the junit website.[2] For the full array of xunit testing frameworks, point your browser at the xprogramming website.[4]

data into the application—that is, via the database—before every build.[2] While tools can be used to make the SQL generation process quick and painless, and while loading a database happens much faster than constructing objects in memory, a back-end loading process has two serious drawbacks. First, because the loaded data is available for any test to use, there is no guarantee that one test will not corrupt another test's data. Second, because the data is being maintained in a system separate from the application code, keeping the test data in sync with the code is a never-ending struggle. Granted, in our experience, it does take longer to build an ObjectMother than a data-loading framework, but ObjectMother pays for itself by being both more dependable and more flexible.

Finally, the pattern is neither Stubs nor Mock Objects[3]. The purpose of the pattern is to generate business objects that resemble as closely as possible actual objects that will exist in production. (Because of this, on one of our projects, we actually had developers pairing with analysts in order to ensure that the objects being generated by the pattern were as close as possible to the real thing.) So, the closer the test data is too the real data, the better the unit tests will be able to test for real problems that may surface in the application.

3 OBJECTMOTHER BASICS

ObjectMother manages the lifecycle of test objects, including creation, customization and, when necessary, deletion. The pattern relies largely on two distinct types of methods: creation methods that return all manner of business objects and attachment methods that assist in tailoring the created objects. These methods are used by both ObjectMother and the unit tests that implement it. The pattern, itself, can be architected in two distinct fashions. For smaller applications, it can take the form of a single utility class with a phalanx of creation and attachment methods. In larger applications, ObjectMother may manifest itself as a framework of objects diffused across the application, each generating and capable of tailoring a specific business object, accessible both to the test writer and to other nodes within the ObjectMother framework. This bifurcation in the pattern is due entirely to size; for larger applications, no one would want to maintain a ten-thousand line utility class.

Writing an ObjectMother is a lot like building with legos. Smaller, simpler objects are locked together to create larger, more complex objects. An ObjectMother method might look something like this: [4]

```
public static Invoice createNewInvoice() {
  Invoice invoice = new Invoice();
  invoice.setInvoiceNumber("InvTest001");
  Address address = createAddress();
                                // #1
    invoice.setBillToAddress(address);
    attachInvoiceLineAsCharge(
              invoice,
              new Money("4999.95","USD"));
                                 // #2
    invoice.setStatus(InvoiceStatus.NEW);
    return invoice;
}
```

The first major point in the above sample is the createAddress() method call, which shows how ObjectMother's creation methods are designed to serve two purposes. First, the creation methods are employed by each other to create ever-more complex test objects. Second, unit tests will use these same methods to obtain test objects from ObjectMother. This means that ObjectMother's creation methods should be made public and accessible by any test.

The second point of interest is the attachInvoiceLineAsCharge() method call. Accepting an Invoice object and a Money object as parameters, this method gets an InvoiceLine object that is a Charge based on the Money parameter, then adds that InvoiceLine to the Invoice object. It might look something like this:

```
public static void attachInvoiceLineAsCharge
                           (Invoice invoice,
                            Money money) {
  InvoiceLine invoiceLine =
           createInvoiceLineAsCharge(money);
  invoiceLine.setInvoice(invoice);
}
```

First, note that, in our current code sample, the createInvoiceLineAsCharge() method performs all the dirty work involved in building our InvoiceLine, setting its status, and adding the desired Money object (all code that would otherwise have to be written into the actual test). Second, the attachment method, itself, does little more than call another creation method. Why push this small bit of code out into its own method? Because the primary role of attachment methods should be to resolve relationships between objects. Therefore, if the application that this ObjectMother supports was updated with a many-to-many relationship between Invoice and InvoiceLine, possibly the only change that one would need to make to ObjectMother would be to retool the above method to look something like this:

[2] This is a process we attempted on some ThoughtWorks projects with very limited success.

[3] For a good introduction to Stubs, see Binder.[1] To learn more about the Mock Object pattern—which can be a very useful testing tool—check out the paper by Mackinnon et al.[3]

[4] For simplicity's sake, this code sample and all that follow will assume that ObjectMother is being implemented as a single class. It should be easy enough to imagine how the sample may look different if taken from an ObjectMother framework.

```
public static void attachInvoiceLineAsCharge
                           (Invoice invoice,
                            Money money) {
  InvoiceToInvoiceLine invToInvLine
              = new InvoiceToInvoiceLine();
  InvoiceLine invLine =
             getInvoiceLineAsCharge(money);
  invToInvLine.setInvoiceLine(invoiceLine);
  invToInvLine.setInvoice(invoice);
}
```

Admittedly, other changes might need to be made; nonetheless, ObjectMother has begun to take on the look, feel and advantages of well-rendered OO code. And, similar to creation methods, attachment methods generally should be made public, so that they may be used by tests to augment prefabricated test objects.

4 THE DETAILS

In the sections above, we have referred to ObjectMother as a factory for test objects; this is true, but it is now time to move beyond analogies and get busy with the particulars—specifically, a definition for our pattern. We propose that ObjectMother is an object or set of objects that:

1. provides a fully-formed business object along with all of its required attribute objects,
2. returns the requested object at any point in its lifecycle,
3. facilitates customization of the delivered object,
4. allows for updating of the object during the testing process, and
5. when required, terminates the object and all its related objects at the end of the test process.

Just as applications should be thought out before they are designed and designed before the are written, so to should test suites. Therefore, it is worth spending some time on the components of our definition.

First, whenever possible, ObjectMother should return a valid, test-ready business object. This should be common sense, since running tests against incomplete or invalid business objects is simply an invitation for disaster. However, this principle is worth noting because there are times it may actually have to be broken. On occasion, in order to avoid code duplication, incomplete objects may need to be built by one creation method for use by other creation or attachment methods. When possible, these methods should be made private; however, if your ObjectMother is a framework of nodes spread across numerous test packages, one is forced to rely on good documentation and competent developers. Either way, such instances should be kept to a minimum.

Second, ObjectMother should provide any given business object in any of the various incarnations or statuses it may be found. Therefore, building on our code sample above, ObjectMother might contain a gaggle of the following methods:

```
public static Address createAddress(
                               String city,
                               String state,
                               String zip) {
  Address address = new Address();
  address.setAddressLine1("1011 Bit Lane");
  address.setCity(city);
  address.setState(state);
  address.setZip(zip);
  address.setStatus(AddressStatus.ACTIVE);
  return address;
}

public static Address createAddress() {
  Address address = createAddress("Chicago",
                                  "IL",
                                  "60647");
  return address;
}

public static Address
                 createInactiveAddress() {
  Address address = createAddress();
  address.setStatus(AddressStatus.INACTIVE);
  return address;
}
```

Notice how each creation method builds on the one above it in order to generate its result. The goal is to return any given object in a variety of formats any number of times, in order to provide for a wide array of easily-accessible test objects. Ideally, the constructor call to any given object will appear within ObjectMother only once—or, in reality, as few times as is absolutely necessary. This should be done both to avoid code duplication and simplify maintenance.

Third, when they return valid business objects, ObjectMother's attachment methods should be made public so that tests may customize objects received via creation methods.[5] For example, a test that requires an invoice with four lines might contain code something like this:

```
public void testInvoice() throws Exception {
  Invoice invoice =
    ObjectMother.createNewInvoice();
         // Remember that createNewInvoice()
         // returns an Invoice with one
         // InvoiceLine already attached
  ObjectMother.attachInvoiceLineAsCharge(
               invoice,
               new Money("199.95","USD"));
  ObjectMother.attachInvoiceLineAsCharge(
```

[5] Similar to creation methods, attachment methods should return valid business objects whenever possible. It is understandable, however, that at times ObjectMother may have attachment methods for internal use that do not return valid objects.

```
                invoice,
                new Money("100","USD"));
  ObjectMother.attachInvoiceLineAsCharge(
                invoice,
                new Money("20","USD"));
    ... //And on with the test
  }
```

By making attachment methods public, test-writers may customize the objects they receive however they see fit. When several tests are found to be customizing a test object in the same fashion that code may be pushed up into ObjectMother as either a creation or attachment method.

Fourth, and similar to the above point, tests should be able to update ObjectMother-provided objects as necessary in the middle of the test process. Often, this will mean nothing more than running a test on a created object, calling an attachment method to alter the object, then performing another test. Sometimes, however, this requirement may involve more complex procedures, such as altering a business object's status. Procedures of this type might involve the implementation of new types of ObjectMother methods. For example, to alter a test object's status a development team may implement a makeStatus() method. The ObjectMother code for such a method might look something like this:

```
public static void makeGenerated(
                          Invoice invoice) {
  invoice.setGeneratedDate
                (new Date("10 Jan 2001"));
  invoice.setDueDate
                (new Date("10 Feb 2001"));
  invoice.setStatus
                (InvoiceStatus.GENERATED);
      //Invoice status has been updated

  Iterator it =
    invoice.getAllInvoiceLines().iterator();

  while (it.hasNext()) {
    InvoiceLine invoiceLine =
                (InvoiceLine) it.next();
    invoiceLine.setStatus
            (InvoiceLineStatus.GENERATED);
  }
     //All the InvoiceLines are also updated
}
```

The key point here is that, however it is done, ObjectMother must allow for changes to its created objects that emulate how those objects are manipulated within the application itself.

5 OBJECTMOTHER ADD-ONS

The astute reader might have noticed that we have yet to address the fifth, and final, point of our ObjectMother definition: test object deletion. This is because the best solution here is to avoid having to fiddle with deletion in the first place. If you have architected your application so that it may be run entirely in memory—via the use of POJOs (Plain Ordinary Java Objects[6]) or some other layer of abstraction—then, when desired, the application may be run without ever writing a bit to the database. If nothing is recorded, then nothing needs to be removed, and the final component of our pattern is yours for free.

If, however, your application must hit the database while the unit tests are running, you will have to implement a registration method within ObjectMother. This registration method will need to be called for every new object created, so that ObjectMother can compile a list of all of its created objects and delete them when the tests have completed. Furthermore, each individual test must notify ObjectMother when it is complete. This can be easily accomplished with a complete(), purge() or apocalypse() method call, but it must be done at the completion of any test that employs ObjectMother. [7]

Finally, whether or not you need to delete objects at the end of your test run, there are additional services a registration method may provide. For example, most applications have audit fields, such as createdBy and createdDate. A registration method may be used to perform such rudimentary, object-related tasks as setting those attributes.

6 CONCLUSION

This paper has detailed what we believe are the essential ingredients to be found in any ObjectMother recipe (salt to taste, serve with bread). The three major benefits that the ObjectMother pattern offers are (1) simplified and standardized test object creation, (2) ease of maintenance, because test object creation is entrusted to a specific class or group of classes, and (3) test object clean-up. The one potential downside is the added time spent building the pattern. We believe ObjectMother more than makes up for this cost. First, the pattern recovers even greater amounts of time that would otherwise be spent writing and maintaining unit tests. Second, by removing a significant hurdle from the test-writing process, ObjectMother encourages developers to write more tests. Lastly, the test suite benefits from being written with the same think-more write-less mentality that should be found in good code everywhere.

[6] POJOs is a term we have begun using around ThoughtWorks to reintroduce and reinvigorate the reputation and utility of the simple stand-alone Java object.

[7] You may ask whether this step is really necessary; what harm will it do to just let the data pile up, or why can't we simply reload a fresh database? Two reasons. First, if your developers are writing tests and testing their code as often as they should be, data will pile up much quicker than you might imagine. Second., by running "clean" ObjectMother allows you to swap in copies of UAT or production data, and to test, code and debug against that data without contaminating it.

ACKNOWLEDGEMENTS
Thanks to Martin Fowler for his consultation and guidance. Also, credit must be given to the members of the Atlas and E-lese teams—two ThoughtWorks projects that pioneered the ObjectMother pattern.

REFERENCES

1. Binder, Robert V. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.
2. JUnit Website, online at: http://www.junit.org/
3. Mackinnon, Tim, Steve Freeman and Philip Craig, "Endo-Testing: Unit Testing with Mock Objects." Available at: http://www.connextra.com/about/xp_approach.htm
4. XProgramming Website, software section, online at http://www.xprogramming.com/software.htm