Mistaeks I Hav Made

“Good judgement is the result of experience … Experience is the result of bad judgement.” — Fred Brooks

Test Data Builders: an alternative to the Object Mother pattern

If you are strict about your use of constructors and immutable value objects, constructing objects in a valid state can be a bit of a chore.

Usually in application code, such objects are constructed in few places and all the information required by the constructor is at hand, having been provided by user input, obtained from a database query or received in a message for example. In tests, on the other hand, you have to provide all those constructor arguments every time you want to create an object, whether to test its behaviour or to create a value to use as input to the code being tested.

```
Invoice invoice = new Invoice(
    new Recipient("Sherlock Holmes",
        new Address("221b Baker Street",
                    "London",
                    new PostCode("NW1", "3RX"))),
    new InvoiceLines(
        new InvoiceLine("Deerstalker Hat",
            new PoundsShillingsPence(0, 3, 10)),
        new InvoiceLine("Tweed Cape",
            new PoundsShillingsPence(0, 4, 12))));
```

The code to create all those objects makes tests messy and hard to read and fills the tests with lots of unnecessary information that has nothing to do with the behaviour being tested. It also makes tests brittle: changes to the constructor arguments or the structure of the objects will break many tests.

The Object Mother pattern is one attempt to avoid this problem. An Object Mother is a class that contains a number of (usually static) Factory Methods that create objects for use in tests. For example, we could create an Object Mother for invoices we want to use in tests:

```
Invoice invoice = TestInvoices.newDeerstalkerAndCapeInvoice();
```

An Object Mother helps keep tests readable by moving the code that creates new objects out of the tests themselves and giving clear names to the objects being constructed. It also helps maintain the test data by gathering the code that creates new objects together into the Object Mother class and allowing it to be reused between tests.

However, the Object Mother pattern does not cope at all well with variation in the test data. Every time programmers need some slightly different test data they add another factory method to the Object Mother.

```
Invoice invoice1 = TestInvoices.newDeerstalkerAndCapeAndSwordsti
Invoice invoice2 = TestInvoices.newDeerstalkerAndBootsInvoice();
...
```

Over time, the Object Mother becomes bloated, messy and hard to maintain. Either programmers add new factory methods without refactoring, in which case the Object Mother becomes full of duplicated code, or programmers refactor diligently, in which case the Object Mother becomes full of many, many fine-grained methods that each contain little more than a single new statement.

A solution is to use the Builder Pattern. For each class you want to use in a test, create a Builder for that class that:

1. Has an instance variable for each constructor parameter
2. Initialises its instance variables to commonly used or safe values
3. Has a `build` method that creates a new object using the values in its instance variables
4. Has "chainable" public methods for overriding the values in its instance variables.

For example, a builder of Invoice objects might look like:

```
public class InvoiceBuilder {
    Recipient recipient = new RecipientBuilder().build();
    InvoiceLines lines = new InvoiceLines(new InvoiceLineBuilder
    PoundsShillingsPence discount = PoundsShillingsPence.ZERO;

    public InvoiceBuilder withRecipient(Recipient recipient) {
        this.recipient = recipient;
        return this;
    }

    public InvoiceBuilder withInvoiceLines(InvoiceLines lines) {
        this.lines = lines;
        return this;
    }

    public InvoiceBuilder withDiscount(PoundsShillingsPence disc
        this.discount = discount;
        return this;
    }

    public Invoice build() {
        return new Invoice(recipient, lines, discount);
        }
    }
```

Tests that don't care about the precise values in an Invoice can create one in a single line:

```
    Invoice anInvoice = new InvoiceBuilder().build();
```

Tests that want to use specific values can define them inline without filling the test with unimportant details:

```
    Invoice invoiceWithNoPostcode = new InvoiceBuilder()
        .withRecipient(new RecipientBuilder()
            .withAddress(new AddressBuilder()
                .withNoPostcode()
                .build())
            .build())
        .build();
```

I've used Builders for creating test data on a couple of projects now and I've found that, compared to Object Mothers, they make it much easier to create test data in-line in the test code without making tests brittle or creating lots of duplication. Tests are isolated from those aspects of the objects' structure that have no bearing on the test. For example, code that creates the invoice with no postcode needs to know that an invoice has a recipient, that has an address, that has a postcode, but has no further dependencies on the structure of invoices, recipients and addresses. You can add constructor arguments without breaking tests at all. Removing constructor arguments is easy as well with modern refactoring IDEs.

Another benefit is that the test code is easier to write and read because the parameters are clearly identified. Compare:

```
    TestAddresses.newAddress(
        "Sherlock Holmes",
        "221b Baker Street",
        "London",
        "NW1");
```

to:

```
    new AddressBuilder()
        .withName("Sherlock Holmes")
        .withStreet("221b Baker Street")
        .withCity("London")
        .withPostCode("NW1", "3RX")
        .build();
```

Nothing in the first example will tell you that "London" has been accidentally passed as the second street line instead of the city name.

In some cases, Builders have so improved the code that they ended up being used in the production code as well.

Further techniques for using Test Data Builders:

- Define common state and avoid aliasing problems
- Combining builders
- Emphase the domain model with factory methods
- Factoring out duplicated logic creates a domain-specific embedded language for testing

Credits: builder picture from the Keep Scotland Beautiful campaign.

Update: thanks to Richard Hansen for pointing out a typo in the builder code, which is now fixed.
