The ‘Self’-Shunt Unit Testing Pattern

Michael Feathers
Object Mentor, Inc.
mfeathers@objectmentor.com

Let's say that you are a test case. One of the things that you can do is pass yourself to the objects you are testing so that you can get more information.

Test first design is fun, but in the beginning, it can be a bit overwhelming. There are all sorts of little fears. You sit at your computer, with that empty screen just staring at you. You pick a test that you want to write, but you stop. “What if I write this test and it passes, and I write another, and another, and then I discover that my objects need to put things on the GUI. How do I write a test for that?” If you are not careful, those sorts of thoughts will derail you for a while. Hopefully, you do hunker down to write the first test and move on, confident that you’ll find a way to write that test when the time comes. This article is about one very interesting strategy you can use when that time comes.

Let’s suppose that we are writing a point of sale system. The first user story tells us that when a sales clerk swipes an item past a barcode scanner, its name and price come up on an LCD display. We don’t have any wiggle room here. It doesn’t look like there are any tests between the display problem and us.

Let’s think about this. We can have a scanner object and we can have a display object, and we can get the item from the scanner and pass it to the display. If we write this up in JUnit, it looks like this:

```
public class ScannerTest extends TestCase
{
      public ScannerTest (String name)  {
            super (name);
      }

      public void testScanAndDisplay ()  {
            Scanner scanner = new Scanner ();
            Display display = new Display ();

            Item item = scanner.scan ();
            display.displayItem (item);

      }
}
```

But, what do we assert? Do we add a method to the display to ask it whether it has displayed anything? That seems pretty artificial. The name of our test case seems suspicious also. The name “testScanAndDisplay” screams out the fact that we are testing two things. Worse, the test case is acting as an intermediary between the scanner and the display. In your application, some object is going to fulfill that role. We might as well confront that now.

```
public class ScannerTest extends TestCase
{
      public ScannerTest (String name)  {
            super (name);
      }

      public void testScan ()  {
            Display display = new Display ();
            Scanner scanner = new Scanner (display);

            scanner.scan ();
      }
}
```

- Test-First Design Tip -
If your test acts as a mediator between two objects,
pick one object and let it talk to the other

We’ve passed the display to the scanner, but we still need to find out whether *scan ()* did the right thing. Will the display be updated?

What forces do we have? The display will know if it has been updated, but the test case needs to know. Under test, we don’t need a real display object. In fact, a real display object could be downright irritating, flickering wildly as we run hundreds of unit tests. Why don’t we make the test case impersonate a display and pass it to the scanner?

```
                                          // act like a display
public class ScannerTest extends TestCase implements Display
{
       public ScannerTest (String name)  {
              super (name);
       }

       public void testScan ()  {

              // pass self as a display
              Scanner scanner = new Scanner (this);

              // scan calls displayItem on its display
              scanner.scan ();

              assertEquals (new Item ("Cornflakes"), lastItem);
       }

       // impl. of Display.displayItem ()
       void displayItem (Item item) {
              lastItem = item;
       }

       private Item lastItem;
}
```

And that's the 'self'-shunt unit testing pattern. When a test case can impersonate one of your collaborators, it can check things that only a collaborator would know.

'Self'-shunt is a nice way to start a class when you need collaborators immediately, but often it is just a stepping-stone. 'Self'-shunting test cases can become unwieldy, as they get larger. At that point, normal refactoring rules apply. With an interface like Display in place, you can factor out a Mock Object [1], or even use a real object as it is developed, provided it sets up easily and allows you decent coverage.

History:

'Self'-shunt is one of those techniques that many people have discovered independently. I remember an XP Immersion class where at least two people jumped up and down and said "Hey, I've done that." when Robert Martin described it on a flip chart. Kent Beck said he'd been doing it for years, but I don't remember him jumping.

In any case, three uses make it a pattern. Me? I've been using it for a long time, but I have no idea whether I re-discovered it or heard about it somewhere.

Naming:

The name ‘Self’-shunt is really a ‘tip of the hat’ to the Shunt Pattern [2], where Alistair Cockburn describes a testing shunt as:

> “…basically a wire that goes out the back of one jack, and into an input, so the machine is connected to itself. When you run the machine, it thinks it is connected to the world, but it is only talking to itself. In software, the trick is to fake communication against the outside world, then run the tests locally. Then your testing is partitioned.”

A ‘Self’-shunt is a shunt made by passing yourself to another object.

References:

[1] Tim MacKinnon, Steve Freeman, Philip Craig, Endo-Testing: Unit Testing with Mock Objects. *eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*

[2] Anonymous. Shunt Pattern. *Portland Pattern Repository*. Dec 12, 2000. http://c2.com/cgi/wiki?ShuntPattern