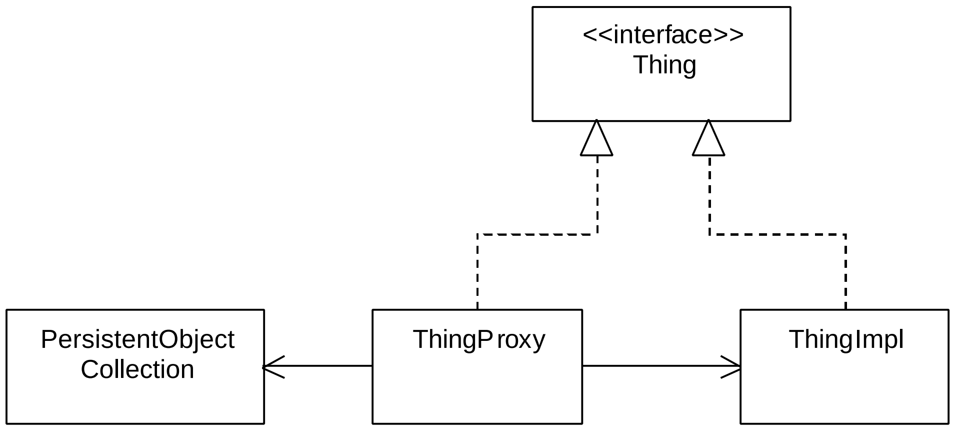
Evolution of an Inside-Outside Test

By James Grenning

The tests created using Test Driven Design are used to design the interface of a class, provide an example of the use of class and show that the class works. Some tests can reveal the implementation of the class. We often need these tests during the incremental development of the class, allowing development to proceed in small steps. The tests revealing the inside knowledge are undesirable once the class has been designed and is working. When the tests reveal the implementation of the class: the tests do not provide a good model for using the class and the tests are likely to need changes when the underlying implementation is changed. Here is a small case study showing the evolution of a test needing inside knowledge to a test needing only outside knowledge.

Mike and I sat down to add persistence for a class that contains a collection of objects. The object type is not important for this story, so I'll just call the object Thing. We have a persistence helper class that we used called PersistentObjectCollection. PersistentObjectCollection holds a container of objects that are initially loaded from a file and later saved to a file. We don't want to reveal the persistence mechanism to the rest of the application that uses Thing, so Mike and I decided that a proxy pattern met our needs. The design looks like this.



James W. Grenning
grenning@objectmentor.com

We were developing using Test Driven Design. We setup a test class for testing our new class ThingStoreProxy using JUnit. Our first test established how to create a ThingStoreProxy and what happens when we create the ThingStoreProxy with a file that does not exist. If we try to get messages from the proxy and the persistence file is empty, we will get a message Collection that is empty. This test also establishes that the proxy data is stored in a file.

```
//Code snip #1

public class ThingStoreProxyTest extends TestCase {
    private String fileName = "test.test";
    private File file = new File(fileName);

    public ThingStoreProxyTest(String name) {
        super(name);
    }

    public void testCreateEmpty() {
        file.delete();
        ThingStoreProxy proxy = new ThingStoreProxy(fileName);
        Collection things = proxy.getThings();
        assertNotNull(things);
        assertEquals(0, things.size());
    }
}
```

We decided next to have the proxy created with a file that has data in it. We had a bit of a chicken and egg problem. How can we create a proxy from a persistence file if we never saved persistent data before? Mike had the bright idea that we use the underlying mechanism in PersistentObjectCollection to create a persistence file and initialize from that. Then we can check if the objects that should be there are actually there.

```
//Code snip #2

    public void testCreate() throws Exception {
        file.delete();
        Collection initial = new ArrayList();
        initial.add("foo");
        initial.add("bar");

        PersistentObjectCollection persist =
                                  new PersistentMessageIO(file);
        persist.writeObjects(initial);

        ThingStoreProxy proxy = new ThingStoreProxy(fileName);
        Collection things = proxy.getThings();
        assertNotNull(things);
        assertEquals(2, things.size());
        assertTrue(things.contains("foo"));
        assertTrue(things.contains("bar"));
    }
```

We got our green bar. Notice that both tests have a file.delete() at their beginning. We saw this duplication and moved it to the setUp() method.

The next test we involved the creation of a proxy with an empty file, update the proxy with a couple objects, and then using the underlying mechanism of PersistentObjectCollection we checked to see if the file contained the new data.

```
//Code snip #3

    public void testUpdate() throws Exception {
        ThingStoreProxy proxy = new ThingStoreProxy(fileName);
        Collection initial = new ArrayList();
        initial.add("foo");
        initial.add("bar");
        proxy.update(initial);

        PersistentObjectCollection persist =
                              new PersistentObjectCollection (file);
        Collection things = persist.readObjects();
        assertNotNull(things);
        assertEquals(2, things.size());
        assertTrue(things.contains("foo"));
        assertTrue(things.contains("bar"));
    }
```

This test passes, so we finished adding persistence to the proxy. Wait! Not so fast. Notice that testCreate and testUpdate have intimate knowledge of the internals of ThingProxy. It was very useful to know that ThingProxy was being built using PersistentObjectCollection. That allowed us to incrementally build the ThingProxy. We put some of the implementation in our test. Once ThingProxy passed testCreate and testUpdate, we don't need that special inside knowledge in the tests. We changed the test from having special inside knowledge to needing only outside knowledge, shifting and inside test to an outside test.

```
//Code snip #4
    public void testPersist() throws Exception {
        ThingStoreProxy proxy = new ThingStoreProxy(fileName);
        Collection initial = new ArrayList();
        initial.add("foo");
        initial.add("bar");
        proxy.update(initial);

        proxy = new ThingStoreProxy(fileName);
        Collection things = proxy.getThings();
        assertNotNull(things);
        assertEquals(2, things.size());
        assertTrue(things.contains("foo"));
        assertTrue(things.contains("bar"));
    }
```

We created this test by combining testCreate and testUpdate into one test called testPersist. testPersist is a better test because it provides and example of how ThingProxy behaves. It also makes no assumption about the ThingProxy implementation. This is actually the first test I wanted to write. If we wrote this test first we would have had to take a much bigger step to get it to pass. The intermediate disposable tests, testCreate and testUpdate, allowed us to code in smaller steps.

I think this is an interesting story for a couple reasons. It shows a technique for starting with tests that is not obvious. The programmer can put some of the implementation into the test code. You could use this technique when you can't find a small first test to write that only has a small amount of code behind it. The other interesting thing this shows is that you are not done until you step back and look at your work. Are the tests providing an example of how to use the class just developed? Do the tests have unnecessary internal knowledge? Are any of my tests obsolete?