Mistaeks I Hav Made

"Good judgement is the result of experience ... Experience is the result of bad judgement." — Fred Brooks

State vs. Interaction Based Testing Example

In a comment to my previous post on this topic, Jason Yip asked for some example code. Here's a simplified example from a previous project. Note, this is all from memory and I haven't compiled or run any of this code; consider it pseudocode.

I was writing an interactive, graphical simulation. Well, a video game but that doesn't sound so impressive on a CV. The simulation rendered graphics, represented as `Drawable` objects, and ran simulation activities, represented as `Animated` objects, every timeslice. The `Drawable` and `Animated` interfaces are shown below:

```
public interface Drawable {
        void draw( GraphicsSurface g );
}
```

```
public interface Animated {
        void animate( double deltaT );
}
```

I implemented animated sprites as objects that implement both the `Drawable` and the `Animated` interfaces. A Sprite's animation was defined by a fixed number of Drawable cels and the duration that the cels will be displayed for. The Sprite displays each cel for the cel duration before stepping to the next in the loop.

Using mock objects (interaction based testing), my tests specified that a sprite draws its cels in order, stepping to the next after the cel duration.

```
public class SpriteTest extends MockObjectTestCase
        static final double CEL_DURATION = 1.0;

        GraphicsSurface display;
        Mock cel1;
        Mock cel2;
        Sprite sprite;

        public void setUp() {
                display = (GraphicsSurface )newDummy(GraphicsSurface.class,"graphics");

                cel1 = mock(Drawable.class,"cel1");
                cel2 = mock(Drawable.class,"cel2");
                List cels = new ArrayList();
                cels.add( cel1.proxy() );
                cels.add( cel2.proxy() );

                Sprite sprite = new Sprite( CEL_DURATION, cels );

        }

        public void testInitiallyDrawsFirstCel() {
                cel1.expects(once()).method("draw").with(same(display));

                sprite.draw( graphics );
        }

        public void testDrawsNextCelAfterCelDuration() {
                cel1.expects(once()).method("draw").with(same(display))
                        .id("draw tick 1");
                cel1.expects(once()).method("draw").with(same(display))
                        .after("draw tick 1")
                        .id("draw tick 2");
                cel2.expects(once()).method("draw").with(same(display))
                        .after("draw tick 2")
                        .id("draw tick 3");
                sprite.animate( CEL_DURATION/2 );
                sprite.draw( graphics );
                sprite.animate( CEL_DURATION/2 );
                sprite.draw( graphics );
        }

        ...
}
```

Here's a possible implementation that will pass these tests.

```
class Sprite implements Animated, Drawable {
        private List cels;
        private double celDuration;

        int currentCelIndex = 0;
        double celTimer = 0.0;

        public Sprite( double celDuration, List cels, ) {
                this.cels = (List)cels.clone();
                this.celDuration = celDuration;
        }

        public void draw( GraphicsSurface display ) {
                cels[currentCelIndex].draw(display);
        }

        public void animate( double deltaT ) {
                celTimer += deltaT;
                while (celTimer > celDuration) {
                        celTimer -= celDuration;
                        currentCelIndex = (currentCelIndex+1) % cels.size();
                }
        }
}
```

Now, how would I test this with state-based testing? I would have to expose the current cel as a property:

```
public class SpriteTest extends TestCase {
        ...

        public void testInitiallyDrawsFirstCel() {
                assertEquals( "should be showing first cel", 0, sprite.getCurrentCelIndex() );
        }

        public void testDrawsNextCelAfterCelDuration() {
                sprite.animate( CEL_DURATION/2 );
                assertEquals( "should be showing first cel", 0, sprite.getCurrentCelIndex() );
                sprite.animate( CEL_DURATION/2 );
                assertEquals( "should be showing second cel", 1, sprite.getCurrentCelIndex() );
        }
}
```

```
public class Sprite ... {
        ...
        public Drawable getCurrentCelIndex() {
                return currentCelIndex;
        }
}
```

But now I've had to create a new method on the Sprite class just for testing. This is not a good idea:

1. I'll have to maintain the method when I change the internals of Sprite.
2. The API is now more complex than it needs to be -- the `getCurrentCelIndex()` method is just noise that does not contribute to the required functionality of the class and will confuse maintenance programmers who have to learn how to use the class.
3. The tests are now misleading because they don't express how one should use the class: domain code should never call `getCurrentCelIndex()` but the tests say the opposite.
4. The tests do not actually test that the sprite draws the cel with the current cel index, only that it changes the current cel index. That's because drawing is done in a "Tell, Don't Ask" style — there is no state to assert about.

Drawback 1 is the one that hits you first when extending code. Later in the project I extracted the concept of a "clip" of cels so that a single clip could be shared by many sprite instances, and introduced finite and looped clips. Sprites used an iterator over cels instead of maintaining a current index. Looped animations were represented as a clip of infinite size: the iterator looped around the clip.

```
public interface CelIterator implements Drawable {
        boolean hasNext();
        void next();
}
```

```
public class Sprite ... {
        private CelIterator currentCel;
        private double celDuration;

        public Sprite( double celDuration, Clip clip ) {
                this.currentCel = clip.iterator();
                this.celDuration = celDuration;
        }

        public void draw( GraphicsSurface display ) {
                currentCel.draw(display);
        }

        public void animate( double deltaT ) {
                celTimer += deltaT;
                while (celTimer > celDuration && currentCel.hasNext() ) {
                        celTimer -= celDuration;
                        currentCel.next();
                }
        }
}
```

Apart from the `setUp()` method which passed a Clip to the Sprite instead of a List, I didn't have to change any tests. If I had used state based testing, on the other hand, what could I have done? The Sprite doesn't store the current cel any more. I could have exposed the current cel through the CelIterator, but now my tests are pushing a bad design decision into my domain code. They should be guiding me towards *good* designs, not bad designs.

What I care about is what my objects do, not what state they happen store to coordinate what they do, or happen to leave lying about in memory after they've finished doing what they do. The only visible manifestation of their behaviour are the messages that they send to other objects. That's why I find interaction based testing easier than state based testing when writing object-oriented code.

Posted on August 06, 2004 [Permalink | TrackBack (0)]

Comments

appreciate the article, definately helped clear up some confusion on state vs interaction based. As a newcomer to TDD I've been leary about having to create methods JUST for my tests to find out if they have succeeded. Do you find that you always use mocks for your object interactions and if so, how do you get test coverage on those helper objects?

thanks :)

Posted by: Jim Plush at July 6, 2005 12:18 AM

I'm not sure what you mean about getting test coverage for helper objects. I write all objects in the same way, and test them with mock objects if necessary, and then write integration tests to test aggregates of objects and do end-to-end tests through the system. Obviously I don't care about test coverage of the mock objects themselves since they are part of the test and usually generated dynamically by jMock or nMock.

Posted by: Nat at July 9, 2005 04:00 PM

Post a comment

Name:

Remember personal info?

Yes [✓ No]

Email Address:

URL:

Comments:

Preview Post