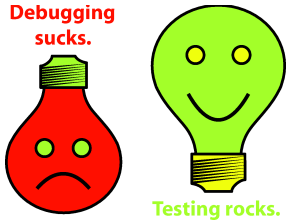

Testing on the Toilet

Testing Against Interfaces

July 24, 2008

To quell a lingering feeling of inadequacy, you took the time to build your own planetary death ray, a veritable rite of passage in the engineering profession. Congratulations. And you were feeling pretty proud until the following weekend, when you purchased the limited-edition Star Wars trilogy with Ewok commentary, and upon watching the Death Star destroy Alderaan, you realized that you had made a bad decision: Your planetary death ray has a blue laser, but green lasers look so much cooler. But it's not a simple matter of going down to Radio Shack to purchase a green laser that you can swap into your existing death ray. You're going to have to build another planetary death ray from the ground-up to have a green laser, which is fine by you because owning two death rays instead of one will only make the neighbors more jealous.

Both your planetary death rays should interoperate with a variety of other bed-wettingly awesome technology, so it's natural that they export the same Java API:

```
public interface PlanetaryDeathRay {
  public void aim(double xPosition, double yPosition);
  public boolean fire();  /* call this if she says the rebel base is on Dantooine */
}

public class BlueLaserPlanetaryDeathRay implements PlanetaryDeathRay { /* implementation here */ }
public class GreenLaserPlanetaryDeathRay implements PlanetaryDeathRay { /* implementation here */ }
```

Testing both death rays is important so there are no major malfunctions, like destroying Omicron Persei VIII instead of Omicron Persei VII. You want to run the same tests against both implementations to ensure that they exhibit the same behavior – something you could easily do if you *only once* defined tests that run against *any* `PlanetaryDeathRay` implementation. Start by writing the following abstract class that extends `junit.framework.TestCase` and belongs to a build rule that includes `java/junit` as a dependency:

```
public abstract class PlanetaryDeathRayTestCase extends TestCase {
  protected PlanetaryDeathRay deathRay;
  @Override protected void setUp() {
    deathRay = createDeathRay();
  }
  @Override protected void tearDown() {
    deathRay = null;
  }
  protected abstract PlanetaryDeathRay createDeathRay();  /* create the PlanetaryDeathRay to test */

  public void testAim() { /* write implementation-independent tests here against deathRay.aim() */ }
  public void testFire() { /* write implementation-independent tests here against deathRay.fire() */ }
}
```

Note that the `setUp` method gets the particular `PlanetaryDeathRay` implementation to test from the abstract `createDeathRay` method. A subclass needs to implement only this method to create a complete test: the `testAim` and `testFire` methods it inherits will be part of the test when it runs:

```
public class BlueLaserPlanetaryDeathRayTest extends PlanetaryDeathRayTestCase {
  protected PlanetaryDeathRay createDeathRay() { return new BlueLaserPlanetaryDeathRay(); }
}
```

You can easily add new tests to this class to test functionality specific to `BlueLaserPlanetaryDeathRay`.

More information, discussion, and archives:
http://googletesting.blogspot.com