

```

package player;

import static org.junit.Assert.*;
import edu.neu.ccs.demeterf.demfgen.lib.List;
import gen.RelationNr;
import gen.Type;
import gen.TypeInstance;

import org.junit.Test;

import player.Util;

public class UtilTest {

    @Test
    public void testVreakEvenValueInteresting() {
        for (int i = 2; i < 128; i += 2) {
            double value = Util.breakEvenValue(makeType(i));
            assertTrue("Break even value of " + i + " is " + value +
                "but should be between 0 and 1 (exclusive)", value < 1 && value > 0);
        }
    }

    @Test
    public void testBreakEvenValueBoring() {
        for (int i = 1; i < 128; i += 2) {
            assertEquals("Index: " + i, 1, Util.breakEvenValue(makeType(i)));
        }
        for (int i = 128; i < 256; ++i) {
            assertEquals("Index: " + i, 1, Util.breakEvenValue(makeType(i)));
        }
    }

    @Test
    public void testBreakEvenPointInteresting() {
        for (int i = 2; i < 128; i += 2) {
            double point = Util.breakEvenPoint(makeType(i));
            assertTrue("Break even point of " + i + " is " + point +
                "but should be between 0 and 1 (exclusive)", point < 1 && point > 0);
        }
    }

    @Test
    public void testBreakEvenPointBoring() {
        for (int i = 1; i < 128; i += 2) {
            assertEquals("Index: " + i, 0, Util.breakEvenPoint(makeType(i)));
        }
        for (int i = 128; i < 256; i += 2) {
            assertEquals("Index: " + i, 1, Util.breakEvenPoint(makeType(i)));
        }
        // Odd values > 128 have maximums at both 0 and 1 so as long as it returns
        // it's correct
        for (int i = 129; i < 256; i += 2) {
            double point = Util.breakEvenPoint(makeType(i));
            assertTrue("Break even point of " + i + " is " + point +
                "but should be either 0 or 1", point == 1 || point == 0);
        }
    }

    @Test
    public void testBreakEvenValueKnownInteresting() {
        assertEquals(4.0/27.0, Util.breakEvenValue(makeType(2)), 0.00001);
        assertEquals(4.0/27.0, Util.breakEvenValue(makeType(4)), 0.00001);
        assertEquals(8.0/27.0, Util.breakEvenValue(makeType(6)), 0.00001);
        assertEquals(4.0/27.0, Util.breakEvenValue(makeType(8)), 0.00001);
        assertEquals(4.0/9.0, Util.breakEvenValue(makeType(22)), 0.00001);
        assertEquals(1.0/4.0, Util.breakEvenValue(makeType(24)), 0.00001);
        assertEquals(1.0/2.0, Util.breakEvenValue(makeType(60)), 0.00001);
        assertEquals(1.0/2.0, Util.breakEvenValue(makeType(90)), 0.00001);
        assertEquals(1.0/2.0, Util.breakEvenValue(makeType(102)), 0.00001);
    }

    @Test
    public void testBreakEvenPointKnownInteresting() {
        assertEquals(1.0/3.0, Util.breakEvenPoint(makeType(2)), 0.00001);
        assertEquals(1.0/3.0, Util.breakEvenPoint(makeType(4)), 0.00001);
        assertEquals(2.0/3.0, Util.breakEvenPoint(makeType(8)), 0.00001);
        assertEquals(1.0/3.0, Util.breakEvenPoint(makeType(6)), 0.00001);
        assertEquals(1.0/3.0, Util.breakEvenPoint(makeType(22)), 0.00001);
        assertEquals(0.5, Util.breakEvenPoint(makeType(24)), 0.00001);
    }
}

```

```
    assertEquals(1.0/2.0, Util.breakEvenValue(makeType(60)), 0.00001);
    assertEquals(1.0/2.0, Util.breakEvenValue(makeType(90)), 0.00001);
    assertEquals(1.0/2.0, Util.breakEvenValue(makeType(102)), 0.00001);
}

private static Type makeType(int relationNr) {
    return new Type(List.<TypeInstance>create(
        new TypeInstance(new RelationNr(relationNr))));
}
}
```