

```

package player.playeragent;

import static org.junit.Assert.*;
import org.junit.Test;

import player.Util;

import edu.neu.ccs.demeterf.demfgen.lib.List;
import gen.Derivative;
import gen.Price;
import gen.RelationNr;
import gen.Type;
import gen.TypeInstance;

public class BuyAgentTest {

    @Test
    public void testAccountNotOverdrawn() {
        BuyAgent b = new BuyAgent();
        double account = 5.0;
        List<Derivative> market = List.<Derivative>create(
            makeDerivative(.5, 1),
            makeDerivative(.4, 3),
            makeDerivative(.6, 5),
            makeDerivative(.9, 128),
            makeDerivative(.8, 130),
            makeDerivative(.8, 132),
            makeDerivative(.8, 134),
            makeDerivative(.8, 138));
        for (Derivative purchased : b.buyDerivatives(market, account)) {
            account -= purchased.price.val;
        }
        assertTrue(account > 0);
    }

    @Test
    public void testDontBuyAtBreakEven() {
        BuyAgent b = new BuyAgent();
        List<Derivative> market = List.<Derivative>create(
            makeDerivativeAtBreakEven(2),
            makeDerivativeAtBreakEven(4),
            makeDerivativeAtBreakEven(6),
            makeDerivativeAtBreakEven(8),
            makeDerivativeAtBreakEven(120),
            makeDerivativeAtBreakEven(62));
        assertTrue(b.buyDerivatives(market, 5.0).isEmpty());
    }

    @Test
    public void testBuyGoodDeals() {
        BuyAgent b = new BuyAgent();
        List<Derivative> market = List.<Derivative>create(
            makeDerivativeJustBelowBreakEven(2),
            makeDerivativeJustBelowBreakEven(4),
            makeDerivativeJustBelowBreakEven(6),
            makeDerivativeJustBelowBreakEven(8),
            makeDerivativeJustBelowBreakEven(120),
            makeDerivativeJustBelowBreakEven(62));
        List<Derivative> bought = b.buyDerivatives(market, 5.0);
        assertEquals(market.length(), bought.length());
    }

    @Test
    public void testBuyOnlyGoodDeals() {
        BuyAgent b = new BuyAgent();
        List<Derivative> market = List.<Derivative>create(
            makeDerivativeJustBelowBreakEven(2),
            makeDerivativeAtBreakEven(4),
            makeDerivativeJustBelowBreakEven(6),
            makeDerivativeAtBreakEven(8),
            makeDerivativeJustBelowBreakEven(120),
            makeDerivativeAtBreakEven(62));
        assertEquals(3, b.buyDerivatives(market, 5.0).length());
    }

    @Test
    public void testPrioritizeBestDeals() {
        BuyAgent b = new BuyAgent();
        List<Derivative> market = List.<Derivative>create(
            makeDerivativeJustBelowBreakEven(2),
            makeDerivative(0.5, 1),

```

```

        makeDerivativeJustBelowBreakEven(6),
        makeDerivative(0.5, 3),
        makeDerivativeJustBelowBreakEven(120),
        makeDerivative(0.5, 5),
        makeDerivative(0.5, 7),
        makeDerivative(0.5, 9),
        makeDerivative(0.5, 11),
        makeDerivative(0.5, 13),
        makeDerivative(0.5, 15),
        makeDerivative(0.5, 17),
        makeDerivative(0.5, 19));
    double account = 5.0;
    List<Derivative> ds = b.buyDerivatives(market, account);
    for (Derivative d : ds) {
        account -= d.price.val;
    }
    assertTrue(account < 0.5);
    assertEquals(11, ds.length());
}

private static Derivative makeDerivative(double price, Type t) {
    return new Derivative("", null, new Price(price), t);
}

private static Derivative makeDerivative(double price, int relationNr) {
    return makeDerivative(price, makeType(relationNr));
}

private static Derivative makeDerivativeAtBreakEven(Type t) {
    return makeDerivative(Util.breakEvenValue(t), t);
}

private static Derivative makeDerivativeAtBreakEven(int relationNr) {
    return makeDerivativeAtBreakEven(makeType(relationNr));
}

private static Derivative makeDerivativeJustBelowBreakEven(Type t) {
    return makeDerivative(Util.breakEvenValue(t) - 0.01, t);
}

private static Derivative makeDerivativeJustBelowBreakEven(int relationNr) {
    return makeDerivativeJustBelowBreakEven(makeType(relationNr));
}

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

```