```
        return p.m_x0 + p.m_x1*b + p.m_x2*b*b + p.m_x3*b*b*b;
    }
    */


}
```

```
/* *********************************
 *   Util.java
 *     Player Agent Utilities
 * *********************************/
package player;

import gen.*;

import java.util.Arrays;
import java.util.Date;
import java.util.Iterator;
import java.util.Random;
import java.util.Set;
import java.util.ArrayList;
import java.util.UUID;
import utils.DocumentHandler;
import utils.DerivativesFinder;
import config.GlobalConfig;
import config.PlayerConfig;
import edu.neu.ccs.demeterf.demfgen.lib.List;
import edu.neu.ccs.satsolver.MaxBiasOutput;
import edu.neu.ccs.satsolver.OutputI;
import edu.neu.ccs.satsolver.PairI;
import edu.neu.ccs.satsolver.Polynomial;
import edu.neu.ccs.satsolver.PolynomialI;

/* TODO: This should be changed so we can provide a Non-FileSystem
 *         interface to the Administrator.  Many of the functions below
 *         just need to ask the Administrator for values from the store
 *         or accounts
 */

/** Various Player based utilities */
public class Util{
        static Random rand = new Random();
        public Util(){

        }



    /** Random Double between 0..1 */
    public static double random(){ return rand.nextDouble(); }

    /** Random Integer between 0..(bound-1)*/
    public static int random(int bound){ return rand.nextInt(bound); }

    /** Random coin flip of the given bias */
    public static boolean coinFlip(double bias){ return (Util.random() < bias); }

    /** Random coin flip, bias of 0.5 */
    public static boolean coinFlip(){ return coinFlip(0.5); }

    /** Returns the upper bound at which a given derivative should be purchased for profit */
    /** Write a player transaction */
    public static void commitTransaction(PlayerTransaction pTrans){
        String fileName = pTrans.player.name+GlobalConfig.DONE_FILE_SUFFIX;
        DocumentHandler.write(pTrans.print(),(PlayerConfig.BLACKBOARD_PATH+GlobalConfig.SEPAR+fileName)
);
    }

    /** Find the account for the given player */
    public static double getAccount(Player p){ return getAccounts().getAccount(p); }

    /** Find the Derivatives (that Player is selling) that need RawMaterials */
    public static List<Derivative> needRawMaterial(Player player){
        return DerivativesFinder.findDersThatNeedRM(getStore().stores, player);
    }

    /** Find the Derivatives (that Player is buying) that need to be Finished */
    public static List<Derivative> toBeFinished(Player player){
        return DerivativesFinder.findDersThatNeedFinishing(getStore().stores, player);
    }

    /** Get the minimum price decrement when reoffering */
    public static double getMinPriceDec(){ return getConfig().MPD; }

    /** Get the current Types in the Store */
    public static List<Type> existingTypes(){
        return DerivativesFinder.findExistingDerTypes(getStore().stores);
    }
```

```
/** Gets the derivative (int[]) of a PolynomialI instance */
public static double[] computeDeriv(PolynomialI p) {
    double[] deriv = new double[3];
    deriv[0] = p.getCoefficient(1);
    deriv[1] = p.getCoefficient(2)*2;
    deriv[2] = p.getCoefficient(3)*3;
    return deriv;
}



/** compute the maximum Y value of the given polynomial **/

public static double computeMaxY(PolynomialI p, double max) {
    // Original Polynomial
    double p0 = p.getCoefficient(0);
    double p1 = p.getCoefficient(1);
    double p2 = p.getCoefficient(2);
    double p3 = p.getCoefficient(3);

    return (p3*max*max*max) + (p2*max*max) + (p1*max) + p0;
}


/** computes the break even price for a given derivative */
public static double breakEven(Derivative d){
    OutputI o = getBias(d);
  PolynomialI p = o.getPolynomial();
    if(d.type.instances.length() == 1){
            if(d.type.instances.top().r.v > 127 || isOdd(d.type.instances.top().r.v)) return 1.0;
            if(d.type.instances.top().r.v == 0) return 0.0;
    }
    else if(d.type.instances.length() >= 2){
            if(d.type.instances.contains(new TypeInstance(new RelationNr(0)))) return 0.0;
            //if(isTrickster(d)) {return Math.min(.5, computeMaxY(p, o.getMaxBias()));}
    }
    //System.out.println(p.toString());
    return computeMaxY(p, o.getMaxBias());
}

/** determines whether this derivative is tricky
 * (contains an odd number and also an even number over 127)
 *  Not a very useful function, as it turns out.
 **/
public static boolean isTrickster(Derivative d) {
   boolean odd = false;
   boolean over = false;
   Iterator<TypeInstance> iter = d.type.instances.iterator();
   while(iter.hasNext()) {
      TypeInstance type = iter.next();
      int r = type.r.v;
      if(r > 127 && !isOdd(r)) {
         over = true;
      }
      else if (isOdd(r)) {
         odd = true;
      }
   }
   return odd && over;
}

/** Calculates the max bias & polynomial given a Derivative.
 * In this version, just take the lowest breakeven of the relation numbers. This minimizes risk.

 */
public static OutputI getBias(Derivative d) {
    List<TypeInstance> types = getImportantTypes(d);
    Iterator<TypeInstance> iter = types.iterator();

    double this_bEven, this_maxB, maxB = 0;
    double bEven = 1.0;
    Polynomial final_poly = null;

    while (iter.hasNext()) {
            Polynomial poly = Polynomial.getPolynomial(iter.next().r.v);
            if (final_poly == null) final_poly = poly;
            this_maxB = computeMaxBias(poly);
            this_bEven = computeMaxY(poly, this_maxB);
            if(this_bEven <= bEven){
```

```
                        bEven = this_bEven;
                        maxB = this_maxB;
                        final_poly = poly;
                }

        }
        return new MaxBiasOutput(final_poly, maxB);
    }

    public static OutputI getBias_v2(Derivative d) {
        List<TypeInstance> types = getImportantTypes(d);
        double length = (double)types.length();
        Iterator<TypeInstance> iter = types.iterator();

        Polynomial final_poly = null;
        double fraction = 1/length;

        while (iter.hasNext()) {
                Polynomial poly = Polynomial.getPolynomial(iter.next().r.v);
                poly = poly.multiplyByDouble(fraction);

                if (final_poly == null) final_poly = poly;
                else final_poly = final_poly.add(poly);
        }

        double max_bias = computeMaxBias(final_poly);

        return new MaxBiasOutput(final_poly, max_bias);
    }


    /**
     * Calculates the max bias and polynomial for finishing a derivative given an InputInitialI instanc
e.
     */
    public static OutputI getBiasForFinishing(InputInitial input) {
      Set<PairIt> pairs = input.p;
      Iterator<PairIt> p_iter = pairs.iterator();

      Derivative d = input.d;

      /****** THIS IS WHERE I CHANGED THE CODE **********/
      List<TypeInstance> types = relationFocus(d);
      /****** RIGHT HERE RIGHT HERE RIGHT HERE **********/

          Iterator<TypeInstance> t_iter = types.iterator();

      Polynomial final_poly = null;

      while (p_iter.hasNext()) {
                PairI p = (PairI) p_iter.next();
                double fraction = p.getFraction();
                while(t_iter.hasNext()){
                        int this_r = t_iter.next().r.v;

                        if(p.getRelationNumber() == this_r){
                                Polynomial poly = Polynomial.getPolynomial(this_r);

                                poly = poly.multiplyByDouble(fraction);
                                if (final_poly == null) {final_poly = poly;     }
                                else {final_poly = final_poly.add(poly);}
                        }
                }//end of inner while loop
                t_iter = types.iterator();
      }//end of outer while loop
      double max_bias = computeMaxBias(final_poly);
      return new MaxBiasOutput(final_poly, max_bias);


    }

    /**
     * Returns whether a given relation is contained in another relation given
     * the relation numbers
     */
    public static boolean isContained(int rsmaller, int rlarger) {
       Pair<Integer, int[]> rs = getRelationReduction(rsmaller);
       Pair<Integer, int[]> rl = getRelationReduction(rlarger);

       return isContained(rs.b, rl.b);
    }
```

```
/** Returns whether a given int array is contained in another int array
 *  Used to say whether r1 implies r2
 */
public static boolean isContained(int[] smaller, int[] larger) {
    for (int i = 0; i < 8; i++) {
            if(smaller[i] == 1 && larger[i] == 0) {
                    return false;
            }
    }
    //if we get here, it's contained
    return true;
}


/** sorts a list of type instances small to big
 *  need to use java arrays sort because DemeterF sort doesnt work*/
public static List<TypeInstance> sort(List<TypeInstance> lot){
    Iterator<TypeInstance> iter = lot.iterator();
    TypeInstance[] types = new TypeInstance[lot.length()];
    int count = 0;
    while(iter.hasNext()){
            types[count] = iter.next();
            count++;
    }
    List<TypeInstance> sorted_to_return = List.<TypeInstance> create();
    Arrays.sort(types, new SmallToBig());

    for(int i = 0; i < types.length; i++){
            sorted_to_return = sorted_to_return.append(types[i]);
    }
    return sorted_to_return;
}

/** returns a List of the important types, reduced from a full list of a given derivative */
public static List<TypeInstance> getImportantTypes(Derivative d) {
    List<TypeInstance> lot = sort(d.type.instances);
    Iterator<TypeInstance> iter = lot.iterator();

    // alist: list of Pairs of relation numbers, and their prime factorization
    ArrayList<Pair<Integer, int[]>> alist = new ArrayList<Pair<Integer, int[]>>();

    //fill alist
    while(iter.hasNext()) {
            int r = iter.next().r.v;
            alist.add(getRelationReduction(r));
    }

    ArrayList<Pair<Integer, int[]>> rList = new ArrayList<Pair<Integer, int[]>>(alist);
    // data to be returned
    Iterator<Pair<Integer, int[]>> a_iter = alist.iterator();
    while (a_iter.hasNext()) {
            Pair<Integer, int[]> p = a_iter.next();
            for(int i=1; i<rList.size(); i++){
                    Pair<Integer, int[]> temp_p = rList.get(i);
                    if(p.a < temp_p.a){
                            if(isContained(p.b, temp_p.b)){
                            rList.remove(i);
                            i--;
                            }
                    }
            }
    }

    List<TypeInstance> returnList = List.<TypeInstance>create();
    TypeInstance this_t;
    for(int i = 0; i < rList.size(); i++){
            this_t = new TypeInstance(new RelationNr(rList.get(i).a));
            returnList = returnList.append(this_t);
    }
    return returnList;
}

/**
 * Calculates which relation number (or numbers) should be focused on for finishing, based on
 * the weight of the numbers in the Raw Material
 */
public static List<TypeInstance> relationFocus(Derivative d) {
   List<Constraint> loc = d.optraw.inner().instance.cs;
   List<TypeInstance> lot = d.type.instances;
   ArrayList<RelationStuff> rstuffs = new ArrayList<RelationStuff>();
```

```
      Iterator<TypeInstance> iterlot = lot.iterator();

      while(iterlot.hasNext()) {
         TypeInstance t = iterlot.next();

         int r = t.r.v;

         RelationStuff rstuff = new RelationStuff(0,0,0,0);
         rstuff.clauses = getNumClauses(r);
         rstuff.breakeven = getBreakEven(r);
         rstuff.relation = r;

         Iterator<Constraint> iterloc = loc.iterator();
         while(iterloc.hasNext()) {
            Constraint c = iterloc.next();
            if(c.r.v == r) {
               rstuff.weight += c.w.v;
            }
         }

         rstuff.value = rstuff.breakeven * rstuff.weight;
         rstuffs.add(rstuff);
      }

      for(int i = 0; i < rstuffs.size(); i++) {
         for(int n = i+1; n < rstuffs.size(); n++) {
            RelationStuff ri = rstuffs.get(i);
            RelationStuff rn = rstuffs.get(n);
            if(isContained(ri.relation, rn.relation)) {
               rstuffs.get(i).chain.append(rstuffs.get(n));
               rstuffs.get(n).chain.append(rstuffs.get(i));
               rstuffs.get(i).value += (ri.clauses/rn.clauses) * ri.breakeven * rn.weight;
               rstuffs.get(n).value += (ri.clauses/rn.clauses) * rn.breakeven * ri.weight;
            }
         }
      }

      List<TypeInstance> types = List.<TypeInstance>create();

      Iterator<RelationStuff> iter = rstuffs.iterator();
      while(iter.hasNext()) {
         RelationStuff rel = iter.next();
         if(rel.isGreatestInChain()) {
            types = types.push(new TypeInstance(new RelationNr(rel.relation)));
         }
      }
      return types;
   }

   public static int getNumClauses(int r){
      int count = 0;
      for (int i = 7; i >= 0; i--) {
         if (r >= Math.pow(2,i)) {
            count++;
            r -= Math.pow(2, i);
         }
      }
      return count;
   }

   /** returns an array representing the 'primes' this relation number reduces to **/
   public static Pair<Integer, int[]> getRelationReduction(int r) {
      int original_R =r;

      int[] reduction = new int[8];

      for (int i = 7; i >= 0; i--) {
         if (r >= Math.pow(2, i)) {
            reduction[i] = 1;
            r -= Math.pow(2,i);
         }
      }
      return new Pair<Integer, int[]>(original_R, reduction);
   }

   /** gets the breakeven value for a given relation number **/
   public static double getBreakEven(int r){
      Polynomial poly = Polynomial.getPolynomial(r);
      double maxB = computeMaxBias(poly);
      return computeMaxY(poly, maxB);
   }
```

```
/** Gets the Max Bias: used to compute by solving the derivative to zero, but with
 * too many exceptions, was reworked to plugin numbers and find the closest value*/
private static double computeMaxBias(Polynomial p) {
    // Original Polynomial
    double p0 = p.getCoefficient(0);
    double p1 = p.getCoefficient(1);
    double p2 = p.getCoefficient(2);
    double p3 = p.getCoefficient(3);


            double current_plugin_max = 0.0;
            double bMax = 0.0;
            for(double i = 0.0; i <= 1.0 ; i = i + .005){
                    double this_answer = (p3*i*i*i) + (p2*i*i) + (p1*i) + p0;
                    //System.out.println("i: " +i+ " this answer = " + this_answer);
                    if(this_answer > current_plugin_max){
                            bMax = i;
                            current_plugin_max = this_answer;
                    }

            }
            return (bMax > .99)? 1.0:bMax;  // if .99, make 1 to account for java math
  }

/** Find all Derivatives ForSale */
public static List<Derivative> forSale(PlayerID id){
    return DerivativesFinder.findDerivativesForSaleByOthers(getStore().stores, id);
}

/** Find uniquely typed Derivatives ForSale */
public static List<Derivative> uniquelyTyped(List<Derivative> forSale){
    return DerivativesFinder.findUniqueDerivatives(forSale);
}

/** Returns true if a number is odd */
public static boolean isOdd(int x){ return ((x%2) == 1); }

/** Get a Fresh Derivative name */
public static String freshName(Player p){
    UUID newID = UUID.randomUUID();
    return p.name+"_"+newID.toString().replace('-', '_');
}

/** Returns an even random number between 1 and max */
public static int randomEvenOneToMax(int max){
    int x;
    do{x = Util.random(max+1);}
    while(x % 2 ==1 || x == 0);
    return x;
}

/** Returns an odd random number between 1 and max */
public static int randomOddOneToMax(int max){
    int x;
    do{x = Util.random(max+1);}
    while(x % 2 == 0 || x == 0);
    return x;
}

/** Returns a list of unique variables contained in a given Derivative's RawMaterial */
public static List<Variable> getUniqueVariables(Derivative d){
    Date uni_start = new Date();
    List<Constraint> lcon = d.optraw.inner().instance.cs;
    List<Variable> lov = List.<Variable> create();
    //Date iter_start = new Date();
    Iterator<Constraint> iter = lcon.iterator();
    //Date iter_end = new Date();
    //System.out.println("Iterator create time: " + (iter_start.getTime() - iter_end.getTime()));
    Iterator<Variable> viter;
    Variable v;
    while(iter.hasNext()){
            viter = iter.next().vs.iterator();
            //Date d_start = new Date();
            while(viter.hasNext()){
                    v = viter.next();
                    if(!lov.contains(v)) lov = lov.push(v);
            }
            //Date d_end = new Date();
            //System.out.println("inner loop time: " + (d_start.getTime() - d_end.getTime()));
    }
```

```
        Date uni_end = new Date();
        //System.out.println("Unique variables time: " + (uni_end.getTime() - uni_start.getTime()) + "
ms");
        return lov;
    }

    /** Generates a binary number given the integer and the desired 'length' of binary number **/
    public static int[] generateBinary(int number, int length) {

       int[] reduction = new int[length];

       for (int i = length; i >= 0; i--) {
          if (number >= Math.pow(2, i)) {
             reduction[i] = 1;
             number -= Math.pow(2,i);
          }
       }
       return reduction;
    }

    /** Get a Fresh Type, not in the Store. Not really nec. as the Players will be more complex */
    public static Type freshType(List<Type> existing){
        Type type, type2;
        int r1, r2;

        do{
                r1 = 2;
                r2 = Util.randomOddOneToMax(255);
                type = new Type(new Classic(), List.create(new TypeInstance(new RelationNr(r1))).push(n
ew TypeInstance(new RelationNr(r2))));
                type2 = new Type(new Classic(), List.create(new TypeInstance(new RelationNr(r2))).push(
new TypeInstance(new RelationNr(r1))));
        }

        while(existing.contains(type) || existing.contains(type2) || (r1 == r2));
        return type;
    }

    private static Accounts getAccounts()
    { return DocumentHandler.getAccounts(PlayerConfig.BLACKBOARD_PATH); }
    private static Store getStore()
    { return DocumentHandler.getStore(PlayerConfig.BLACKBOARD_PATH); }
    private static Config getConfig()
    { return DocumentHandler.getConfig(PlayerConfig.BLACKBOARD_PATH); }

    //---------------------------------------------------------------------
    // OBSOLETE CODE:
    /** Calculates the max bias & polynomial given a Derivative.*/
    /*
    public static OutputI getBias_v1(Derivative d) {
        List<TypeInstance> types = getImportantTypes(d);
        double length = (double)types.length();
        Iterator<TypeInstance> iter = types.iterator();

        Polynomial final_poly = null;
        double fraction = 1/length;

        while (iter.hasNext()) {
                Polynomial poly = Polynomial.getPolynomial(iter.next().r.v);
                poly = poly.multiplyByDouble(fraction);

                if (final_poly == null) final_poly = poly;
                else final_poly = final_poly.add(poly);
        }

        double max_bias = computeMaxBias(final_poly);

        return new MaxBiasOutput(final_poly, max_bias);
    }
    */

    /** returns a List of the important types, reduced from a full list of given a derivative */
    /*public static List<TypeInstance> getImportantTypes(Derivative d){
        List<TypeInstance> lot = d.type.instances;
        Iterator<TypeInstance> iter = lot.iterator();
        ArrayList<Integer> primes = new ArrayList<Integer>();
        primes.add(1);
        primes.add(2);
        primes.add(4);
        primes.add(8);
        primes.add(16);
```

```
        primes.add(32);
        primes.add(64);
        primes.add(128);

        ArrayList<Integer> presentprimes = new ArrayList<Integer>();
        ArrayList<Integer> nonprimes = new ArrayList<Integer>();
        //fills the 'present' array of 'primes' that are present in lot,
        //and fills the 'nonprimes' array of nonprimes that are present
        while(iter.hasNext()) {
                Integer n = iter.next().r.v;
                if (primes.contains(n)) {
                        presentprimes.add(n);
                }
                else {
                        nonprimes.add(n);
                }
        }

        List<TypeInstance> importantTypes = List.<TypeInstance>create();

        Iterator<Integer> np_iter = nonprimes.iterator();

        //only add nonprimes that don't have reduced primes in the list
        while(np_iter.hasNext()) {
                int n = np_iter.next();
                Pair<Integer, int[]> reduction = getRelationReduction(n);
                boolean remove = false;
                for(int i = 0; i < 8; i++) {
                        if(presentprimes.contains(reduction.b[i])) remove = true;

                }
                if(!remove) {
                        //System.out.println("important type for " + d.name + ": " + n);
                        importantTypes = importantTypes.push(new TypeInstance(new RelationNr(n)));
                }
        }

        //now add the already present primes to the list
        for (int i = 0; i < presentprimes.size(); i++) {
                //System.out.println("important type for " + d.name + ": " + presentprimes.get(i));
                importantTypes = importantTypes.push(new TypeInstance(new RelationNr(presentprimes.get(
i))));
        }

        return importantTypes;
    }
    */


    /** computes a polynomial encoded into an int array given an array of q values
     *          matrix q:[polynomial values for q[i] b^0, b^1, b^2, b^3]
     *          note: each row represents the resulting expanded polynomials from q(0) - q(3)
     */
    /*
    public static int[] computePolynomial(int[] cat) {
        int[] polynomial = new int[]{0,0,0,0};
        int[][] q = new int[][] {{1,-3,3,-1},
                                                        {0,1,-2,1},
                                                        {0,0,1,-1},
                                                        {0,0,0,1}};

        // Run through and build polynomial from q matrix values
        for (int i=0; i < 4; i++) {
                if(cat[i] > 0) {
                        for(int j=0; j < 4; j++) {
                                polynomial[j] += cat[i] * q[i][j];
                        }
                }
        }
        return polynomial;
    }
    */

    /** Find the break even point for the given derivative */
    /*
    public static double breakEven(Derivative d){
        int rank = 3;
        // Always 3 for this version of the game.
        int relation = d.type.instances.top().r.v;                                      // Grabs the re
lation # out of the derivative
```

```
        if (relation%2 == 1) return 1;          // Special cases.  Odd #s are always 1
        if (relation == 0) return 0;            // 0 returns 0
        if (relation > 127) return 1;           // Anything above 127 is also 1

        Relation r = new Relation(rank, relation);
        int[] cat = new int[4];                         // Array of q values

        for (int i = 0; i < 4; i++) {           // Populate array
                cat[i] = r.q(i);
        }
        int [] poly = computePolynomial(cat);
        return computeMax(poly);
    }
    */

    /** compute the maximum bias of the given polynomial from a CSP formula **/
    /*
    public static double computeBMaxCSP(int[] poly) {
        int[] deriv = computeDeriv(poly);
        double a = deriv[2];    // ax^2
        double b = deriv[1];    // bx
        double c = deriv[0];    // c

        if(a == 0) {                    // if a is zero, use -c/b
                return -(c/b);
        }

        else {                                  // if quadratic equation is needed, find both roots, an
d return the max value of the two
                double pos = ((-b + Math.sqrt((b*b) - (4*a*c)))/(2*a));
                double neg = ((-b - Math.sqrt((b*b) - (4*a*c)))/(2*a));
                double checkpos = poly[0] + (poly[1]*pos) + poly[2]*pos*pos + poly[3]*pos*pos*pos;
                double checkneg = poly[0] + (poly[1]*neg) + poly[2]*neg*neg + poly[3]*neg*neg*neg;

                if(Math.max(checkpos, checkneg) == checkpos) return pos;
                else return neg;
        }
    }
    */



    /** compute the maximum bias of the given relation **/
    /*
    public static double computeBMax(int relation) {
        Relation r = new Relation(3, relation);

        if (relation%2 == 1) return 0;          // Special cases.  Odd #s are always 0
        if (relation == 0) return 0;            // 0 returns 0

        int[] cat = new int[4];                         // Array of q values

        for (int i = 0; i < 4; i++) {           // Populate array
                cat[i] = r.q(i);
        }
        int[] poly = computePolynomial(cat);
        int[] deriv = computeDeriv(poly);
        double a = deriv[2];    // ax^2
        double b = deriv[1];    // bx
        double c = deriv[0];    // c

        if(a == 0) {                    // if a is zero, use -c/b
                double n = -(c/b);
                return n;
        }

        else {                                  // if quadratic equation is needed, find both roots, an
d return the max value of the two
                double pos = ((-b + Math.sqrt((b*b) - (4*a*c)))/(2*a));
                double neg = ((-b - Math.sqrt((b*b) - (4*a*c)))/(2*a));
                double checkpos = poly[0] + (poly[1]*pos) + poly[2]*pos*pos + poly[3]*pos*pos*pos;
                double checkneg = poly[0] + (poly[1]*neg) + poly[2]*neg*neg + poly[3]*neg*neg*neg;
                if (Math.max(checkpos, checkneg) == checkpos) return pos;
                else return neg;
        }
    }
    */

    /** given a Polynomial p and an int b , return p(b) */
    /*
     * public static double pluginToPoly(Polynomial p, double b){
```