```
package player.playeragent;

import edu.neu.ccs.demeterf.demfgen.lib.List;
import edu.neu.ccs.demeterf.demfgen.lib.ident;
import gen.Assignment;
import gen.Constraint;
import gen.Derivative;
import gen.RawMaterial;
import gen.RawMaterialInstance;
import gen.Secret;
import gen.Type;
import gen.TypeInstance;
import gen.Variable;
import gen.Weight;

import java.util.HashMap;
import java.util.HashSet;

import player.Bias;
import player.Minimizer;
import player.Permutations;
import player.PlayerI;
import player.Util;

/**
 * Class for delivering raw material for a derivative.
 */
public class DeliverAgent implements PlayerI.DeliverAgentI {

    /**
     * Adds raw materials to the given derivatives.
     */
    public Derivative deliverRawMaterial(Derivative needRM) {
        RawMaterialInstance raw = rawMaterialInst(needRM.type);

        if (needRM.type.kind instanceof Secret) {
            raw = addSecretAssignment(raw);
        }

        return needRM.deliver(raw);
    }

    /**
     * Adds the secret assignment to the raw material instance.
     */
    public RawMaterialInstance addSecretAssignment(RawMaterialInstance r) {
        Assignment assign = new FinishAgent()
                .bestAssignment(new RawMaterial(r));

        return new RawMaterialInstance(r.cs, assign);
    }

    /**
     * Compute a raw material instance for the given derivative.
     */
    public RawMaterialInstance rawMaterialInst(Type t) {
        List<Constraint> constraints = List.<Constraint> create();

        Type simple = new Minimizer().simplify(t);

        // Compute relative weights
        HashMap<Integer, Integer> weights = computeRelativeWeights(simple);

        // Build the list of variables
        int max = maxVars(simple.instances.length());
        List<Variable> vars = createVars(max);

        Permutations p = new Permutations(vars);
        HashSet<List<Variable>> comb = p.combinations();
        HashSet<List<Variable>> perms = p.permutations(comb);

        Weight w;

        for (TypeInstance i : simple.instances) {
            // Build the list of constraints
            for (List<Variable> v : perms) {
                // Get this type's relative weight
                w = new Weight(weights.get(i.r.v));

                constraints = constraints.append(new Constraint(w, i.r, v));
            }
```

```
        }

        return new RawMaterialInstance(constraints);
    }

    /**
     * Generate the correct number of variables to ensure we never exceed the
     * maximum number of constraints for a raw material.
     */
    private int maxVars(int t) {
        int right = Util.getMaxRawMaterialLen() / t;

        // At least 3
        int n = 3;

        while (true) {
            // (n choose 3) * 6 * t <= max, find n
            int left = n * n * n - 3 * n * n + 2 * n;

            // Maximum constraints exceeded
            if (left > right) {
                n--;
                break;
            } else {
                n++;
            }
        }

        return n;
    }

    /**
     * Create a list of variables using letters in the alphabet.
     *
     * @param num
     *            The number of variables to create (1-26).
     * @return The list of variables.
     */
    private List<Variable> createVars(int num) {
        List<Variable> result = List.<Variable> create();

        String alphabet = "abcdefghijklmnopqrstuvwxyz";

        for (int i = 0; i < num; i++) {
            result = result.append(new Variable(new ident(alphabet.substring(i,
                    i + 1))));
        }

        return result;
    }

    /**
     * Create a map of relative weights given a type.
     *
     * @param t
     *            The type.
     * @return A map representing each type's relative weight.
     */
    private HashMap<Integer, Integer> computeRelativeWeights(Type t) {
        int numTypes = t.instances.length();

        HashMap<Integer, Integer> weights = new HashMap<Integer, Integer>(
                numTypes);

        // Compute the maximum break-even across the derivative
        double maxBE = new Bias(t).breakEven();

        // Compare each break-even to the maximum
        for (TypeInstance i : t.instances) {
            double breakEven = new Bias(new Type(List.<TypeInstance> create(i)))
                    .breakEven();

            int relativeWeight = new Double(Math.ceil(maxBE / breakEven))
                    .intValue();

            weights.put(i.r.v, relativeWeight);
        }

        return weights;
    }
}
```