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

import gen.*;

import java.util.Random;
import java.util.UUID;
import java.util.Set;

import utils.DocumentHandler;
import utils.DerivativesFinder;
import config.GlobalConfig;
import config.PlayerConfig;
import edu.neu.ccs.demeterf.demfgen.lib.List;
import java.util.Hashtable;
import java.util.Iterator;
import edu.neu.ccs.evergreen.ir.*;

/* 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();

    /** 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); }

    /** 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);
    }

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

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

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

    /** 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;
        do{ type = new Type(List.create(new TypeInstance(new RelationNr(Util.random(256))))); }
        while(existing.contains(type));
        return type;
    }

    /** Reduces the Raw Materials Based on a Literal*/
    public static RawMaterial getShannon(RawMaterial rawMat, Literal lit){
        List<Constraint> retCons = List.create();
        Iterator<Constraint> iter = rawMat.instance.cs.iterator();
        while(iter.hasNext()){
                Constraint tempConst = iter.next();
                if(tempConst.vs.contains(lit.var)){
                        int varPosin = tempConst.vs.index(lit.var);
                        Relation newRel = new Relation(tempConst.w.v, tempConst.r.v);
                        RelationNr reducNum = new RelationNr(newRel.reduce(varPosin, lit.value.sign()))
;
                        tempConst = new Constraint(tempConst.w, reducNum, tempConst.vs);
                }
                retCons = retCons.append(tempConst);
        }
        return new RawMaterial(new RawMaterialInstance(retCons));
    }

    /** Returns the set of variables in the given raw material. */
    public static Set<Variable> getVars(RawMaterial raw){
        java.util.HashSet<Variable> vars = new java.util.HashSet<Variable>();
        Iterator<Constraint> cIter = raw.instance.cs.iterator();
        while(cIter.hasNext()){
                Iterator<Variable> vIter = cIter.next().vs.iterator();
                while(vIter.hasNext()){
                        vars.add(vIter.next());
                }
        }
        return vars;
    }

    /** Returns all the Relation Numbers in the Raw Materials Constraints */
    public static List<RelationNr> getRelationNumbers(RawMaterial rawMat){
        List<RelationNr> relationNumbers = List.create();
        Iterator<Constraint> iter = rawMat.instance.cs.iterator();
        while(iter.hasNext()){
                Constraint temp = iter.next();
                if (!relationNumbers.contains(temp.r)){
                        relationNumbers = relationNumbers.append(temp.r);
                }
        }
        return relationNumbers;
    }

    /** Returns all the Weighted Fractions for a given Raw Material's Constraints*/
    public static Hashtable<RelationNr, Double> getWeightedFracts(RawMaterial rawMat){
        Hashtable<RelationNr, Double> mapFracts = new Hashtable<RelationNr, Double>();
        Double total = new Double(0);
        Iterator<Constraint> iter = rawMat.instance.cs.iterator();
        while(iter.hasNext()){
                Constraint temp = iter.next();
                total += new Double(temp.w.v);
                if(mapFracts.containsKey(temp.r))
                        mapFracts.put(temp.r, mapFracts.get(temp.r) + new Double(temp.w.v));
                else
                        mapFracts.put(temp.r, new Double(temp.w.v));
        }
        Iterator<RelationNr> mapKeyIter = mapFracts.keySet().iterator();
        while(mapKeyIter.hasNext()){
                RelationNr temp = mapKeyIter.next();
                mapFracts.put(temp, mapFracts.get(temp)/total);
        }
        return mapFracts;
    }

    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); }

    /** Returns true if a implies b. */
    public static boolean implies(RelationNr a, RelationNr b){
        return a.v == (a.v & b.v);
    }

    /** Prune all relation numbers from the given set that are implied by other relation numbers in it.
 */
    public static List<RelationNr> pruneImplied(List<RelationNr> relations){
        List<RelationNr> emptyNrs = List.create();
        return Util.pruneImpliedHelper(emptyNrs, relations);
    }

    /** Iterates through all basic implicators so far checking the first element of the rest and prunin
g any implicated  */
    private static List<RelationNr> pruneImpliedHelper(List<RelationNr> passed, List<RelationNr> rest){
        if(rest.isEmpty())
                return passed;
        Iterator<RelationNr> iter = passed.iterator();
        RelationNr top = rest.top();
        rest = rest.pop();
        while(iter.hasNext()){
                RelationNr a = iter.next();
                if(Util.implies(a, top))
                        return Util.pruneImpliedHelper(passed, rest);
                if(Util.implies(top, a)){
                        passed = passed.remove(a);
                }
        }
        return Util.pruneImpliedHelper(passed.append(top), rest);
    }

    /** Returns a list of strings that represent all implications present in the given list of Relation
s excluding self-implications. */
    public static List<String> listImplications(List<RelationNr> rels){
        List<String> relStrs = List.create();
        for(int i=0; i<rels.length(); i++){
                for(int j=0; j<rels.length(); j++){
                        if(i != j){
                                RelationNr iRel = rels.lookup(i), jRel = rels.lookup(j);
                                if(Util.implies(iRel, jRel)){
                                        relStrs = relStrs.append(iRel.toString() + "=>" + jRel.toStrin
g());
                                }
                        }
                }
        }
        return relStrs;
    }
}
```